Imperial College London
Department of Electrical and Electronic Engineering

CONCURRENCY AND DATA LOCALITY FOR SPARSE LINEAR ALGEBRA ON MODERN PROCESSORS

ANDREA PICCIAU

Supervised by Prof. George A. Constantinides and Dr. Eric C. Kerrigan

Subitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering of Imperial College London
and the Diploma of Imperial College London

Declaration of originality

I, Andrea Picciau, declare that all of the work in this dissertation is either my own or appropriately referenced.

ABSTRACT

Graphics processing units (GPUs) are used as accelerators for algorithms in which the same instructions are carried out on different data. Algorithms for sparse linear algebra can achieve good performance on GPU, although they tend to have an irregular pattern of accesses to memory. The performance of these algorithms is highly dependent on input data. In fact, the parallelism these algorithms can achieve is limited by the opportunities for concurrency given by the data.

Focusing on the solution of sparse riangular linear systems of equations, this thesis shows that a good partitioning of the data and a good scheduling of the computation can greatly improve performance on GPUs. For this class of algorithms, a partition of the data that maximises concurrency in the execution does not necessarily achieve the best performance. Instead, improving data locality by reducing concurrency reduces the latency of memory access and consequently the execution time.

First, this work characterises the problem formally using graph theory and performance models. Then, algorithms that can be used effectively to partition the data are described. These algoritms aim to balance concurrency and data locality automatically. This approach is evaluated experimentally on the solution of linear equations with the preconditioned conjugate gradient method. Also, the thesis shows that the proposed approach can be used in the case when a matrix changes during the execution of an algorithm from one iteration to the other, like in the simplex method. In this case, the approach proposed in this thesis allows to update the partition of the matrix from one iteration to the other. Finally, the algorithms and performance models developed in the thesis are used to discuss the limitations of the acceleration of the simplex method with GPUs.

ACKNOWLEDGMENTS

The work presented in this thesis was carried out under the supervision of George A. Constantinides and Eric C. Kerrigan and was funded by Siemens Corporate Research.

I would like to thank my supervisors George and Eric for their support in the past four years. Their insight into research and their attention to detail is of inspiration to me and my meetings with them have always been occasions for my professional and personal development.

I am especially grateful for the love, encouragement, and tolerance of my partner Enrica without whom this work would not have been possible. Also I would like to thank my family, in particular my parents and my sister, for their emotional support during my studies.

During these four years at Imperial College, I have been blessed with the friendship of many people in the Circuit and Systems Group. Among current and former members of the research group, I would like to thank in particular Andrea Suardi, Shane Fleming, Gordon Inggs, Josh Levine, James Davis, Ivan Beretta, Nadesh Ramanathan, and John Wickerson.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF LISTINGS

1 INTRODUCTION

Driven by the limitations of microchip manufacturing technologies, in the past ten years high-performance computing has moved from a homogeneous system model, in which all computing devices are of the same type, to a heterogeneous system model, in which hardware accelerators carry out a part of the computation. This change in the computing model, however, has introduced the necessity to partition data and organise the computation across the devices to achieve the best performance.

This thesis discusses techniques to automatically partition data and organise the computation on accelerators for some linear algebra algorithms. The motivations and context for this work are examined in Section 1.1 and the ojectives of the thesis are explained in Section 1.2. Also, an overview of the thesis is shown in Section 1.3. Section 1.4 contains a list of research contributions in this thesis.

1.1 MOTIVATION

Graphics processing units (GPU) are by far the most popular type of hardware accelerator for high-performance computing, with a market share of about 80 % [1]. The success of GPUs is due to their higher performance, measured in floating-point operations per second (FLOPS), with respect to multi-core CPUs. Figure 1.1 shows that, as of 2016, the theoretical peak performance of a high-end GPU is about 2.6 times that of the latest announced multi-core CPU and it will be almost twice as high in two years' time [2]. Also Intel Xeon Phi accelerators, which are many-core processors, have a theoretical peak performance close to that of a high-end GPU. Furthermore, many linear algebra algorithms, such as matrix-matrix multiplication, can be carried out on a GPU and achieve a performance close to the peak [3, 4].

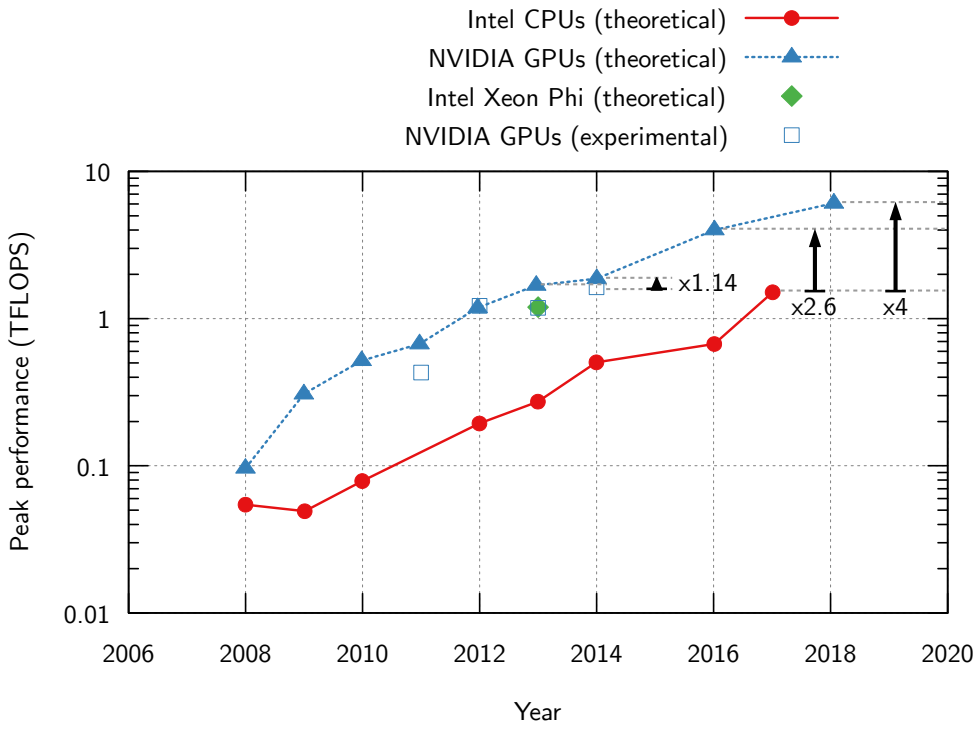


Figure 1.1: Increase in the theoretical peak performance of Nvidia GPUs, Intel CPUs, and Intel Xeon Phi on double floating point precision data, between 2008 and 2018. Data from [2–7]. The experimental data was obtained by executing a matrix-matrix multiplication benchmark.

In general, linear algebra algorithms achieve high performance on hardware accelerators such as GPUs and many-core processors if some part of the software can carry out computations while another part is idle, waiting for data transfer. This is also called *memory access latency hiding*.

Good memory access latency hiding is mainly achieved by *dense* linear algebra algorithms, such as those implemented in the libraries BLAS [8], LAPACK [9], and their GPU equivalents CUBLAS [10] and MAGMA [11]. These algorithms operate on data structures in which contiguous matrix entries have contiguous addresses in memory.

However, a great number of linear algebra algorithms use a different type of data structure, called *sparse*, in which matrix entries that are equal to zero are not stored. As a result, the algorithms that use these data structures carry out much fewer operations than their dense equivalent. In algorithms that operate

on sparse data structures, contiguous matrix entries do not have contiguous memory addresses and the pattern of memory accesses is irregular. Linear algebra algorithms that operate on sparse data structures are called *sparse* linear algebra algorithms.

Sparse linear algebra has applications in mathematical optimisation. For example, the linear programming (LP) problems in the benchmark library Netlib [12] have constraint matrices with 96 % of the entries equal to zero, on average. Also, the constraint matrices of the mixed-integer programming (MIP) problems in the benchmark suite MIPLIB2010 [13] have on average 99.3 % of their entries equal to zero. The problems in these benchmark libraries arise in many fields, including power network management, logistics and circuit design.

The preconditioned counjugate gradient method (PCG), which is used in mathematical optimisation to solve both linear and nonlinear optimisation problems [14, ch. 9], demands fast solution of systems of linear equations. In particular, because the matrix is factorised, these systems are sparse, triangular, and linear (STLs). Similarly, the solution of LPs and MIPs is often obtained by carrying out the simplex method, which requires the solution of a sequence of STLs [15].

Depending on the sparse data structures used, STLs can be solved with high performance using concurrency. The conventional approach is used in most sparse linear algebra algorithms on GPUs and is implemented by the Nvidia CUSPARSE library [16, 17], which is used by the ViennaCL library [18], the Paralution library [19], the MAGMA library [20], the LAMA library [21], and the GHOST library [22]. This approach is to partition data so that as many numerical operations as possible can be carried out concurrently. If this is achieved during the execution of the GPU software, part of these numerical operations can be carried out while others are waiting for the transfer of data from the GPU memory and memory access latency is good. However, this approach does not guarantee that enough numerical operations can be carried out concurrently since this depends on the matrix of the STL. Moreover, because of sparse data structures, the pattern of accesses to the GPU memory is

irregular, which causes the memory access latency to be higher than in dense linear algebra. This approach is rarely able to achieve good memory access latency hiding and high performance in practice.

This thesis discusses a different approach for solving STLs and sequences of STLs on GPUs. By partitioning data automatically, this approach matches the structure of the matrix of the STL to the computer architecture, which reduces the access latency of the GPU memory. Compared to existing approaches, the one discussed in this thesis achieves a better spatial locality of memory accesses and as a result performance is improved in most test cases.

1.2 OBJECTIVES

The main objective of this thesis is to investigate how the performance of sparse linear algebra algorithms is affected by both the structure of data and architectural parameters of the GPU. Focusing on the solution of STLs, in which the structure of data is given by the matrix of the STL, this work provides a methodology for solving STLs and sequences of STLs that is faster than current methodologies for GPUs. The name of this approach is TASSL, which is short for *time slot sub-graph level*. TASSL can be considered a building block for more advanced sparse linear algebra algorithms that rely on fast solution of STLs, like the PCG method.

At a high level, the operation of TASSL can be described as in Figure 1.2. A preliminary analysis phase partitions the matrix of an input STL to match the architectural parameters of the GPU. The analysis phase needs to be executed only once per matrix. The matrix is partitioned by making sure that the data associated with a set of rows or columns can be stored in the GPU's local memories. These memories are fast but small memories which are present in the architecture of GPUs, many-core processors, and multi-core CPUs. For this reason, TASSL can be used on processors other than GPUs.

TASSL allows to reduce the execution time of the partitioning if the STL matrix changes. This is the case of sparse linear algebra algorithms such as

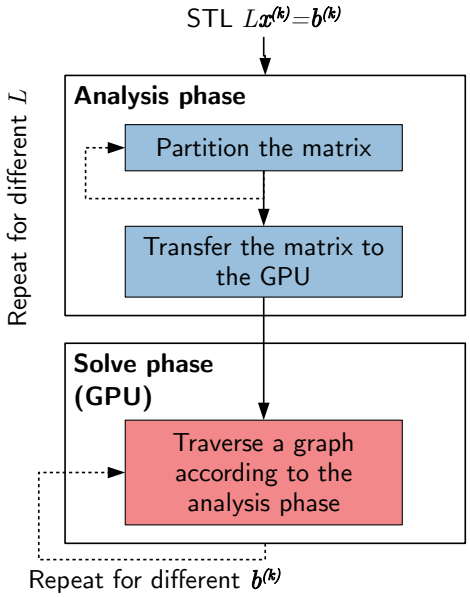


Figure 1.2: Overview of the operation of TASSL. A preliminary analysis phase is executed once per matrix. Then, the matrix is transferred to the GPU and one or more STLs are solved with different right-hand side vectors $b^{(k)}$ using the information obtained during the analysis phase. The analysis phase does not need to be repeated unless the matrix L changes. In the latter case, TASSL can reduce the execution time of the new analysis phase by updating the result of the previous one.

the left-looking LU decomposition, and optimisation algorithms such as the simplex method.

After the partitioning, the matrix is transferred to the GPU and one or more STLs are be solved with different right-hand side vectors. This is called the solve phase. The analysis phase does not need to be repeated unless the matrix changes from one STL to the other. The solve phase finds the solution to the STLs by executing the operations in the order obtained during the analysis phase.

The design of TASSL is based on performance models that put together the structure of the matrix of the STL and the architecture of the GPU. Using these performance models, this thesis also discusses the limitations of parallel implementations the simplex method.

1.3 OVERVIEW

This thesis is organised as follows. Chapter 2 introduces the concepts of GPU architectures and sparse linear algebra that are necessary to understand this work, and it provides a survey of the research in the field of concurrent sparse linear algebra algorithms.

Part I of this thesis discusses an approach to solving single STLs on GPUs and it is composed of Chapter 3, Chapter 4 and Chapter 5. Chapter 3 illustrates the shortcomings of the CUSPARSE approach and an algorithm to solve STLs that achieves better data locality. Chapter 4 discusses a technique for organising the computation so that the algorithm in Chapter 3 can achieve high performance. Chapter 4 includes two algorithms: one that partitions the input matrix depending on the architectural parameters of the GPU, and one that schedules numerical operations. Chapter 5 shows the application of the proposed approach to the PCG method.

Part II of this thesis discusses a generalisation of the approach discussed in the previous part to solving sequences of STLs on GPUs. This part is composed of Chapter 6 and Chapter 7. Chapter 6 discusses how to extend the techniques developed in Part I to a sequence of STLs. Chapter 6 includes an algorithm to update a partition from one matrix to the following in the sequence of STLs. Chapter 7 illustrates the application of the generalised technique to the simplex method.

Finally, Chapter 8 discusses the conclusions of the thesis, its key insights and potential futher research.

A list of the acronyms used in this thesis (page 197) and a glossary (page 199) are also provided.

1.4 STATEMENT OF ORIGINALITY

The main original contributions of this work are summarised below.

1. An algorithm to solve sparse triangular linear systems of equations (STLs) on GPUs. This algorithm uses local memory to reduce memory access latency, and, in most of practical test cases, is faster than the CUSPARSE algorithm, which is used by the majority of software libraries for scientific computing (Chapter 3) [23].

2. A heuristic algorithm that partitions a directed acyclic graph (DAG) across the local memories of a GPU automatically (Chapter 4) [23].

3. A generalisation of the approach described in Chapter 3 and Chapter 4 to solve sequences of STLs in which the matrix changes from one element of the sequence to the next (Chapter 6).

Another contribution is the following:

i. an analysis of the limitations of GPUs in accelerating the simplex method. This analysis shows that the performance of the GPU is limited by the bandwidth of the data transfer between the CPU and the GPU.

ii. A theoretical framework to describe the performance of GPU algorithms that solve STLs (Chapter 3).

2

BACKGROUND AND RELATED WORK

Sparse linear algebra algorithms can be considered a special case of irregular applications, because they are characterised by irregular data structures, control, and communication patterns [24]. For example, in sparse linear algebra software it is not uncommon to have an algorithm like that shown in Listing 2.1, with *for loops* whose first or last indices are stored in an array and cannot be determined at compile-time.

This chapter discusses theoretical concepts and current research work carried out in the field of the parallelisation of irregular applications and sparse linear algebra. First, Section 2.1 gives an overview of the architectures of hardware accelerators and of those of GPUs in particular. Then, Section 2.2 is a survey of research in irregular applications. Section 2.3 explains the basic mathematical concepts of sparse linear algebra and discusses the research work carried out in the field of parallelising sparse linear algebra algorithms.

```
for(int i=0; i<M; i++)
{
   Y[i]=0.0;
   for(int k=C[i]; k<C[i+1]; k++)
      Y[i]+=X[J[k]]*V[k];
}
```

Listing 2.1: An example irregular application. This piece of software computes the sparse matrix-vector multiplication if the matrix stored in the compressed sparse row (CSR) format [25, ch. 1], which uses three arrays: `C`, `J`, and `V`. The first *for loop* iterates on the rows of the matrix with index `i`. For each row, column indices and entry values are accessed with the index `k`, whose values are between `C[i]` and `C[i+1]`. Column indices are stored in the array `J` and matrix entry values are stored in the array `V`. The output is written to the array `Y`.

2.1 ARCHITECTURES OF HARDWARE ACCELERATORS

All modern hardware accelerators implement some form of parallelism. GPUs, Intel Xeon Phi, and also multi-core CPUs implement both *single-instruction-multiple-data* (SIMD) and *multiple-instruction-multiple-data* (MIMD) parallelism [26]. In SIMD parallelism, a single instruction is executed on many operands at the same time, while in MIMD parallelism different instructions can be executed on different operands at the same time.

For example, in a multi-core CPU different parts of the same software, called *threads*, can be executed at the same time on different CPU cores, each one operating on a different piece of data (MIMD parallelism). However, inside each CPU core a single instruction from a thread can be executed at the same time on up to eight operands [27], thanks to specialised hardware (SIMD parallelism). GPU architectures mainly implement SIMD parallelism, since each instruction is executed on up to thirty-two operands at the same time, but they also support a certain degree of MIMD parallelism.

This section aims to provide the reader with all the concepts of GPU architectures that are needed to understand the content of the thesis. Those concepts that are common to most hardware accelerators are discussed in Section 2.1.1, which illustrates the OpenCL model. Section 2.1.2 instead delves into the details of GPU architectures and Section 2.1.3 gives a short survey of the research in GPU architectures.

2.1.1 *The OpenCL architecture model*

The architecture of most accelerators, including GPUs, can be described with the OpenCL model [28], shown in Figure 2.1. The advantage of the OpenCL framework is in that a piece of software written using OpenCL can be executed on a GPU or on a multi-core CPU with few or no changes. However, OpenCL software usually requires some changes to be implemented on a field-programmable gate array (FPGA) before achieving good performance.

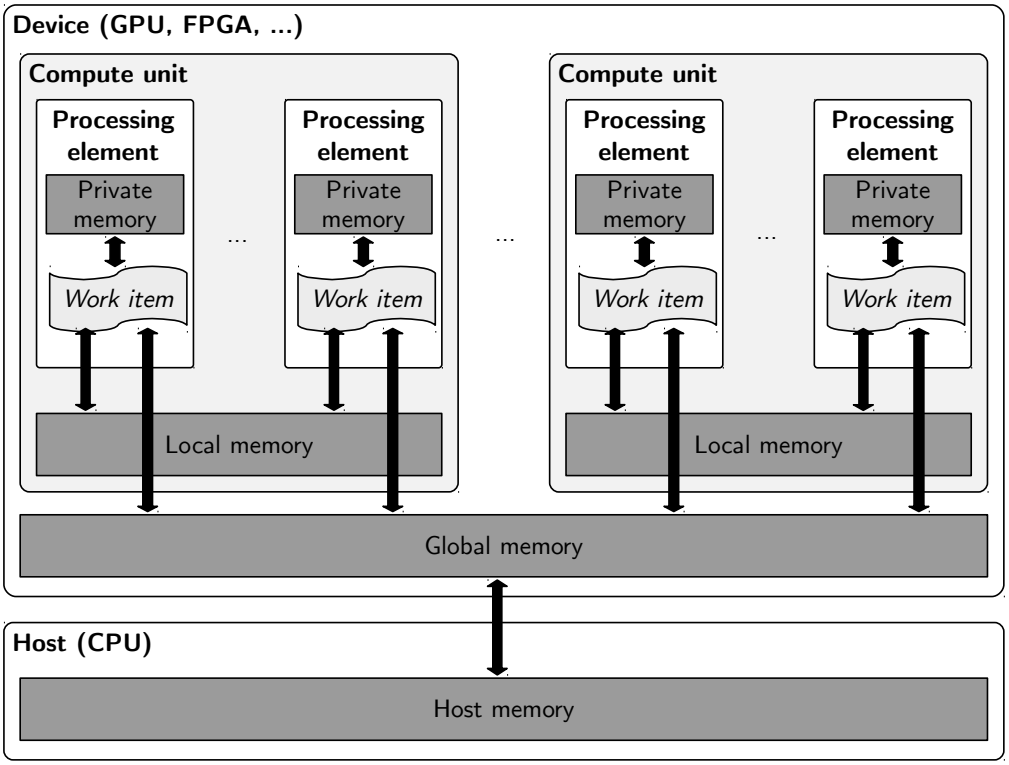


Figure 2.1: The OpenCL model of the architecture of a parallel accelerator. An OpenCL device is composed of a number of compute units (CUs) which access a global memory. Each CU contains a local memory and a number of processing elements (PEs), and each PE contains a private memory.

An OpenCL device is composed of a number of *processing elements* (PEs), which are the most basic computational blocks in the architecture. A PE executes an elementary piece of software, called a *work item*, and accesses a portion of memory called a *private memory*. If the private memory is small, its contents are stored into registers inside the PE. PEs are packed into *compute units* (CUs) and share a *local memory*. This local memory has a higher access latency compared to that of a PE's private memory but it is also larger.

The set of all work items being processed by a single compute unit is called a *work group*. A device can have many compute units, all sharing a *global memory* and each one processing a different work group. Work groups are usually executed out-of-order. Also, the execution of work items in the same work group can be synchronised, but that of work item in different work groups cannot be synchronised. Hence, the computations of work items in different

```
kernel void example(global int const *x, global int *y)
{
   private int global_id  = get_global_id(0);
   private int temp = 0;

   temp = x[global_id];
   if(temp%2==0)
      temp++;
   else
      temp--;

   y[global_id] = temp;
}
```

Listing 2.2: Source code of example OpenCL kernel. Each work item in the kernel accesses an element of the array x and stores it to private memory. Then, if the value retrieved is even, the value is incremented, else it is decremented. Finally, the data is written from the private memory to the array y, which is stored in global memory.

work groups have to be mutually independent. The set of all work groups processed on a device is called a *kernel*. A device can process more work groups than its compute units, because at the end of the execution of a work group the local memory is emptied and a new work group is loaded. For this reason, any data in common between different kernels has to be stored in the global memory, which is the largest memory in the hierarchy but also that with the highest access latency.

Table 2.1 illustrates how the model discussed above maps onto the actual hardware of three different types of parallel devices: GPUs, multi-core CPUs and FPGAs [29, 30].

2.1.2 *Architecture of a GPU in detail*

In a GPU, a number of work items, usually 32, are mapped to a single *SIMD thread*, also called a *warp*. A SIMD thread is a sequence of SIMD instructions and each SIMD instruction executes the same operation on 32 operands.

Table 2.1: Correspondence between the OpenCL abstract model [28] and actual accelerator architectures. In the table, *mCPU* stands for multi core CPUs. Hennessy and Patterson's nomenclature [31, ch. 4] is used for the GPU case.

Concept	GPU	mCPU	FPGA
Processing element (PE)	A SIMD lane. Each SIMD lane has up to 8 arithmetic-logic units (ALUs). Hence it can perform the same operation on more integer or single-precision floating point numbers at the same time.	Each core is composed of a single PE, which usually supports a limited vectorisation of operations.	Defined ad-hoc for a given kernel.
Private memory	A set of registers private to each SIMD lane. Its size is in the order of tens of bytes.	Banks of registers inside each core.	Configurable by the designer, usually based on flip-flops.
Compute unit (CU)	A multithreaded SIMD processor. It is composed of up to 32 lanes or processing elements.	A CPU core.	Ad-hoc parallel architecture for the given kernel.
Local memory	Fast SRAM memory whose size is in the order of kilobytes. An average GPU has 32 KB local memories. A local memory is available to each CU.	Cache memory. An average mCPU has 32 KB local memories.	Block RAM (BRAM). The size is configurable by the designer.
Global memory	DRAM memory accessible to all multithreaded SIMD processors in a GPU.	A region of the RAM memory on the computer's motherboard.	Configurable by the designer. Either off-chip RAM in the FPGA board or BRAM.

For example, consider the OpenCL source code in Listing 2.2. In the OpenCL model, each work item in the kernel accesses an element of the array `x` and stores it to private memory. Then, if the value retrieved is even, the value is incremented, else it is decremented. Finally, the data is written from the private memory to the array `y`, which is stored in global memory.

In practice, the first 32 work items are mapped to the first SIMD thread. This SIMD thread reads 32 integer values from global memory and writes it to registers in one SIMD instruction. Then, the modulus with respect to 2 is computed in parallel for each of the 32 values. Afterwards, the increment SIMD instruction is executed but only on even values, hence a *predicate* is set before it. This control mechanism works like a mask that prevents the increment instruction from being executed on some of the operands. Next, the predicate is inverted and the decrement instruction is executed on the other operands. On GPUs, the mechanism of predicates is used to implement *conditional branching*, such as the `if-else` and `case` statements and its effect is called SIMD thread divergence. Finally, the 32 values are copied at the same time from registers to global memory with a single SIMD instruction.

The GPU hardware that executes SIMD threads is the multithreaded SIMD processor, which corresponds to a CU in OpenCL and is illustrated in Figure 2.2.

A multithreaded SIMD processor is composed of a number of SIMD lanes that operate in parallel. Work items are mapped onto a single SIMD thread so that the operations of one work item are always executed by the same SIMD lane. Two SIMD lanes cannot execute different SIMD instructions simultaneously [31, ch. 4]. Modern GPUs have only 16 physical SIMD lanes so one SIMD lane is used for two operands. Each SIMD lane has vector arithmetic-logic units that can operate on more data at the same time if vector data types like `int16` or `float8` are used [32–34].

The SIMD threads in execution on the SIMD processor are listed in a SIMD thread scheduler. A SIMD thread can be in three states: *execution*, *ready* for execution, or *waiting* for data. Whenever a SIMD thread in execution carries out a memory operation, it enters the waiting state and its ID is stored in a *scoreboard*.

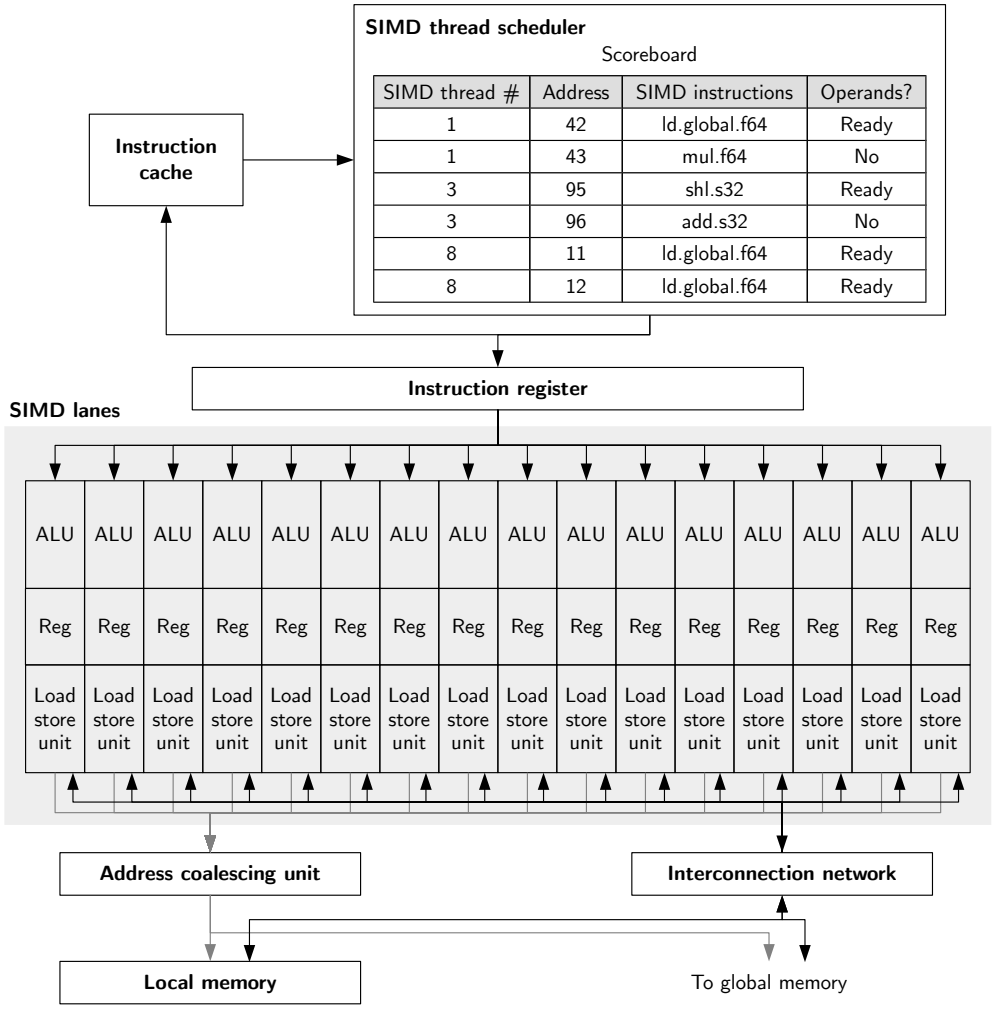


Figure 2.2: Simplified architecture of a multithreaded SIMD processor, adapted from [31, ch. 4]. A SIMD processor is composed of a number of SIMD lanes. Since work items are mapped onto SIMD threads, each work item is associated with a SIMD lane. SIMD threads selected among those ready for execution in a warp scheduler and dispatched to the SIMD. Each SIMD lane can use a portion of a register file and can access the memory though a *load-store unit*. Memory accesses are organised automatically by an *address coalescing unit* and then propagated either to the local memory or to the global memory.

The SIMD thread scheduler then selects one of the other SIMD threads in the scoreboard among those ready for execution and starts the context switching.

GPUs achieve the best performance when there are enough SIMD threads ready for execution to keep the SIMD lanes, in particular the arithmetic-logic

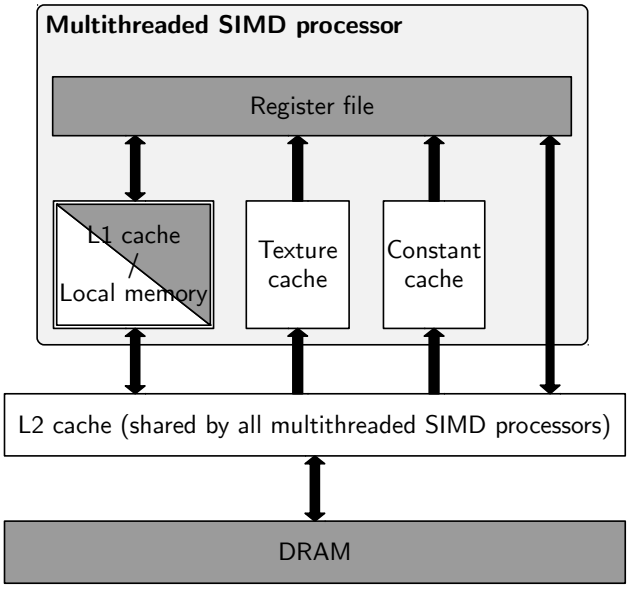


Figure 2.3: General scheme of the memory sub-system of a GPU. An off-chip DRAM contains all the global memory data of the OpenCL model. Multithreaded SIMD processors share a L2 cache memory but each one contains some specialised cache memories and L1 cache memory. In the Nvidia Fermi architecture [34], the same physical memory is used both as a L1 cache memory and as the local memory. Temporary results and private memory data of the OpenCL model is stored in a register file unless part of it is offloaded to the DRAM.

units, busy at all times. Since at the same time other SIMD threads can be waiting for memory operations, this effect is called *memory access latency hiding*. In practice, maximising memory access latency hiding is the aim of most GPU software optimisations. A specialised *address coalescing unit* helps by analysing all the memory instructions being executed on the multithreaded SIMD processor, reordering them partially, and automatically translating them to as few memory access requests as possible. Nonetheless, to exploit this specialised unit, a programmer has to take data locality into account while writing GPU software.

Figure 2.3 represents the memory sub-system of a GPU. This scheme was implemented first in Nvidia Fermi GPUs [34] but in its variants it has become standard.

An off-chip DRAM contains all the data belonging to global memory in the OpenCL abstraction. The data in this DRAM is accessed by an on-chip L2 cache memory which is shared by multithreaded SIMD processors. *Atomic operations* on global memory data, which are special operations that appear uninterruptible to the rest of the system, are managed by this L2 cache memory. For example, when an atomic increment

$$x \xleftarrow[\text{atomic}]{} x + 1$$

is executed by a SIMD thread, first the value of x in the right-hand side of the expression is retrieved either from the L2 cache memory or the DRAM. All other values of x which may be stored in other memories in the memory sub-system are immediately discarded. Then, the increment is executed by the multithreaded SIMD processor and a new value of x is written back in the L2 cache memory. To all the other SIMD threads that access x, the atomic operation appears uninterruptible and for these threads the updated value of x needs to be retrieved from the L2 cache memory.

Each multithreaded SIMD processor contains at least three different types of cache memory: an L1 cache memory, a texture cache memory and a constant cache memory.

In the Nvidia Fermi architecture [34], the L1 cache memory and the local memory reside on the same SRAM, and the programmer can chose how to partition the SRAM between the two. Instead, on AMD architectures [35] L1 cache memory and local memory reside on two separate memory devices. Choo *et al.* [36] observed that the hit rate in the L1 cache memory on GPUs tends to be lower compared to CPU cache memory because data tends not to be reused in most applications. For this reason, on Nvidia architectures it is usually preferable to assign a large part of the SRAM to the local memory, whose use can be higher if the programmer uses information on the application to write the GPU software. Also, in both Nvidia and AMD architectures, local memory is organised in memory banks and the best performance is achieved only if as many memory banks as possible are accessed at the same time. For example, let us assume an array of integer numbers `A` is stored in local memory and one

of the memory banks can store b integer numbers. Then, the best performance is achieved if a SIMD instruction accesses the element `A[0]`, `A[1]`, `A[2]` at the same time, and not `A[0]`, `A[b]`, `A[2*b]`. In the last case, two or more SIMD lanes access the same memory bank at the same time, which is called a *local memory bank conflict*. This causes a reduction of performance because all the accesses to the same memory bank are sequentialised.

The texture cache memory has a size of about 8 KB and is used extensively in graphics applications but less in computing applications [37]. In the latter case, it is usually called a *read-only cache* memory and it is used in applications where spatial locality can be enforced, like image rendering. The constant cache memory has a similar size to that of the texture cache memory but it can only be accessed by static indexing, as determined by the compiler, and only when data is declared to be constant.

Also, each multithreaded SIMD processor contains a register file. A GPU compiler tries to store all data belonging to private memory in the OpenCL model in these registers. However, if this is not possible, a part of the data is stored in the DRAM and cached in the L2 cache memory. This technique is called *register spilling* and causes a reduction in performance because it increases the number of accesses to the L2 cache memory.

2.1.3 *Related work on GPU architectures*

GPU simulators can be used to improve the performance of GPU software using knowledge of the architecture. One of the most popular GPU simulators is GPGPU-Sim [38], which was developed developed by Bakhoda *et al.* and can be used to simulate both Nvidia and AMD GPU architectures. Using this simulator, it was observed that, although having many SIMD threads increases the chances for good memory access latency hiding, having fewer SIMD threads than otherwise possible can reduce contention in the memory system and improve performance [38]. Another popular GPU simulator, Multi2Sim [39], is designed to study in depth the communication between CPU and GPU, and recently Candel *et al.* showed that it is possible to use this tool to carry out

a top-down analysis of the whole memory system [40, 41]. The conclusion of their analysis was that the performance of GPU software can also vary depending on the particular memory access coalescing mechanism implemented by the architecture. This was observed in particular in the case of a GPU kernel that transposes a matrix, for which merging memory requests from different SIMD threads required up to 75 % more memory accesses than combining requests from the same SIMD thread [40].

Theoretical models are an alternative to GPU simulators and can give insight on GPU software without executing any actual software. Hong and Kim's work provided a theoretical model for both performance [42, 43] and power consumption [43]. This model does not take into consideration the GPU cache memories in Figure 2.3, nonetheless it achieves a maximum relative error of 13 % with respect to the execution time measured on the GPU. The model is based on the idea that every application can be described in terms of *memory parallelism* and *computation parallelism*: the first is the number M_P of SIMD threads that access memory concurrently in the GPU software, and the second is the number C_P of SIMD threads that can carry out computations while another SIMD thread is waiting for memory access. If computation parallelism is greater than memory parallelism, like in sparse linear algebra applications, the application is said to be *memory bound* and the model [42] gives the following expression for the execution time, measured in clock cycles:

$$t_{\text{exec}} = c_{\text{mem}} \frac{n_{\text{threads}}}{M_P} + c_{\text{comp}} C_P, \tag{2.1}$$

where c_{mem} is the number of clock cycles needed to access any memory location, n_{threads} is the number of SIMD threads, and c_{comp} is the number of clock cycles needed to carry out a SIMD numerical instruction, for example a sum. From (2.1), one can note that, if the memory parallelism M_P is small, the only way to reduce the execution time is to obtain a small c_{mem} by optimising the GPU software for high data locality.

The literature of high-performance computing is rich in techniques for GPU software optimisation. Garland *et al.* categorised a number of these techniques by application domain [44]. For example, they discussed that in dense linear

algebra, the best performance is achieved if the matrix is sliced into blocks of data and a small part of each of these blocks is associated with a work item. However, Ryoo *et al.* observed that in practice the design space for GPU software is much more complex than this categorisation [45], and they showed a technique for pruning the optimisation space based on information captured from the GPU software at compile-time. This technique performs a partial design space exploration and compares the performance of the different designs. A similar approach was also used by Thoman *et al.* to characterise the architecture of a GPU by executing a number of micro-benchmarks [46]. The drawback of both approaches is in that they can be time consuming, while compiler optimisations for CPUs usually carry out the transformations quickly and automatically.

As an alternative to this approach, Jang *et al.* proposed a compiler optimisation that optimises the transformed code on a model of the GPU memory system and achieves a speedup of up to 13 compared to unoptimised code [47]. However, the problem with this other approach is that it cannot be applied to irregular code such as that in Listing 2.1, because loop bounds are stored in irregular data structures and their values are known only at runtime.

A third approach to improve the performance of GPU software is that to provide the GPU hardware with the ability of optimising the execution at runtime. For example, Kayıran *et al.* studied a SIMD thread scheduler that allocates the optimal number of SIMD threads to each multithreaded SIMD processors such that memory access contention is reduced [48]. Li *et al.* designed an optimised L1 cache memory with a locality-monitoring mechanism that gives an average of 30.3 % performance improvement [49]. These improvements can help to achieve good performance but they are effective only if the GPU software is written to make use of them. For example, if all work items in a work-group access memory locations that are distant to one another, with an irregular access pattern, no L1 cache memory can help reducing access latency and little can be done by an optimised SIMD thread scheduler. For this reason, much research is being carried out in the field of parallel irregular applications.

2.2 PARALLELISING IRREGULAR APPLICATIONS

Seminal work on parallel irregular application was carried out by Pingali *et al.* at the University of Texas, Austin [50–58]. Pingali provided a theoretical framework to categorise irregular applications by describing them as the repeated application of an *operator* to a set of graph nodes or edges called the *active elements* [50]. In this framework, the active elements are processed in a specific order by the operator, which uses data from the edges or nodes close to the active elements. Different types of data topology, active elements, and operators give rise to an elaborate taxonomy whose scope is beyond this thesis. In terms of Pingali's taxonomy, the scope of this thesis is that of *unstructured topologies*, which means general graphs, where the operator carries out *local computation* and the choice of active elements is *topology-driven*, which means that at each iteration the computation is carried out only on active elements chosen only based on the structure of the graph.

The performance of applications with these properties is mainly determined by graph topology. For example, consider an application that executes the breadth-first traversal of a graph and the graph has a topology like that shown in Figure 2.4. In this case, the set of active elements will contain only one vertex at the time because the data associated with one vertex can be processed only after that of its parent. Kulkarni *et al.* produced a software tool that, given an irregular application and a graph topology, computes how many work items can be used to execute the irregular application [51]. This gives the user an idea of how profitable it is to execute the application in parallel, but it cannot give an estimate of performance since that depends on the computer architecture. In fact, on GPUs the performance depends on how the memory system is used and having too many work items can lead to memory access contention.

For this reason, suites of irregular programs have been proposed separately for CPUs [52] and GPUs [59], which shows that different approaches to the parallelisation of irregular programs have to be devised for different architectures, although some concepts can be adapted from one architecture to the

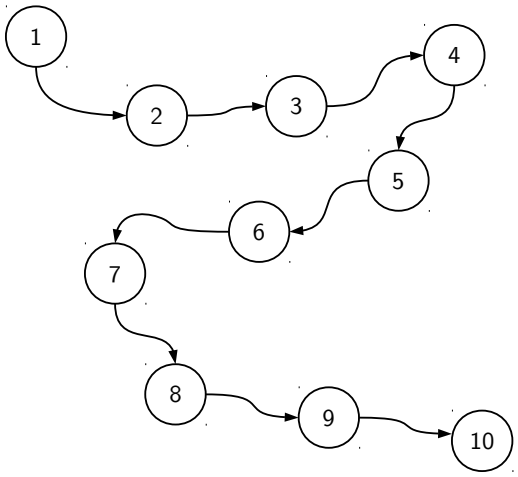


Figure 2.4: An example of graph topology that allows for no concurrency. In this directed acyclic graph (DAG), the data associated with one vertex can be processed only after that of its parent, which forces the processing in sequence of all vertices one by one.

other. CPU approaches are discussed in Section 2.2.1 and those for GPUs are discussed in Section 2.2.2.

2.2.1 *Irregular applications on CPUs*

In multi-core CPUs and computer clusters, the input data is initially stored in a *shared memory*, for example the computer's RAM, and it needs to be assigned to different CPU cores or cluster nodes. Although *message passing* [60, 31, ch. 5] could be used to transfer small blocks of data whenever needed without partitioning, in practice partitioning reduces communication and improves performance [61]. Therefore, on these systems an initial partitioning is carried out and then data are transferred from the shared memory to one CPU core or to one node.

The partitioner METIS [62], developed by Karypis and Kumar, uses a particularly fast heuristic which aims to balance the number of vertices in each partition but does not take into consideration the topology of the graph. As a consequence, vertices that are close in the graph can be in different partitions, which reduces data locality and performance [63]. This limitation was

overcome by Mellor-Crummey *et al.* using space-filling curves [63], and Han and Tseng with a runtime partitioner that uses multi-level clustering [64]. In the last case, performance increased by a minimum of 60 % and a maximum of 80 %. However, none of these approaches can be applied to directed acyclic graphs (DAG), which is the case of many sparse linear algebra algorithms.

Understanding the behaviour of cache memories on CPUs when irregular applications are executed is the key towards achieving good performance. Probabilistic models were produced for the performance of different types of cache memories [65, 66] without taking into account a complete hierarchy of memories. The case of a simple but complete memory system was discussed by Zhang *et al.* using a deterministic model [67], which, however, cannot be used for realistic memory systems without repeatedly sampling memory access requests during the execution.

Since CPUs do not suffer from branch divergence like GPUs, instead of modelling the behaviour of the memory system before executing the software, sophisticated runtime systems can be used to reorder memory accesses during the execution of the software itself. The PILAR project developed by Lain and Banerjee [68] was one of the first runtime systems to do this by finding regular patterns in the memory access requests. However, this approach is not effective for general graphs, for which alternative approaches were proposed. The *inspector-executor* approach [69] was shown to be effective for single-core CPUs but not applicable for other CPU-based architecture, where the memory accesses of all CPU cores or cluster nodes are not synchronised. For these architectures, the inspection of future memory accesses of each CPU core or cluster node does not give a good prediction of how the complete system will behave. To address this issue, Han and Tseng proposed a reordering algorithm called Z-SORT [64] which improves performance of 64 % with respect to a baseline implementation and complements their runtime partitioner.

A locality-aware runtime job scheduler was introduced by Yoo *et al.* [70]. Although its use is not specific to irregular applications, it was shown to achieve speedup with respect to other runtime job schedulers and an average energy consumption reduction of 47 %. This shows that taking data locality into con-

sideration is profitable on CPUs in the general case as well as that of irregular applications.

Synchronisation is also a cause of performance reduction in irregular applications [31, ch. 5]. The Cilk framework [71], which extends the C programming language to use multi-core CPUs, was used by Cong *et al.* to implement *work-stealing* in the software library XWS [72]. This approach is the opposite of partitioning since it leaves the assignment of work, and consequently the transfer of data, to the CPU cores themselves at runtime. The execution of the application with work-stealing is *asynchronous* and uses atomic operations. In contrast to the partitioning approach, work-stealing does not usually require an initial pre-processing phase, which can be time consuming, but work-stealing can lead to poor data locality because data is assigned dynamically as soon as a CPU core is available for computation. An intermediate solution is that implemented in KLA [73], a graph processing framework that first carries out a coarse-grain partitioning of the graph and then processes each partition using work-stealing. Partitions are grouped into *levels* and at the end of the execution of each level the operation in all the CPU cores is synchronised with a mechanism called a *barrier*. When the execution of the software in a CPU core encounters a barrier, the CPU core stays idle until the execution of all the other CPU cores reaches the same point. The balance between synchronicity and asynchronicity is determined with a model. A similar approach is applied by the EPIC framework [74] to schedule asymmetric sets of tasks. Although it achieves good performance on multi-core CPUs, the KLA approach is not suitable to all GPUs since every atomic operation requires discarding data in other caches, which can cause notable performance degradation.

On multi-core CPUs and computer clusters, the problems of achieving good data locality and low synchronisation overhead have been addressed in the development of the libraries and domain-specific languages (DSLs) listed in Table 2.2.

The Parallel Boost Graph Library (BGL) is designed such that the implementation of algorithms is as independent as possible from the details of data structures [75]. Similarly, the Standard Template Adaptive Parallel Library

Table 2.2: Software libraries and domain-specific languages (DSLs) for irregular applications on multi-core CPUs and computer clusters.

Category	Name	Year	Features
API	Parallel BGL [75]	2005	Implementation of algorithms separated from data structures.
	XWS [72]	2008	Work stealing.
	STAPL [76]	2010	Extends the Standard Template Library by adding abstractions for communication, synchronisation, distrubuted data structures.
	Pregel [77]	2010	Vertex-centric approach.
	Galois [50, 55]	2011	Runtime system based on Pingali's taxonomy of irregular applications [50].
	GraphLab [78]	2012	Data-structure-independent algorithms and asynchronous computation.
	Power-Graph [79]	2012	Optimised for *power-law* graphs.
	Ligra [80]	2013	Lightweight library, balances the workload depending on the density of vertices in the graph.
	KLA [73]	2014	Trade-off between synchronous and asynchronous computations.
	Grappa [81]	2014	Runtime system that improves data locality and balances the workload. Can be used together with GraphLab.
	HeLP [82]	2014	Based on Pregel. Set of primitives to build graph algorithms.
DSL	STAR-P [83]	2006	Parallel dialect of MATLAB for graph algorithms.
	Green-Marl [84]	2012	Can be used with OpenMP and other backends.
	Elixir [56]	2015	Contraints on synchronisation and scheduling can be supplied by the user.

(STAPL) extends the C++ Standard Template Library by adding abstractions for communication and synchronisation, distributed data structures, and data-structure independent algorithms that can be customised [76]. Although the independence of algorithms from data structures makes both libraries very usable and extensible, it also causes a reduction in performance compared to implementations that are optimised for specific data sructures. For example, the Pregel library only implements algorithms that are vertex-centric and are not independent on data structures [77]. Compared to the parallel BGL, this

approach reduces the amount of data that is stored locally and improves scalability. The framework HeLP is based on Pregel and gives a set of primitives that can be used as building blocks for graph algorithms [82]. The Galois API, based on Pingali's taxonomy of irregular applications [50], is more flexible than the HeLP API because it can be used to implement any graph algorithm. Also, Galois does not aim to separate the implementation of algorithms from the details of data structures like the parallel BGL or STAPL, and it provides a set of standard data structures. Because of this, the Galois user is encouraged to design the application focusing how the data is accessed and to let the framework select the most suitable optimisations.

Another approach, implemented in the software library GraphLab, is to have data-structure-independent algorithms but reduce the cost of communication and memory accesses with asynchronous computations [78]. Moreover, GraphLab can be used together with the Grappa framework, a runtime system that schedules computations to improve data locality and balance the workload with work-stealing [81]. A trade-off between asynchronous and synchronous computations is implemented by KLA [73].

Research on structure-exploiting algorithms was carried out by Gonzalez *et al.* who implemented Power-Graph, a software library optimised for graphs whose degree distribution follows a power law [79]. Using Pingali's nomenclature [50], Power-Graph implements graph operators that do local computations and balance the workload across all the CPU cores or cluster nodes. The lightweight software library Ligra also balances the workload depending on the density of vertices in the graph [80].

Domain-specific languages have been used to implement irregular applications on multi-core CPUs and computer clusters. Star-P, a parallel dialect of MATLAB [85], was designed to implement graph algorithms with application to sparse linear algebra [83]. The software architecture of Star-P is of the type master-slave, with one Star-P master server instance distributing the workload over a number of slave CPU cores or cluster nodes. This software architecture is not scalable because communication latency increases with the number

of slave cluster nodes available, which limits the size of graphs that can be processed.

The DSLs Green-Marl [84] and Elixir [56] implement more scalable approaches. Green-Marl is used to generate C++ software that uses OpenMP or Pregel [77] but can only do high-level code optimisations. Hence, for example, it cannot take data locality into account. A different approach is that of Elixir [56], a domain specific language companion to the Galois runtime system [50, 55] and designed to help programmers think in terms of accesses to data structures. Elixir is used to generate C++ programs which use the Galois runtime and are optimised following Pingali's taxonomy of irregular applications [50]. Also, constraints on synchronisation and scheduling can be provided to Elixir to guide code generation.

Summarising, on CPUs and computer clusters the best performance is achieved with implementations that mix synchronous and asynchronous computation [73, 81]. The absence of thread divergence, which is a problem on GPUs, allows for the decision on how to partition the data and schedule computation to be taken at run-time [50, 55, 64].

2.2.2 *Irregular applications on GPUs*

To implement high-performance irregular applications on GPUs it is necessary to avoid thread divergence and atomic operations. Consequently, in most GPU implementations the computations in SIMD threads are synchronised with specialised techniques [53].

For example, some graph algorithms require storing a work-list to associate graph vertices or graph edges to GPU threads and to distribute the workload [54, 86]. Although atomic operations can be used to manage this work-list [53], Merrill *et al.* used the barrier mechanism in their implementation of the breadth-first traversal of a graph [86]. On GPUs, a barrier is implemented using either a global or a local memory location that acts as a counter. Whenever a work item encounters a barrier, it increments the counter and it remains idle until the execution of all the other work items in the same work group

reaches the same point. This mechanism is handled by specialised hardware instructions [31, ch. 5], which Nasre *et al.* showed to be faster than atomic operations on GPUs [54]. Also, in their implementation of the breadth-first traversal of a graph, Merrill *et al.* observed that non-coalesced memory accesses can cause a reduction of performance that is worse than the overhead of managing the work-list [86]. Nonetheless, managing the work-list causes a reduction of performance that becomes worse the more vertices are in the work-list. Hence, this approach does not achieve good performance on graphs where significant SIMD parallelism can be utilised.

The use of atomics on GPUs was studied for an algorithm that identifies disjoint parts of a graph by Sutton *et al.* who observed that in practice atomic operations can improve performance with respect to barriers for graphs with a specific structure [58]. Kaleem *et al.* showed that, starting from the Nvidia Tesla models, the overhead of atomic operations on Nvidia GPUs is small compared to that in AMD GPUs but ultimately the choice of synchronisation mechanism should depend on the structure of the graph [57].

In practice the performance of irregular applications on GPUs is very much dependent on the structure of the graph, which determines how the data is transferred from the memory and whether memory accesses can be coalesced [53, 57]. Reorganising memory accesses to improve data locality was shown to benefit stencil computations [87]. Unkule *et al.* discussed how to improve data locality automatically by adding more workload to work items without incurring in register spilling [88], but their work did not examine the use of local memory. Work by Lee and Wu on the scheduling of work items in multithreaded SIMD processors showed that the execution time can be reduced of at most 20 % if the GPU is equipped with a specially designed hardware scheduler [89], while Hbeika and Kulkarni showed that by partitioning the data on the CPU to take data locality into account, an irregular application can be up to 50 times faster than one that does not take data locality into account [90].

The issues of synchronisation overhead and data locality in irregular applications have been addressed by three GPU software libraries: Cusha, Medusa, and Gunrock.

Cusha is a GPU software library developed by Khorasani *et al.* to implement graph applications based on a graph data format called *concatenated windows* (CW) [91]. To use this format, the graph is first partitioned in *shards*, also called *levels* in other application domains [16], which are grouped together depending on the structure of the graph to achieve high data locality. However, when shards are imbalanced, this approach is slower than specialised implementations [86].

The API Medusa is used to implement graph algorithms on one or more GPUs [92]. Applying the same idea of vertex-centric processing of Pregel [77], Medusa hides the implementation details from the user and leaves the task of balancing the workload across work items and work groups to its runtime system. However, this API does not take into account the issue of data locality in irregular applications.

Gunrock is a GPU software library that can be used both for vertex-centric and edge-centric graph algorithms [93]. The operation of Gunrock is divided into three steps: *advance*, which adds edges of vertices to the set of those that need to be processed, *filter*, which removes duplicates or other edges of vertices from this set, and *compute*, which carries out the processing of the elements in the set. This approach is seemingly faster than both Cusha and Medusa on a number of graphs [93]. However, like Medusa, this approach does not take into account the issue of data locality.

Summarising, GPU implementations of irregular applications are affected by the issue of thread divergence and the particular architecture of GPU memory systems. While on CPUs and computer clusters the best performance is achieved with implementations that mix synchronous and asynchronous computation [73, 81], on GPUs the computation is synchronous to avoid thread divergence [54] and specialised approaches have been studied to improve memory access coalescing. Although workload balancing was mostly used to indirectly achieve this aim [86, 92, 93], only Cusha addresses the issue of data

locality directly but its implementation has shortcomings with respect to custom implementations.

2.3 SPARSE LINEAR ALGEBRA

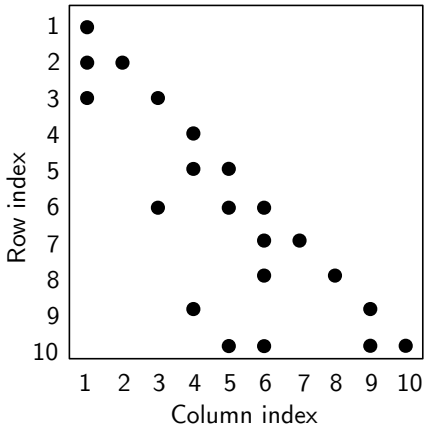
This thesis discusses sparse linear algebra applications as a particular case of irregular applications and addresses specifically the issue of data locality on GPU implementations.

This section illustrates first, in Section 2.3.1, those concepts used in the field of sparse linear algebra that are necessary to understand the content of the thesis. In Section 2.3.2, the literature of sparse linear algebra algorithms on GPUs is discussed in general, while Section 2.3.3 focuses on the solution of STLs.

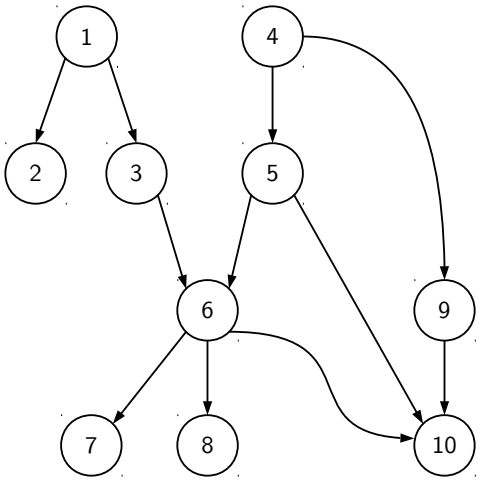
2.3.1 *Fundamental concepts in sparse linear algebra*

Given a square lower triangular matrix $L \in \mathbb{R}^{n \times n}$ with n rows and n columns, the graph G associated with it is an ordered pair of sets $G := (\mathcal{V}, \mathcal{E})$. Figure 2.5 shows an example sparse lower triangular matrix and the graph associated with it. The set of vertices $\mathcal{V} := \{1, 2, \ldots, n\}$ has as many elements as the number of rows and columns in the matrix L. A non-zero entry l_{ij} at row i and column $j \neq i$ in the matrix L is represented with a directed edge $e_{ij} := (i, j)$ from vertex j to vertex i in the graph G. The set $\mathcal{E}$ contains all the edges in the graph.

It can be shown that the graph G is an *acyclic* graph, which means there is no sequence of edges that starts from a vertex and ends in the same vertex without interruption [25, ch. 1, 94, ch. 1]. An interrupted sequence of edges from a vertex to another is also called a *path*. If a graph is composed of a number of parts that are not connected to each other, those parts are called *disjoint components* or weakly connected components [94, ch. 1].



(a) *Sparsity pattern of matrix* $L \in \mathbb{R}^{10\times 10}$.



(b) *Graph associated with the matrix L.*

Figure 2.5: An example sparse lower triangular matrix with ten rows and ten columns and the graph associated with it. In the representation of the sparsity pattern, the non-zero entries are shown with dots.

In the context of linear systems of equations, a path can be interpreted as dependency between two variables. Let us consider the sparse lower triangular linear system

$$L\boldsymbol{x} = \boldsymbol{b},$$

where $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors in $\mathbb{R}^n$. Let G be the graph associated with the matrix L. The system of equations can be solved by forward subsitution [95, ch. 3].

For example, in case L has a sparsity pattern as in Figure 2.5a, the following equation for x_6 can be written:

$$l_{66}x_6 = b_6 - l_{63}x_3 - l_{65}x_5,$$

which shows that the variable x_6 depends on the variables x_3 and x_5. On the graph in Figure 2.5b, one can see that there are in fact two edges incident to vertex 6, which start from vertex 3 and vertex 5 respectively. Similarly, equations can be written for x_3 and x_5 and it can be observed that they depend on x_1 and x_4. Therefore, x_6 also depends on these variables. In the graph, vertex 6 can also be reached with paths that go from vertices 1, 3, 4, and 5.

Because of the correspondence between sparse matrices and graphs, one can solve a sparse triangular system of linear equations (STL) by carrying out a traversal of the associated graph. Although the dependencies of the variables in the linear system depend on the sparsity pattern of the matrix L, the type of computation that can be carried out depends on the architecture of the computing system. For example, single-core CPUs perform a single numerical operation on a single datum, so a single-core CPU implementation can only traverse one graph edge at the time. Architectures that implement MIMD and SIMD parallelism [26], such as multi-core CPUs, computer clusters, and GPUs can traverse more than one edge at a time and solve the STL faster.

2.3.2 *Data structures for sparse linear algebra algorithms*

Sparse matrix-vector multiplication (SPMVM) has been used as a reference application for the study of data structures and techniques for sparse linear algebra in the past thirty years. In particular, GPU implementations were discussed by Filippone *et al.* who listed about seventy publications on the topic in the years from 2008 to 2017 [96]. Most of these publications are based on new data structures or optimisations of existing data structures that improve data locality or reduce thread divergence on GPUs. The most important data structures in sparse linear algebra are: the *coordinate* format (COO), the *compressed*

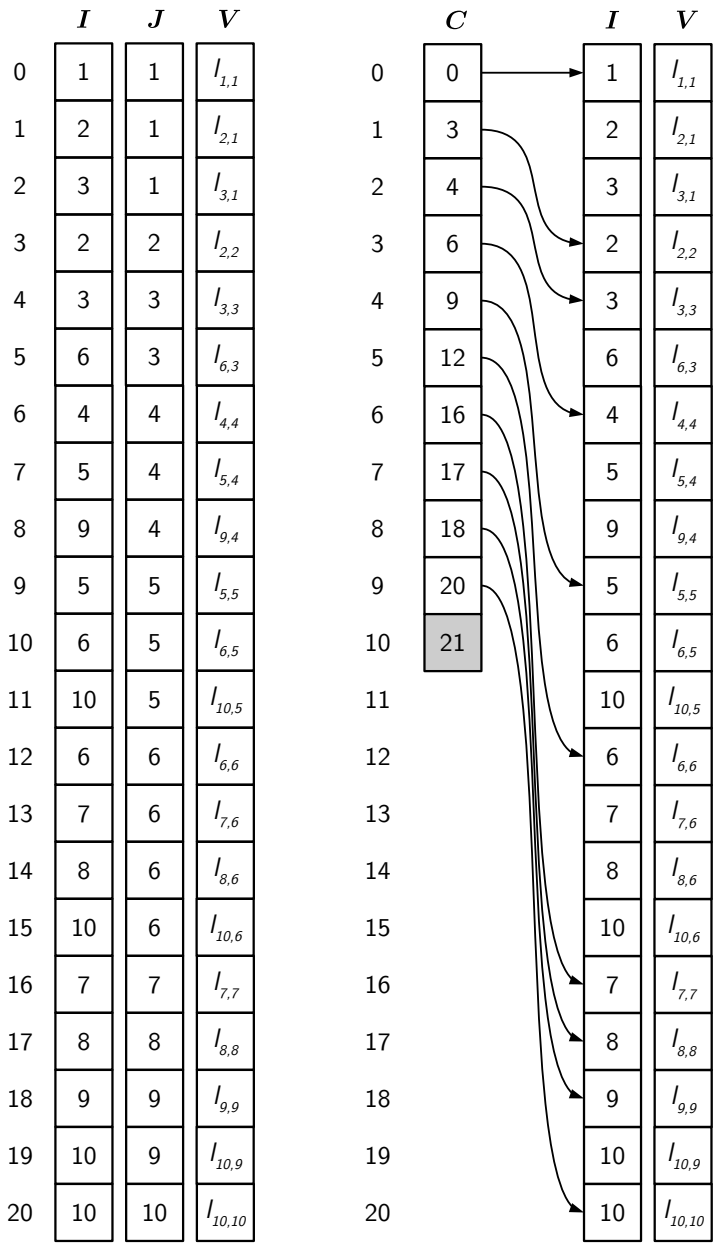
sparse rows or columns formats (CSR or CSC), and the ELLPACK format [25, ch. 1, 96, 97].

COO consists of simply listing all the entries in the matrix, storing their row indices, their column indices, and their value. Instead, CSC and CSR use a compressed form of either the array of column indices or row indices as shown in Figure 2.6. To obtain this data structure, the entries in the matrix are sorted by column or row index first, then the array `C` is used to store the offset for the entries in each column. With this technique, the values in column j are stored between the offsets `C[j-1]` and `C[j]`. With respect to the COO format, CSC and CSR require less storage space. However, if CSC is used it is difficult to find all the entries in a specific row, and vice-versa for CSR.

As alternative to both approaches, the ELLPACK format uses two-dimentional arrays to store the entries in the matrix [97]. Figure 2.7 illustrates that each row in the two-dimensional arrays is padded with zeros to reach the maximum number of entries in a row n_{row} such that the values of the entries in row i are stored between `Sa[(i-1)*n_row]` and `Sa[i*n_row]`. Since the length of each row in the data structure is fixed, ELLPACK has been used for vector processors and GPUs [96] but the overhead of zero-padding and the large memory footprint usually make this data structure less appealing than CSC and CSR for practical applications.

To overcome the limitations of each of these data structures, combinations of their core ideas have been proposed and implemented in other data structures [96]. Hugues and Petiton [98] and Langr and Tvrdik [99] have suggested a set of evaluation criteria for these other data structures which are based on the performance of SPMVM.

The work of Baskaran and Bordawekar [100] evaluates the performance of SPMVM with CSR but applies some minimal zero-padding to align memory addresses to cache lines, uses local memories to store parts of the multiplicand vector, and achieves a maximum improvement of 15 % in performance over Nvidia's library function [16]. However, because of the zero-padding this implementation transfers zeros that are useless for computation and cause avoidable contention in the memory sub-system for large matrices. Specialised



(a) *COO* (b) *CSC*

Figure 2.6: Comparison of the coordinate format (COO) and compressed sparse column format (CSC) for the matrix in Figure 2.5a. The COO format lists each entry in the matrix, storing its row index in the array `I`, its column index in the array `J`, and its value in the array `V`. Instead, in the CSC format the entries are sorted by column index first, then the array `C` is used to store the offset for the entries in each column. With this technique, the values in column j are stored between the offsets `C[j-1]` and `C[j]`.

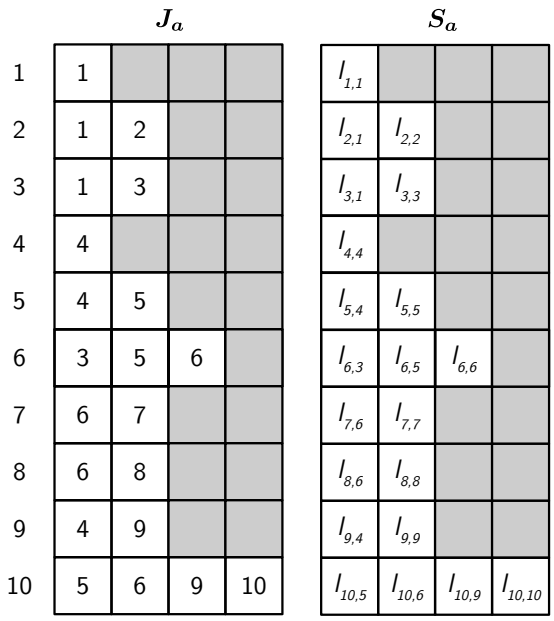


Figure 2.7: The sparse ELLPACK format [97] for the matrix in Figure 2.5a. The two-dimentional array `Ja` stores the column indices of the entries in each row and the two-dimentional array `Sa` stores their values. Each row in the array is padded with zeros, here represented with dark boxes, to the maximum number of entries in a row in the matrix.

memory sub-systems for SPMVM that address this specific issue have been studied for FPGA architectures [101, 102] but these are beyond the scope of this thesis. Rafique *et al.* used a performance model for their matrix-power algorithm, which applies SPMVM repeatedly, on both GPUs and FPGAs [103]. The performance model helped determine the value of an implementation parameter minimising communication. However, the GPU implementation of this algorithm did not use local memory, and it resulted in register spilling for large matrices, which greatly limited performance.

Reguly and Giles evaluated the performance of SPMVM using CSR on the GPU when the L1 cache memory is enabled and observed that the performance gap with ELLPACK can be filled if implementation parameters such as the number of work items in a work group are chosen using knowledge of the application [104]. Moreover, Reguly and Giles noticed that *cache misses* in the L1 cache memory cause a large reduction in performance, so the partition

of memory into L1 cache memory and local memory is key to performance improvement [104]. The issue of memory partitioning was also investigated by Liu and Vinter, who devised a data structure based on CSR, called CSR5, the aim of which is to distribute the non-zero entries in the matrix uniformly across compute units [105]. With CSR5, the non-zero entries in the matrix are organised in two-dimensional arrays of the same size, called *tiles*, which are stored in data structures similar to CSR. This approach achieves a maximum speedup of up to 141 compared to the standard CSR format and has the advantage of having low sensitivity to the structure of the graph associated with the sparse matrix. However, this approach also requires a preprocessing phase for the partitioning and a tuning of the size of the tiles, which are time consuming tasks and vary from sparse matrix to sparse matrix. Also, two matrices with similar sparsity patterns can have two very different CSR5 representations, so the preprocessing phase has to be repeated every time the matrix changes. The additional problem of scheduling tiles of CSR5 to compute units has led to the derivation of other specialised data structures based on CSR5 [106], which also require providing an optimal set of parameters.

To reduce the amount of zero-padding in ELLPACK, Monakov *et al.* devised a data structure called SELL-C [107], in which zero-padding is done over groups of C rows and not over all the rows. Kreutzer *et al.* applied an extra parameter σ which specifies how to sort the rows in the matrix so to have rows with almost the same number of non-zero entries grouped together [108]. This optimised data structure, called SELL-C-σ, is at the base of a library of building blocks for sparse linear algebra algorithms named GHOST [22], which can be used in multi-core CPUs, computer clusters, GPUs, and Intel Xeon Phi accelerators.

A different technique to reduce the memory footprint of sparse data structures was investigated by Tang *et al.* who applied a compression algorithm to the row indices in COO [109]. To use the data structure, two phases of operation are necessary: compression and decompression. The first can be carried out on the multi-core CPU host, while the second can be executed on the GPU.

Table 2.3: Software libraries for sparse linear algebra on GPUs. This list only include software libraries that provide GPU kernels and not high-level libraries that provide functionalities at a higher level of abstraction.

Name	Year	Features
CUSPARSE [16, 17]	2007	Proprietary library from Nvidia. BLAS and LAPACK equivalent for GPUs.
MAGMA [20]	2009	Does not implement LAPACK routines.
Matrix-template [110]	2013	Never released to the public.
SPRAL [111]	2014	Does not implement all LAPACK routines.
CLSPARSE [112]	2016	Does not implement all LAPACK routines.
ViennaCL [18]	2016	Does not implement all LAPACK routines.
Paralution [19]	2016	Does not implement all LAPACK routines.
GHOST [22]	2016	Developed around the SELL-C-σ sparse data structure [108]. Does not implement all LAPACK routines.

In both cases, using the data structure requires incurring an overhead that is difficult to amortise.

On GPUs, the problems of achieving good data locality, low synchronisation overhead, and little thread divergence have been addressed in the development of the libraries listed in Table 2.3.

Nvidia started distributing its sparse linear algebra library for GPUs, the CUSPARSE library [17], in 2007. CUSPARSE implements operations between sparse matrices and vectors and dense matrices and vectors using all the main data structures: COO, CSR, CSC, and ELLPACK. Moreover, it also provides specialised data structures such as the *block compressed sparse row* format (BSR), which uses CSR to store dense matrix blocks instead of single matrix entries. Since its functionalities are meant to be used as building blocks for other algorithms, one can consider CUSPARSE a GPU version of the numerical libraries for single-core CPUs BLAS [8] and LAPACK [9]. Because of its long development history, CUSPARSE is considered the reference standard of sparse linear algebra libraries on GPUs and is used inside other numerical linear algebra libraries such as the LAMA library [21].

The Matrix-template library [110] was developed by Gottschling with the aim of providing a more natural notation than that of CUSPARSE and the

automatic choice of the underlying data structure given the sparsity pattern of the matrix. Unfortunately, this library was never released to the public.

Most sparse linear algebra software for GPUs focuses on iterative algorithms for the solution of linear systems [14] and, unlike CUSPARSE, does not implement all BLAS and LAPACK routines. CLSPARSE [112] is the effort of AMD, Nvidia's principal competitor in the GPU market, towards a sparse linear algebra library. Initially developed internally at AMD and then transformed into an open source project, CLSPARSE only uses COO and CSR, and only implements the sparse matrix-vector multiplication, the conjugate gradient algorithm [14, ch. 6], and the biconjugate gradient stabilized algorithm [14, ch. 7]. Similarly, the ViennaCL library [18], the Paralution library [19], the MAGMA library [20], and the GHOST library [22] do not implement basic algorithms apart from SPMVM and focus on more complex iterative algorithms for solving linear systems. Of these, the GHOST library [22] is developed around the SELL-C-σ data structure and can be used with GPUs, Intel Xeon Phi accelerators, multi-core CPUs, and computer clusters. Instead, the SPRAL library [111], developed at the Rutherford-Appleton Laboratory, implements algorithms such as the solution of a sparse system of equations in the sense of the least squares [113] and internally uses customised data structures [114].

Summarising, the performance of sparse linear algebra algorithms is given by a combination of the structure of the graph associated with the sparse matrix and the data structure used. On GPUs, specialised data structures aim to reduce contention in the memory sub-system and either require the sorting rows and columns of the matrix or padding row or column entries with zeroes.

2.3.3 *Solving sparse triangular systems of equations*

The same data structures that are studied on SPMVM are also used in other sparse linear algebra algorithms [16, 18, 112], among which there is the solution of STLs. However, such is the importance of SPMVM in sparse linear algebra that in 1993 Peyton *et al.* suggested to solve STLs by using SPMVM on the matrix inverse [115] since a large amount of concurrency can be obtained with

SPMVM. In a period when the memory bandwidth of modern GPUs was not available, the inverse of the triangular matrix was represented as a product of matrices with as few non-zero entries as possible [116] and STLs were solved as a sequence of SPMVMs. The issue of data locality in sparse linear algebra applications was well known and addressed with probabilistic models [117] and specialised techniques for improving data layout at compile-time [118]. On single-core CPUs, the use of partitioning at run-time was examined by Strout *et al.* [119], and the issue of tuning the parameters of the algorithm when solving STLs was examined by Vuduc *et al.* [120].

The basic algorithm to solve STLs on single-core CPUs [25, ch. 1], first performs a depth-first search on the graph. This phase examines all the dependencies between variables by traversing the edges in the graph. The search phase returns an ordering of the vertices. For example, for the matrix in Figure 2.5 (page 45), the search phase could give the sequence $\{1, 2, 3, 4, 9, 5, 6, 7, 8, 10\}$. This sequence of vertices is in topological order, because no vertex appears in the sequence before its parents. After the search phase, each element of the sequence is considered a row index, and all the elements in the row are multiplied and summed. The complexity analysis for this algorithm depends on the sparsity pattern of the matrix [25, ch. 1, 120]. Also, the performance analysis on single-core CPUs showed that if the non-zero entries are close to the diagonal of the matrix, cache hits increase and computational latency is reduced [120].

The problem of implementing a parallel version of the solution of STLs has been examined in the context of MIMD architectures, such as multi-core CPUs and computer clusters, and GPUs. In general, it was observed that synchronisation has a greater impact on performance in STLs compared to SPMVM, since the latter can be implemented without any synchronisation [16, 121, 122].

The algorithm implemented by Totoni *et al.* [123], specific for multi-core CPUs, consists of analysing the dependencies and streaming rows to CPU cores whenever a dependency is satisfied. This approach achieves close-to-linear speedup with the number of CPU cores used, so it is very scalable, but it requires the synchronisation of threads in two or more cores. Although this technique achieves high performance on multi-core CPUs, it cannot be applied

to GPUs because this would require the execution of a GPU kernel at each step of the algorithm.

Two other algorithms devised by Mayer [124], also specific for multi-core CPUs, illustrate more limitations imposed by the architecture of GPUs. Both algorithms first permute the matrix into a block-diagonal form and then assign each block to a CPU core. The data structure used to store the partitioned matrix is a variation of CSR called *parallel solve*, and was devised specifically for this application. With this approach, the computation carried out in a CPU core is independent from that in another CPU core. If these algorithms were implemented on a GPU, the number of GPU kernels to execute would be proportional to the number of blocks. For example, if rows and columns were divided into s_M parts, $2s_M - 1$ GPU kernels would be executed. The results on multi-core CPUs show that by partitioning the matrix into blocks one can make better use of cache memories. Nonetheless, this approach does not make use of structure in the graph associated with the triangular matrix, since the partitioning does not take into account any opportunity for SIMD parallelism, so it is of little applicability to GPUs.

On GPUs, it was observed that the synchronisation of SIMD threads by stopping their execution in all of the multithreaded SIMD processors is extremely time-consuming [16, 122, 125], and that sparse linear algebra algorithms on GPUs tend to be bandwidth-bound, so the use of local memory to save bandwidth is necessary [125]. Also, while the design aim of GPU architectures is to hide memory latency with parallelism, not all applications are concurrent enough to achieve this aim efficiently [89].

The *level-set* algorithm, which is implemented in the Nvidia CUSPARSE library [16], uses CSR and is carried out in two phases: a preliminary analysis phase and a solve phase. The analysis phase associates vertices in the associated graph with *levels*, such that vertices in the same levels are guaranteed to depend only on vertices in previous levels. The solve phase traverses the graph level by level, either executing one GPU kernel per level or grouping levels with few vertices together into one single GPU kernel execution. This approach, which aims to achieve good memory latency hiding, is more generally applied

in GPU implementations of irregular applications including Cusha, Medusa, and Gunrock [91–93]. The CUSPARSE algorithm is also used by the ViennaCL library [18], the Paralution library [19], the MAGMA library [20], the LAMA library [21], and the GHOST library [22].

In the level-set algorithm, the number of levels and vertices per level is uniquely determined by the sparsity pattern of the matrix. For example, if the graph is a chain of vertices, each connected to the next with an edge as in Figure 2.4, there will be only one vertex per level. To overcome this limitation, a technique for transforming the matrix was introduced [122]. This technique increases the number of vertices per level and reduces the number of levels needed. However, it is not guaranteed that the transformation can always increase the amount of SIMD parallelism that can applied when the solve phase is executed. Li and Saad implemented both the conventional level-set technique and level-reducing matrix permutation [25, ch. 7], but obtained much worse performance than with a single-core CPU because of the large overhead of executing a GPU kernel per each level.

Hogg *et al.* implemented both the $LDL^\intercal$ factorisation algorithm [95, ch. 4] and a triangular solve algorithm on GPUs [114]. The latter uses the data structures of the factorisation to partition the data across multithreaded SIMD processors. These data structures are a set of small dense matrix blocks, indexed with a system that is similar to CSR. Also, the triangular solve was compared to a specialised version of the same algorithm that consists of two phases: a preprocessing phase and an SPMVM phase [127]. The specialised algorithm was shown to be faster if the preprocessing phase could be amortised across about 20 or 30 solves with different right-hand sides.

Chen *et al.* implemented a sparse triangular solve algorithm that uses a hybrid data structure, with some features of ELLPACK and some features of CSR [128]. This hybrid data sructure stores the denser parts of the matrix using ELLPACK and the sparser parts using CSR. The implemented sparse triangular solve algorithm is a customised version of the level-set algorithm for the hybrid data structure. By using ELLPACK in part, this data structure

Table 2.4: Algorithms for the solution of STLs on different architectures.

Architecture	Author	Year	Features
Single-core CPUs	Strout *et al.* [119]	2001	Run-time partition of the STL matrix to improve cache use.
	Davis [25]	2006	Orders the vertices before solving the STL. Used by MATLAB.
Computer clusters	Peyton *et al.* [115]	1993	SPMVM of the matrix inerse.
	Wolf *et al.* [121]	2011	Improves performance of the level-set algorithm by reducing the synchronisation overhead .
Multi-core CPUs	Mayer [124]	2009	Algorithms that divide the STL matrix into blocks to improve cache memory ususe.
	Totoni *et al.* [123]	2014	Streaming matrix rows to CPU cores when needed. Not suitable to be implemented on GPUs.
GPUs	Naumov [16]	2011	Nvidia CUSPARSE implementation of the level-set algorithm. Used by most scientific computing libraries.
	Suchoski *et al.* [122]	2012	Transforms the STL matrix, trying to improve the performance of the level-set algorithm.
	Hogg *et al.* [114]	2016	Specialised algorithm for matrices obtained with the $LDL^\intercal$ factorisation algorithm, based on SPMVM.
	Chen *et al.* [128]	2016	Specialised algorithm based on the level-set algorithm, specific for matrices with dense blocks.

achieves more regular memory acesses with respect to standard CSR, but it is not applicable in general to matrices with any structure.

Table 2.4 summarises the state of the art in solving STLs. The level-set algorithm is the most used algorithm to solve STLs, and a GPU implementation of it is provided by the CUSPARSE library. Work has been carried out to improve the performance of this algorithm on both GPUs and computer clusters, but the improvement that can be obtained is also dependent on the sparsity pattern of the STL matrix. Specialised algorithms, different from the level-set algorithm, exist for matrices with very specific sparsity patterns, but they are

not applicable to the general case. Other algorithms aim to improve cache use by partitioning the STL matrix, but this approach has not been explored on GPU architectures.

2.4 CONCLUSIONS FROM THIS CHAPTER

Irregular applications are characterised by irregular memory access patterns, which can cause a large number of *cache misses* in the memory sub-system of single-core CPUs, multi-core CPUs, and GPUs. To achieve good performance, different techniques have been developed for different computer architectures. On GPUs, because of thread divergence and the structure of the memory sub-system, it is not possible to use adaptive and computation-asynchronous techniques like those developed for multi-core CPUs and computer clusters. Instead, GPU implementations of irregular applications usually have synchronous computations and aim to distribute the computation evenly across work items. To reduce the execution time of these GPU implementations, the overhead of synchronisation needs to be minimised and memory accesses need to be coalesced.

Sparse linear algebra algorithms are a type of irregular application because they use specialised data structures such as COO, CSC, and CSR. Other sparse data structures that aim to make the pattern of accesses more regular, such as ELLPACK, require zero-padding and consequently the transfer of a large amount of data that are not useful for computations. Sparse linear algebra algorithms are designed around these data structures. In practice, the performance of sparse linear algebra algorithms is given by a combination of the structure of the graph associated with the sparse matrix and the data structure used.

On GPUs, techniques to minimise the overhead of synchronisation and thread divergence have been studied both for sparse linear algebra algorithms and in general for irregular applications. Techniques to improve data locality have also been studied, but the relationship between data locality and the structure of the graph has not been fully investigated. Also, it has been observed that lo-

cal memory can be used to reduce contention in the GPU memory sub-system, but the the relationship between this and the structure of the graph has not been examined yet in the literature. This thesis addresses specifically these issues.

Part I

SOLVING A SINGLE SYSTEM OF LINEAR EQUATIONS

This part the thesis discusses an approach to solving single STLs on GPUs and it is composed of Chapter 3, Chapter 4 and Chapter 5. Similarly to that of CUSPARSE [16], the approach described in this thesis is composed of two phases: a preliminary *analysis phase* and a *solve phase*. The analysis phase preprocesses and partitions input data on a multi-core CPU. The solve phase uses the information obtained in the previous phase to solve the STL on the GPU. Since the aim of the analysis phase is to achieve high performance during the solve phase, the latter is discussed first in Chapter 3. Then, the analysis phase is discussed in Chapter 4. The following Chapter 5 evaluates this approach on the preconditioned conjugate gradient method.

3

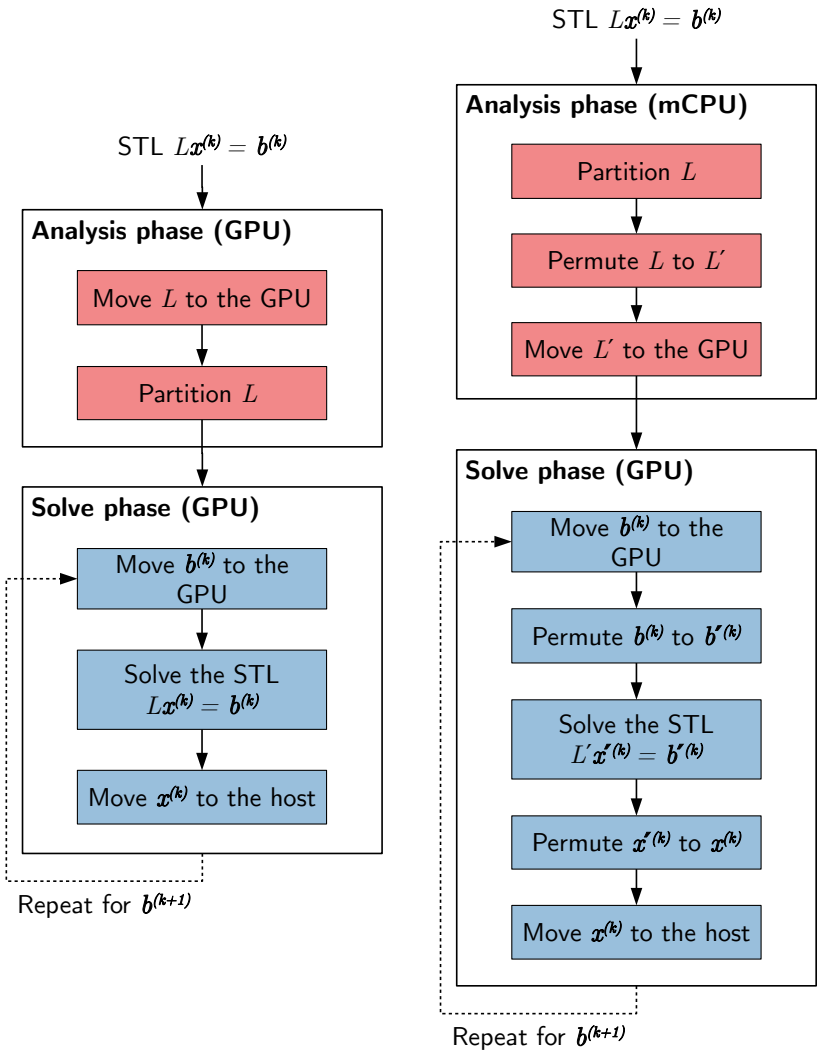
THE SOLVE PHASE

Any GPU software is composed of concurrent parts called work items. The GPU compiler groups 32 work items together into a SIMD thread, and an instruction from a SIMD thread is called SIMD instruction.

Work items must satisfy some conditions for the GPU software to achieve high performance. First, in high-performance GPU software there are many SIMD threads ready to carry out computations while others are waiting for memory accesses. If this happens, the GPU software achieves good *memory access latency hiding*. Also, in high-performance GPU software, when branching instructions like the `if-else` and `case` statements are executed the work items in the same SIMD thread all take the same branch. If this happens, the GPU software achieves low *thread divergence*. Section 2.1 of Chapter 2 explains why these features are tipical of high-performance GPU software by discussing the architecture of GPUs.

If the application is irregular, as in the case of the solution of sparse triangular linear systems of equations (STLs), good memory access latency hiding and low thread divergence can be achieved depending on the system matrix. The input data is a *sparse* matrix L defined in the space $\mathbb{R}^{n\times n}$ with which a directed acyclic graph $G := (\mathcal{V}, E)$ is associated. The set of graph vertices $\mathcal{V} := \{1, 2, \ldots, n\}$ has as many elements as the number of rows and columns if the matrix L. A non-zero entry $l_{i,j}$ at row i and column $j \neq i$ in the matrix L is represented with a directed edge $e_{i,j} := (i, j)$ from vertex j to vertex i in the graph G. The set $\mathcal{E}$ contains all the edges in the graph.

CUSPARSE [17], that is the most used GPU library for sparse linear algebra, aims to satisfy the conditions on memory access latency hiding and low thread divergence by ensuring as many work items as possible carry out the same operations at the same time. However, the CUSPARSE algorithm does not guarantee a good balance between SIMD threads waiting for memory accesses and those carrying out numerical operations.



(a) *CUSPARSE's approach.* (b) *TASSL's approach.*

Figure 3.1: Comparison of CUSPARSE's and TASSL's approach to solving STLs. In the figure, *mCPU* stands for multi-core CPU. TASSL executes the analysis phase on a multi-core CPU and the solve phase on the GPU, while CUSPARSE executes both of them on the GPU. Moreover, TASSL internally solves the STL $L'\boldsymbol{x'}^{(k)} = \boldsymbol{b'}^{(k)}$, where L', $\boldsymbol{x'}^{(k)}$, and $\boldsymbol{b'}^{(k)}$ are permuted versions of L, $\boldsymbol{x}^{(k)}$, and $\boldsymbol{b}^{(k)}$.

CUSPARSE's approach to solving STLs is organised in two phases: a preliminary *analysis phase*, which preprocesses and partitions input data, and a *solve phase*, which uses the information obtained in the previous phase to solve STLs on the GPU. The solve phase of CUSPARSE is a parallel form of the forward

substitution algorithm [25, ch. 2] and corresponds to a breadth-first traversal of the edges in the graph G [17].

The approach described in this thesis is called *time slot sub-graph level* (TASSL). Similarly to CUSPARSE, TASSL is organised in a preliminary analysis phase and a solve phase. The two approaches are compared in Figure 3.1 at a high level. This chapter illustrates the solve phase of TASSL, which overcomes the limitations of CUSPARSE's technique and achieves better performance. Section 3.1 discusses the shortcomings of the CUSPARSE approach in more detail. In Section 3.2 an algorithm for the solve phase of TASSL is developed and a performance model is derived for it. Experimental results that compare the two solve phases are shown in Section 3.3.

The research contributions discussed in this chapter are the following:

- an algorithm to solve STLs with GPUs given a partition of the corresponding graph [23].
- A nomenclature that describes the common elements in the implementation of irregular applications on GPUs, which consists of:
  - The concepts of sub-graph, sub-graph level, and time slot, which describe the scheduling of numerical operations on a GPU.
  - The concept of *concurrency factor*, which is a value between zero and one that captures the level of concurrency in the execution of an irregular algorithm.
  - The concept of *efficiency factor*, which is a value between zero and one that captures the effects of data locality, thread divergence, and memory access latency hiding on a GPU.
  - The concept of *synchronisation overhead*, which captures how much time is spent synchronising the execution of the work items with respect to the execution time of numerical operations.
  - The concept of *local transfer overhead*, which captures how much time is spent copying data from and to local memory with respect to the execution time of numerical operations.

- Two performance models, one for the solve phase of CUSPARSE and one for that of TASSL that use the concepts introduced in this chapter.
- A comparison between the TASSL and the CUSPARSE approaches, which shows that the solve phase of TASSL is faster than that of CUSPARSE in about 70 % of the examined test cases [23].

3.1 SHORTCOMINGS OF THE CUSPARSE ALGORITHM

Consider the following STL:

$$L\boldsymbol{x} = \boldsymbol{b} \tag{3.1}$$

where L is a triangular matrix in $\mathbb{R}^{n\times n}$ and the vectors $\boldsymbol{x}$ and $\boldsymbol{b}$ are both in $\mathbb{R}^n$. In particular, $\boldsymbol{b}$ is called the *right-hand side vector* of the STL. Also, most of the entries in L are equal to zero, so in the matrix is stored in a sparse data structure like the *compressed sparse row* format (CSR) which is discussed in Section 2.3.2 of Chapter 2 (page 46). Instead, most of the entries in the arrays $\boldsymbol{x}$ and $\boldsymbol{b}$ are not zero, so they are stored in conventional arrays, which are *dense* data structures.

CUSPARSE's algorithm to solve (3.1) [16], also called the *level-set* algorithm, is executed after a preliminary analysis phase. The analysis phase produces a partition of the vertices of the graph $G \coloneqq (\mathcal{V}, \mathcal{E})$ associated with the matrix L into *levels* $\mathcal{V}_k$ such that

$$\mathcal{V} = \mathcal{V}_0 \cup \mathcal{V}_1 \cup \mathcal{V}_2 \cup \cdots \cup \mathcal{V}_{n_{\text{levels}}-1}.$$

Each level $\mathcal{V}_k$ contains only vertices that do not have entering edges starting from vertices in the same level or other levels with a higher index k, hence:

$$\mathcal{V}_k \coloneqq \left\{ v \in \mathcal{V} \mid \nexists\, (v,j) \in \mathcal{E} \wedge\ j \in \mathcal{V}_p \quad p = k, k+1, \ldots, n_{\text{levels}} - 1 \right\}. \tag{3.2}$$

The algorithm of the solve phase of CUSPARSE, shown in Algorithm 3.1, carries out numerical operations in parallel with the SIMD paradigm at each level. Each edge (i,j) in the graph represents the numerical operation

Input: $\boldsymbol{b}$, $\mathcal{V}_0$, $\mathcal{V}_1$, $\mathcal{V}_2$, $\ldots$, $\mathcal{V}_{n_{\text{levels}}-1}$,
Output: $\boldsymbol{x}$
1: $\boldsymbol{x} \leftarrow \boldsymbol{b}$
2: **for** $p \leftarrow 1, 2, \ldots, n_{\text{levels}} - 1$
3: **for all** $i \in \mathcal{V}_p$ **do in parallel** ▷ SIMD
4: **for all** $(i, j) \in \mathcal{E}$
5: $x_i \leftarrow x_i - l_{i,j} x_j$

Algorithm 3.1: Algorithm of the solve phase of CUSPARSE [16]. The vectors $\boldsymbol{x}$ and $\boldsymbol{b}$ are defined in $\mathbb{R}^n$, while $l_{i,j}$ is the entry in row i and column j of the matrix $L \in \mathbb{R}^{n \times n}$. In the pseudocode, $\mathcal{V}_p$, with $p = 1, 2, \ldots, n_{\text{levels}}$ are the levels.

$$x_i \leftarrow x_i - l_{i,j} x_j \tag{3.3}$$

where $l_{i,j}$ is the entry of matrix L in row i and column j. Since numerical operations carried out at the same time must read the correct, up-to-date value of x_j, edges entering the same vertex i are executed at different times, as shown in line 4.

3.1.1 *Nomenclature used in this thesis*

The sequence of numerical operations of the solve phase of CUSPARSE can be described by introducing the concept of *synchronisation point*. In this thesis, a synchronisation point is either the start of the execution of a GPU kernel, the end of the execution of a GPU kernel, or a barrier. The set of numerical operations executed between two synchronisation points is called a *time slot*. Hence, a time slot does not necessarily have a constant duration in clock cycles. In each time slot, one or more numerical operations are associated with an *active element*. Numerical operations associated with the same active element cannot be executed concurrently. For example, consider the following numerical operations of the type in (3.3)

$$
\begin{aligned}
x_i &\leftarrow x_i - l_{i,j_1} x_{j_1} \\
x_i &\leftarrow x_i - l_{i,j_2} x_{j_2} \\
x_i &\leftarrow x_i - l_{i,j_3} x_{j_3} \\
x_i &\leftarrow x_i - l_{i,j_4} x_{j_4}
\end{aligned}
$$

where i is a vertex in a graph G and is the active element, and j_1, j_2, j_3 and j_4 are other vertices in G. All these numerical operations associated with i update the same value x_i. Therefore, these numerical operations are executed either in sequence or as atomic operations. In this thesis, the case of numerical operations executed in sequence is considered, because atomic operations on GPUs can have a large latency, as discussed in Section 2.1.2 of Chapter 2 (page 26). The duration of time slot $\mathcal{T}_k$ is determined by the largest number of numerical operations associated with an active element of $\mathcal{T}_k$, and this number is denoted with o_k. The concept of active element was first introduced by Pingali *et al.* [50].

In the solve phase of CUSPARSE, there is one time slot $\mathcal{T}_k$ per level $\mathcal{V}_k$. In each time slot, the vertices in $\mathcal{V}_k$ are the active elements. Since one numerical operation corresponds to one edge in the graph, the numerical operations associated with a vertex i are those corresponding to its entering edges. Figure 3.2 illustrates how the nomenclature introduced in this section matches CUSPARSE's algorithm.

In CUSPARSE, there is no limit to how many vertices can be in a level. Since line 3 in Algorithm 3.1 is executed concurrently, if the number vertices in a level is very large, that level is processed with a GPU kernel. Otherwise, the level is executed in the same kernel as previous or following levels [16]. Since the source code of CUSPARSE is proprietary, it is not known how this decision is implemented but the number of GPU kernels executed, denoted with n_{kernel}, can be measured with the profiling tool *nvprof* [129].

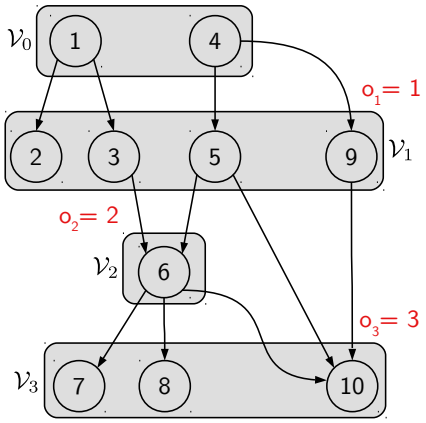


Figure 3.2: An example graph and the corresponding operation scheduling with CUSPARSE's algorithm [16]. Levels are represented with shaded blocks and there is only one time slot per level. The duration o_k of each time slot corresponds to the largest number of entering edges per vertex in $\mathcal{V}_k$.

3.1.2 *Analysis of the CUSPARSE algorithm*

To analyse the performance of the solve phase of CUSPARSE, one can derive a model of its execution time, which is denoted with t_{CUSPARSE}.

Before and after the execution of the solve phase of CUSPARSE, the right-hand side vector $\boldsymbol{b}$ is transferred to and from the GPU memory. The execution time of the transfer is given by

$$t_{\text{transf}} \approx a_1 n + a_2 \tag{3.4}$$

where a_1 and a_2 are two positive real numbers and n is the number of rows and columns of the input matrix L.

The execution time of Algorithm 3.1 is given by the execution time of the levels. The execution time t_k of level $\mathcal{V}_k$ is determined by

$$t_k \approx \frac{o_k}{T\eta_k} + \sigma_k = \frac{o_k}{\eta_k}\iota + \sigma_k \tag{3.5}$$

where o_k is the maximum number of numerical operations per active element, σ_k is the synchronisation time, and the η_k is called *operation efficiency*.

In (3.5), T is the theoretical throughput T of SIMD instructions of the type in (3.3). The value of T or its inverse ι can be either given by the GPU documen-

tation or obtained experimentally [130]. The value of ι captures only the time required by the GPU's arithmetic-logic units to carry out the SIMD instructions but it does not include any memory transfer. For example, in Nvidia GPUs this value can be found in the CUDA programming guide [131]. The operation efficiency η_k captures the effects of data locality, memory access coalescing, and thread divergence in the execution of level $\mathcal{V}_k$, while excluding synchronisation overheads. If the value of η_k is equal to one, the throughput achieved with the algorithm is equal to T, which is the ideal case of no thread divergence and a null memory access latency.

The maximum number of numerical operations per active element o_k in (3.5) depends on the partitioning the graph G into levels. The synchronisation time σ_k depends on what type of synchronisation point is used at the end of the execution of the level $\mathcal{V}_k$. If the synchronisation point requires stopping and starting a new GPU kernel, then σ_k is equal to a value κ, while if requires executing a barrier in global memory, then σ_k is equal to a value σ_{global}. Table 3.1 shows values of the parameters in the model obtained either from the documentation for the GPU or experimentally for an Nvidia Quadro K4000 GPU [132].

To obtain the value of the parameter κ experimentally, an empty GPU kernel was executed on the GPU 100 times and the average execution time was taken. To obtain the value of the parameters σ_{global} and σ_{local}, GPU kernels containing only barriers were executed and the numerical value of the parameter were obtained by interpolation. For example, in the case of σ_{global} the following polynomial was used

$$n_{\text{barriers}}\sigma_{global} + t_{\text{offset}}$$

where n_{barriers} is the number of barriers in the GPU kernel and t_{offset} is a positive constant. Although the execution time of synchronisation points varies slightly depending on how many work items are synchronised, this was neglected to obtain a simplified performance model.

For the complete CUSPARSE algorithm, the execution time is given by:

Table 3.1: Features of the Nvidia Quadro K4000 GPU [132] used in the execution time model of both CUSPARSE and TASSL. The value of the execution time of a SIMD instruction ι for an instruction of the type in (3.3) with 64-bit operands is given in the CUDA programming guide [131]. The other values were instead obtained experimentally. In the table, κ is the time required for starting and stopping the execution of a GPU kernel, σ_{global} and σ_{local} are the execution time of a barrier in global and in local memory respectively, β is the time required to transfer a an array element from global to local memory and back if the array stores double precision floating point numbers.

Model parameter	Value
κ	2.929×10^{-5} s
ι	4.938×10^{-9} s/SIMD instruction
σ_{global}	3.659×10^{-6} s
σ_{local}	1.247×10^{-7} s
β	1.596×10^{-8} s/element

$$\begin{aligned} t_{\text{algo}} &= \sum_{k=1}^{n_{\text{levels}}-1} \left(\frac{o_k}{\eta_k}\iota + \sigma_k \right) \\ &= \iota \sum_{k=1}^{n_{\text{levels}}-1} \frac{o_k}{\eta_k} + \kappa n_{\text{kernel}} + \sigma_{\text{global}}\left(n_{\text{levels}} - n_{\text{kernel}}\right) \end{aligned} \tag{3.6}$$

where n_{kernel} is the number of GPU kernels that are executed during the solve phase of CUSPARSE and can be determined experimentally. Some algebraic manipulation can be done on the expression in (3.6):

$$\begin{aligned} t_{\text{algo}} &= \iota \sum_{k=1}^{n_{\text{levels}}-1} \frac{o_k}{\eta_k} + \kappa n_{\text{kernel}} + \sigma_{\text{global}}\left(n_{\text{levels}} - n_{\text{kernel}}\right) \\ &= \iota \left(\sum_{k=1}^{n_{\text{levels}}-1} \frac{o_k}{\eta_k} \right) \frac{\sum_{k=1}^{n_{\text{levels}}-1} o_k}{\sum_{k=1}^{n_{\text{levels}}-1} o_k} \frac{n_{\text{edges}}}{n_{\text{edges}}} + \kappa n_{\text{kernel}} + \sigma_{\text{global}}\left(n_{\text{levels}} - n_{\text{kernel}}\right) \\ &= \underbrace{\iota \left(\sum_{k=1}^{n_{\text{levels}}-1} \frac{o_k}{\eta_k} \right) \frac{1}{\sum_{k=1}^{n_{\text{levels}}-1} o_k}}_{1/\eta} \underbrace{\frac{\sum_{k=1}^{n_{\text{levels}}-1} o_k}{n_{\text{edges}}}}_{1-C} n_{\text{edges}} + \\ &\quad + \kappa n_{\text{kernel}} + \sigma_{\text{global}}\left(n_{\text{levels}} - n_{\text{kernel}}\right). \end{aligned} \tag{3.7}$$

In (3.7),

$$C := 1 - \frac{\sum_{k=1}^{n_{\text{levels}}-1} o_k}{n_{\text{edges}}} \tag{3.8}$$

is the *concurrency factor* of the matrix L with respect to the CUSPARSE algorithm, and it is a quantity between zero and one. When the numerator of the ratio in (3.8) is equal to one, the algorithm requires sequences of SIMD instructions, called SIMD threads, of length one. In fact, work items execute only one operation per time slot and there is only one time slot in total. This is the case of maximum concurrency and C is closer to one the more the edges in the graph. If the numerator of the ratio in (3.8) is equal to n_{edges}, all the numerical operations are executed in sequence one after the other. This is the case of a graph whose structure is that of a *chain*, where concurrency cannot be utilized. In this case, the value of the concurrency factor is closer to zero the more the edges in the graph. Hence, the concurrency factor C captures the scheduling of the SIMD instructions given the structure of the graph. It does not capture the effects of synchronisation, data locality, memory access latency hiding, thread divergence, and other effects like *register spilling*, which were discussed in Section 2.1 of Chapter 2 (page 24).

Instead, in (3.7),

$$\eta := \frac{\sum_{k=1}^{n_{\text{levels}}-1} o_k}{\sum_{k=1}^{n_{\text{levels}}-1} \frac{o_k}{\eta_k}} \tag{3.9}$$

is the *efficiency factor* of the matrix L with respect to the CUSPARSE algorithm, and it is also a quantity between zero and one. Similarly to the operation efficiency, η can be equal to one only in the ideal case, in which the operation throughput is equal to T. The efficiency factor captures all effects which were discussed in Section 2.1 of Chapter 2 (page 24) like those of of data locality, memory access latency hiding, and thread divergence. However, the efficiency factor does not capture the effect of synchronisation points on the execution time.

Using (3.8) and (3.9), one can write (3.7) as

$$
\begin{aligned}
t_{\text{algo}} &= \frac{1-C}{\eta}\iota n_{\text{edges}} + \kappa n_{\text{kernel}} + \sigma_{\text{global}}\left(n_{\text{levels}} - n_{\text{kernel}}\right) \\
&= \frac{1-C}{\eta}\iota n_{\text{edges}}\left[1 + \underbrace{\frac{\kappa n_{\text{kernel}} + \sigma_{\text{global}}\left(n_{\text{levels}} - n_{\text{kernel}}\right)}{\frac{1-C}{\eta}\iota n_{\text{edges}}}}_{\sigma}\right]
\end{aligned}
\tag{3.10}
$$

In (3.10)

$$
\sigma := \frac{\kappa n_{\text{kernel}} + \sigma_{\text{global}}\left(n_{\text{levels}} - n_{\text{kernel}}\right)}{\frac{1-C}{\eta}\iota n_{\text{edges}}} \tag{3.11}
$$

is the *synchronisation overhead*. The synchronisation overhead is the ratio of the part of execution time of the algorithm that is spent on synchronisation points to the part that is spent carrying out numerical operations.

The formulation in (3.10) can be simplified even further by introducing the *effective operation time*, which is defined as

$$
\tau := \frac{\iota}{\eta}\left(1 + \sigma\right) \tag{3.12}
$$

and captures at the same time all the effects captured by the efficiency factor and the synchronisation overhead. Using this notation, (3.10) becomes

$$
t_{\text{algo}} = \left(1 - C\right)\tau n_{\text{edges}}, \tag{3.13}
$$

which is the most concise form of the performance model. The physical meaning of τ is given by its inverse, which has the dimensions of a throughput. In fact, $1/\tau$ is the apparent rate of SIMD operations of the type in (3.3) during the execution of the CUSPARSE algorithm.

In this derivation, the dependency of t_{algo} on the number of compute units in the GPU is neglected. This requires the number of vertices in each level to be not much greater than the number of compute units. This can be verified by introducing another metric, called *per-thread concurrency* and denoted in this thesis with W, which estimates how many work items are used to execute the algorithm on the GPU. The per-thread concurrency of matrix L with respect to the CUSPARSE solve phase is defined as:

$$W := \frac{\max_k |\mathcal{V}_k|}{n_{\text{edges}}}. \tag{3.14}$$

As shown in line 3 of Algorithm 3.1, CUSPARSE executes one work item per active element in each level. Hence, the numerator of the ratio in (3.14) represents the maximum number of work items used during the execution of the algorithm. If W is close to one, there is almost one work item per vertex in the graph, and consequently C is also close to one because the longest SIMD thread is composed of few SIMD instructions. This metric is similar to the *degree of parallelism* metric first defined by Kulkarni *et al.* [51]. Given the number of compute units in the GPU n_{CU} and the maximum number of work items per compute unit n_{w}, the model developed in this section assumes

$$W \leq \frac{n_{\text{CU}} n_{\text{w}}}{n_{\text{edges}}}\,, \tag{3.15}$$

which is the case when not enough SIMD threads are used to occupy all the compute units in the GPU and the overhead of scheduling work groups of compute units can be neglected.

3.1.3 *Evaluation of the performance model*

The form used for evaluation was obtained from (3.4) and (3.10):

$$\begin{aligned} t_{\text{CUSPARSE}} &\approx t_{\text{transf}} + t_{\text{algo}} \\ &= a_1 n + \iota \frac{1-C}{\eta} n_{\text{edges}} + \kappa n_{\text{kernel}} + \sigma_{\text{global}} \left(n_{\text{levels}} - n_{\text{kernel}} \right) + a_2 \end{aligned} \tag{3.16}$$

where the parameters C and n_{levels} are known from how CUSPARSE partitions the graph [16], n_{kernel} is found using *nvprof* [129], n_{edges} is known from the graph, and κ, σ_{global}, and ι are known from either the GPU documentation or benchmarks. The values of a_1, η, and a_2 can be obtained experimentally, considering that a_1 and a_2 are the same for any test case.

To evaluate the model in (3.16), some tests were carried out using the Florida matrix collecton [133], a standard benchmark set for sparse linear algebra algorithms that contains non-artificial test cases. Only the lower triangular part of

Table 3.2: Features of the test cases from the Florida matrix collection [133] and range of their values.

Feature	Parameter	min	max
Number of rows/columns	n	10261	525824
Number of graph edges	n_{edges}	760	4356551

square matrices in this set of benchmarks was considered. Hence, the resulting triangular matrices had the same sparsity patterns as the lower half of the matrices in the benchmark. CUSPARSE was executed on an Nvidia Quadro K4000 GPU [132], whose parameters are shown in Table 3.1. Of all the test cases examined, those for which the standard deviation of the measured execution time was more than 10 % of the mean value were discarded to filter the noisiest experimental data. Of the remaining test cases, the first 200 starting from that with the fewest rows and columns were taken. Table 3.2 summarises the features of these matrices.

After the filtering, all the experimental data was used to fit the model in (3.16), using the non-negative least squares algorithm [134]. Since the algorithm is known in its entirety, the model fitting did not require checking with a validation set. In fact, only implementation-dependent parameter, n_{kernel}, was measured with *nvprof* and hence is known for each test case. For these reasons, instead of using a validation set, all the data was used to for fitting the model to achieve higher accuracy in the estimate of the parameters.

The parameters in (3.16) are shown in Table 3.3. Figure 3.3 shows how the performance model compares to the experimental data. In general the model overestimates the execution time, because the model describes synchronisation with constant parameters. For example, the execution time of a global memory barrier, described in Section 2.1.2 of Chapter 2, depends on how many work items are involved and it is not constant in reality. However, Figure 3.3 shows that the model is an acceptable first-order approximation of the performance of CUSPARSE and can be used to make the following observations on the algorithm.

Table 3.3: Parameter values for the performance model of CUSPARSE [16] in (3.16). These parameters were obtained by executing CUSPARSE on 200 test cases from the Matrix Collection [133]. In this table, $\bar{\eta}$ and $\bar{C}$ are respectively the average efficiency factor and the average concurrency factor.

Parameter	Value
a_1	2.294×10^{-8} s/element
a_2	5.990×10^{-4} s
$\bar{\eta}$	1.695×10^{-1}
$\bar{C}$	8.871×10^{-1}

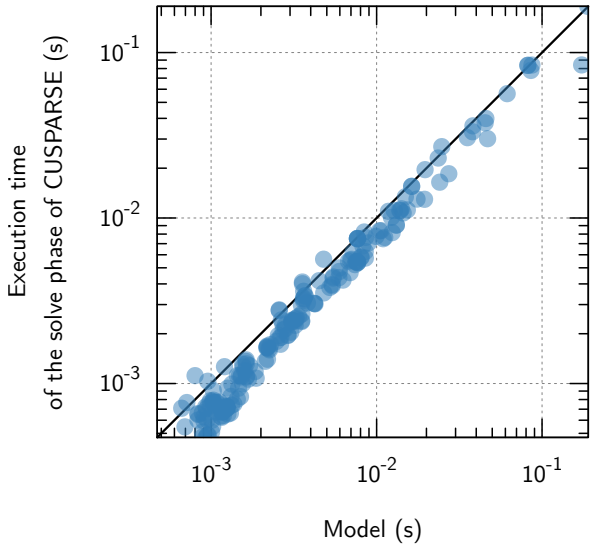


Figure 3.3: Comparison between the experimental results on the execution time of CUSPARSE's solve phase [16] and the theoretical model in (3.16). Each point in the diagram represents one of the 200 test cases taken from the Florida matrix collection [133].

First, the model gives a mean value $\bar{\eta}$ of the efficiency factor of about 17 %. To check if this metric can be improved, a follow-up test was carried out with the GPU profiler *nprof* [129]. This showed that about 20 % of branching instructions in the solve phase of CUSPARSE caused branch divergence. This is because different active elements require a different number of instructions, as shown in line 4 in Algorithm 3.1. Although the model in (3.16) does not capture the effect of thread divergence with a specific parameter, this follow-

up experiment showed that there is margin of improvement of the efficiency factor with respect to CUSPARSE.

3.2 THE TASSL APPROACH

This thesis proposes an approach to the solution of STLs different than that of CUSPARSE called *time slot sub-graph level* (TASSL). The aim of this approach is, with respect to CUSPARSE, to reduce the time spent in synchronisation points, to reduce thread divergence, and to improve memory access latency hiding.

Similarly to CUSPARSE, TASSL is composed of two phases: an analysis phase and a solve phase. Similarly to the previous section, which described the solve phase of CUSPARSE, this section describes the solve phase of TASSL and is organised as follows: Section 3.2.1 defines some concepts that are used in the development of the algorithm, Section 3.2.2 explains the algorithm, and in Section 3.2.3 a performance model for the solve phase of TASSL is developed.

3.2.1 *External and internal edges*

Consider the system of linear equations in (3.1), where L is the matrix of the STL and $G := (\mathcal{V}, \mathcal{E})$ the graph associatied with it. The solve phase of TASSL requires the set of vertices $\mathcal{V}$ to be partitioned into $n_s + 1$ disjoint subsets $\mathcal{V}_0, \mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_{n_s}$, such that

$$\mathcal{V} = \mathcal{V}_0 \cup \mathcal{V}_1 \cup \mathcal{V}_2 \cup \cdots \cup \mathcal{V}_{n_s}, \tag{3.17}$$

where $\mathcal{V}_0$ is the subset of vertices in the graph G with neither entering nor exiting edges.

Given the partition of G in (3.17), an edge (i, j) is an *internal edge* if both vertex i and vertex j are in the same set $\mathcal{V}_k$. Otherwise, it is an *external edge*. Also, for the scope of this thesis a *sub-graph* $G_k := (\mathcal{V}_k, \mathcal{E}_k)$ of G is composed of the vertices in the subset $\mathcal{V}_k$, and the union of the set of internal edges for $\mathcal{V}_k$, which is defined by

$$\mathcal{E}_{\text{int},k} \coloneqq \{(i,j) \in \mathcal{E} \mid i \in \mathcal{V}_k \wedge j \in \mathcal{V}_k\} \,,$$

and the set of external edges for $\mathcal{V}_k$, which is defined by

$$\mathcal{E}_{\text{ext},k} \coloneqq \{(i,j) \in \mathcal{E} \mid i \in \mathcal{V}_k \wedge j \in \mathcal{V} \wedge j \notin \mathcal{V}_k\},$$

hence $\mathcal{E}_k = \mathcal{E}_{\text{int},k} \cup \mathcal{E}_{\text{ext},k}$.

The solve phase of TASSL requires the graph G to be partitioned such that:

CONDITION I: if an edge goes from a vertex in $\mathcal{V}_k$ to a vertex in $\mathcal{V}_p$, then $p \geq k$.

CONDITION II: the number of vertices in the set $\mathcal{V}_k$, denoted with n_k, must not be greater than a positive integer $n_{\max}$.

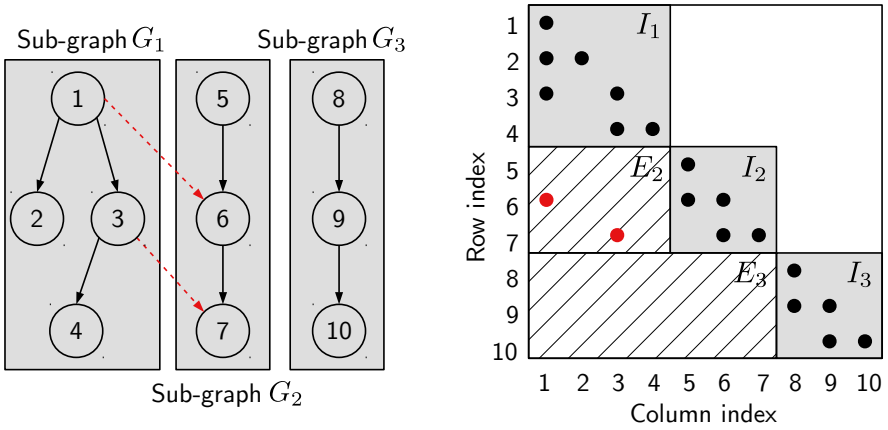
If both these conditions are met, then the partition is called a *feasible partition*. In particular, condition I implies that there is no cyclical dependency between variables associated with vertices in different $\mathcal{V}_k$ sets. Condition II implies that the number of vertices in each sub-graph is bounded. Using the OpenCL nomenclature discussed in Section 2.1.1 of Chapter 2 (page 24), if data is represented in the double precision floating point format, the upper bound of the number of vertices in $\mathcal{V}_k$, is equal to

$$n_{\max} \coloneqq \left\lfloor \frac{M_{\text{local}}}{8} \right\rfloor , \tag{3.18}$$

where M_{local} is the size in bytes of the local memory inside a compute unit.

If the partition is feasible, it is possible to permute rows and columns of the matrix to have indices in the same set $\mathcal{V}_k$ close together, while keeping a triangular structure. This permutation $Q \in \mathbb{R}^{n \times n}$ can be carried out by reordering the vertices of the original graph by sub-graph index. For example, the new indices of the rows and columns in the set $\mathcal{V}_1$ will be smaller than those of the rows and columns in $\mathcal{V}_2$. A lookup table can be used to keep track of the change in the indices.

By applying the permutation Q to to (3.1), one obtains



(a) *An example partitioned graph.*

(b) *The block form corresponding to the example graph.*

Figure 3.4: An example graph, partitioned into three sub-graphs, and the sparsity pattern of the corresponding matrix with the block form in (3.19). In the representation of the sparsity pattern, the non-zero entries are shown with dots, while in the representation of the graph external edges are represented with dashed lines.

$$\underbrace{\begin{bmatrix} I_0 & & & & 0 \\ & I_1 & & & \\ & E_2 & I_2 & & \\ 0 & E_3 & I_3 & & \\ & \vdots & & \ddots & \\ & E_{n_s} & & & I_{n_s} \end{bmatrix}}_{L'} \underbrace{\begin{bmatrix} \boldsymbol{x'_0} \\ \boldsymbol{x'_1} \\ \boldsymbol{x'_2} \\ \boldsymbol{x'_3} \\ \vdots \\ \boldsymbol{x'_{n_s}} \end{bmatrix}}_{x'} = \underbrace{\begin{bmatrix} \boldsymbol{b'_0} \\ \boldsymbol{b'_1} \\ \boldsymbol{b'_2} \\ \boldsymbol{b'_3} \\ \vdots \\ \boldsymbol{b'_{n_s}} \end{bmatrix}}_{b'} \tag{3.19}$$

where $L' := Q^\intercal L Q$, $\boldsymbol{x'} := Q^\intercal \boldsymbol{x}$, and $\boldsymbol{b'} := Q^T \boldsymbol{b}$. In (3.19), the blocks I_k and E_k are associated with $\mathcal{V}_k$. As shown in Figure 3.4, the elements in block E_k are associated with external edges, while the elements below the diagonal in block I_k are associated with internal edges of sub-graph G_k. The vectors $\boldsymbol{x'_k}$ and $\boldsymbol{b'_k}$ are defined in $\mathbb{R}^{n_k}$, where n_k is the number of vertices in $\mathcal{V}_k$. Also, I_k is a n_k-by-n_k lower triangular real matrix, while matrix E_k is a real matrix with n_k rows and $\sum_{p=1}^{k-1} n_p$ columns. By construction, as required by condition II, n_k is always smaller or equal to $n_{\max}$.

Equation (3.19) can be written in terms of $\boldsymbol{x'_k}$, thus obtaining

$$E_k \begin{bmatrix} \boldsymbol{x'_1} \\ \boldsymbol{x'_2} \\ \vdots \\ \boldsymbol{x'_{k-1}} \end{bmatrix} + I_k \boldsymbol{x'_k} = \boldsymbol{b'_k} \qquad \text{for } k = 1, 2, \ldots, n_s \tag{3.20}$$

and $E_1 = 0$.

3.2.2 *Algorithm of the solve phase*

The solve phase of TASSL computes the solution of n_s equations of the form in (3.20). Although one can solve the n_s equations in sequence, from $k = 1$ to $k = s$, some of the equations may be independent since some sub-graphs may not have external edges that connect them. For example, in Figure 3.4, sub-graph G_1 and sub-graph G_3 are not connected, so their traversal can be carried out in parallel. In general, the set of sub-graphs can be partitioned into n_{levels} subsets of independent sub-graphs

$$\mathcal{L}_1 \cup \mathcal{L}_2 \cup \cdots \cup \mathcal{L}_{n_{\text{levels}}}$$

where the set $\mathcal{L}_p$ is called the *sub-graph level* of index p, for $p = 1, 2, \ldots, n_{\text{levels}}$.

The proposed solution technique for (3.19) is outlined in Algorithm 3.2. A GPU kernel is executed n_{levels} times, each time for a different sub-graph level, in sequence. At each kernel execution, a group of work items, called a work group, is associated with each sub-graph G_k. This is because the algorithm stores the values of vector $\boldsymbol{x}_{\text{loc}} \in \mathbb{R}^{n_k}$ in local memory which can only be accessed by work items in the same work group. This allows data is to be shared across all the work items that process G_k. The values of $\boldsymbol{x}_{\text{loc}}$ are initialised with $\boldsymbol{b'_k}$, then external and internal edges of sub-graph G_k are processed. The computation in the two cases is of the same type: a sequence of multiply-and-add numerical operations like that in (3.3).

The aim of the analysis phase, which precedes the solve phase, is to find a feasible partition of G with the minimum number of sub-graph levels. In practice, if the partition has few sub-graph levels, few GPU kernels are executed

Input: $\boldsymbol{b}'$, $\mathcal{L}_1$, $\mathcal{L}_2$, ..., $\mathcal{L}_{n_{\text{levels}}}$, $\mathcal{T}_{kq}$ and $\mathcal{T}'_{kq}$ $\quad \forall q, k = 1, 2, \ldots, s$
Output: $\boldsymbol{x}'$

```
1: x' ← b'
2: for p ← 1, 2, ..., n_levels
3:     for all g ∈ L_p do in parallel
4:         x_loc ← x'_g
5:         for q ← 1, 2, ..., n'_slots,g
6:             for all (i, j) ∈ T'_g,q do in parallel                ▷ SIMD
7:                 x_loc,i ← x_loc,i − l'_i,j x'_j
8:         for q ← 1, 2, ..., n_slots,g
9:             for all (i, j) ∈ T_g,q do in parallel                 ▷ SIMD
10:                x_loc,i ← x_loc,i − l'_i,j x_loc,j
11:        x'_g ← x_loc
```

Algorithm 3.2: The solve phase of TASSL for a STL like that in (3.19). The vector $\boldsymbol{x}_{\text{loc}}$ is stored in local memory, and it is defined in $\mathbb{R}^{n_k}$, where n_k is the number of vertices in $\mathcal{V}_k$. In the pseudocode, $\mathcal{L}_p$, with $p = 1, 2, \ldots, n_{\text{levels}}$ are the sub-graph levels, and l'_{ij} is the entry in row i and column j in the matrix L'. Also, $\mathcal{T}_{g,q}$ and $\mathcal{T}'_{g,q}$ are the q-th time slot of internal and external edges respectively, in sub-graph G_g.

and synchronisation has a small effect on performance. Also, if the number of GPU kernels is small, a high number of sub-graphs are traversed concurrently and consequently the GPU is used efficiently.

Using the nomenclature introduced in Section 3.1.1, the computation of each sub-graph G_k is organised in one or more time slots, unlike in the solve phase of CUSPARSE where the computation was organised in one time slot per sub-graph. Also, in TASSL the active elements are graph edges and not vertices as in CUSPARSE. Hence, in TASSL only one numerical operation is associated with an active element.

Also, for sub-graph G_k, numerical operations associated with external edges are distributed over $n'_{\text{slots},k}$ time slots, while numerical operations associated with internal edges are distributed over $n_{\text{slots},k}$ time slots. For TASSL, the q-th time slot in sub-graph G_k is represented with $\mathcal{T}_{k,q}$ when referring to internal edges, and $\mathcal{T}'_{k,q}$ when referring to external edges. At the end of the algorithm, $\boldsymbol{x}_{\text{loc}}$ is copied to $\boldsymbol{x}'_{\boldsymbol{k}}$. The worst-case computational complexity of the algorithm is $O(n_{\text{edges}})$, where n_{edges} is the number of edges in the graph G.

The difference between the numerical operations associated with time slots of internal and external edges is the memory locations accessed. This is because when processing internal edges most memory accesses are in local memory, as shown in line 10 of Algorithm 3.2. Instead, when processing external edges most of the locations are in global memory, as shown in line 7. As discussed in Section 2.1.2 of Chapter 2 (page 26), the access latency of local memory is much lower than that of global memory. Also, since TASSL requires only one numerical operation per active element, one can expect a much smaller thread divergence compared to that of CUSPARSE. Because of these features, the solve phase of TASSL is expected to achieve a higher efficiency factor compared to that of CUSPARSE.

Moreover, as seen both in line 7 and in line 10 the results of both types of numerical operations are written to locations in local memory. This allows for the use of local memory barriers instead of global memory barriers as in CUSPARSE. However, compared to the solve phase of CUSPARSE, that of TASSL has two overheads: the application of the permutation Q to the elements in (3.1) to obtain the block form in (3.19), and the copying of data to and from local memory in line 4 and line 11 of Algortithm 3.2. To evaluate how these overheads affect the performance of the solve phase of TASSL, a performance model was developed.

3.2.3 *Analysis of the algorithm*

In this section, a performance model for Algorithm 3.2 is derived, which is then used in Chapter 4 to work out a technique to partition the graph G. The purpose of the performance model is to predict the execution time accurately and to characterise the execution time in terms of graph attributes.

In Algorithm 3.2, at each sub-graph level some numerical operations that are associated with either internal or external edges are carried out. These numerical operations are shown at line 7 and line 10 in Algorithm 3.2, where $l'_{i,j}$ is the entry of matrix L' in row i and column j. The time to execute a numerical operation depends on the location of each of the operands. If (i, j)

is an internal edge, then x'_i and x'_j are both in local memory, and their values can be accessed with relatively low latency. On the other hand, if (i, j) is an external edge, x'_j is in global memory, while x'_i is in local memory. The access latency of the value x'_j depends on whether this value is in some cache memory or not but, on average, this latency is greater than that of data in local memory. Moreover, the *cache hit rate* when accessing the location of $l'_{i,j}$ depends on how the matrix is stored. For these reasons, the model developed in this section considers two different values of operation efficiency: $\eta_{g,q}$ is the operation efficiency of the q-th time slot of internal edges $\mathcal{T}_{g,q}$ of sub-graph G_g, and $\eta'_{g,q'}$ is the operation efficiency of the q'-th time slot of external edges $\mathcal{T}_{g,q}$.

The development of the performance model is similar to that of the solve phase of CUSPARSE, which was carried out in Section 3.1.2. Given a sub-graph G_g, the execution time $t_{g,q}$ of the time slot of internal edges $\mathcal{T}_{g,q}$, and the execution time $t'_{gq'}$ of the time slot of external edges $\mathcal{T}'_{g,q'}$ can be written as

$$\begin{aligned} t_{g,q} &\approx \frac{\iota}{\eta_{g,q}} + \sigma_{g,q} \\ t'_{g,q'} &\approx \frac{\iota}{\eta'_{g,q'}} + \sigma'_{g,q'} \end{aligned}$$

where $\sigma_{g,q}$ and $\sigma'_{g,q'}$ are always equal to the execution time of a barrier in local memory σ_{local}. If $n_{\text{slots},g}$ and $n'_{\text{slots},g}$ are respectively the number of time slots of internal and external edges, the execution time of sub-graph G_g is given by

$$\begin{aligned} t_{\text{sub-graph},g} &\approx \sum_{q=1}^{n_{\text{slots},g}} t_{g,q} + \sum_{q'=1}^{n'_{\text{slots},g}} t'_{g,q'} + \beta n_g \\ &= \sum_{q=1}^{n_{\text{slots},g}} \frac{\iota}{\eta_{g,q}} + \sum_{q'=1}^{n'_{\text{slots},g}} \frac{\iota}{\eta'_{g,q'}} + (n_{\text{slots},g} + n'_{\text{slots},g})\sigma_{\text{local}} + \beta n_g \end{aligned} \tag{3.21}$$

In (3.21), β is the transfer rate from global memory to local memory and from local memory to global memory, $n_g = |\mathcal{V}_g|$ is the number of vertices in sub-graph G_g, and ι is the inverse of the theoretical throughput T of SIMD instructions of the type in (3.3). The execution time of sub-graph level $\mathcal{L}_p$ is given by the longest execution time of a graph in the sub-graph level, which is

$$
\begin{aligned}
t_{\text{level},p} &\approx \max_{g \in \mathcal{L}_p} \left\{ t_{\text{sub-graph},g} \right\} \\
&= \max_{g \in \mathcal{L}_p} \left\{ \iota \sum_{q=1}^{n_{\text{slots},g}} \frac{1}{\eta_{g,q}} + \iota \sum_{q'=1}^{n'_{\text{slots},g}} \frac{1}{\eta'_{g,q'}} + (n_{\text{slots},g} + n'_{\text{slots},g})\sigma_{\text{local}} + \beta n_g \right\}.
\end{aligned}
$$

This is then summed across all sub-graph levels, obtaining

$$
\begin{aligned}
t_{\text{algo}} = \sum_{p=1}^{n_{\text{levels}}} \max_{g \in \mathcal{L}_p} \left\{ \iota \sum_{q=1}^{n_{\text{slots},g}} \frac{1}{\eta_{g,q}} + \iota \sum_{q'=1}^{n'_{\text{slots},g}} \frac{1}{\eta'_{g,q'}} + (n_{\text{slots},g} + n'_{\text{slots},g})\sigma_{\text{local}} + \beta n_g \right\} \\
+ \kappa n_{\text{levels}}.
\end{aligned}
\tag{3.22}
$$

where κ is the time required for starting and stopping the execution of a GPU kernel.

Becase of the *max* operator, the values of $\eta_{g,q}$ and $\eta'_{g,q'}$ in (3.22) are difficult to compute from experimental data. One technique to overcome the issue is to consider these values constant across all sub-graphs and time slots, requiring $\eta_{g,q} \approx \eta_{\text{int}} \quad \forall g, q$ and $\eta'_{g,q'} \approx \eta_{\text{ext}} \quad \forall g, q'$. If this approximation is applied, η_{int} and η_{ext} can be computed using a nonlinear optimisation method [135].

If $\hat{p}$ is the index of the sub-graph in sub-graph level $\mathcal{L}_p$ with the longest execution time

$$
\hat{p} := \text{argmax}_{g \in \mathcal{L}_p} \left\{ \iota \sum_{q=1}^{n_{\text{slots},g}} \frac{1}{\eta_{g,q}} + \iota \sum_{q'=1}^{n'_{\text{slots},g}} \frac{1}{\eta'_{g,q'}} + (n_{\text{slots},g} + n'_{\text{slots},g})\sigma_{\text{local}} + \beta n_g \right\},
\tag{3.23}
$$

substituting in (3.22), it results:

$$
\begin{aligned}
t_{\text{algo}} &= \sum_{p=1}^{n_{\text{levels}}} \left[\iota \sum_{q=1}^{n_{\text{slots},\hat{p}}} \frac{1}{\eta_{\hat{p},q}} + \iota \sum_{q'=1}^{n'_{\text{slots},\hat{p}}} \frac{1}{\eta'_{\hat{p},q'}} + (n_{\text{slots},\hat{p}} + n'_{\text{slots},\hat{p}})\sigma_{\text{local}} + \beta n_{\hat{p}} \right] \\
&\quad + \kappa n_{\text{levels}} \\
&= \iota \sum_{p=1}^{n_{\text{levels}}} \left(\sum_{q=1}^{n_{\text{slots},\hat{p}}} \frac{1}{\eta_{\hat{p},q}} + \sum_{q'=1}^{n'_{\text{slots},\hat{p}}} \frac{1}{\eta'_{\hat{p},q'}} \right) \\
&\quad + \sigma_{\text{local}} \sum_{p=1}^{n_{\text{levels}}} (n_{\text{slots},\hat{p}} + n'_{\text{slots},\hat{p}}) + \kappa n_{\text{levels}} \\
&\quad + \beta \sum_{p=1}^{n_{\text{levels}}} n_{\hat{p}}
\end{aligned}
\tag{3.24}
$$

Then, similarly to the case of the analysis of CUSPARSE, the first term of (3.24) can be multiplied and divided by $n_{\text{edges}} \sum_{p=1}^{n_{\text{levels}}} \left(n_{\text{slots},\hat{p}} + n'_{\text{slots},\hat{p}} \right)$ to define the concurrency factor C as

$$
C := 1 - \frac{\sum_{p=1}^{n_{\text{levels}}} \left(n_{\text{slots},\hat{p}} + n'_{\text{slots},\hat{p}} \right)}{n_{\text{edges}}} \tag{3.25}
$$

and the efficiency factor η as

$$
\eta := \frac{\sum_{p=1}^{n_{\text{levels}}} \left(n_{\text{slots},\hat{p}} + n'_{\text{slots},\hat{p}} \right)}{\sum_{p=1}^{n_{\text{levels}}} \left(\sum_{q=1}^{n_{\text{slots},\hat{p}}} \frac{1}{\eta_{\hat{p},q}} + \sum_{q'=1}^{n'_{\text{slots},\hat{p}}} \frac{1}{\eta'_{\hat{p},q'}} \right)}. \tag{3.26}
$$

Substituting (3.25) and (3.26) in (3.24) one also obtains:

$$
t_{\text{algo}} = \frac{1 - C}{\eta} \iota n_{\text{edges}} + \sigma_{\text{local}} \sum_{p=1}^{n_{\text{levels}}} (n_{\text{slots},\hat{p}} + n'_{\text{slots},\hat{p}}) + \kappa n_{\text{levels}} + \beta \sum_{p=1}^{n_{\text{levels}}} n_{\hat{p}}, \tag{3.27}
$$

from which, using (3.25), one can define the synchronisation overhead σ

$$
\sigma := \frac{\eta}{\iota} \left[\sigma_{\text{local}} + \frac{\kappa n_{\text{levels}}}{(1 - C) n_{\text{edges}}} \right] \tag{3.28}
$$

and the *local transfer overhead* λ

$$
\lambda := \frac{\beta \sum_{p=1}^{n_{\text{levels}}} n_{\hat{p}}}{\frac{1-C}{\eta} \iota n_{\text{edges}}}. \tag{3.29}
$$

In particular, the local transfer overhead represents how much time is spent transferring data to and from the local memory with respect to the time spent carrying out numerical operations. Apart from the local transfer overhead, these metrics have also been defined for the solve phase of CUSPARSE in Section 3.1.2, from which one can also generalise the concept of effective operation time τ. In fact, by defininining $\tau := \frac{\iota}{\eta}(1 + \sigma + \lambda)$ it results

$$t_{\text{algo}} = (1 - C)\tau n_{\text{edges}}, \tag{3.30}$$

which is the same expression that was obtained for the case of CUSPARSE in (3.13).

As for the solve phase of CUSPARSE, one can define the per-thread concurrency of the matrix L with respect to the solve phase of TASSL as:

$$W := \frac{\max_{l \in \{\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_{n_{\text{levels}}}\}} \left\{ \sum_{g \in l} \max_{q,q'} \left\{ |\mathcal{T}_{g,q}|, |\mathcal{T}'_{g,q'}| \right\} \right\}}{n_{\text{edges}}}, \tag{3.31}$$

which represents the maximum number of work items used to execute the algorithm on the GPU. When W is close to one, also C is close to one because the number of SIMD instructions in the longest SIMD thread are few. Moreover, the validity of the model developed in this section is limited to the case

$$W \leq \frac{n_{\text{CU}} n_{\text{w}}}{n_{\text{edges}}}. \tag{3.32}$$

where n_{CU} is the number of compute units in the GPU, ad n_{w} is the maximum number of work items per compute unit. This is the case in which not enough SIMD threads are used to occupy all the compute units in the GPU and the overhead of scheduling work groups of compute units can be neglected.

The solve phase of TASSL also requires applying the permutation matrix Q to the right-hand side vector $\boldsymbol{x}$ in (3.1) to obtain the vector $\boldsymbol{x}'$ of the block form in (3.19), and then its inverse. The total execution time of these permutations is given by

$$t_{\text{perm}} \approx a_3 n + a_4 \tag{3.33}$$

where a_3 and a_4 are positive real numbers and n is the number of elements of $\boldsymbol{x}'$.

3.2.4 *Evaluation of the performance model*

The model of the execution time of the solve phase of TASSL was

$$t_{\text{TASSL}} \approx t_{\text{transf}} + t_{\text{perm}} + t_{\text{algo}} \ ,$$

hence

$$t_{\text{TASSL}} \approx (a_1 + a_3)\, n + \sum_{p=1}^{n_{\text{levels}}} \left[\iota \frac{n_{\text{slots},\hat{p}}}{\eta_{\text{int}}} + \iota \frac{n'_{\text{slos},\hat{p}}}{\eta_{\text{ext}}} + (n_{\text{slots},\hat{p}} + n'_{\text{slots},\hat{p}})\sigma_{\text{local}} + \beta n_{\hat{p}} \right] + \kappa n_{\text{levels}} + a_2 + a_4 \quad (3.34)$$

where the values of a_1 and a_2 were taken from Table 3.3, the values of ι, κ, σ_{local}, and β were taken from Table 3.1 for the Nvidia Quadro K4000 GPU [132], and all the other parameters but η, a_3, and a_4 were known from the execution of the analysis phase of TASSL. The values of $\hat{p}$ for each sub-graph level $\mathcal{L}_p$ were obtained using MATLAB's nonlinear optimisation solver, which implements an interior point method [85].

To obtain the values of η, the solve phase of TASSL was executed over the same test cases from the Florida matrix collection used in Section 3.1.3 for CUSPARSE, whose features are summarised in Table 3.2 (page 72).

The numerical values of the obtained parameters are shown in Table 3.4 and a visual comparison between the experimental data and the proposed model is shown in Figure 3.5. In general the model overestimates the execution time of those test cases whose execution time is less than 1 ms. This is because the model assumes the the execution time of a barrier in local memory, denoted with σ_{local}, and the memory transfer rate between global to local memory, denoted with β, to be constant. In reality, these values change depending on the number of work items used by the GPU software.

Table 3.4: Parameter values for the performance model of TASSL in (3.34). These parameters were obtained by executing TASSL on 200 test cases from the Florida matrix collection [133]. In this table, $\bar{\eta}$ and $\bar{C}$ are respectively the average efficiency factor and the average concurrency factor.

Parameter	Value
a_3	2.391×10^{-8} s/element
a_4	2.296×10^{-4} s
$\bar{\eta}$	4.407×10^{-1}
$\bar{C}$	9.907×10^{-1}

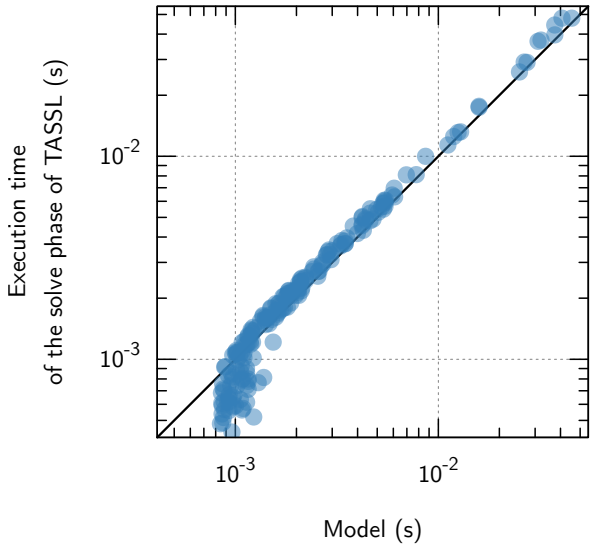


Figure 3.5: Comparison between the experimental results on the execution time of the solve phase of TASSL and the theoretical model in (3.34). Each point in the diagram represents one of the 200 test cases taken from the Florida matrix collection [133].

With respect to the model in (3.24), that in (3.34) is an application the technique described earlier in this chapter (page 81). For this technique, the value of the operation efficiency $\eta_{g,q}$ for time slots associated with internal edges was approximated with a constant value η_{int}, and that for time slots associated with external edges $\eta'_{g,q'}$ was approximated with a constant value η_{ext}. The values of η_{int} and η_{ext} change from problem to problem but it was observed that, on average, η_{int} was 7.8 times higher than η_{ext}.

Table 3.3 (page 73) showed that CUSPARSE obtained a mean value of the efficiency factor $\bar{\eta}$ across the test cases of 17 % and a mean value of the concurrency factor $\bar{C}$ of 89 %. Instead, TASSL obtained a value of $\bar{\eta}$ of about 50 % and a mean value of $\bar{C}$ of about 99 %. The two approaches are compared more in detail in the next section.

3.3 EXPERIMENTAL RESULTS

To study how the performance of the solve phase of TASSL changes with the sparsity pattern of L, five experiments were carried out. The experiments were divided into two categories: experiments with automatically generated test cases and experiments with test cases from the Florida matrix collection [133].

The solve phase of both CUSPARSE and TASSL was executed 100 times per matrix in every experiment, each time with a different randomly generated right hand side, and the mean of the results was taken. The experimental results consider: the transfer of the right-hand side vector from the CPU to the GPU, the additional vector permutations required by TASSL (included only in the TASSL case), the execution of the solution of the STL, and the transfer of the result back from the GPU to the CPU. The transfer of the STL matrices is not included because both TASSL and CUSPARSE are to be used in iterative methods, where the matrix is transfered only once at the beginning of the algorithm [16]. The double-precision floating point format was used for matrix entries in both types of test.

3.3.1 *Automatically generated test cases*

Automatically generated test cases are matrices that were generated to have specific properties, with the aim of studying the performance of the algorithms in function of specific graph attributes. The execution time of the solve phase of TASSL, which was implemented with OpenCL, was measured and it was compared to to the execution time of the corresponding CUSPARSE library

function [16] on the same GPU. On these automatically generated cases, two experiments were carried out.

The objective of the first experiment was to measure how the execution time of the solution algorithm scales in block-diagonal matrices. In fact, in block-diagonal triangular matrices, if the size n_c of each block is less than $n_{\max}$, there are no external edegs in the graph. To carry out the experiment matrices with 16 diagonal blocks were generated, with block size n_c that ranged from 250 to 12000. The block-diagonal lower triangular matrices were generated such that the ratio

$$\delta := \frac{n_{\text{elements},c}}{n_c^2}$$

of the number of the non-zero elements in the blocks $n_{\text{elements},c}$ to the square of block size n_c, was 0.1 %. This ratio is called *block density* and is about the average measured on the test cases from the Florida matrix collection considered in Section 3.1.3 and Section 3.2.3. For each of the automatically generated matrices, the software was executed on an Nvidia K80 GPU with a peak double-precision performance of 2.91 TFLOPS [136].

Figure 3.6 summarises the results of the experiment. In this experiment, TASSL outperformed CUSPARSE in most test cases. In the smallest test case, the overhead of permuting the right-hand side vector caused TASSL to be slower than CUSPARSE. Also, the execution times of TASSL and CUSPARSE became closer as the block size increased. This is because TASSL partitioned the matrix in many sub-graph levels, which reduced the concurrency factor with respect to the smaller test cases. The maximum speedup achieved was 2.5.

The objective of the second experiment was to measure how the execution time of the solution algorithm scales with the number of diagonal blocks in the input matrix. Block-diagonal matrices were generated with blocks of size 6000, which is approximately the value of $n_{\max}$ for the Nvidia Quadro K4000 GPU. The number of diagonal blocks ranged from 8 to 512, while the block density was the same as the previous experiment. For each test case, the software

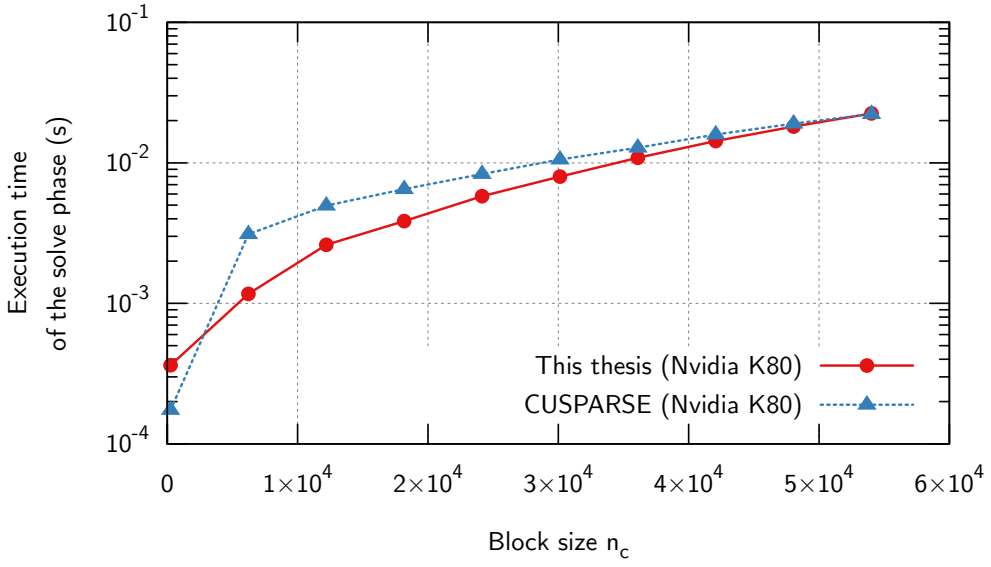


Figure 3.6: Execution time of the solve phase with randomly generated block-diagonal matrices, each having 16 diagonal blocks.

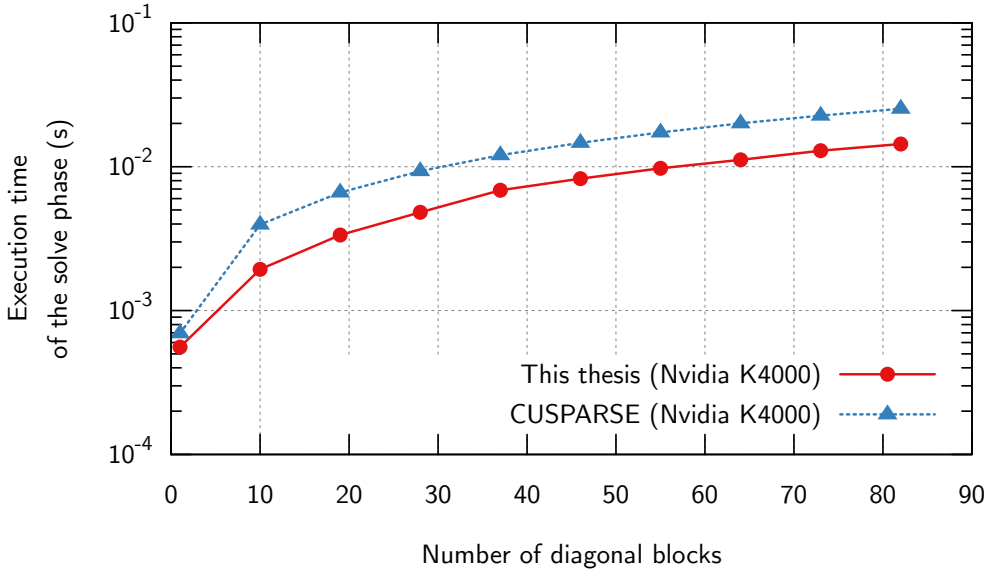


Figure 3.7: Execution time of the solve phase with randomly generated block-diagonal matrices, each having blocks of size 6000.

was executed on an Nvidia Quadro K4000 GPU with a peak double-precision performance of 1.27 TFLOPS [132].

Figure 3.7 shows that TASSL outperformed CUSPARSE in all cases, because TASSL always obtained one sub-graph level and a concurrency factor close to one. The maximum speedup achieved was 2.

Table 3.5: Experimental results on the test cases from the Florida matrix collection [133]. The mean values for the solve phase of CUSPARSE and TASSL are shown. In the table, C is the concurrency factor, η is the effiency factor, σ is the synchronisation overhead, and λ is the local transfer overhead. The other metrics are used by the Nvidia profiler *nvprof* [129].

Parameter	Mean value	
	CUSPARSE	TASSL
C	8.871×10^{-1}	9.801×10^{-1}
η	3.269×10^{-1}	4.408×10^{-1}
σ	1.880×10^{2}	4.782×10^{1}
λ		4.741×10^{1}
Thread divergence	20.4 %	0 %
Write L2 cache memory transactions	4.067×10^{4}	6.270×10^{3}
Write DRAM transactions	3.724×10^{4}	6.001×10^{3}
Read L2 cache memory transactions	8.869×10^{5}	4.282×10^{5}
Read DRAM transactions	4.750×10^{5}	1.750×10^{5}
Occupancy	28.0 %	50.0 %

3.3.2 *Test cases from the Florida matrix collection*

For each of the square matrices in the Florida matrix collection, only the lower triangular part was taken so that the resulting triangular factors had the same sparsity patterns as the lower half of the input matrix. Each of the obtained matrices was executed on the Nvidia Quadro K4000. Of these matrices, only those for which the execution time of the solve phase was known with a confidence interval smaller than 10 % were considered. Table 3.2 (page 72) summarises the features of the lower triangular factors that were used.

On the Nvidia Quadro K4000 GPU, the average execution time of the solve phase of TASSL was compared to the execution time of that of CUSPARSE, as shown in Figure 3.8, where each point corresponds to a test case. Points above the diagonal line correspond to test cases in which the solve phase of TASSL outperformed CUSPARSE's function, while points below the diagonal line correspond to test cases in which TASSL was outperformed. In 71 % of all test cases, the execution time of TASSL was less than that of CUSPARSE.

The experimental results summarised in Table 3.5, show that on average the solve phase of TASSL achieved a higher concurrency factor and a higher efficiency factor than CUSPARSE. Also, the synchronisation overhead for the solve phase of TASSL was less than that of CUSPARSE because of the use of local memory barriers instead of global memory barriers. However, the average time spent transferring data from and to the local memory during the solve phase of TASSL was about 47 times the time spent carrying out computations. The higher efficiency factor with TASSL compared to CUSPARSE was also investigated with the Nvidia profiler *nvprof* [129]. This tool showed that the solve phase of TASSL had no thread divergence, while during the solve phase of CUSPARSE, 20 % of branching instructions caused thread divergence.

Also, *nvprof* showed that the solve phase of TASSL achieved better data locality than that of CUSPARSE. In the nomenclature used by *nvprof*, each SIMD memory instruction causes one or more data transfers, called transactions, between memories. For example, if a SIMD instruction accesses a datum that cannot be found in the L2 cache memory, the memory sub-system counts one unsuccessful transaction to the L2 cache memory and one transaction to the DRAM memory. These memories and the complete memory sub-system of a GPU are discussed in Section 2.1.2 of Chapter 2 (page 26). Table 3.5 shows that during the solve phase of TASSL all the write transactions in the L2 cache memory caused a write transaction in the DRAM. However, the number of write transactions was about one order of magnitude less than that of CUSPARSE. Moreover, the read transactions to the L2 cache memory recorded during the solve phase of TASSL were about half of those recorded during the solve phase of CUSPARSE.

The result of these improvements in terms of thread divergence and data locality is summarised by the *occupancy* metric of *nvprof*. This metric is the ratio of the number of SIMD threads available for execution, which means not waiting for memory transfers, and the maximum number of SIMD threads that can be executed on a compute unit. The occupancy metric captures memory latency access hiding, and the value of TASSL was about two thirds higher than that of CUSPARSE.

Table 3.6: Comparison between three test cases in the Florida matrix collection [133]: *circuit_3* is the test case for which TASSL shows the largest slowdown compared to CUSPARSE, equal to 4.533, *human_gene2* is the test case for which TASSL shows the largest speedup compared to CUSPARSE, equal to 5.872, and *qpband* is the test case in which TASSL shows the largest speedup compared to CUSPARSE even if the value of the concurrency factor C for TASSL is lower than that for CUSPARSE. In the table, t is the execution time of the solve phase of CUSPARSE or TASSL, *CS* and *TS* mean CUSPARSE and TASSL respectively, and C, η, σ, and λ are the metrics defined in this chapter.

	circuit_3		*human_gene2*		*qpband*	
	CS	TS	CS	TS	CS	TS
t (s)	7.799×10^{-4}	3.535×10^{-3}	1.902×10^{-1}	3.239×10^{-2}	7.678×10^{-4}	5.762×10^{-4}
C	8.313×10^{-1}	8.565×10^{-1}	3.444×10^{-1}	9.979×10^{-1}	9.998×10^{-1}	9.996×10^{-1}
η	5.432×10^{-1}	4.479×10^{-1}	5.822×10^{-1}	5.002×10^{-1}	4.956×10^{-1}	5.399×10^{-1}
σ	1.144×10^{1}	5.048×10^{1}	2.273	8.630×10^{1}	5.720×10^{3}	1.755×10^{1}
λ		2.167×10^{1}		8.753×10^{1}		3.556

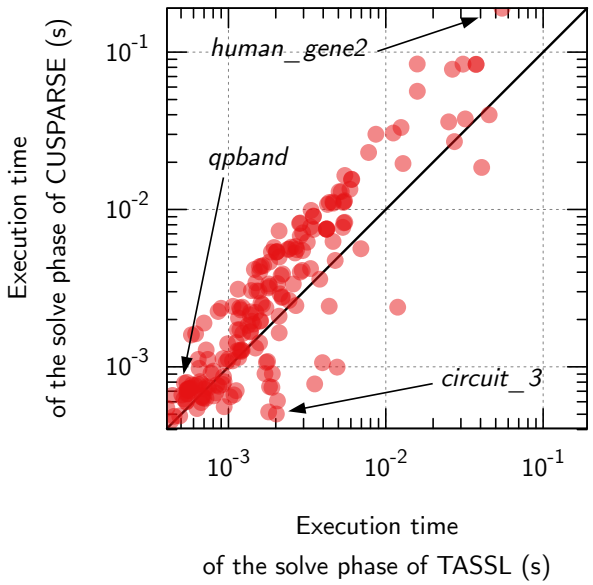


Figure 3.8: Comparison between the execution time of the solve phase of TASSL and that of CUSPARSE [16] on 200 test cases from the Florida matrix collection [133]. The points above the diagonal, which are 71 % of the total, represent test cases in which the solve phase of TASSL outperforms that of CUSPARSE. The software was executed on the Nvidia Quadro GPU.

Table 3.6 summarises the experimental results for three particular test cases. The worst slowdown achieved was 4.5 and the test case was named *circuit_3*. In this test case, the concurrency factor of TASSL was almost equal to that of CUSPARSE. However, the efficiency factor of CUSPARSE was higher than that of TASSL, because of bank conflicts when accessing local memory during the execution of TASSL. Also, the synchronisation overhead of TASSL was about 5 times higher than that of CUSPARSE. The best speedup achieved was of about 5.8, on a test case named *human_gene2*. For this test case, the concurrency factor was almost equal to one for TASSL and it was about one third for CUSPARSE. Hence, although the synchronisation overhead for TASSL was almost 40 times that of CUSPARSE and the local transfer overhead was of about 88 %, TASSL was faster. Finally, test case *qpband* is the test case among those for which $C_{\text{CUSPARSE}} \geq C_{\text{TASSL}}$ and the speedup was the highest, equal to about 1.3. For this test case, the synchronisation overhead of CUSPARSE was more than 300 times that of TASSL, which is because TASSL uses local memory barriers instead of global memory barriers.

In the final experiment, it was investigated how the architectural attributes of the GPU impact on the performance of the solve phase of TASSL. For this, two different GPUs were used: an Nvidia Quadro K4000 and an AMD Firepro W5000. The two GPUs chosen have about the same peak performance, which is 1.26 TFLOPS for the Nvidia GPU and 1.27 TFLOPS for the AMD GPU [132, 137]. However, while the AMD Firepro has a local memory capacity of 32 kB, the Nvidia Quadro has a local memory capacity of 48 kB. Also, as specifications of the two GPUs in Table 3.7 show, the compute units of the Nvidia Quadro GPU contain more than double the processing elements of the AMD GPU. On the other hand, the AMD Firepro GPU contains three times as many compute units as the Nvidia Quadro GPU.

Figure 3.9 illustrates the results of the experiment, where points above the diagonal line correspond to test cases in which the Nvidia GPU outperformed the AMD one, while points below the diagonal line correspond to test cases in which the Nvidia GPU was outperformed. In 74 % of all test cases, the execution time on the Nvidia GPU was less than that of the AMD GPU.

Table 3.7: Specifications of the GPUs used to carry out the experiments. *CU* stands for compute units, *PE* for processing elements, and *LM* for local memory size.

Platform	Clockrate	CU	PE/CU	LM
Nvidia Quadro K4000	0.82 GHz	4	192	48 kB
AMD Firepro W5000	0.83 GHz	12	64	32 kB

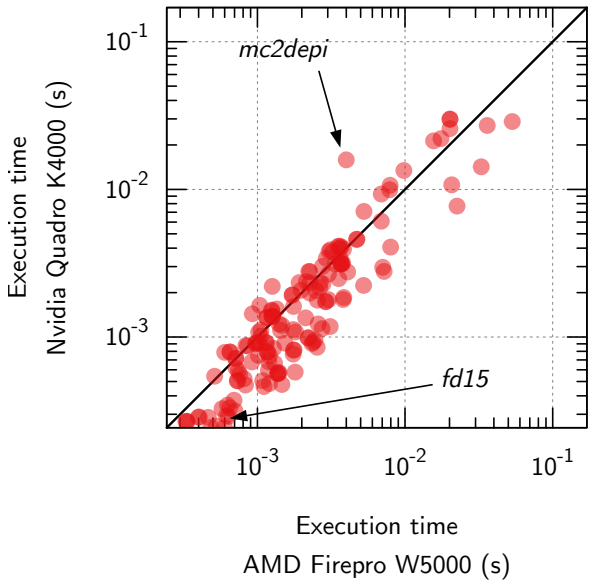


Figure 3.9: Comparison between the execution time of the solve phase of TASSL on the Nvidia Quadro K4000 and the AMD Firepro W5000 GPUs on 200 test cases from the Florida matrix collection [133]. The points below the diagonal, which are 74 % of the total, represent test cases in which the Nvidia GPU outperforms the AMD GPU.

In particular, the maximum speedup achieved with the Nvidia GPU against the AMD GPU was 4, on a test case named *fd15*. For this test case, because of larger local memory on the Nvidia GPU, sub-graphs had a larger number of vertices than on the AMD GPU. As a result, on the Nvidia GPU this test case required only one sub-graph level with a single sub-graph. Hence, all edges in the graph were internal edges on the Nvidia GPU. Instead, on the AMD GPU, the solve phase of TASSL required two sub-graph levels with one sub-graph each, which caused a reduction of performance compared to the Nvidia GPU. The maximum speedup achieved with the AMD GPU with respect to the Nvidia GPU was 3, on a test case named *mc2depi*. For both GPUs, the execution

of the solve phase of TASSL on *mc2depi* required one sub-graph level with 47 sub-graph in it. Because the AMD GPU has three times as many compute units as the Nvidia GPU, the AMD GPU carried out all computations three times faster.

3.4 CONCLUSIONS FROM THIS CHAPTER

When solving STLs, the structure of the input matrix has a significant impact on performance. As the experimental results show, the same algorithm carried out on the same data with two GPUs of comparable peak performance can have significantly different execution times. The algorithm illustrated in this chapter overcomes the limitations of conventional sparse linear algebra algorithms for GPUs by using local memory and taking advantage of an architecure-specific partitioning of the data.

The approach described in this chapter, called TASSL, was shown to be faster compared to the level-set approach implemented by the CUSPARSE library [17], which is used by the ViennaCL library [18], the Paralution library [19], the MAGMA library [20], the LAMA library [21], and the GHOST library [22]. In particular, TASSL takes advantage of the local memories inside the GPU to increase data locality and reduce memory access latency. Also, a theoretical framework was introduced in this chapter to describe sparse linear algebra algorithms. Using this framework, it was shown that accepting the overhead of copying data to the local memory can improve performance if the data is reused. In practice, this happens in most of practical test cases.

The next chapter discusses how to partition the data of the input matrix of a STL so that the solve phase, which was described in this chapter, can achieve high performance.

4

THE ANALYSIS PHASE

Given a STL

$$L\boldsymbol{x} = \boldsymbol{b} \tag{4.1}$$

where the vectors $\boldsymbol{x}$ and $\boldsymbol{b}$ are in $\mathbb{R}^n$ and L is a matrix in $\mathbb{R}^{n\times n}$, let $G \coloneqq (\mathcal{V}, \mathcal{E})$ be the directed acyclic graph associated with L. The aim of the analysis phase of both TASSL and CUSPARSE is to find a partition of G that minimises the execution time of the solve phase. The analysis phase of TASSL partitions the graph G into the sub-graphs G_1, G_2, ..., G_{n_s}, which are then organised in sub-graph levels $\mathcal{L}_1$, $\mathcal{L}_2$, ..., $\mathcal{L}_{n_{\text{levels}}}$. Inside each sub-graph, numerical operations between two synchronisation points belong to the same time slot.

In this thesis, as described by Section 3.2.1 of Chapter 3, the sub-graph $G_k \coloneqq (\mathcal{V}_k, \mathcal{E}_{\text{int},k} \cup \mathcal{E}_{\text{ext},k})$ of G is composed of the vertices in the subset $\mathcal{V}_k \subseteq \mathcal{V}$, and the union of the sets of internal and external edges for $\mathcal{V}_k$, which are defined by

$$\begin{aligned}\mathcal{E}_{\text{int},k} &\coloneqq \{(i,j) \in \mathcal{E} \mid i \in \mathcal{V}_k \wedge j \in \mathcal{V}_k\},\\ \mathcal{E}_{\text{ext},k} &\coloneqq \{(i,j) \in \mathcal{E} \mid i \in \mathcal{V}_k \wedge j \in \mathcal{V} \wedge j \notin \mathcal{V}_k\}.\end{aligned}$$

For TASSL, as discussed in Section 3.2.3 of Chapter 3 (page 79), the execution time of the solve phase is given by

$$t_{\text{solve}} \approx (1 - C) n_{\text{edges}} \frac{1}{T\eta}(1 + \sigma + \lambda) \tag{4.2}$$

where C is the concurrency factor, n_{edges} is the number of edges in the graph G, T is the maximum throughput of multiply-and-add SIMD instructions of the GPU, η is the efficiency factor, σ is the synchronisation overhead, and λ is the local transfer overhead. The analysis phase aims to minimise (4.2) by achieving a high value of the concurrency factor C, defined as

$$C \coloneqq 1 - \frac{\sum_{p=1}^{n_{\text{levels}}} \left(n_{\text{slots},\hat{p}} + n'_{\text{slots},\hat{p}} \right)}{n_{\text{edges}}} \tag{4.3}$$

where $\hat{p}$ is the index of the sub-graph in sub-graph level $\mathcal{L}_p$ with the longest execution time. Also, the aim of the analysis phase is to achieve a low value of the syncrhonisation overhead σ, defined as

$$\sigma := \eta T \left[\sigma_{\text{local}} + \frac{\kappa n_{\text{levels}}}{(1 - C) n_{\text{edges}}} \right] \tag{4.4}$$

where σ_{local} and κ are the execution times of two synchronisation mechanisms used in GPUs, both discussed in Section 2.1.2 of Chapter 2 (page 26). The first is the execution time of a *local memory barrier* and is in the order of magnitude of 10^{-7} s, while the second is the time required to start and stop the execution of a GPU kernel and is in the order of magnitude of 10^{-5} s. A discussion on all these metrics can be found in this thesis in Section 3.2.3 of Chapter 3.

While maximising (4.3) and minimising (4.4), the analysis phase guarantees the satisfaction of two conditions:

CONDITION I: if an edge goes from a vertex in $\mathcal{V}_k$ to a vertex in $\mathcal{V}_p$, then $p \geq k$.

CONDITION II: the number of vertices in the set $\mathcal{V}_k$, denoted n_k, must not be greater than a positive integer $n_{\max}$.

If both these conditions are met, then the partition is called a *feasible partition*. After the partitioning, the analysis phase schedules the numerical operations to be executed during the solve phase.

The analysis phase of TASSL is illustrated in Figure 4.1. The analysis phase is divided in two stages: the *partitioning stage*, which is discussed in Section 4.1 and the *scheduling stage*, which is discussed in Section 4.2. The execution time of the two phases depends on the structure of the graph G. This is discussed in Section 4.3, which illustrates the experimental results.

While the analysis phase of CUSPARSE is executed on the GPU, the two steps of the analysis phase of TASSL are executed on a multi-core CPU. This is because the performance of GPU software is affected more than that of CPU software by branching instructions, as `if-else` and `case` statements, of which the algorithms described in this chapter are rich. This issue, called *thread divergence*, is discussed in Section 2.1.2 of Chapter 2 (page 26).

The research contribution discussed in this chapter are the following:

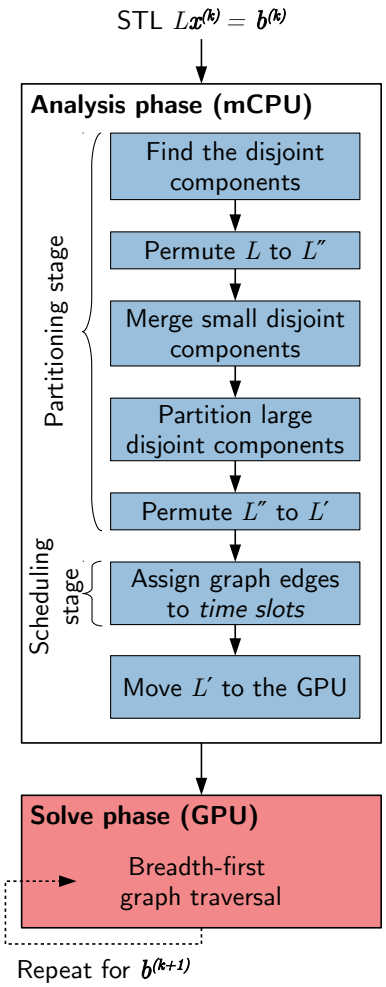


Figure 4.1: Overview of the analysis phase of TASSL. In the figure, *mCPU* stands for multi-core CPU. Internally, TASSL's analysis phase creates two permuted versions of the STL $L\boldsymbol{x}^{(k)} = \boldsymbol{b}^{(k)}$: $L''\boldsymbol{x}''^{(k)} = \boldsymbol{b}''^{(k)}$ and $L'\boldsymbol{x}'^{(k)} = \boldsymbol{b}'^{(k)}$.

- An algorithm that partitions the vertices of a directed acyclic graph into sets of a given size, which is dependent on the local memory of a GPU [23].
- An algorithm that schedules the numerical operations required to solve a STL according to the partition of the directed acyclic graph [23].

4.1 PARTITIONING STAGE

The partitioning stage partitions the graph $G := (\mathcal{V}, \mathcal{E})$ associated with the input matrix L in (4.1) into n_s sub-graphs. The value of n_s is determined by the

size of the GPU's local memory, a small but fast memory inside each compute unit of the GPU which can be used to reduce the latency of memory accesses. The number of sub-graphs n_s is defined as follows:

$$n_s := \left\lceil \frac{n}{n_{\max}} \right\rceil \tag{4.5}$$

where n is the number of vertices in the graph G, and $n_{\max}$ is given by the size of the local memory of the GPU, which is denoted with M_{local}. If the data is stored using the double precision floating point format, it is

$$n_{\max} := \left\lfloor \frac{M_{\text{local}}}{8} \right\rfloor. \tag{4.6}$$

The aim of the partitioning stage is to obtain a small value of n_{levels} in (4.3) and (4.4). In fact, a small value of n_{levels} means many sub-graphs in each sub-graph level. On one hand, this increases the concurrency factor C. On the other hand, a small value of n_{levels} can mitigate the synchronisation overhead σ defined in (4.4). In fact, assuming that the number of time slots is approximately the same in each sub-graph level, from (4.3) one has

$$C := 1 - \frac{\sum_{p=1}^{n_{\text{levels}}} \left(n_{\text{slots},\hat{p}} + n'_{\text{slots},\hat{p}} \right)}{n_{\text{edges}}} \approx 1 - \frac{n_{\text{levels}} M}{n_{\text{edges}}} \tag{4.7}$$

where $M = n_{\text{slots},\hat{p}} + n'_{\text{slots},\hat{p}}$ is constant for $p = 1, 2, \ldots, n_{\text{levels}}$. Substituting (4.7) in (4.4) and in (4.2) it is

$$t_{\text{solve}} \approx n_{\text{levels}} \left[M \left(\frac{1}{T\eta} + \sigma_{\text{local}} \right) + \kappa \right]. \tag{4.8}$$

Hence, in a first order approximation, the execution time of the solve phase can be considered linearly dependent on the number of levels. Also, in the ideal case that the efficiency factor η is close to one and the total number of time slots per sub-graph M is very small, it is $t_{\text{solve}} \approx n_{\text{levels}} \kappa$, which means the execution time of the solve phase is dominated by the time required to launch n_{levels} GPU kernels.

Because the number of sub-graphs n_s is fixed, the technique used by TASSL to reduce n_{levels} tries to put as many sub-graphs as possible in the first set $\mathcal{L}_1$, while making sure that the partition is feasible when the algorithm terminates.

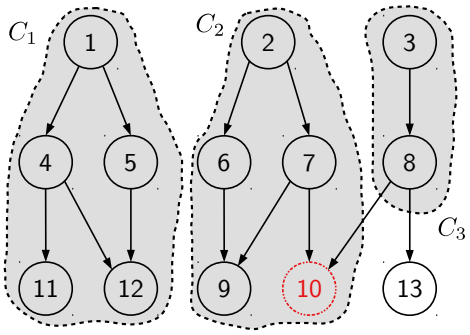


Figure 4.2: Execution of Tarjan's algorithm for finding the disjoint components of a DAG. When vertex 10 is reached by the algorithm from vertex 8, since vertex 10 was already assigned to disjoint component C_2, both vertex 8 and vertex 3 are also re-assigned to disjoint component C_2.

The partitioning stage is divided into three *steps*: partitioning into disjoint components, merging small disjoint components, and partitioning large disjoint components.

4.1.1 *Partitioning into disjoint components*

The first step of the partitioning stage is to partition the graph G into its n_{comp} disjoint components, which are the parts of the graph that are not connected to any other. If G can be partitioned into its disjoint components and the partition is feasible, each of the components can be considered a sub-graph. As a result n_{levels} is one, which is the smallest possible value.

The partitioning into disjoint compoenents can be done using Tarjan's algorithm [138], which has complexity $O(n + n_{\text{edges}})$, where n is the number of vertices and n_{edges} is the number of edges in the graph G.

TASSL implements a form of Tarjan's algorithm for directed graphs. First, the implementation finds the *root vertices* of the graph, which are those vertices that do not have entering edges. Each of these vertices is assigned a *label* and a depth-first traversal of the graph G is executed starting from it. All the edges found during the traversal that do not have a label are marked with the label of the current root vertex.

If during the traversal a vertex is found with a different label, the current root vertex belongs to the same disjoint component as another one. This is shown in Figure 4.2. All the vertices found previously and marked with the label of the current root vertex are marked with the new label, then the traversal continues starting from those vertices that still have unexplored exiting edges.

At the end of this partitioning step, there can be disjoint components that have more than $n_{\max}$ vertices, which means the partition at this stage is not feasible, or a number of disjoint components that have very few vertices. The latter case is addressed in the following step of the partitioning stage.

4.1.2 *Merging small disjoint components*

The second step of the partitioning stage is to merge together disjoint components with less than $n_{\max}$ vertices using a heuristic algorithm.

This step is necessary because each sub-graph is processed during the solve phase using one compute unit of the GPU, as explained in Chapter 3. Inside each compute unit of the GPU, the computation is divided in concurrent parts called work items and the set of the work items executed by one compute unit is called a work group. Groups of 32 work items are merged into a single SIMD thread by the GPU compiler. If on a compute unit there are enough SIMD threads ready for executing numerical operations while other SIMD threads are waiting for memory operations, the GPU software can achieve high performance. As discussed in Section 2.1.2 of Chapter 2, this is called *memory access latency hiding*.

If the graph G is composed of many disjoint components with few vertices, all the compute units in the GPU are used but the amount of numerical operations per each compute unit is small. Therefore, memory access latency hiding is poor and, in (4.2), this is captured by a small value of the efficiency factor η. The merging step aims to mitigate this effect.

First in the merging step, all disjoint components are sorted with respect to their number of vertices, from the smallest to the largest. Then, starting from the smallest, disjoint components are merged together, such that the number of

vertices of each merged component is never greater than $n_{\max}$. A lookup table can be used to keep track of the assignment of vertices to merged components.

Using this lookup table, the matrix L in (4.1) is then permuted into a block diagonal form by constructing a permutation P. This can be done as follows: the vertex with the lowest index i that does not have neither entering nor exiting edges is permuted to index 1, the second one with the lowest index that does not have neither entering nor exiting edges is permuted to index 2, and so on until all the n_0 vertices that do not have neither entering nor exiting edges are considered. Then, the vertex in $\mathcal{V}_1$ with the lowest index is permuted to index $n_0 + 1$, and the process is repeated until all the vertices in the graph G associated with the matrix L have been considered. This keeps the lower triangular structure in L'', because each entry in L $l_{i,j}$ with $j > i$ is permuted to an entry $l''_{i'',j''}$ of L'' with $j'' > i''$. Using P, the matrix L is permuted to a block-diagonal matrix:

$$L'' = P^{\intercal} L P = \begin{bmatrix} \boxed{L''_0} & & & & \\ & \boxed{L''_1} & & 0 & \\ & & \boxed{L''_2} & & \\ & & & \boxed{L''_3} & \\ & 0 & & & \ddots \\ & & & & \boxed{L''_{n_{\text{merged}}}} \end{bmatrix} \tag{4.9}$$

where L''_k with $k = 0, 1, \ldots, n_{\text{merged}}$ are lower triangular matrix blocks. Each block is associated with one of the resulting merged components of the graph G. In particular, L''_0 is a diagonal block, and it is associated with the vertices of the graph G that have neither entering nor exiting edges.

At the end of this step of the partitioning stage, all conditions for the feasibility of the partition are met except possibly condition II (page 96), which gives an upper bound on the number of vertices in each sub-graph. Hence, further processing of the components is needed.

Because of the block diagonal structure of L'', the processing of each resulting component is independent from the others, therefore the third step of the partitioning stage can be carried out concurrently on a multi-core CPU.

```
1: function ASSOCIATE_SUBG(v, p, n_max, V_1, V_2, ..., V_{n_split})
2:     k ← p
3:     while n_k ≥ n_max
4:         if k < n_split
5:             k ← k + 1
6:         else return error
7:     V_k ← V_k ∪ {v}
8:     return success
```

Algorithm 4.1: Function that associates a vertex v with a sub-graph. The function looks for a sub-graph whose index is between p and n_{split}, and that contains fewer than $n_{\max}$ vertices. If there is no such sub-graph, the function terminates with an error.

4.1.3 *Partitioning large disjoint components*

The third step of the partitioning stage is a heuristic that partitions those disjoint components that have a number of vertices greater than $n_{\max}$ into sub-graphs. This is done to obtain a feasible partition, such that the local memories on the GPU can be used during the solve phase. If n_C is the number of vertices in the disjoint component $C := (\mathcal{W}, \mathcal{Y})$, this disjoint component is to be split into

$$n_{\text{split}} := \left\lceil \frac{n_C}{n_{\max}} \right\rceil$$

sub-graphs $G_1 := (\mathcal{V}_1, \mathcal{E}_1)$, $G_2 := (\mathcal{V}_2, \mathcal{E}_2)$, ..., $G_{n_{\text{split}}} := (\mathcal{V}_{n_{\text{split}}}, \mathcal{E}_{n_{\text{split}}})$.

The basic operation of the partitioning heuristic is to put a given vertex v in a subset $\mathcal{V}_k$ associated with sub-graph G_k by making sure that the partition is kept feasible. Hence, as illustrated in Algorithm 4.1, the vertex v is put in the first subset $\mathcal{V}_k$ with fewer than $n_{\max}$ vertices.

The partitioning heuristic is illustrated in Algorithm 4.2. The heuristic uses two containers of vertices: V_{next} and V_{cur}, on which it is possible to execute three basic operations, which are the addition of an element (*put*), the sorting according to a criterion κ (*sort*), and counting the number of elements (*numel*).

If (i, j) is an edge that goes from vertex j to vertex i, vertex j is called a *parent* of vertex i and that i is called a *child* of vertex j. The set of the parents of a vertex j is denoted with $parents(j)$ in Algorithm 4.2, and the set of its children

Input: $C \coloneqq (\mathcal{W}, \mathcal{Y})$, n_{split}, $n_{\max}$, κ
Output: $\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_{n_{\text{split}}}$

1: $\mathcal{V}_k \leftarrow \{\emptyset\} \quad \forall k = 1, 2, \ldots, n_{\text{split}}$
2: $V_{\text{next}} \leftarrow$ empty
3: $V_{\text{next}} \leftarrow \text{put}(r, V_{\text{next}}) \quad \forall r \in \mathcal{W} \mid \text{parents}(r) = \{\emptyset\} \wedge \text{children}(r) \neq \{\emptyset\}$
4: $n_{\text{init}} \leftarrow \min\{n_{\text{split}}, \text{numel}(V_{\text{next}})\}$
5: **if** $n_{\text{init}} > 0$
6: $\quad V_{\text{next}} \leftarrow \text{sort}(V_{\text{next}}, \kappa)$
7: $\quad p \leftarrow 0$
8: $\quad$ **for all** $k \in V_{\text{next}}$
9: $\quad\quad$ **if** $p > n_{\text{init}}$ **then** $p \leftarrow 1$, **else** $p \leftarrow p + 1$
10: $\quad\quad e \leftarrow$ ASSOCIATE_SUBG$(k, p, n_{\max}, \mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_{n_{\text{split}}})$
11: $\quad\quad$ **if** $e =$ error
12: $\quad\quad\quad n_{\text{init}} \leftarrow \lfloor n_{\text{init}}/2 \rfloor$ **and go to line** 5
13: $\quad\quad$ **else for all** $v \in \text{children}(k)$
14: $\quad\quad\quad$ **if** $\nexists j \in \text{parents}(v) \mid j \notin \mathcal{V}_g \quad \forall g = 1, 2, \ldots, n_{\text{split}}$
15: $\quad\quad\quad\quad V_{\text{next}} \leftarrow \text{put}(v, V_{\text{next}})$
16: $\quad$ **while** $\text{numel}(V_{\text{next}}) \neq 0$
17: $\quad\quad V_{\text{cur}} \leftarrow \text{sort}(V_{\text{next}}, \kappa)$
18: $\quad\quad V_{\text{next}} \leftarrow$ empty
19: $\quad\quad$ **for all** $k \in V_{\text{cur}}$
20: $\quad\quad\quad p \leftarrow \max\{q \in \mathbb{N} \mid \exists j \in \text{parents}(k) \wedge j \in \mathcal{V}_q\}$
21: $\quad\quad\quad e \leftarrow$ ASSOCIATE_SUBG$(k, p, n_{\max}, \mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_{n_{\text{split}}})$
22: $\quad\quad\quad$ **if** $e =$ error
23: $\quad\quad\quad\quad n_{\text{init}} \leftarrow \lfloor n_{\text{init}}/2 \rfloor$ **and go to line** 5
24: $\quad\quad\quad$ **else for all** $v \in \text{children}(k)$
25: $\quad\quad\quad\quad$ **if** $\nexists j \in \text{parents}(v) \mid j \notin \mathcal{V}_g \quad \forall g = 1, 2, \ldots, n_{\text{split}}$
26: $\quad\quad\quad\quad\quad V_{\text{next}} \leftarrow \text{put}(v, V_{\text{next}})$
27: $\quad$ **if** $\exists v \in \mathcal{W} \mid v \notin \mathcal{V}_k \quad \forall k = 1, 2, \ldots, n_{\text{split}}$
28: $\quad\quad n_{\text{init}} \leftarrow \lfloor n_{\text{init}}/2 \rfloor$ **and go to line** 5
29: $\quad$ **else terminate with** success
30: **else terminate with** error

Algorithm 4.2: Heuristic algorithm that partitions a graph component $C \coloneqq (\mathcal{W}, \mathcal{Y})$, into n_{split} sub-graphs $G_1 \coloneqq (\mathcal{V}_1, \mathcal{E}_1)$, $G_2 \coloneqq (\mathcal{V}_2, \mathcal{E}_2)$, $\ldots$, $G_{n_{\text{split}}} \coloneqq (\mathcal{V}_{n_{\text{split}}}, \mathcal{E}_{n_{\text{split}}})$. Each sub-graph has at most $n_{\max}$ vertices. The algorithm uses the containers of vertices V_{cur} and V_{next}, on which the *put*, *sort* and *numel* operations are defined. In the pseudocode, κ is the sorting criterion, and $\text{parents}(v)$ and $\text{children}(v)$ denote the set of parents and children of vertex v respectively.

is denoted with $children(j)$. In line 3 of Algorithm 4.2, V_{next} is initialised with the root vertices of the component to partition, which are those vertices that do not have any parents but do have children.

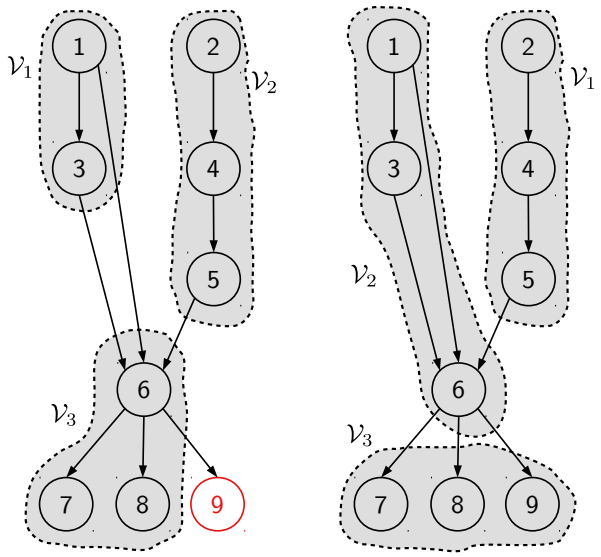
Initially, the root vertices are distributed across n_{init} initial sub-graphs. As shown in line 4, if the number of sub-graphs n_{split} is less than the number

of root vertices, then n_{init} is chosen equal to n_{split}. If instead n_{split} is greater than the number of root vertices, the aim is to maximise the sub-graphs in the first sub-graph level by having one root vertex per initial sub-graph. The root vertices in V_{next} are then sorted according to a sorting criterion κ.

At each iteration of the partitioning heuristic, in line 19, a vertex k is taken from the container V_{cur}, and it is associated with the sub-graph with the highest possible index. This is described in line 21. If the *associate_subg* routine is succesful, the children of the vertex k are considered, and those whose parents are all assigned to sub-graphs are put in the container V_{next}. When all the vertices in V_{cur} have been processed, the container V_{next} is sorted and its content are moved to V_{cur}. Then, as shown in line 18, V_{next} is emptied before the following iteration.

The partitioning heuristic is executed with different criteria κ until a feasible partition is found. Initially, the criterion is *descending order of out-degree*, which means that the edges with the most children are also those in the sub-graphs with the smallest indices. For example consider the case in which there are two root vertices. The root vertex with the highest out-degree, denoted with v_1, is put in the first subset $\mathcal{V}_1$ associated with sub-graph G_1, while the other root vertex, denoted with v_2, is put in the second subset $\mathcal{V}_2$ associated with sub-graph G_2. With this assignment of vertices to subsets, the children of v_1, which are more than those of v_2, can be assigned to more subsets.

If a feasible partition is not found immediately, it is because the execution of the *associate_subg* routine was terminated with an error, which can occur either in line 10 or line 21. When an error occurs, n_{init} is halved and the execution of the heuristic is started from the beginning. This is because it is assumed that the failure of the heuristic was caused by a too optimistic value of n_{init}. If n_{init} becomes equal to zero, then the whole heuristic terminates with an error. Whenever this happens, the execution of the heuristic is repeated and κ is changed: from *descending order of out-degree* to *ascending order of out-degree*. The latter criterion is more suitable to graphs that have a long an narrow shape like that shown in Figure 4.3, because sorting vertices in descending order of out-degree causes some vertices to be left out of the partitioning, while sorting



(a) *Descending order of out-degree.*

(b) *Ascending order of out-degree.*

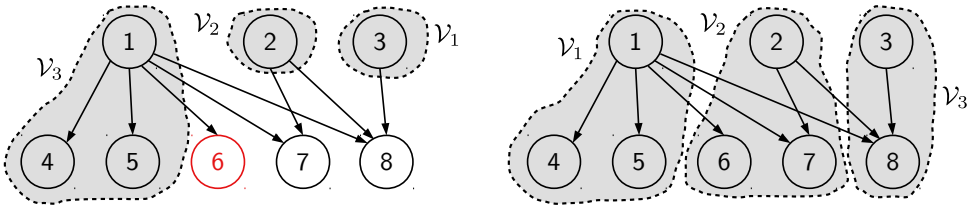
Figure 4.3: Comparison of two criteria for sorting vertices on long and narrow graphs when $n_{\max}$ is 3. When the graph is long and narrow, sorting vertices in descending order of out degree causes the partitioning heuristic in Algorithm 4.2 to fail because at the final steps of the algorithm some vertices cannot be put in any subset $\mathcal{V}_k$ On the contrary, if vertices are sorted in ascending order of out-degree, a feasible partition is obtained. The vertex in red is that at which the partitioning algorithn gives the error.

vertices in ascending order of out-degree does not. For the same reason, sorting vertices in descending order of out-degree is more suitable to short and wide graphs like that in Figure 4.4.

Finally, if also this criterion does not achieve a feasible partition, κ is changed to *ascending order of vertex index*, which puts the vertices with the lowest index in the subsets with the lowest index. Consequently, finding a feasible solution that does not violate condition I (page 96) becomes more likely.

The worst-case computational complexity of each iteration of the partitioning algorithm is $O(n + n_{\text{edges}})$, where n is the total number of vertices in the graph and n_{edges} is the number of edges.

At the end of the partitioning stage a permutation Q is obtained, so that input matrix L in (4.1) can be permuted to the block form



(a) *Ascending order of out-degree.* (b) *Descending order of out-degree.*

Figure 4.4: Comparison of two criteria for sorting vertices on short and wide graphs when $n_{\max}$ is 3. When the graph is short and wide, sorting vertices in ascending order of out degree causes the partitioning heuristic in Algorithm 4.2 to fail because at the final steps of the algorithm some vertices cannot be put in any subset $\mathcal{V}_k$ On the contrary, if vertices are sorted in descending order of out-degree, a feasible partition is obtained. The vertex in red is that at which the partitioning algorithn gives the error.

$$\underbrace{\begin{bmatrix} I_0 & & & & \\ & I_1 & & 0 & \\ & E_2 & I_2 & & \\ 0 & & E_3 & I_3 & \\ & \vdots & & \ddots & \\ & & E_{n_s} & & I_{n_s} \end{bmatrix}}_{L'} \underbrace{\begin{bmatrix} \boldsymbol{x'_0} \\ \boldsymbol{x'_1} \\ \boldsymbol{x'_2} \\ \boldsymbol{x'_3} \\ \vdots \\ \boldsymbol{x'_{n_s}} \end{bmatrix}}_{x'} = \underbrace{\begin{bmatrix} \boldsymbol{b'_0} \\ \boldsymbol{b'_1} \\ \boldsymbol{b'_2} \\ \boldsymbol{b'_3} \\ \vdots \\ \boldsymbol{b'_{n_s}} \end{bmatrix}}_{b'} \tag{4.10}$$

where $L' := Q^\intercal L Q$, $\boldsymbol{x'} := Q^\intercal \boldsymbol{x}$, and $\boldsymbol{b'} := Q^T \boldsymbol{b}$. The form in 4.10 is used by the solve phase of TASSL, as discussed in Section 3.2.1 of Chapter 3. Since a a number of vertices smaller or equal to $n_{\max}$ is associated with each sub-graph G_k because of condition II (page 96), also the number of rows in the blocks I_k and E_k is less or equal than $n_{\max}$.

At the end of the partitioning the sequence of the equations in (4.10) is known, together with the sub-graph levels $\mathcal{L}_p$. In fact, one can build a graph where each vertex represents a sub-graph and each edge represents the presence of one or more edges (i, j) where i and j belong to two different sub-graphs. These edges are called *external*. An example of such graph is shown in Figure 4.5. In this figure, the edge between sub-graph G_1 and sub-graph G_3 shows that there is at least one edge between vertices associated with sub-

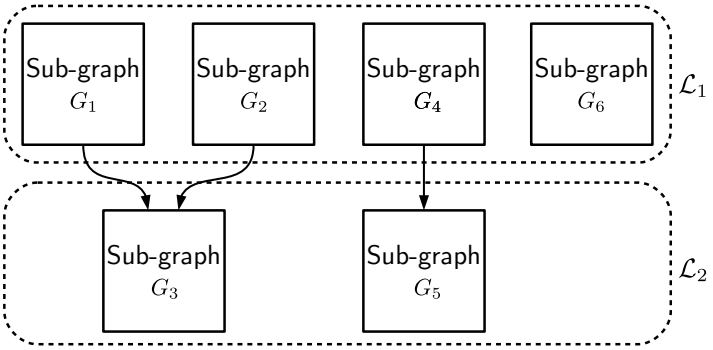


Figure 4.5: Example graph representing the relationship between sub-graphs. Each vertex in this graph represents a sub-graph and each edge represents an external edge in the original G graph after the partitioning. Sub-graphs are into two sets: $\mathcal{L}_1$ and $\mathcal{L}_2$. There is no path between sub-graphs in each of the sets, because this graph is always a directed acyclic graph if the partition is feasible.

graph G_1 and vertices associated with sub-graph G_3. Also, sub-graphs 1, 2, 4 and 6 belong to the set $\mathcal{L}_1$ because they do not depend on any other sub-graph. Similarly, sub-graphs 3 and 5 belong to the set $\mathcal{L}_2$ because they are connected to sub-graphs in the set $\mathcal{L}_1$ and they do not depend on each other, therefore in this example $n_{\text{levels}} = 2$. Since condition I for feasibility (page 96) is always satisfied if the partitioning is successful, this type of graph is always a directed acyclic graph.

Although the partitioning stage always obtains a feasible partition if L is lower triangular, it is not known whether the obtained partition is optimal, which means if the obtained partition minimises (4.2). This depends only on the partition of large components obtained with Algorithm 4.2. The technique to find the optimal partition consists in posing an equivalent mixed-integer nonlinear problem which can be solved by a solver like Scip [139]. Since finding such a partition is a *NP-complete* problem [62], it can require many hours of computation even for a small matrix. In turn, this makes evaluating the optimality of partitions obtained with Algorithm 4.2 impractical for any test case of interest. Nonetheless, in general it can be observed that partitions obtained with Algorithm 4.2 are farthest from the optimal if the graph associated with L has much fewer root vertices than *leaf vertices*, which are vertices that have

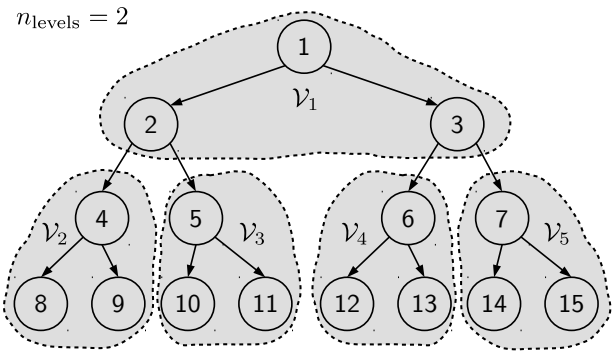
parents but not children. This is shown in Figure 4.6, where the optimal partition has two sub-graph levels and the partition obtained with Algorithm 4.2 has four sub-graph levels. This is because Algorithm 4.2 does not use any assumption on leaf vertices and only puts vertices in sub-graphs depending on an initial assignment of the root vertices. Hence, for graphs like that shown in Figure 4.6, the partition can be far from optimal and, by consequence, the solution of the STL on the GPU can be less performant than with the optimal partition.

In the worst case, it is $n_{\text{levels}} = n_s$ in the obtained partition and $n_{\text{levels}} = 2$ in the optimal partition. Using (4.8) and assuming M and η are the same for both the obtained partition and the optimal partition, the slowdown in the worst case is equal to $\frac{n_s}{2}$. For example, considering $n_{\max} = 6000$ like in most of currently available GPUs, a STL with a matrix of 60 000 rows and 60 000 columns would be solved approximately 5 times faster with the optimal partition than with the one obtained during the solve phase in the worst case.

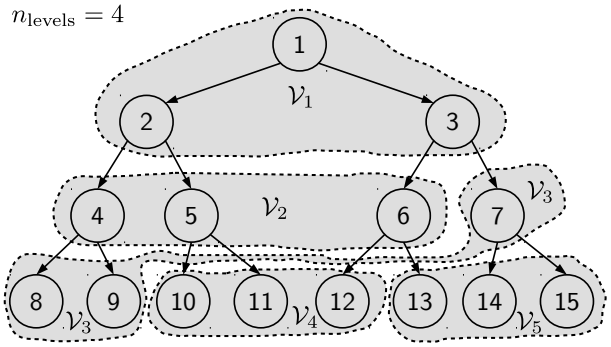
The partitioning stage gives coarse-grained information about data locality. However, it does not give any indication about the order of numerical operations within each block in (4.10). This information is captured by the concept of time slot, that is a set of numerical operations between two synchronisation points. In TASSL, each numerical operation in a time slot is associated with an edge. Given a sub-grah G_g, the q-th time slot is denoted with $\mathcal{T}'_{g,q}$ if it contains external edges. If the edges connect vertices in the same sub-graph they are called *internal* and the time slot is denoted with $\mathcal{T}_{g,q}$. The organisation of edges into time slots is obtained by the scheduling stage.

4.2 SCHEDULING STAGE

During the scheduling stage, each edge in the sub-graphs G_1, G_2, ..., G_s is mapped to a time slot. Edges with the same time slot represent multiply-and-add operations that can be executed at the same time, with the SIMD paradigm. However, the computation represented by each sub-graph requires further analysis in order to minimize conflicting memory accesses. For ex-



(a) *Optimal partition.*



(b) *Partition obtained with Algorithm 4.2*

Figure 4.6: Comparison between the optimal partition and one obtained with Algorithm 4.2 assuming $n_{\max} = 3$. In the figures, the leaf vertices are represented at the bottom, while the root vertices are represented at the top. In a graph with leaf vertices than vertices at the top, the heuristic is less able to achieve a low number of sub-graph levels than in the opposite case because it does not use any assumption on the leaf vertices. In the figure, subset $\mathcal{V}_5$ is in sub-graph level $\mathcal{L}_4$, because it depends on subset $\mathcal{V}_3$ ($\mathcal{L}_3$), which depends on subset $\mathcal{V}_2$ ($\mathcal{L}_2$). The only subset in $\mathcal{L}_1$ is $\mathcal{V}_1$.

ample, consider the situation represented in the left half of Figure 4.7. The variable x_6 is obtained by the execution of two operations in any order, but if the two numerical operations are carried out at the same time, a read-write access conflict occurs. This is because each operation reads the value of the variable x_6 and overwrites it with a new value. In general, this issue occurs whenever two or more edges are directed towards the same vertex.

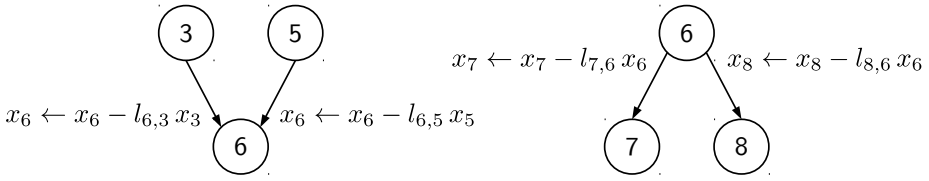


Figure 4.7: Examples of the two edge patterns in a sub-graph. On one hand, if two or more edges are directed towards a vertex, the numerical operations associated with them are in read-write conflict. On the other hand, if two or more edges are exiting a vertex, the numerical operations associated with them can be carried out in parallel with the SIMD paradigm.

1: **function** FIX_CONFLICTS
   $(v,\, G_k \coloneqq (\mathcal{V}_k, \mathcal{E}_{\text{int},k} \cup \mathcal{E}_{\text{ext},k}),\, \mathcal{T}_{k,1},\, \mathcal{T}_{k,2},\, \ldots,\, \mathcal{T}_{k,n_{\text{slots},k}},\, \mathcal{T}'_{k,1},\, \mathcal{T}'_{k,2},\, \ldots,\, \mathcal{T}'_{k,n'_{\text{slots},k}})$
2: **for all** $j \in \text{parents}(v) \mid j \in \mathcal{V}_k$
3: **if** $\exists q \in \text{parents}(v) \mid q \neq j \wedge (v,q) \in \mathcal{T}_{k,p} \wedge (v,j) \in \mathcal{T}_{k,p}$
4: $t \leftarrow \min\{h \in \mathbb{N} \mid h > p,\, \nexists u \in \text{parents}(v) \in \mathcal{T}_{k,h}$
5: $\mathcal{T}_{k,t} \leftarrow \mathcal{T}_{k,t} \cup \{(v,j)\}$
6: **for all** $j \in \text{parents}(v) \mid j \notin \mathcal{V}_k$
7: **if** $\exists q \in \text{parents}(v) \mid q \neq j \wedge (q,v) \in \mathcal{T}'_{k,p} \wedge (v,j) \in \mathcal{T}'_{k,p}$
8: $t \leftarrow \min\{h \in \mathbb{N} \mid h > p,\, \nexists u \in \text{parents}(v) \in \mathcal{T}'_{k,h}$
9: $\mathcal{T}'_{kt} \leftarrow \mathcal{T}'_{k,t} \cup \{(v,j)\}$
10: **return** success

Algorithm 4.3: Function that checks and solves possible read-write conflicts in numerical operations. In the pseudocode, v is a vertex in sub-graph $G_k = (\mathcal{V}_k, \mathcal{E}_{\text{int},k} \cup \mathcal{E}_{\text{ext},k})$, so $v \in \mathcal{V}_k$. Also, parents(v) and children(v) denote the set of parents and children of vertex v respectively, $\mathcal{T}_{k,p}$ and $\mathcal{T}'_{k,p}$ denote the p-th internal and external time slot of sub-graph G_k, respectively. The function always terminates with success.

The proposed solution to this problem consists of delaying one or more of the operations and assigning the edges to different time slots. This is carried out with the routine shown in Algorithm 4.3. There is no read-write access conflict in the situation shown in the right half of Figure 4.7: in this case, the operations overwrite the value of two different variables. Since the two operations can be done at the same time, this is also an opportunity for concurrency.

Input: $G_k := (\mathcal{V}_k, \mathcal{E}_k)$
Output: $\mathcal{T}_{k,1}, \mathcal{T}_{k,2}, \ldots, \mathcal{T}_{k,n_{\text{slots},k}}, \mathcal{T}'_{k,1}, \mathcal{T}'_{k,2}, \ldots, \mathcal{T}'_{k,n'_{\text{slots},k}}$

$V_{\text{cur}} \leftarrow$ empty
$V_{\text{cur}} \leftarrow \text{put}(r, V_{\text{next}}) \quad \forall r \in \mathcal{V}_k \mid \text{parents}(v) = \{\emptyset\} \wedge \text{children}(v) \neq \{\emptyset\}$
$\mathcal{T}_{k,p} \leftarrow \{\emptyset\} \quad \forall p$
$\mathcal{T}'_{k,p} \leftarrow \{\emptyset\} \quad \forall p$
for all $v \in V_{\text{cur}}$
    for all $j \in \text{parents}(v) \mid j \notin \mathcal{V}_k$
        $\mathcal{T}'_{k,1} \leftarrow \mathcal{T}'_{k,1} \cup \{(v, j)\}$
    FIX_CONFLICTS$(v, G_k, \mathcal{T}_{k,1}, \mathcal{T}_{k,2}, \ldots, \mathcal{T}_{k,n_{\text{slots},k}}, \mathcal{T}'_{k,1}, \mathcal{T}'_{k,2}, \ldots, \mathcal{T}'_{k,n'_{\text{slots},k}})$
    for all $i \in \text{children}(v) \mid i \in \mathcal{V}_k$
        $\mathcal{T}_{k,1} \leftarrow \mathcal{T}_{k,1} \cup \{(i, v)\}$
        if $\nexists j \in \text{parents}(i) \mid (i, j) \notin \mathcal{T}_{k,p} \wedge (j, i) \notin \mathcal{T}'_{k,p} \quad \forall p$
            $V_{\text{next}} \leftarrow \text{put}(i, V_{\text{next}})$
while $\text{numel}(V_{\text{next}}) \neq 0$
    $V_{\text{cur}} \leftarrow V_{\text{next}}$
    $V_{\text{next}} \leftarrow$ empty
    for all $v \in V_{\text{cur}}$
        FIX_CONFLICTS$(v, G_k, \mathcal{T}_{k,1}, \mathcal{T}_{k,2}, \ldots, \mathcal{T}_{k,n_{\text{slots},k}}, \mathcal{T}'_{k,1}, \mathcal{T}'_{k,2}, \ldots, \mathcal{T}'_{k,n'_{\text{slots},k}})$
        $t \leftarrow 1 + \max\{q \in \mathbb{N} \mid \exists j \in \text{parents}(v) \wedge (v, j) \in \mathcal{T}_{k,q}\}$
        for all $i \in \text{children}(v) \mid i \in \mathcal{V}_k$
            $\mathcal{T}_{k,t} \leftarrow \mathcal{T}_{k,t} \cup \{(i, v)\}$
            if $\nexists j \in \text{parents}(i) \mid (i, j) \notin \mathcal{T}_{k,p} \wedge (j, i) \notin \mathcal{T}'_{k,p} \quad \forall p$
                $V_{\text{next}} \leftarrow \text{put}(i, V_{\text{next}})$
terminate with success

Algorithm 4.4: Algorithm that assigns the edges of a sub-graph G_p to time slots while making sure no read-write conflict occurs. The algorithm uses the containers of vertices V_{cur} and V_{next}, on which the *put* and *numel* operations are defined. In the pseudocode, $\text{parents}(v)$ and $\text{children}(v)$ denote the set of parents and children of vertex v respectively, while $\mathcal{T}_{k,p}$ and $\mathcal{T}'_{k,p}$ are the p-th time slot of internal and external edges respectively, for sub-graph G_k.

4.2.1 *Algorithm of the scheduling stage*

The proposed scheduling algorithm, which is illustrated in Algorithm 4.4, assigns edges to time slots while taking advantage of any opportunity for concurrency. Also, it avoids any read-write access conflict by delaying operations if necessary by using the routine in Algorithm 4.3. Since each vertex is associated with only one sub-graph during the partitioning stage, sub-graphs can be processed in parallel.

Algorithm 4.4 assigns each edge to the first available time slot. Given a vertex j, the conflicts in its entering edges are first solved with the function *fix_conflicts*. Then, if $\mathcal{T}_{k,q}$ is the last time slot to which one of the entering edges was assigned, then the exiting edges are assigned to $\mathcal{T}_{k,q+1}$. This is repeated for all the vertices in the graph whose entering edges have all been assigned to time slots, until all the vertices in the graph have been processed. The algorithm is initialised with those vertices that belong to sub-graph G_k and do not have any parent vertex that belongs to the same sub-graph.

The worst-case computational complexity of the scheduling algorithm corresponds to the case in which every edge is reassigned to a time slot by the function *fix_conflicts*. This gives $O\big(2n_{\text{edges}}\big)$, where n_{edges} is the number of edges in the graph G associated with the input matrix L of the STL.

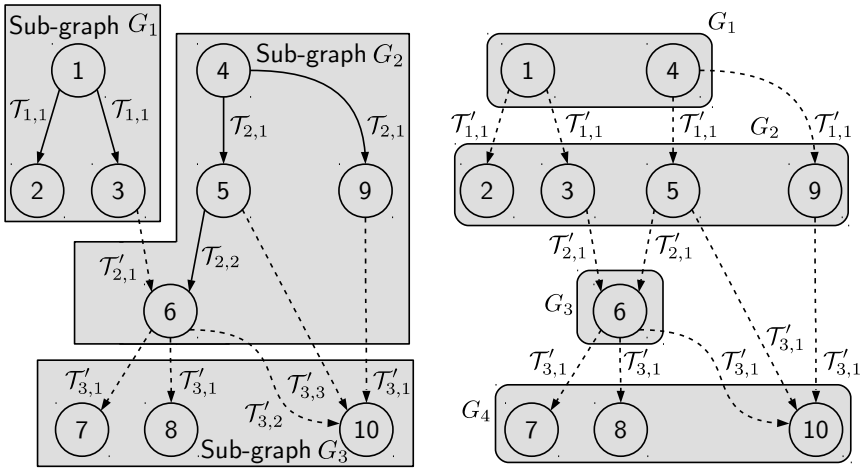
4.2.2 *Comparison with the analysis phase of CUSPARSE*

At the end of the scheduling stage, all information is available to carry out the execution of the solve phase, and each vertex of the graph is associated with a sub-graph. During the solve phase of TASSL, each sub-graph is processed by a compute unit and all the edges in each time slot execute a numerical operation of the type

$$x'_i \leftarrow x'_i - l'_{i,j} x'_j$$

where $l'_{i,j}$ is the entry in row i and column j of the matrix L' in (4.10).

A preliminary analysis phase before solving a STL is also required by the CUSPARSE library. This library is the most popular sparse linear algebra library for GPUs and it is used within other sparse linear algebra libraries such as the ViennaCL library [18], the Paralution library [19], the MAGMA library [20], the LAMA library [21], and the GHOST library [22]. Chapter 3 discussed the shortcomings of CUSPARSE and showed that TASSL achieves better performance in most practical test cases. In particular, this is because numerical operations associated with internal time slots during the solve phase of TASSL mostly access local memory inside the GPU, which has a low access



(a) *TASSL.* (b) *CUSPARSE.*

Figure 4.8: Comparison between the result of the analysis phase on an example graph, with $n_{\max} = 4$, and the analysis phase as carried out by the algorithm of CUSPARSE [16]. For each edge the corresponding time slot is shown and external edges are denoted with dashed lines. The analysis phase of TASSL resulted in six external edges and five internal edges, while the analysis phase of CUSPARSE results only in external edges. Moreover, the analysis phase of CUSPARSE resulted in four sub-graph organised in four sub-graph levels, while the analysis phase of TASSL resulted in three sub-graphs organised in three sub-graph levels.

latency. Instead, most numerical operations CUSPARSE only access global memory locations, which have higher access latency. A comparison between the analysis phase of CUSPARSE and that of TASSL is shown in Figure 4.8, where it can be seen that the latter also results in fewer sub-graph levels.

However, with the analysis phase of TASSL, the maximum number of operations that can be carried out concurrently in one time slot is $n_{\max}$. Instead, with CUSPARSE, the maximum number of operations that can be carried out in one time slot is equal to the number of vertices in the graph G, denoted with n, which is usually greater than $n_{\max}$. For example, consider the graph represented in Figure 4.9. The analysis phase of CUSPARSE assigns all the edges in the graph to the same time slot, thus the concurrency factor C is equal to one. Also the *per-thread concurrency* W, which is defined in Section 3.1.2 of Chap-

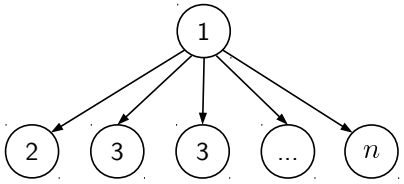


Figure 4.9: Structure of the graph that can achieve the highest degree of parallelism. The analysis phase of CUSPARSE [16] assigns all the edges in the graph to the same time slot and obtains only one sub-graph level. Instead, the analysis of TASSL partitions the graph into $n/n_{\max}$ sub-graphs that are organised on two sub-graph levels.

ter 3 (page 71), is equal to one. This metric captures how many work items can be used to execute the solve phase of both TASSL and CUSPARSE. The solve phase of TASSL partitions the graph in Figure 4.9 into $n/n_{\max}$ sub-graphs, such that the first sub-graph includes vertex 1 and is in the first sub-graph levels, while all the other sub-graphs are in the second sub-graph level. Considering that the maximum number of work items per compute unit n_w is usually 1024, TASSL also requires more than one time slot per sub-graph. As a result, the value of W of TASSL is much lower than that of CUSPARSE. When n is 10^5, for example, W is 1.638×10^{-1}.

Summarising, although TASSL usually gives better data locality than CUSPARSE, the maximum per-thread concurrency it can achieve is lower. A more detailed comparison of the two approaches is given in the following section.

4.3 EXPERIMENTAL RESULTS

The experiments carried out on the analysis phase were of two types: those that compare the analysis phase of TASSL to that of CUSPARSE are described in Section 4.3.1, while those that analyse the execution time of the analysis phase of TASSL are described in Section 4.3.2.

4.3.1 *Comparison with the analysis phase of CUSPARSE*

Some matrices were generated with the aim of studying the performance of the analysis phase of TASSL in function of specific graph attributes. For each of these matrices, the sofware was executed on an an Nvidia K80 GPU with a peak double-precision performance of 2.91 TFLOPS [136]. The objective of the first experiment was to measure how the execution time of the analysis algorithm scales with the number of vertices in the disjoint components of a graph. To achieve this, the matrices were generated with 16 diagonal blocks, each one associated with a disjoint component, with block size that ranged from 250 to 55000. The block density

$$\delta := \frac{n_{\text{elements},c}}{n_c^2}$$

which is the ratio of non-zero elements in each block $n_{\text{elements},c}$ to the square of block size n_c, was of 0.1 %. This is also the average density of the test cases from the Florida matrix collection [133] that were considered in Chapter 3 to evaluate the solve phase of TASSL.

The analysis phase was carried out only once per data point on an Intel Xeon Processor E5-2640 with 20 MB of L3 cache [140]. The algorithms for the analysis were implemented in C++ and parallelised over a maximum of 6 CPU threads. Figure 4.10 shows that the execution times of both the analysis phase of CUSPARSE and the that of TASSL grow linearly with the number of non-zero elements in the matrix. However, it was observed that the slope was 166 times steeper in the case of TASSL. Because of this, when using TASSL in practice, the solve phase needs to be repeated before the execution time of the analysis phase is amortised. The number of repetitions increases with the size of disjoint components, as illustrated in Figure 4.11.

Another experiment was executed on a set of test cases from the Florida matrix collection [133]. The analysis phase of CUSPARSE was executed on the Nvidia Quadro K80 GPU [136], while that of TASSL was executed on the Intel Xeon E5 CPU [140]. Similarly to the experiments carried out for the solve phase on Chapter 3, only the lower triangular part of square matrices in this

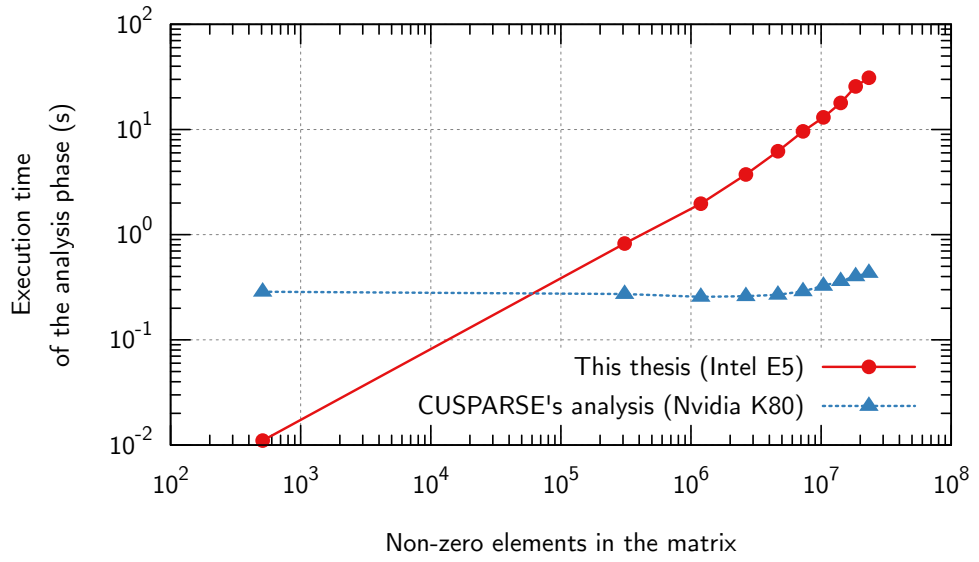


Figure 4.10: Execution time of the analysis phase with randomly generated block-diagonal matrices, each having 16 disjoint components, as a function of the number of non-zero elements in the matrix.

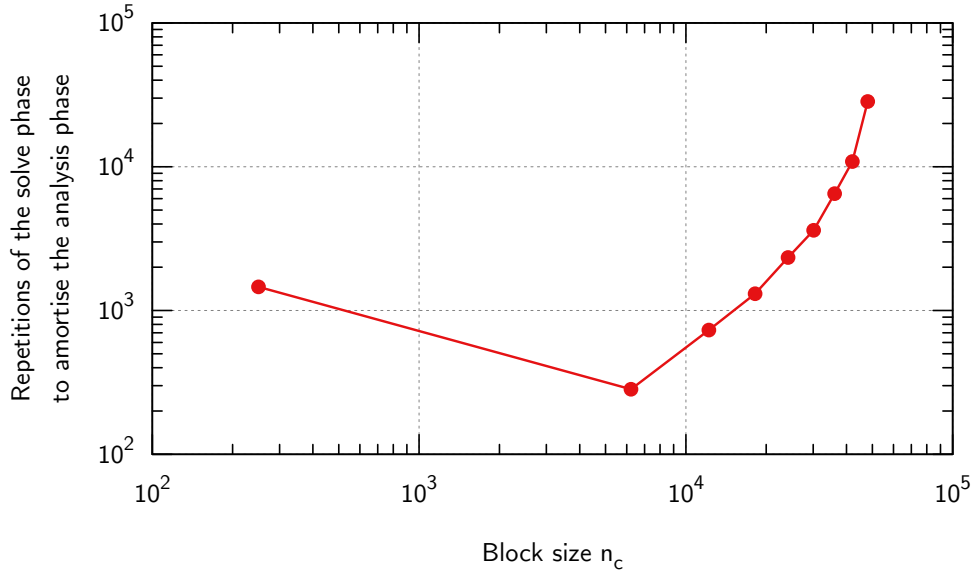


Figure 4.11: Number of times the solve phase should be repeated before the execution time of the analysis phase is amortised, and the total execution time is less than that of CUSPARSE. Each of the test cases has 16 disjoint components of the given size.

set of benchmarks was considered, so that the resulting triangular matrices had the same sparsity patterns as the lower half of the original ones. Of all the test cases examined, those for which the standard deviation of the measured

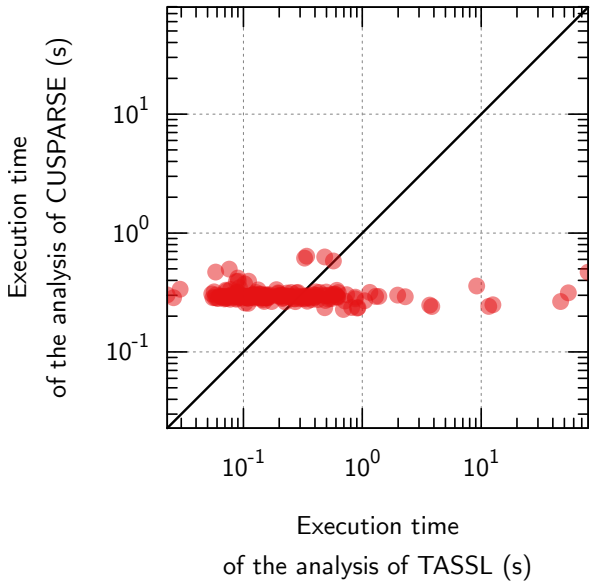


Figure 4.12: Comparison between the execution time of the analysis phase of TASSL and that of CUSPARSE [16] on 200 test cases from the Florida matrix collection [133]. The points above the diagonal, which are 70 % of the total, represent test cases in which the analysis phase of TASSL outperforms that of CUSPARSE. The analysis phase of TASSL was executed on the Intel E5 CPU and that of CUSPARSE on the Nvidia K80 GPU.

execution time was more than 10 % of the mean value were discarded, to filter out the noisiest experimental data. Of the remaining test cases, the first 200 starting from that with the fewest rows and columns were considered.

The results of this experiment are shown in Figure 4.12. Of all the test cases examined, the execution of the analysis phase of TASSL on the CPU was faster than that of CUSPARSE on the GPU in 70 % of the cases. Although one could say that a comparison between the analysis phase of CUSPARSE and TASSL is not fair because they are executed on two different computer architectures, this experiment showed that TASSL can be used for real applications and achieve better performance than CUSPARSE in practice.

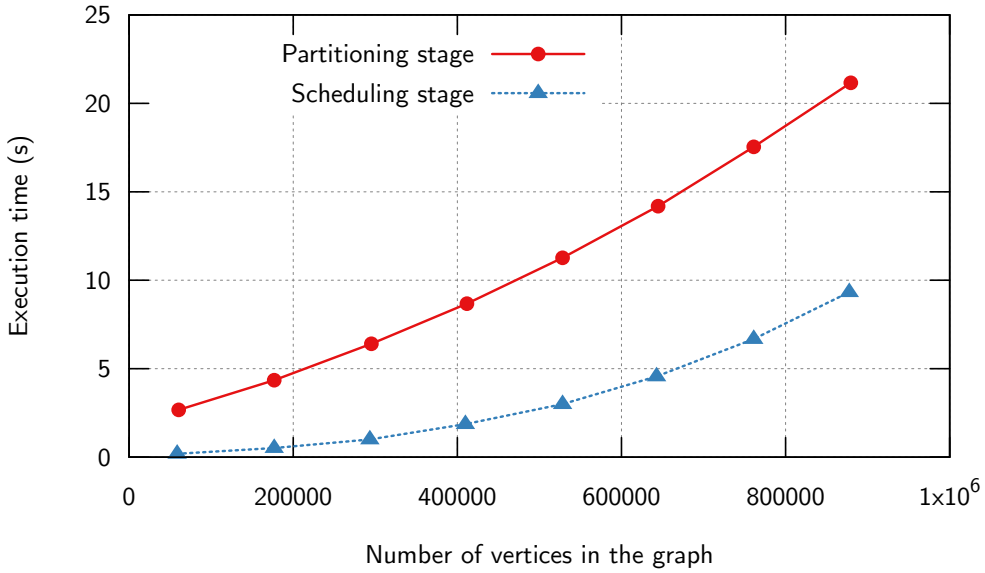


Figure 4.13: Execution time of the partitioning and scheduling stages over randomly generated block-diagonal matrices, each having 16 diagonal blocks.

4.3.2 *Execution time of the partitioning stage and the scheduling stage*

The execution time of the partitioning stage and the scheduling stage of TASSL were measured when the analysis phase of TASSL was executed on the block-diagonal matrices in the first experiment, which all have 16 blocks whose size ranged from 250 to 55000. The CPU used was an Intel Xeon E5 processor [140]. As shown in Figure 4.13, the execution time of the analysis phase was dominated by the partitioning step. In fact, as the size of the component increased, so did the number of times the partitioning heuristic was to be repeated to achieve a feasible partition. This is illustrated in Figure 4.14.

4.4 CONCLUSIONS FROM THIS CHAPTER

The algorithms discussed in this chapter partition the graph associated with the matrix of the STL in a way that marries the sparsity pattern of the matrix to the architecture of the GPU. Depending on the matrix, the analysis phase of TASSL can be more time consuming than that of CUSPARSE, but in most test cases from standard benchmarks it was shown to be faster. The analysis phases

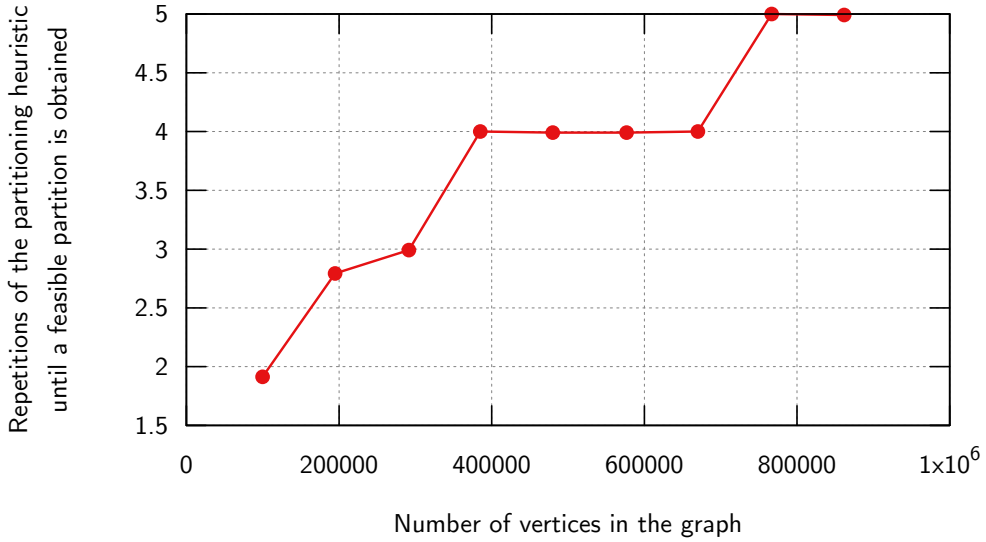


Figure 4.14: Number of repetitions required to achieve the feasible partition of a graph component. For components of the same size, the geometric mean was taken. Results obtained over randomly generated block-diagonal matrices, each having 16 diagonal blocks.

of both CUSPARSE and TASSL can be time consuming, but their benefits are evident in the reduction of the time of the solve phase, which was discussed in Chapter 3. Also, if the solve phase is executed many times with the same matrix, the cost of the analysis can be amortised.

Unlike the partitioning carried out by CUSPARSE's analysis phase, that described in this chapter does not simply aim to maximise concurrency in the execution of the solve phase. Instead, it aims to improve data locality by assigning vertices to sub-graphs that share the GPU's local memory, and to decrease the overhead of synchronisation. The next chapter compares these two design philosophies for parallel sparse linear algebra algorithms. This is done by using CUSPARSE and TASSL as building blocks for a more complex algorithm: the preconditioned conjugate gradient method (PCG).

5

EVALUATION ON PRECONDITIONED ITERATIVE METHODS

Large scale linear systems are systems of linear equations with many rows and columns, typically a hundred thousand or more. For this class of systems, *iterative methods* [14], are usually convenient because they do not change the structure of the matrix and they do not increase the number of non-zero elements [141]. However, iterative methods require the definition of a stopping criterion, usually related to the desired quality of the approximation of the solution. The better the quality of the approximation, the larger the number of iterations.

To reduce the number of iterations, some iterative methods use an approximation of the matrix of the linear system that is easier to invert and is called a *preconditioner*. The preconditioned conjugate gradient method (PCG) [14, ch. 9] and the preconditioned generalised minimal residuals method (PGMRES) [14, ch. 9] belong in this category.

This chapter describes the application of the *time-slot sub-graph level* (TASSL) algorithms to preconditioned iterative methods. While only the PCG is used as an example the same principles also hold for the PGMRES and other algorithms.

The scientific contribution of this chapter is in showing that the TASSL algorithm can substitute CUSPARSE to accelerate preconditioned iterative methods. This is the application domain where CUSPARSE is used by the ViennaCL library [18], the Paralution library [19], the MAGMA library [20], the LAMA library [21], and the GHOST library [22].

Section 5.1 summarises some known results on the PCG method and its convergence when the matrix is sparse. In Section 5.2 two implementations of the PCG method are described: one that uses CUSPARSE and one that uses TASSL. Section 5.3 discusses the experimental results for both artificial test cases and standard benchmarks. In both cases, the TASSL implementation is shown to

be faster than the CUSPARSE implementation. Section 5.4 is a short survey of available GPU implementations of preconditioned iterative methods, which shows that major sparse linear algebra libraries use the CUSPARSE implementation and would benefit from using TASSL instead.

5.1 THE PRECONDITIONED CONJUGATE GRADIENT METHOD

Apart from the experiment described in Section 5.1.2 (page 125), the contents of this section are background information that is needed understand the experimental work illustrated in the other sections.

The PCG method is a modified version of the conjugate gradient method (CG) [14, ch. 6], which is known to converge to the solution of a linear system of equations in the form

$$A\boldsymbol{x} = \boldsymbol{b} \tag{5.1}$$

in n iterations, where n is the number of rows and columns of the matrix $A \in \mathbb{R}^{n \times n}$. In (5.1), A is assumed to be symmetric and positive definite.

5.1.1 *Algorithm of the PCG method*

Both the CG and the PCG methods aim to minimise a quadratic form. For CG, this form is

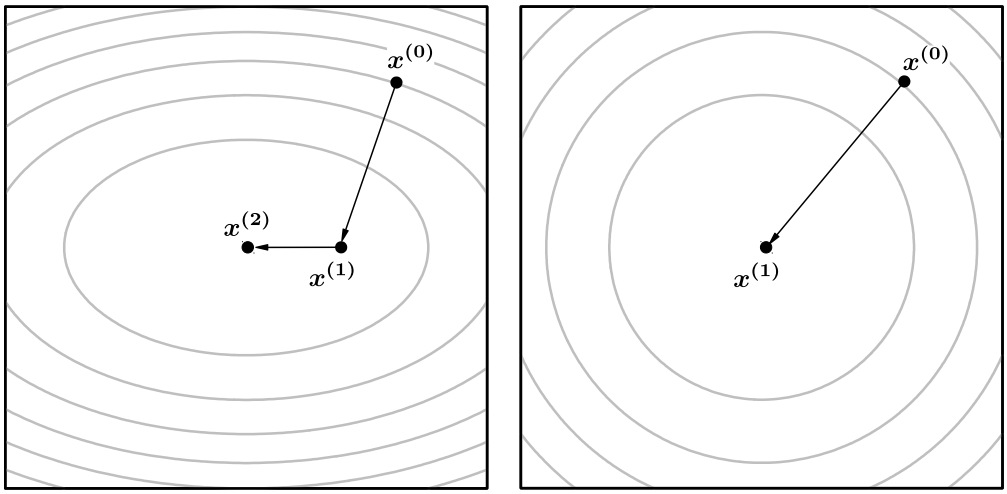
$$\phi(\boldsymbol{x}) := \boldsymbol{x}^{\mathsf{T}} A\boldsymbol{x} - \boldsymbol{x}^{\mathsf{T}}\boldsymbol{b}. \tag{5.2}$$

The minimum of $\phi(\boldsymbol{x})$ is the null vector, which is obtained when (5.1) is satisfied.

At iteration $k+1$ of the CG method, the vector $\boldsymbol{x}^{(k+1)}$ is given by

$$\boldsymbol{x}^{(k+1)} \leftarrow \boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{p}^{(k)},$$

where the vector $\boldsymbol{x}^{(k)} \in \mathbb{R}^n$ is the approximation of the solution of (5.1) at iteration k. The vector $\boldsymbol{p}^{(k)}$ is *conjugate* to the vector $\boldsymbol{p}^{(k-1)}$, which means that



(a) *CG method.* (b) *PCG method.*

Figure 5.1: Operation of the CG and PCG methods. In both cases, the approximation of the solution at iteration $k+1$, that is $\boldsymbol{x}^{(k+1)} \in \mathbb{R}^n$, is obtained from that at step k by moving along the direction $\boldsymbol{p}^{(k)}$, which is conjugate with respect to $\boldsymbol{p}^{(k-1)}$, $\boldsymbol{p}^{(k-2)}$, ..., $\boldsymbol{p}^{(0)}$. In the PCG method, the level surfaces of the quadratic form resemble more to spheres, so convergence is faster.

$$\left(\boldsymbol{p}^{(k)}\right)^{\mathsf{T}} A\boldsymbol{p}^{(k)} = 0.$$

The direction $p^{(0)}$ is orthogonal to the level-surface of the quadratic form ϕ at $\boldsymbol{x}^{(0)}$, as shown in Figure 5.1. The parameter $\alpha^{(k)}$ is chosen to minimize ϕ along $\boldsymbol{p}^{(k)}$.

The PCG method works exactly like the CG method, as shown in Figure 5.1b, but it substitutes ϕ with

$$\phi'(\boldsymbol{x}) := \boldsymbol{x}^{\mathsf{T}} M^{-1} A\boldsymbol{x} - \boldsymbol{x}^{\mathsf{T}} M^{-1}\boldsymbol{b}, \tag{5.3}$$

where M is the preconditioner, an invertible matrix defined in $\mathbb{R}^{n\times n}$. As a result of the preconditioning, the level-surfaces of the quadratic form ϕ' are more similar to spheres than those of ϕ, which reduces the number of iterations required for convergence [14, 142]. In the ideal case, $M = A$, so that level-surfaces are perfect spheres.

Input: A, b, M, $k_{\max}$, τ, $\boldsymbol{x^{(0)}}$
Output: $\boldsymbol{x}$
1: $r^{(0)} \leftarrow \boldsymbol{b} - A\boldsymbol{x^{(0)}}$
2: $z^{(0)} \leftarrow M^{-1}\boldsymbol{r^{(0)}}$
3: $p^{(0)} \leftarrow \boldsymbol{z^{(0)}}$
4: $k \leftarrow 0$
5: **while** $k < k_{\max} \wedge \|\boldsymbol{r^{(k)}}\|_\infty > \|\boldsymbol{b}\|_\infty \tau$
6: $\quad \boldsymbol{s} \leftarrow A\boldsymbol{p^{(k)}}$
7: $\quad \delta \leftarrow \left(\boldsymbol{p^{(k)}}\right)^\intercal \boldsymbol{s}$
8: $\quad \alpha^{(k)} \leftarrow \left(\boldsymbol{p^{(k)}}\right)^\intercal \boldsymbol{r^{(k)}} \;/\; \delta$
9: $\quad \boldsymbol{x^{(k+1)}} \leftarrow \boldsymbol{x^{(k)}} + \alpha^{(k)}\boldsymbol{p^{(k)}}$
10: $\quad \boldsymbol{r^{(k+1)}} \leftarrow \boldsymbol{r^{(k)}} - \alpha^{(k)}\boldsymbol{s}$
11: $\quad \boldsymbol{z^{(k+1)}} \leftarrow M^{-1}\boldsymbol{r^{(k+1)}}$
12: $\quad \beta^{(k)} \leftarrow \boldsymbol{s}^\intercal \boldsymbol{z^{(k+1)}} \;/\; \delta$
13: $\quad \boldsymbol{p^{(k+1)}} \leftarrow \boldsymbol{z^{(k+1)}} - \beta^{(k)}\boldsymbol{p^{(k)}}$
14: $\quad k \leftarrow k + 1$

Algorithm 5.1: The PCG method. In this algorithm, A is a positive definite matrix in $\mathbb{R}^{n\times n}$, b is a vector in $\mathbb{R}^n$, and M is the preconditioner, also defined in $\mathbb{R}^n$. The parameters $k_{\max}$ and τ are respectively the maximum number of iterations and the tolerance. Line is the termination condition and it can be changed depending on the problem. The condition shown in the pseudocode is that implemented in GNU Octave's *pcg* function [143].

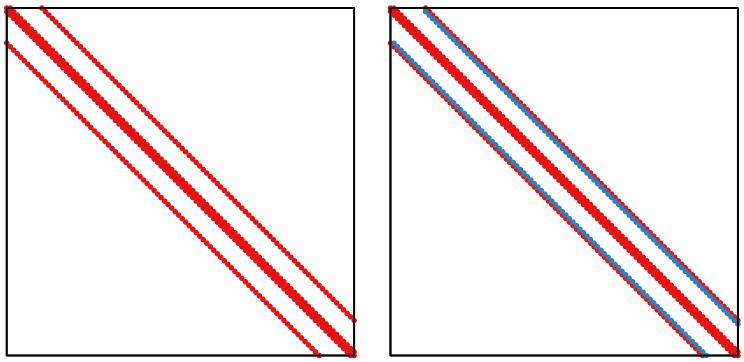
As shown in Algorithm 5.1, the PCG method requires the solution of a system of linear equations (lines 2 and 11) at each iteration. The matrix of this system of equations is the preconditioner M, which is often known in a factorised form of the type

$$M := M_L M_R. \tag{5.4}$$

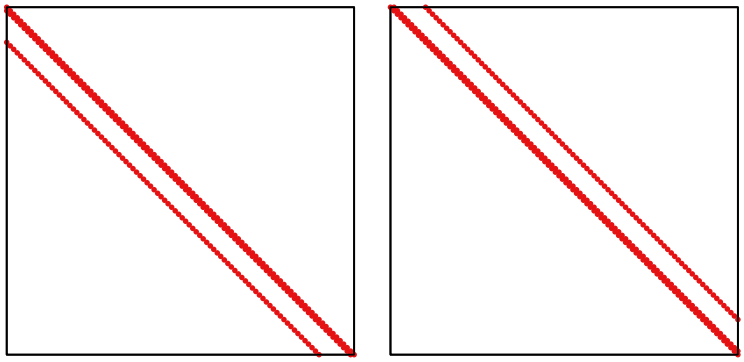
The two factors are obtained either by incomplete LU factorisation (ILU) or incomplete Cholesky factorisation [14, ch. 10] of the system matrix A. Both families of algorithms aim to compute triangular factors such that

$$M \approx A,$$

without M_L and M_R having many more non-zero entries than A. The approximation affects the convergence rate of the PCG method, depending on the sparsity pattern of A.



(a) *System matrix A.* (b) *Preconditioner M.*

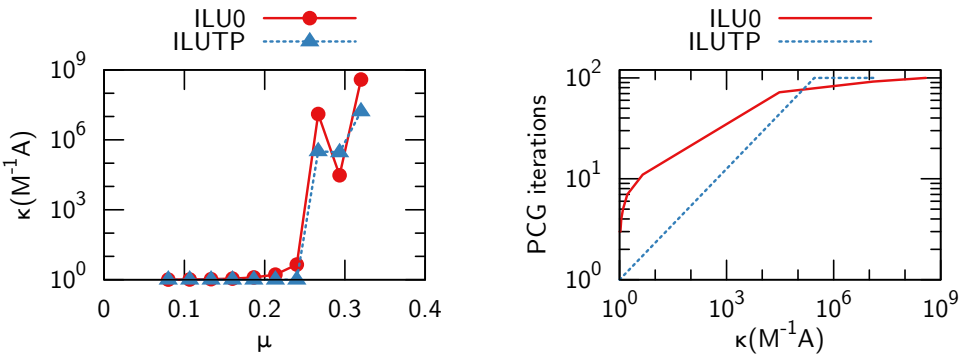


(c) *Lower triangular factor M_L.* (d) *Upper triangular factor M_U.*

Figure 5.2: ILU0 of a matrix with a tri-diagonal sparsity pattern. Although the lower and upper triangular factor have the same sparsity pattern as the lower and upper part of the matrix A respectively, their product $M := M_L M_U$ has a different sparsity pattern than that of A. In the figure, the difference is highlighted in a different colour.

5.1.2 *Preconditioners and convergence*

Suppose the matrix A in (5.1) is a sparse matrix with the sparsity pattern shown in Figure 5.2a. This type of sparsity pattern is called *tri-diagonal* and is common in the field of finite element methods [14, ch. 2]. The incomplete LU factorisation *with zero fill-in* (ILU0) [14, ch. 10] gives triangular factors with the same sparsity pattern as that of the lower and the upper part of A, as shown in Figure 5.2c and Figure 5.2d. However, the product of these factors is not guaranteed to have the same sparsity pattern as A and can have more non-zero entries, called *fill-in*.



(a) *Quality of the preconditioning.* (b) *Number of iterations of the PCG method.*

Figure 5.3: Comparison of different incomplete LU factorisation algorithms carried out on tri-diagonal matrices with bandwidth 10 and size 100. The real parameter μ is the value of the non-zero elements in the secondary diagonals of the matrix. When μ increases, the preconditioning becomes poorer and the number of iterations of the PCG method increases.

Other types of incomplete LU factorisation, such as the incomplete LU factorisation *with threshold and pivoting* (ILUTP) [14, ch. 10], can reduce the fill-in in the product of the triangular factors M_L and M_U by adding more non-zero entries to the factors themselves. Still, if A is very large, ILUTP can result in an increase in memory occupation and if the elements of A have very different values the preconditioner can be poor.

The quality of the preconditioner M for solving a system with matrix A can be measured with the condition number of the matrix $M^{-1}A$, denoted with $\kappa(M^{-1}A)$ [142]. If the condition number is much higher than one, the number of iterations of the PCG method is very high, typically close to n. To measure how the number of iterations of the PCG changes with respect to $\kappa(M^{-1}A)$, an experiment was carried out. This is the only original part of this section of Chapter 5. Given a matrix Q with a hundred rows and columns with entries given below for some $\mu \in \mathbb{R}$

$$q_{ij} := \begin{cases} 1, & \text{if } i = j \vee i = j+1 \vee i = j-1, \\ \mu, & \text{if } i = j+w \vee i = j-w \end{cases},$$

with $w = 10$ and $A := QQ^T$ the matrix of a linear system, the quality of the preconditioning was measured as a function of μ. Figure 5.3 shows that, depending on the value of μ, the quality of the preconditioner can be poor independently of the type of incomplete LU factorisation used so that the number of iterations is close to the number of rows and columns n. Moreover, the number of non-zero entries in the factors obtained with ILUTP is 2.98 times that of the factors obtained with ILU0.

The same numerical issue also occurs in the algorithms that compute incomplete Cholesky factorisations [14, ch. 10] and affects the convergence of the PGMRES method, which also requires the solution of a linear system with the preconditioner at each iteration [14, ch. 9].

Since the numerical effect of fill-in on the preconditioner is difficult to predict, the use of an accelerator, such as a GPU, can counteract possible negative effects on the execution time.

5.2 CUSPARSE AND TASSL IMPLEMENTATIONS

Preconditioned iterative methods can require many iterations to converge to the solution of the linear system if the system matrix A is sparse. Both CUSPARSE and TASSL aim to reduce the execution time of each iteration by using a GPU, but require a preliminary analysis of the matrix A. In Chapter 4 of this thesis it was observed that the preliminary analysis of CUSPARSE is more time consuming than that of TASSL in practice. The additional cost of the analysis is amortized if the number of iterations is large.

To compare CUSPARSE and TASSL, the PCG method was implemented for this thesis with both algorithms.

The CUSPARSE implementation carries out both the linear system solves at line 2 and line 11 with CUSPARSE. The matrix-vector multiplications at line 1 and line 6 are implemented with the same library while all other operations are implemented with the CUBLAS library [10], which is the dense linear algebra library given by the GPU vendor. This implementation is an expression of the conventional design philosophy for parallellising numerical algorithms, which

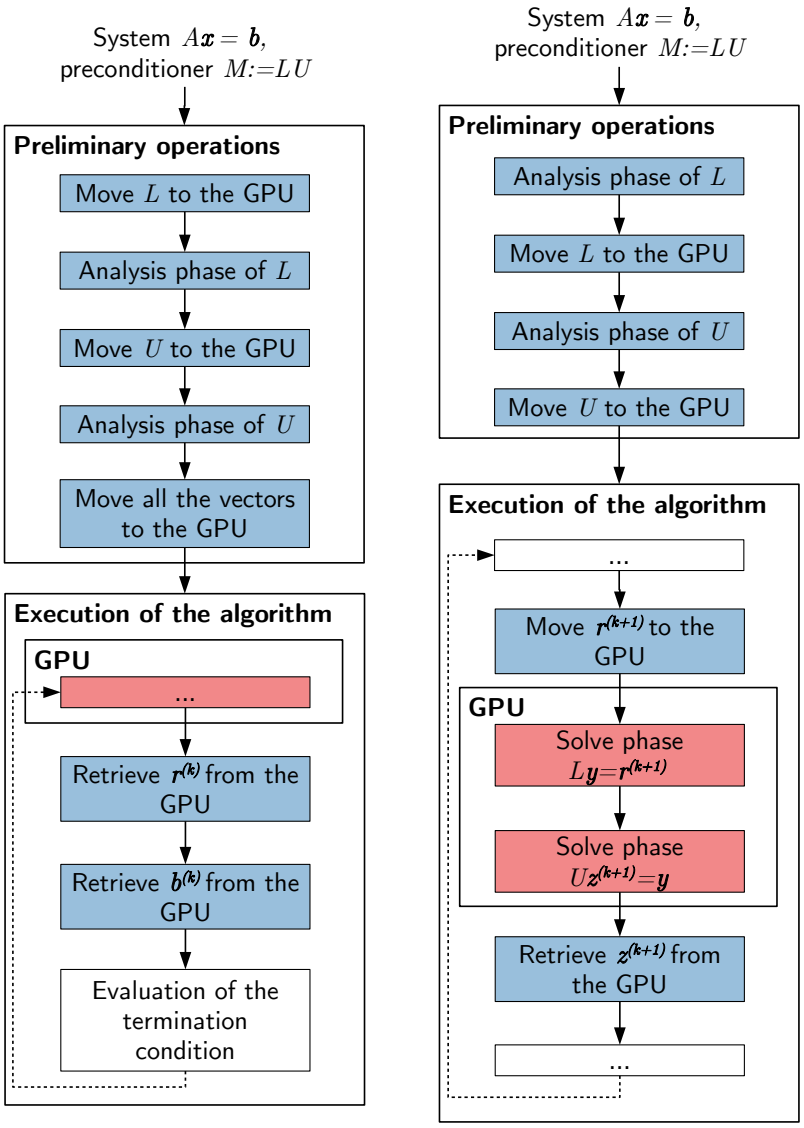
is to transfer the data to the GPU only at the beginning of the algorithm and to execute all the numerical operations on the device, aiming to maximise concurrency.

The TASSL implementation of the PCG method only accelerates the linear system solves at line 2 and line 11 with the GPU while implementing the remaining operations with the CSPARSE library [25], which does not use any accelerator. This implementation is an expression of the design philosophy for parallelising numerical algorithms suggested in this thesis, which consists in not to aim to maximise concurrency only but to balance concurrency and data locality depending on the structure in the data. This is carried out automatically by TASSL, using the algorithms described in Chapter 4.

In terms of GPU operations, the two implementations are illustrated in Figure 5.4. Since no vector operations nor matrix-vector multiplications are accelerated in the TASSL implementation, one could think TASSL is disadvantaged with respect to the CUSPARSE implementation. However, under the hypothesis of the solution of STLs being the most time consuming part of each iteration, this disadvantage can be neglected. Therefore, under this hypothesis, when the TASSL implementation is faster than the CUSPARSE implementation it is only because aiming to balance concurrency and data locality leads to better performance than only aiming to maximise concurrency.

5.3 EXPERIMENTAL EVALUATION

The CUSPARSE and the TASSL implementations were compared experimentally by executing three series of tests. In the first series, artificially generated matrices were used to verify that the solution of STLs is the most time consuming part of each iteration of the PCG method. In the second and in the third series of tests the execution time of the CUSPARSE and the TASSL implementations are compared using artificial and non-artificial test cases.



(a) *The CUSPARSE implementation.* (b) *The TASSL implementation.*

Figure 5.4: Comparison of the CUSPARSE and the TASSL implementations of the PCG method. In the figure, the operations related to the GPU are highlighted. In the CUSPARSE implementation, after some preliminary operations the whole algorithm is executed on the GPU except for the evaluation of the termination condition. In the TASSL implementation, the only part of the algorithm that is executed on the GPU is the solution of two STLs using the factorised form of the preconditioner.

Table 5.1: Breakdown of the execution time of one iteration of the PCG method for the CUSPARSE and TASSL implementations. In both implementations, solving linear systems of equations with the preconditioner M for system matrix is the most time-consuming part of the iteration.

Part of the iteration	Line in Algorithm 5.1	CUSPARSE (%)	TASSL (%)
Termination condition	5	0.0208	0.0285
Computing $\alpha^{(k)}$	6–8	0.623	0.393
Computing $\boldsymbol{x}^{(k+1)}$	9	0.0710	0.0153
Computing $\boldsymbol{r}^{(k+1)}$	10	0.282	0.0301
Computing $\boldsymbol{z}^{(k+1)}$	11	98.6	99.0
Computing $\beta^{(k)}$	12	0.130	0.0452
Computing $\boldsymbol{p}^{(k+1)}$	13	0.258	0.245

5.3.1 *Breakdown of the execution time*

In the first test, both the CUSPARSE and the TASSL implementation of the PCG method were executed on a computer equipped with an Intel i7-3770 CPU with a clock frequency of 3.40 GHz and a L3 cache memory of 8 MB [144]. Also, the computer had an Nvidia GeForce GT 430 GPU, with a peak double-precision performance of 403 GFLOPS [145].

Ten matrices with a random sparsity pattern were generated using GNU Octave [143]. Considering that the GPU used has a local memory size of 48 KB, the maximum number of vertices in a sub-graph $n_{\max}$ in TASSL was 6144, as defined in (4.6) (page 98). Also, the GPU used has one compute unit. Hence, the number of rows and columns in the matrices n was chosen to be 10^4, so that the whole device was used during the computation for both TASSL and CUSPARSE. The density of non-zero elements

$$\delta := \frac{n_{\text{entries}}}{n^2}$$

which is given by the ratio of the number of non-zero entries in the matrix to the total number of entries was 0.1 %, which is the average in the benchmarks used for the solve phase of TASSL in Section 3.3.2 of Chapter 3 (page 89).

Each matrix was factorised with the ILU0 algorithm, which gives factors with less than half the number of non-zero entries compared to the ILUTP, which makes ILU0 more suited to large scale problems. The PCG method was carried out with a random right-hand side vector, after setting a maximum number of iterations equal to the number of rows and columns in the matrix and a tolerance of 10^{-6}. These parameters were chosen because they are the default settings in MATLAB [85], which was used to check the correctness of results for both implementations.

For each implementation the geometric mean of the relative execution time over all test cases was taken. The results are presented in Table 5.1, which shows that solving linear systems of equations with the preconditioner M for the system matrix (line 11 of Algorithm 5.1) is by a large margin the most time-consuming part of each iteration for both implementations. For this reason, the CUSPARSE implementation is not disadvantaged with respect to the TASSL implementation for executing sparse matrix-vector multiplications on the GPU.

Summarising, this experiment shows that none of the two implementations is disadvantaged with respect to the other. Hence, it is possible to compare the execution time of the two implementations and draw conclusions about the two different design philosophies for parallelising sparse linear algebra algorithms.

5.3.2 *Artificial test cases*

The aim of the second series of tests was to show that the total execution time of the TASSL implementation, which includes the preliminary analysis, is smaller than that of the CUSPARSE implementation for a sufficiently high number of iterations.

The two implementations were executed on a computer [146] with an Intel Xeon E5-2640 v3 CPU that has a clock frequency of 2.6 GHz and a L3 cache of 20 MB [140], and an Nvidia K80 GPU with a peak double-precision performance of 2.91 TFLOPS [136]. For the test, some diagonal matrices of bandwidth 20, of the same type as in Section 5.1.2, were generated in the form:

$$\begin{bmatrix} 1 & 1 & & & a_{1,w} & & & \\ 1 & 1 & 1 & & & a_{2,w+1} & & \\ & 1 & 1 & 1 & & & \ddots & \\ & & \ddots & \ddots & \ddots & & & a_{n-w,n} \\ a_{w,1} & & & \ddots & \ddots & \ddots & & \\ & a_{w+1,2} & & & \ddots & \ddots & \ddots & \\ & & \ddots & & & \ddots & \ddots & 1 \\ & & & a_{n,n-w} & & & 1 & 1 \end{bmatrix}.$$

All the matrices had a different number of rows and columns. Also, each a different value μ of the entries on the diagonals at $w = 10$. Each generated matrix was then multiplied by its transpose to obtain positive definite matrices for the PCG method. The matrices were then divided into two sets. As in the previous sets of experiments, the number of rows and columns n was 10^4 in the set a so to keep the whole GPU busy during the experiments. To evaluate the performance of both TASSL and CUSPARSE on large scale problems, n in the set b was chosen to be about 100 times that of the test cases in set a, so to have a maximum execution time of about three hours for each test case. The value of μ was between 10^{-6} and 10^{-2} in both sets, which is a range that covers different qualities of preconditioning. As shown in Section 5.1.2, different values of μ gave matrices for which the PCG method converged in a range of iterations.

The experimental results for the second test session are given in Figure 5.5. In all test cases, the total execution time of the TASSL implementation was less than that of the CUSPARSE implementation, proving that TASSL is preferable to CUSPARSE for large-scale matrices that require a large number of iterations. In particular, the maximum speedup measured was 2.11 and the minimum speedup was 1.4.

5.3.3 *Non-artificial test cases*

In the third series of experiments, the total execution time of each of the two implementations of the PCG method were compared over non-artificial test cases. All band matrices with more than 10^4 rows and columns in the NIST

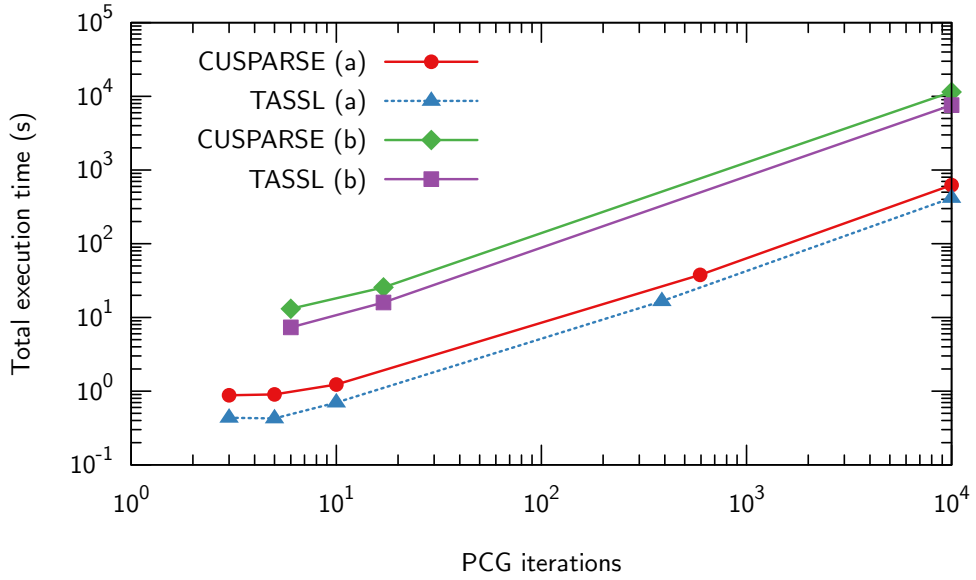


Figure 5.5: Comparison of the total execution time of the CUSPARSE and TASSL implementation of the PCG method when solving artificial test cases.

collection [147] were selected, so that both the TASSL and CUSPARSE implementations used the whole GPU. Also, all tridiagonal matrices were selected to observe the effect of the sparsity pattern on the quality of the preconditioner, as described in Section 5.1.2. All test cases were symmetric positive definite matrices, so to be tested with the PCG method.

Table 5.2: Execution time of the PCG method implemented with CUSPARSE and TASSL on some test cases in the NIST collection [147]. In the table, *TET* means total execution time.

Name	CUSPARSE		TASSL		Speedup
	Iterations	TET (s)	Iterations	TET (s)	
bcsstk17	1.00×10^{5}	1.57×10^{3}	1.00×10^{5}	9.49×10^{2}	1.65
bcsstk18	6.78×10^{2}	4.31	6.77×10^{2}	3.29	1.31
gr_30_30	2.00×10^{1}	3.39×10^{-1}	2.00×10^{1}	5.51×10^{-2}	6.16
nos7	3.00×10^{1}	2.70×10^{-1}	3.00×10^{1}	4.70×10^{-2}	5.74
s3dkq4m2	1.00×10^{5}	6.23×10^{3}	1.00×10^{5}	5.34×10^{3}	1.16
s3dkt3m2	5.09×10^{3}	3.07×10^{2}	5.19×10^{3}	2.37×10^{2}	1.29

The same experimental setup and technique used for the artificial test cases was also used for the non-artificial ones. For the non-artificial test cases, the experimental results are summarised in Table 5.2.

In the results, it can be noticed that for the small tri-diagonal test cases *gr_30_30* and *nos7* the achieved speedup was much higher than in the other test cases. This is because the preliminary analysis of TASSL is carried out on the CPU and all the required data was easily stored in the large cache memory of the Intel Xeon processor.

The smallest speedup was obtained on the large test case *s3dkq4m2*. On this matrix, CUSPARSE's preliminary analysis took 0.24 s while TASSL's analysis took 3.8 s. However, the total execution time of TASSL was less than that of CUSPARSE because of the faster iterations. The average execution time of iteration was 0.52 s for TASSL and 0.61 s for CUSPARSE.

5.4 RELATED WORK

Bolz *et al.* described the first implementation of the CG method on GPU in 2003. This implementation used a specialised technique for computing the sparse matrix-vector multiplication (SPMVM) which required packing off-diagonal matrix entries together in the same array, which is similar to the ELLPACK approach [97]. The cost of communication was shown to be limiting the performance of the implementation. Also, in the implementation of the Jacobi-preconditioned CG algorithm by Buatois *et al.* the only sparse linear algebra function used was SPMVM [149]. Using either the CSR or BSR sparse data structures [17], SPMVM on GPUs was shown to be about 4 times faster than on a single-core CPU with memory bandwidth being the limitation of the implementation.

Li and Saad implemented the PCG method on GPU and single-core CPU and observed that the sparse triangular solve is the most time-consuming part in the execution of the algorithm [126]. Their approach to solving the STL was similar to that of CUSPARSE [16], but their implementation was less sophisticated since it required executing a kernel per each level.

It was observed that communication is the largest overhead in GPU implementations of both iterative and direct methods [150]. As discussed by Naumov, who implemented part of the Nvidia software library CUSPARSE [17], this cost can be amortised only if enough concurrency should be available from the structure of the matrix [16]. Moreover, when preconditioned iterative methods are implemented using CUSPARSE, the cost of the analysis should be taken into account too. The GPU sparse linear algebra libraries ViennaCL [18], SPRAL [111], and GHOST [22] all provide sparse iterative methods for solving systems of linear equations. In particular, MAGMA [20] and PARALUTION [19] provide implementations of the PCG method which use the CUSPARSE functions when handling the preconditioner.

5.5 CONCLUSIONS FROM THIS CHAPTER

This chapter compared two different design philosophies for parallel sparse linear algebra algorithms. CUSPARSE is the expression of the conventional design philosophy, which consists in aiming to maximise concurrency, while TASSL is the expression of the design philosophy proposed in this thesis, which consists in balancing concurrency and data locality. The two design philosophies were used as building blocks for implementing the PCG method.

Depending on the sparsity pattern of the system matrix, solving a linear system of equations with the PCG can require a large number of iterations. Although predicting the quality of the preconditioning is difficult, in Section 5.1.2 it is shown that preconditioners of banded and tri-diagonal linear systems can be poor because of numerical instability due to fill-in. These types of matrices arise from, among other fields, finite element methods [14].

Poor preconditioning results in a high number of iterations. In this case, all experimental results in Section 5.3 show that the TASSL algorithm is faster than the CUSPARSE algorithm because the extra cost of the TASSL analysis is amortised. Moreover, since TASSL's analysis is carried out on a multi-core CPU, there are some test cases in which the execution time of TASSL's analyis is less than CUSPARSE's, because for both algorithms the analysis phase

consists of traversing a directed acyclic graph (DAG), which could offer little opportunity for concurrency depending on graph structure.

The use cases described in the first part of this thesis are the same for the two libraries. However, in the second part of the thesis it is shown that, unlike the CUSPARSE approach, the TASSL approach can be used in the case of a sequence of STLs with matrices that have a similar sparsity pattern. Because of this, TASSL can be used to implement more advanced algorithms than the PCG method described in this chapter. Before describing the implementation of one of these algorithms, the next chapter first discusses how the idea of sub-graphs behind TASSL can be used in this more advanced context.

Part II

SOLVING A SEQUENCE OF SYSTEMS OF LINEAR EQUATIONS

This part the thesis discusses a generalisation of the TASSL approach, described in the previous part, to the solution of sequences of sparse triangular linear systems of equations (STLs) with different matrices of the type $A^{(k)}\boldsymbol{x} = \boldsymbol{b}^{(k)}$, where $k = 1, 2, \ldots, n_{\text{seq}}$. Using CUSPARSE, the analysis phase has to be repeated for every matrix $A^{(k)}$ in the sequence. Instead, using TASSL the analysis of $A^{(k)}$ can be obtained from that of the matrix $A^{(k-1)}$ if the structure of the two matrices is similar. Chapter 6 describes how the TASSL approach can be generalised to the case of a sequence of STL, while experimental results and the application of this approach to the simplex method are discussed in Chapter 7.

6

UPDATING THE RESULT OF THE ANALYSIS PHASE

Some algorithms require the solution to a sequence of sparse triangular systems of equations (STL), the matrices of which have similar sparsity patterns. For example, the left-looking sparse LU decomposition (LL-LU) [25], factorises a square matrix $A \in \mathbb{R}^{n \times n}$ by solving n linear systems with matrices

$$L^{(1)}, L^{(2)}, \ldots, L^{(k-1)}, L^{(k)}, L^{(k+1)}, \ldots, L^{(n_{\text{seq}})} \tag{6.1}$$

where the matrix $L^{(k)}$ is defined in $\mathbb{R}^{n \times n}$ and its sparsity pattern is the same as that of $L^{(k+1)}$ and $L^{(k-1)}$ but for one column at most. The LL-LU algorithm is often used in circuit design and simulation [151], and the numerical solution of the sequence of STLs is usually the most time-consuming part of the algorithm. For instance, it is on average 90 % of the execution time of the package KLU [151], when carried out on an Intel i7 CPU with 8 MB of L3 cache [144] over all square matrices from the Florida Matrix Collection [133].

The same type of sequence of STLs as (6.1) is also solved in flexible iterative algorithms [14, ch. 9], which are preconditioned iterative methods that change the preconditioner at each iteration. However, these algorithms require the solution of two sequences of STLs instead of one: one for the upper triangular factors and one for the lower triangular factors.

Two sequences of STLs are also solved by the simplex method (SM) [15], in which the basic matrix $M \in \mathbb{R}^{n \times n}$ describes the status of the algorithms at each iteration and is stored in factorised form. On an Intel i7 CPU with 8 MB of L3 cache and a clock frequency of 3.40 GHz [144], the sofware SCIP [139] spends on average about 30 % of the execution time solving sequences of STLs when carried out on linear programming problems in the NETLIB collection [12]. Because of this, accelerating the solution of sequences of STLs with GPUs is of much practical interest.

This chapter discusses how to accelerate the solution of STLs with the TASSL algorithm on GPUs. As shown in Figure 6.1, the TASSL algorithm is organised

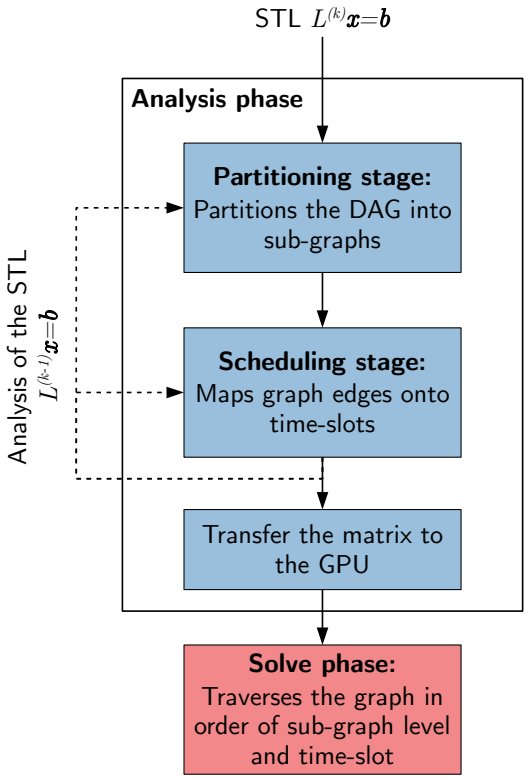


Figure 6.1: Operation of the time slot sub-graph-level (TASSL) algorithm for sequences of STLs. The analysis phase is described in Chapter 4 in this thesis, while the solve phase is described in Chapter 3.

in two phases: the analysis phase and the solve phase. When carried out on a sequence of the type in (6.1), the execution time of the analysis of matrix $L^{(k)}$ can be reduced by updating the results of the analysis of $L^{(k-1)}$, hence the overhead of the analysis is amortised over the whole sequence of matrices. Section 6.3, which illustrates experimental results and practical application of the approach discussed in the present chapter, shows that the execution time of the analysis phase is about halved on non-artificial test cases when the results of the analysis are updated.

The scientific contributions in this chapter are the following:

- A method to solve sequences of sparse matrices with similar sparsity patterns on a GPU.
- A technique to update the partition of a directed acyclic graph (DAG) when one or more edges in the graph change.

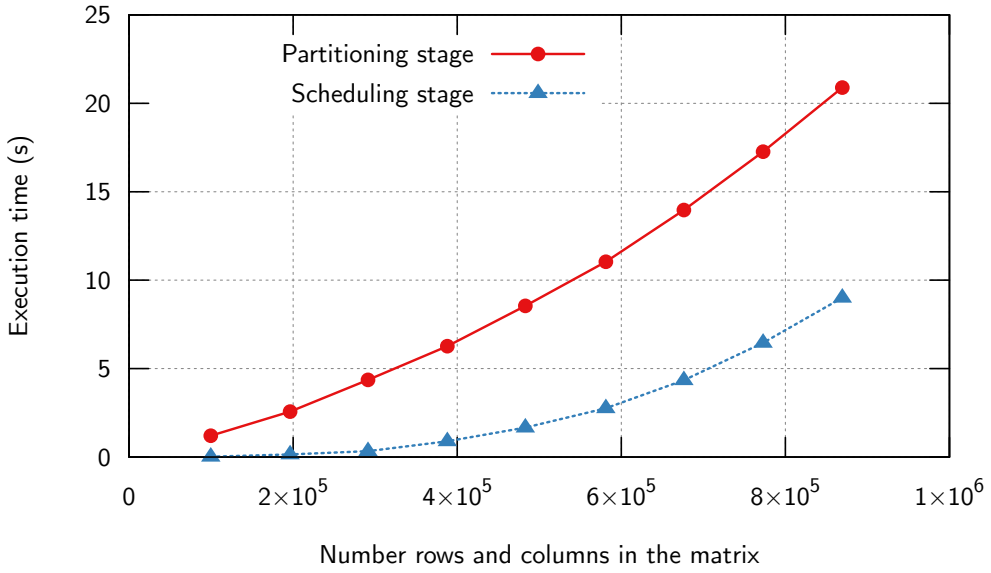


Figure 6.2: Execution time of the partitioning and the scheduling stages: summary of the experimental results in Figure 4.13 (page 118). The matrices considered in the experiment had 16 diagonal blocks of varying size, each one with the same ratio of non-zero entries to the square of block size, called *density* of 0.1 %.

- A technique to update the schedule of numerical operations to solve a STL according to the changes in the matrix of the STL.

This chapter is organised as follows, Section 6.1 discusses updating the partition of the DAG, while Section 6.2 shows how to update the schedule, which is the map of graph edges to time slots. The following Section 6.3 discusses the experimental results.

6.1 UPDATING THE PARTION OF THE GRAPH

In Chapter 4 (page 118) it was shown empirically that the partitioning stage is the most time consuming of the two stages of the analysis phase. The experimental results, which are summarised in Figure 6.2, demonstrate that the time spent on the partitioning stage grows with the number of rows and columns in the matrix. Because of this, the partitioning stage is expected to take most of the execution time of the analysis for large matrices, although the execution

time of the scheduling stage cannot be neglected. If the similarities between the sparsity patterns of the matrices are put to effect, the cost of the partitioning can be amortised by updating the partition from one matrix in the sequence to the other.

A technique to carry out the update can be derived by considering just two matrices L and L' in a sequence such as that in (6.1), assuming $L := L^{(k)}$ and $L' := L^{(k-1)}$, where k is any number from 2 to the length of the sequence n_{seq}. Although for the remainder of this chapter it is assumed that all the matrices in the sequence are lower triangular matrices, the same derivation can be done for sequences of upper triangular matrices. Moreover, the derivation in this chapter assumes that the two matrices L and L' differ in only one column, whose index is j_{diff}. The entries of this column are

$$\left\{(i'_1, j_{\text{diff}}),\ (i'_2, j_{\text{diff}}),\ \ldots,\ (i'_{n'_{\text{elem}}}, j_{\text{diff}})\right\} \subseteq \mathcal{E}'$$

for matrix L' and

$$\left\{(i_1, j_{\text{diff}}),\ (i_2, j_{\text{diff}}),\ \ldots,\ (i_{n_{\text{elem}}}, j_{\text{diff}})\right\} \subseteq \mathcal{E}$$

for matrix L, where n'_{elem} and n_{elem} are the number of non-zero entries in the column in matrix L' and L respectively, while $\mathcal{E}'$ and $\mathcal{E}$ are the set of edges in the DAGs $G' := (\mathcal{V}', \mathcal{E}')$ and $G := (\mathcal{V}, \mathcal{E})$ associated with L' and L respectively. The general case, when L' and L differ in n_{diff} columns of indices $j_{\text{diff},1}, j_{\text{diff},2}, \ldots, j_{\text{diff},n_{\text{diff}}}$ can be obtained by applying the same considerations of the case of a single column to each of the n_{diff} columns, from that with the lowest index to that with the highest index.

During the partitioning step of the TASSL algorithm, the set of vertices $\mathcal{V}'$ in the graph G' is partitioned into n'_{subg} subsets $\mathcal{V}'_0, \mathcal{V}'_1, \mathcal{V}'_2, \ldots, \mathcal{V}'_{n'_{\text{subg}}}$. A non-zero entry l'_{ij} in the set $\mathcal{E}'$ is called external if j and i belong to two different subsets, otherwise the edge is called internal. The partition ofthe graph G', obtained by the partitioning stage, is feasible if both the following conditions, which are also discussed in Chapter 3 (page 75), are met:

CONDITION 1: if an edge goes from a vertex in $\mathcal{V}'_k$ to a vertex in $\mathcal{V}'_p$, then $p \geq k$.

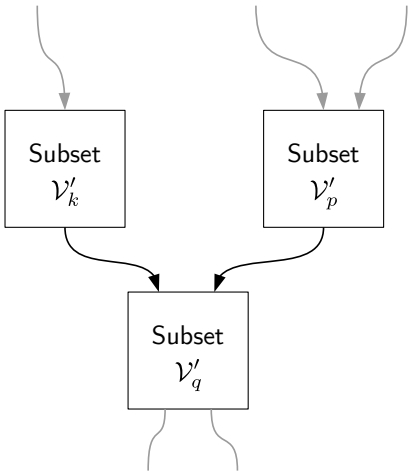


Figure 6.3: An example partition DAG. Each vertex represents a subset of vertices in the original graph and each edge represents the existence of one or more external edges in the original graph G'.

CONDITION II: the number of vertices in the set $\mathcal{V}'_k$, denoted n'_k, must not be greater than a positive integer $n_{\max}$.

Since the partition is feasible, it can be represented with a DAG in which each vertex is a subset and each edge represents one or more external edges. For example, in Figure 6.3 the edge between subset $\mathcal{V}'_p$ and vertex $\mathcal{V}'_q$ shows that there is at least one edge (j, i) in the original graph such that j is in the subset $\mathcal{V}'_p$ and i is in the subset $\mathcal{V}'_q$.

Let us assume that Figure 6.3 represents the partition, of the graph G' associated with matrix L' and that j_{diff} belongs to either set $\mathcal{V}'_0$, $\mathcal{V}'_k$ or $\mathcal{V}'_q$ with $p < q < k$, where $\mathcal{V}'_0$ is the set of rows and columns of L' that do not have any entries. Each sub-graph in the partition is associated with a subset, for example sub-graph G'_k is defined as

$$G'_k \coloneqq (\mathcal{V}'_k, \mathcal{E}_k), \quad \mathcal{E}_k \coloneqq \left\{(i,j) \in \mathcal{E} \mid i \in \mathcal{V}'_k \wedge j \in \mathcal{V}\right\}.$$

In particular, in Figure 6.3 no assumption is made on whether G'_p and G'_q belong to the same sub-graph level (see Figure 4.5 on page 107).

Also, let us assume that the following matrix in the sequence L has a non-zero entry $l_{i,j_{\text{diff}}}$ that is missing in L' or, equivalently, that the graph G has an

Table 6.1: Possible cases for the update or re-partition of a DAG in a sequence. The sets $\mathcal{V}'_0$, $\mathcal{V}'_k$ or $\mathcal{V}'_q$ are subsets of the vertices in the graph G' associated with the matrix L'. *U* stands for *update*, meaning that a feasible partition of the vertices in the graph G can be obtained by updating the partition of G', whereas *R* stands for *repeat*, meaning that the partitioning stage has to be repeated completely.

i	j_{diff}			
	$\mathcal{V}'_0$	$\mathcal{V}'_k$	$\mathcal{V}'_p$	$\mathcal{V}'_q$
$\mathcal{V}'_0$	U	U	U	U
$\mathcal{V}'_k$	U	U	R	R
$\mathcal{V}'_p$	U	U	U	R
$\mathcal{V}'_q$	U	U	U	U

edge (i, j_{diff}) where i belongs to either set $\mathcal{V}'_0$, $\mathcal{V}'_k$ or $\mathcal{V}'_q$. Table 6.1 shows all possible cases for the edge (i, j_{diff}): in some of them, the graph G needs to be re-partitioned completely to obtain a feasible partition, and in some cases the partition of G can be obtained by updating that of G'.

6.1.1 *Cases when re-partitioning is necessary*

Re-partitioning the graph G to obtain a feasible partition is necessary whenever condition I (page 140) cannot be satisfied, that is when

$$(i, j_{\text{diff}}) \in \mathcal{E},\ j_{\text{diff}} \in \mathcal{V}'_k \wedge i \in \mathcal{V}'_p \quad p < k,\ p \neq 0,\ k \neq 0.$$

These cases are illustrated in Figure 6.4. If the vertex i belongs to the subset $\mathcal{V}'_0$ it is always possible to add another subset to the partition by having

$$n_{\text{subg}} \leftarrow n'_{\text{subg}} + 1,$$

and assigning i to the subset $\mathcal{V}_{n_{\text{subg}}}$.

If the vertex j_{diff} belongs to the subset $\mathcal{V}'_p$ and vertex i to the subset $\mathcal{V}'_k$ the structure of the partition DAG can be used to avoid re-partitioning. For example, one could choose

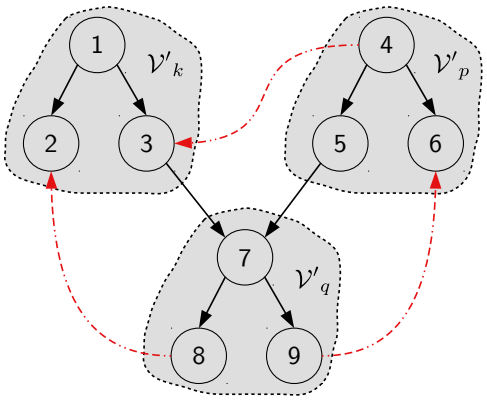


Figure 6.4: Visual example of the cases in which re-partitioning the graph is necessary shown in Table 6.1, in the case of a graph with 9 vertices. The dashed edge in red between the subsets represents the edge added from graph G' to graph G. While the case at the top can be solved by exchanging the indices of $\mathcal{V}'_k$ and $\mathcal{V}'_p$ in the partitioning of G, the other two violate condition I (page 140) and require a complete re-partitioning.

$$\mathcal{V}_k \leftarrow \mathcal{V}'_p$$
$$\mathcal{V}_p \leftarrow \mathcal{V}'_k$$

and check if the partition obtained for G satisfies the definition of feasible partition. However, checking for feasibility requires browsing all n_{edges} edges in the graph G, so it has complexity $O(n_{\text{edges}})$.

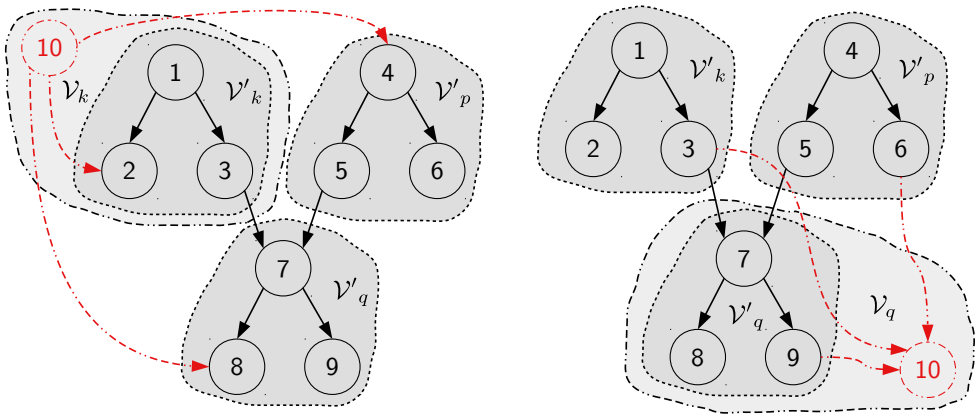
6.1.2 *Cases when the partition can be updated*

As shown in Table 6.1, in most cases re-partitioning can be avoided because a feasible partition for G can be obtained by updating the partition of G'.

The cases in wich either $j_{\text{diff}} \in \mathcal{V}'_0$ or $i \in \mathcal{V}'_0$ can all be solved by adding an extra subset to the partition of G and assigning either j_{diff} or i to it, as shown in Figure 6.5. Otherwise, one can check if there exists a subset $\mathcal{V}'_{\text{nofull}}$ with a number of elements lower than an upper bound $n_{\max}$, which depends on the size of the GPU's local memory. To do this, one can use the function *next_nonfull_subg* in Algorithm 6.1. If such a set exists, the introduction of a new set is avoided if the vertex is put in the set $\mathcal{V}'_{\text{nofull}}$. If both vertices j_{diff} and

```
1: function NEXT_NONFULL_SUBG(V_0, V_1, ..., V_{n_subg}, n_subg, n_max, k_start)
2:     k ← k_start
3:     while |V_k| ≥ n_max ∧ k ≤ n_subg
4:         k ← k + 1
5:     if k > n_subg
6:         n_subg ← n_subg + 1
7:     return k
```

Algorithm 6.1: Function that finds the first subset of vertices with less than $n_{\max}$ elements starting from vertex $\mathcal{V}_{k_{\text{start}}}$.



(a) $j_{\text{diff}} \in \mathcal{V}'_0$. (b) $i \in \mathcal{V}'_0$.

Figure 6.5: Visual examples of the cases in which either the start vertex j_{diff} or the end vertex i of the edge (i, j_{diff}) that is in G but not in G' initially belong to the subset $\mathcal{V}_0$. This is the subset of the vertices in the graph that do not have neither entering not exiting edges. In the figures, the dotted red edges represent the possible cases in Table 6.1 and a possible update of the partition is shown in case the subsets $\mathcal{V}'_k$ and $\mathcal{V}'_q$ have less than $n_{\max}$ vertices.

i belong to $\mathcal{V}'_0$, *next_nonfull_subg* is executed first on j_{diff} starting from subset $\mathcal{V}'_1$ and then on i. By doing this, one makes sure that both vertices are put in subsets while the partition is kept feasible.

Although updating the partition of the previous matrix in the sequence reduces the execution time of the analysis phase, it can make the execution time

of the solve phase longer. Considering the graph G, in Chapter 3 (page 81) the execution time of the solve phase was given by

$$t_{\text{solve}} \approx \sum_{p=1}^{n_{\text{levels}}} \max_{g \in \mathcal{L}_p} \left\{ \sum_{q=1}^{n_{\text{int,slots},g}} \frac{1}{T\eta_{\text{int},g,q}} + \sum_{q=1}^{n_{\text{ext,slots},g}} \frac{1}{T\eta_{\text{ext},g,q}} + (n_{\text{int,slots},g} + n_{\text{ext,slots},g})\sigma_{\text{local}} + \beta n_{\text{vertices},g} \right\} + \kappa n_{\text{levels}}. \quad (6.2)$$

In (6.2), n_{levels} is the number of sub-graph levels in the partition. Sub-graph levels are sets of sub-graphs such that there is no path in G from one vertex in one sub-graph to a vertex in another sub-graph. $\mathcal{L}_p$ denotes the p-th sub-graph level. In TASSL, numerical operations during the processing of a sub-graph in the solve phase are organised in time slots. Time slots are sets of numerical operations executed between two synchronisation points, for example a local memory barrier, whose execution time is denoted in (6.2) with σ_{local}, or the starting or stopping of a kernel, whose execution time is denoted in (6.2) with κ. For the g-th sub-graph G_g, $n_{\text{int,slots},g}$ and $n_{\text{ext,slots},g}$ are the number of time slots associated with internal and external edges respectively. Also, in (6.2) $n_{\text{vertices},g}$ is the number of vertices in G_g, and $\eta_{\text{int},g,q}$ and $\eta_{\text{ext},g,q}$ are the operation efficiencies of the q-th internal and the q-th external time slots of G_g respectively, which are discussed in Chapter 3 (page 66 and page 80). Moreover, in (6.2), T is the maximum throughput of multiply-and-add instructions executed by the GPU, called SIMD instructions, and β is the transfer rate of floating point values in double precision to and from the local memory of the GPU.

As done in Chapter 3 (page 81), the model in (6.2) can be simplified by assuming that operation efficiencies are constant across all sub-graphs and time slots. For example, for internal time slots operation efficiencies can be in

first approximation considered equal to η_{int}, and for external time slots they can be considered equal to η_{ext}. Then, (6.2) becomes

$$t_{\text{solve}} \approx \sum_{p=1}^{n_{\text{levels}}} \max_{g \in \mathcal{L}_p} \left\{ \frac{n_{\text{int,slots},g}}{T\eta_{\text{int}}} + \frac{n_{\text{ext,slots},g}}{T\eta_{\text{ext}}} + (n_{\text{int,slots},g} + n_{\text{ext,slots},g})\sigma_{\text{local}} + \beta n_{\text{vertices},g} \right\} + \kappa n_{\text{levels}}$$

and then, by choosing

$$\tau_{\text{int}} := \frac{1}{T\eta_{\text{int}}} + \sigma_{\text{local}}$$
$$\tau_{\text{ext}} := \frac{1}{T\eta_{\text{ext}}} + \sigma_{\text{local}}$$

one obtains

$$t_{\text{solve}} \approx \sum_{p=1}^{n_{\text{levels}}} \max_{g \in \mathcal{L}_p} \left\{ n_{\text{int,slots},g}\tau_{\text{int}} + n_{\text{ext,slots},g}\tau_{\text{ext}} + \beta n_{\text{vertices},g} \right\} + \kappa n_{\text{levels}}. \quad (6.3)$$

Similarly, for the graph G' it is

$$t'_{\text{solve}} \approx \sum_{p=1}^{n'_{\text{levels}}} \max_{g \in \mathcal{L}'_p} \left\{ n'_{\text{int,slots},g}\tau'_{\text{int}} + n'_{\text{ext,slots},g}\tau'_{\text{ext}} + \beta n'_{\text{vertices},g} \right\} + \kappa n'_{\text{levels}}.$$

Let us assume that subset $\mathcal{V'}_k$ and the subset $\mathcal{V'}_p$ are in the same sub-graph level in the partition of graph G'. If j_{diff} is in $\mathcal{V'}_k$ and i is in $\mathcal{V'}_p$, a feasible partition of G can be obtained from that of G' immediately in the case when both subsets have fewer than $n_{\max}$ elements by choosing

$$\begin{aligned} \mathcal{V}_k &\leftarrow \mathcal{V'}_k \\ \mathcal{V}_p &\leftarrow \mathcal{V'}_p \end{aligned} .$$

However, since in the resulting partition of G there is an edge from $\mathcal{V}_k$ to $\mathcal{V}_p$, the two subsets cannot belong to the same sub-graph level, the number of sub-graph levels for G is given by

$$n_{\text{levels}} \leftarrow n'_{\text{levels}} + 1$$

where n'_{levels} is the number of sub-graph levels of G'. If all sub-graphs are processed in the same time

$$\begin{aligned}\omega &= n_{\text{int,slots},g}\tau_{\text{int}} + n_{\text{ext,slots},g}\tau_{\text{ext}} + \beta n_{\text{vertices},g} \quad \forall G_g \\ &= n'_{\text{int,slots},g}\tau'_{\text{int}} + n'_{\text{ext,slots},g}\tau'_{\text{ext}} + \beta n'_{\text{vertices},g} \quad \forall G'_g\end{aligned}$$

then (6.3) becomes

$$t_{\text{solve}} \approx n_{\text{levels}}\,(\alpha + \omega) \tag{6.4}$$

and, being t'_{solve} the execution time of the solve phase for G', the relative increase in the execution time, or *slowdown* is

$$\frac{t_{\text{solve}} - t'_{\text{solve}}}{t'_{\text{solve}}} \approx \frac{n_{\text{levels}} - n'_{\text{levels}}}{n'_{\text{levels}}} = \frac{1}{n'_{\text{levels}}}.$$

This approximation shows that when the number of sub-graph levels is small, either because the matrix L' is small or because concurrency is high, the addition of one sub-graph level causes a relatively large slowdown of the solve phase.

If the subsets $\mathcal{V}'_k$ and $\mathcal{V}'_p$ belong to two different sub-graph levels, the edge (i, j_{diff}) is external in the partition of graph G, so the scheduling can require an extra time slot. Consequently, in the worst case and if (6.4) holds, the slowdown of the solve phase is

$$\frac{t_{\text{solve}} - t'_{\text{solve}}}{t'_{\text{solve}}} \approx \frac{\tau_{\text{ext}}}{n'_{\text{levels}}(\alpha + \omega)}, \tag{6.5}$$

which is negligible given that τ_{ext} is in the order of microseconds and the denumerator is in the order of milliseconds or seconds. Similarly, if the edge (i, j_{diff}) is internal the slowdown under the same assumptions is

$$\frac{t_{\text{solve}} - t'_{\text{solve}}}{t'_{\text{solve}}} \approx \frac{\tau_{\text{int}}}{n'_{\text{levels}}(\alpha + \omega)} \tag{6.6}$$

for the solve phase, which is also neglegible. Nonetheless, for long matrix sequences (6.5) and (6.6) can sum up and cause a significant slowdown.

Because of the accumlated slowdown, sometimes it is preferable not to avoid re-partitioning the graph. Also, measuring the slowdown is expensive, since it

requires checking all the n_{edges} edges in the graph and has complexity $O(n_{\text{edges}})$. To choose whether to re-partition the graph, an algorithm is required.

Summarising, there are some cases in which re-partitioning the graph is necessary, others in which re-partitioning the graph can be avoided without any effect on the performance of the solve phase, and cases in which re-partitioning is preferable because avoiding it can reduce the performance of the solve phase. The proposed algorithm to update a graph partition ignores the third case to reduce the execution time of the analysis phase.

6.1.3 *Algorithm that updates a feasible graph partition*

The proposed algorithm for constructing a feasible partition for graph G from that of graph G' is shown in Algorithm 6.2. The algorithm does not compute the slowdown and does not repeat the partitioning when the performance is reduced below a threshold. Instead, the algorithm sets a *flag* to repeat the partitioning when one of the conditions in the definition of feasible partition (page 140) is violated, hence the overhead of estimating the slowdown is cut at the risk of an increase in the execution time of the solve phase.

Because of this, Algorithm 6.2 is particularly suitable to the applications in which the sequence of matrices starts from a matrix with few non-zero entries, for example, the sequence in both the LL-LU and the simplex method (SM). Since the partition is built incrementally, for these sequences the algorithm puts the vertices j_{diff} and $i_1, i_2, \ldots, i_{n_{\text{elem}}}$ in the same subset until this subset has $n_{\max}$ elements. Consequently, for long sequences a partition obtained with Algorithm 6.2 is expected to cause a slowdown compared to a partition obtained with a complete re-partititoning.

To overcome the cumulative slowdown, one could re-partition the graph every n_{repart} matrices in the sequence regardless of the possibility of obtaining a feasible partition by a faster update. The value of n_{repart} would then be dependent on the application and the specific sequence and it could be chosen manually.

Input: G', $\mathcal{V}'_0$, $\mathcal{V}'_1$, ..., $\mathcal{V}'_{n'_{\text{subg}}}$, G, j_{diff}, i_1, i_2, ..., $i_{n_{\text{elem}}}$, $n_{\max}$
Output: $\mathcal{V}_0$, $\mathcal{V}_1$, ..., $\mathcal{V}_{n_{\text{subg}}}$, n_{subg}, f_{repart}

1: $n_{\text{subg}} \leftarrow n'_{\text{subg}}$
2: $\mathcal{V}_k \leftarrow \mathcal{V}'_k \quad \forall k \in \{0, 1, 2, \ldots, n_{\text{subg}}\}$
3: $f_{\text{repart}} \leftarrow$ false
4: $s_j \leftarrow k \quad j_{\text{diff}} \in \mathcal{V}_k$
5: **if** $s_j = 0$
6: $\quad s_j \leftarrow$ NEXT_NONFULL_SUBG$(\mathcal{V}_0, \mathcal{V}_1, \ldots, \mathcal{V}_{n_{\text{subg}}}, n_{\text{subg}}, 1)$
7: $\quad \mathcal{V}_{s_j} \leftarrow \mathcal{V}_{s_j} \cup \{j_{\text{diff}}\}$
8: **for** $r \leftarrow 1, 2, \ldots, n_{\text{elem}}$
9: $\quad s_i \leftarrow q \quad i_r \in \mathcal{V}_q$
10: $\quad$ **if** $s_i \neq 0$
11: $\quad\quad$ **if** $s_i < s_j$
12: $\quad\quad\quad f_{\text{repart}} \leftarrow$ true
13: $\quad$ **else**
14: $\quad\quad s_i \leftarrow$ NEXT_NONFULL_SUBG$(\mathcal{V}_0, \mathcal{V}_1, \ldots, \mathcal{V}_{n_{\text{subg}}}, n_{\text{subg}}, s_j)$
15: $\quad\quad \mathcal{V}_{s_i} \leftarrow \mathcal{V}_{s_i} \cup \{i_r\}$

Algorithm 6.2: Algorithm that constructs a feasible partition for the graph G from that of the previous graph in a sequence. In the pseudocode, *next_nonfull_subg* is the function defined in Algorithm 6.2. The previous graph in the sequence is G' and the partition of its vertices is described by the subset $\mathcal{V}'_0$, $\mathcal{V}'_1$, ..., $\mathcal{V}'_{n'_{\text{subg}}}$. The current graph in the sequence G differs from G' for the output edges of vertex j_{diff} which are incident to vertices i_1, i_2, ..., $i_{n_{\text{elem}}}$ in the graph G. Also, the integer number $n_{\max}$ is the maximum number of vertices in a subset. The subsets $\mathcal{V}_0$, $\mathcal{V}_1$, ..., $\mathcal{V}_{n_{\text{subg}}}$ represent the updated partition but if the flag f_{repart} is true, the partitioning has to be repeated as described in Chapter 4 (page 97)

Avoiding to re-partition the graph reduces the complexity of the partitioning stage from $O(n_{\text{vertices}} + n_{\text{edges}})$, that is the complexity of the Algorithm 4.2 (page 103), to $O(n_{\text{elem}})$, that is the complexity of Algorithm 6.2, where n_{elem} is the number of elements in the new column, and n_{vertices} and n_{edges} are respectively the number of vertices and edges in G'. Moreover, if the graph is not re-partitioned and if the scheduling algorithm is suitably modified, it can also be avoided to re-execute the scheduling stage.

6.2 UPDATING TIME SLOTS

The algorithm that carries out the scheduling stage which is described in Algorithm 4.4 (page 111), has complexity $O(2n_{\text{edges}})$ and traverses all edges in each sub-graph starting from the vertices that do not have any parent vertex.

If the partition of the graph is obtained by updating that of the previous graph, the traversal can be limited to a subset of the vertices. This is because the partition update in Algorithm 6.2 does not move already assigned vertices from one subset to the other but just assigns vertices in $\mathcal{V}_0$ to another subset. Consequently, an external edge cannot become an internal edge or vice-versa.

To determine which part of the graph is to be traversed whenever the schedule is updated, this section shows first the effect of the removal of edges and next that of the addition of new edges between two graphs G' and G. Using this information, an algorithm for the update of the schedule is then discussed.

6.2.1 *Effect of the removal of edges*

To study the effect of the removal of edges, consider the structure of the graph and the change in the edges shown in Figure 6.6. Suppose that in the graph G' the vertex $\frac{n_{\text{vertices}}}{2}$ has an output edge $(\frac{n_{\text{vertices}}}{2}+1, \frac{n_{\text{vertices}}}{2})$ which is assigned to time slot $\frac{\mathcal{T}_{n_{\text{vertices}}}}{2}$. If in the graph G this edge is removed and no other edge is added, the assignment of edges to time slots is still valid. Since no partial graph traversal is necessary the total number of time slots is still $n_{\text{vertices}}-1$. Repeating the scheduling completely would result in $\frac{n_{\text{vertices}}}{2}$ time slots.

Assuming only one sub-graph, (6.3) gives

$$t_{\text{solve,norep}} \approx \alpha + \tau_{\text{int}} n'_{\text{int,slots}} = \alpha + \tau_{\text{int}} \left(n_{\text{vertices}} - 1\right)$$
$$t_{\text{solve,rep}} \approx \alpha + \tau_{\text{int}} n_{\text{int,slots}} = \alpha + \tau_{\text{int}} \frac{n_{\text{vertices}}}{2}$$

and the slowdown for the solve phase is

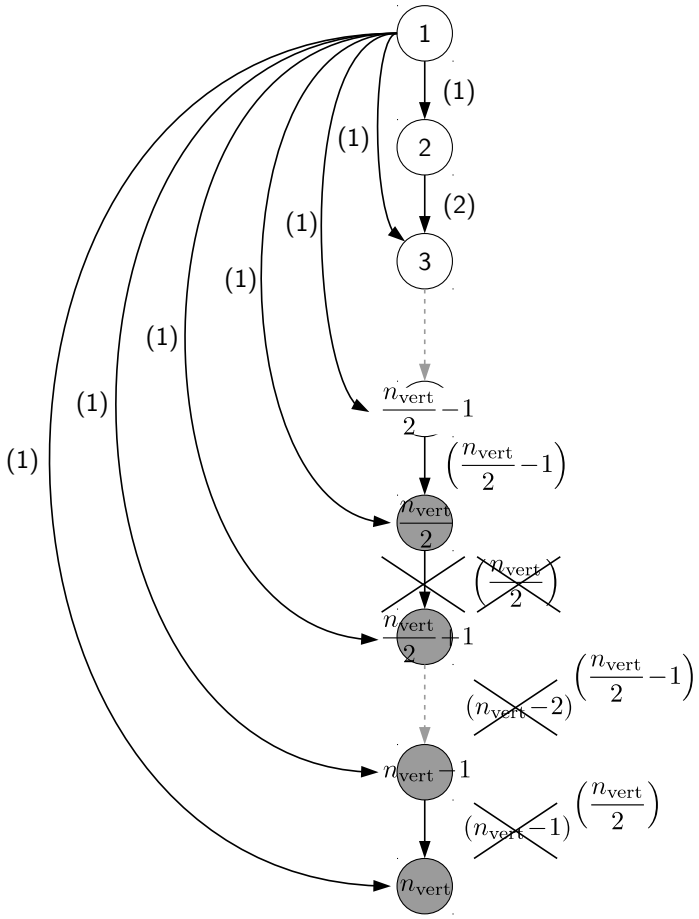


Figure 6.6: Effect of the removal of edges on the schedule. In the figure, time slots are represeted between parentheses, such that (1) is shorthand for time slot $\mathcal{T}_1$. When edge $\left(\frac{n_{\text{vertices}}}{2}, \frac{n_{\text{vertices}}}{2}-1\right)$ is removed, since no edges are added, repeating the scheduling stage would result in $\frac{n_{\text{vertices}}}{2}$ time slots, while a quick update of the schedule results in $n_{\text{vertices}}-1$ time slots.

$$\begin{aligned}\frac{t_{\text{solve,norep}}-t_{\text{solve,rep}}}{t_{\text{solve,rep}}} &\approx \frac{\alpha+\tau_{\text{int}}\left(n_{\text{vertices}}-1\right)-\alpha-\tau_{\text{int}}\frac{n_{\text{vertices}}}{2}}{\alpha+\tau_{\text{int}}\frac{n_{\text{vertices}}}{2}} \\ &= \frac{\tau_{\text{int}}\left(\frac{n_{\text{vertices}}}{2}-1\right)}{\alpha+\tau_{\text{int}}\frac{n_{\text{vertices}}}{2}}\end{aligned}. \tag{6.7}$$

When the execution time of the solve α is negligible, the slowdown is about 1 and the execution time of the solve phase is doubled.

The slowdown in (6.7) is specific to the graph structure shown in Figure 6.6, which is the worst case, but in general the effect of removed edges can acumulate over the matrices in the sequence.

6.2.2 *Effect of the addition of edges*

To study the effect of the addition of edges, consider the case in which vertex j_{diff} has no output edges in the graph G' but has n_{elem} output edges in the graph G. Let $\mathcal{R}_{\text{diff}}$ be the set of vertices reachable from j_{diff} in G, which is the set of all vertices i for which exists a sequence of edges of the type

$$(q_1, j_{\text{diff}}),\ (q_2, q_1),\ \ldots,\ (i, q_{n_{\text{seq}}}) \tag{6.8}$$

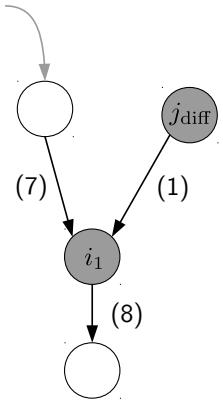
and be $\mathcal{E}_{\text{diff}}$ the set

$$\mathcal{E}_{\text{diff}} \coloneqq \{(i,j) \in \mathcal{E} \mid j \in \mathcal{R}_{\text{diff}} \vee i \in \mathcal{R}_{\text{diff}}\}\,. \tag{6.9}$$

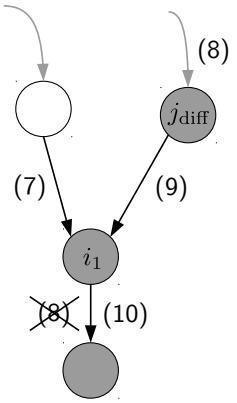
Then, traversing all the edges starting from j_{diff} in G has complexity $O(|\mathcal{R}_{\text{diff}}| + 2|\mathcal{E}_{\text{diff}}|)$. In the worst case this is the same as repeating the scheduling completely, which has complexity $O(2n_{\text{edges}})$, where n_{edges} is the number of edges in the graph respectively. However, the algorithm can be improved in some specific cases.

Figure 6.7a shows one such case. If the new edge (j_{diff}, i_1) is put in a time slot that is not the highest among the input edges of i, it is not necessary to traverse the edges downstream. On the contrary, if the time slot is the highest, as in Figure 6.7b, the change in the schedule is to be propagated to all the edges that can be traversed from i_1. In both cases, the schedule is guaranteed to be correct, which means all the input edges of each vertex are assigned to earlier time slots than the output edges, and all the input edges are assigned to different time slots.

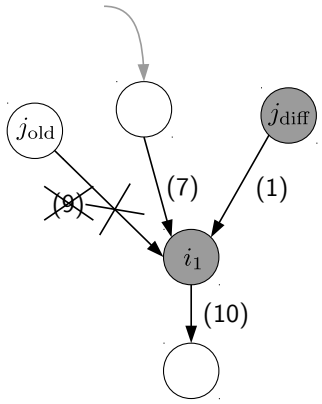
Avoiding traversal of the downstream edges can reduce the execution time of the schedule update depending on the structure of the graph but it can also have an effect on the solve phase. For example, consider Figure 6.7c, showing the case in which edge (i_1, j_{old}) was present in a previous graph in



(a) *Only the edge (i_1, j_{diff}) is to be updated.*



(b) *Edge (i_1, j_{diff}) and all following edges are to be updated.*



(c) *Edge (i_1, j_{diff}) is to be updated, updating the following can be avoided at the cost of introducing empty time slots.*

Figure 6.7: Updating the schedule of edges. In the figures time slots are represented in parentheses next to the corresponding edge, such that (1) is shorthand for time slot $\mathcal{T}_1$, vertex j_{diff} is the vertex whose output edges change in the current graph.

the sequence and was assigned to time slot $\mathcal{T}_9$. In the current graph edge (j_{old}, i_1) is not present but a new one, that is (i_1, j_{old}), is assigned to the time slot $\mathcal{T}_1$. Although it is possible not to traverse the graph starting from i_1, by doing this one makes sure the output edges of i_1 are assigned to time slot $\mathcal{T}_8$ and not $\mathcal{T}_{10}$, which results in a longer execution time of the solve phase.

Summarising, adding and removing edges from the graph can have an effect on the execution time of the solve phase. In some cases, reassigning edges to time slots can be avoided without any effect. In other cases, reassigning edges to time slots cannot be avoided without introducing empty time slots, in which no computation is carried out. These time slots reduce the performance of the solve phase because they require the execution of synchronisation primitives like any other time slot. The proposed algorithm always reassigns edges to time slots to reduce any effect on the solve phase.

6.2.3 *Algorithm to update time slots*

To cancel completely the slowdown effect on the solve phase, the partial traversal of the graph should involve all the vertices reachable from j_{diff} in both graphs G' and G, which are the sets $\mathcal{R}'_{\text{diff}}$ and $\mathcal{R}_{\text{diff}}$ respectively. An algorithm that carries out this traversal would have worst-case complexity

$$O\left(2|\mathcal{E}'_{\text{diff}} \cup \mathcal{E}_{\text{diff}}|\right),$$

where $\mathcal{E}_{\text{diff}}$ is the set of edges defined in (6.9) for graph G while the set $\mathcal{E}'_{\text{diff}}$ is defined similarly but for G'.

The complexity of such algorithm can be very close to that of the complete scheduling stage algorithm, which is $O(2n_{\text{edges}})$ in the worst case. Moreover, considering that the slowdown effect is canceled competely every time the analysis phase is repeated without update, an algorithm with a smaller computational complexity like Algorithm 6.3 is preferable.

Algorithm 6.3 has complexity $O(2|\mathcal{E}_{\text{diff}}|)$. Although this algorithm does not traverse the vertices in $\mathcal{R}'_{\text{diff}}$, it does traverse all vertices reachable from j_{diff} in the graph G. Consequently, it solves the problem shown in Figure 6.7c but not that shown in Figure 6.6. However, this algorithm does not take advantage of the cases like that in Figure 6.7a by reducing the part of graph traversed. Because of this, Algorithm 6.3 is not the algorithm with the lowest computational complexity among those possible, but it balances computational complexity and slowdown of the solve phase, if a limited slowdown is allowed.

Input: $\mathcal{T}'_{\text{int},1}, \mathcal{T}'_{\text{int},2}, \ldots, \mathcal{T}'_{\text{int},n'_{\text{int,slots}}}, n'_{\text{int,slots}},$
$\mathcal{T}'_{\text{ext},1}, \mathcal{T}'_{\text{ext},2}, \ldots, \mathcal{T}'_{\text{ext},n'_{\text{ext,slots}}}, n'_{\text{ext,slots}},$
$G_s := (\mathcal{V}_s, \mathcal{E}_s), j_{\text{diff}}$
Output: $\mathcal{T}_{\text{int},1}, \mathcal{T}_{\text{int},2}, \ldots, \mathcal{T}_{\text{int},n_{\text{int,slots}}}, n_{\text{int,slots}}, \mathcal{T}_{\text{ext},1}, \mathcal{T}_{\text{ext},2}, \ldots, \mathcal{T}_{\text{ext},n_{\text{ext,slots}}}, n_{\text{ext,slots}}$

$n_{\text{int,slots}} \leftarrow n'_{\text{int,slots}}$
$n_{\text{ext,slots}} \leftarrow n'_{\text{ext,slots}}$
$\mathcal{T}_{\text{int},k} \leftarrow \mathcal{T}'_{\text{int},k} \quad \forall k \in \{1, 2, \ldots, n_{\text{int,slots}}\}$
$\mathcal{T}_{\text{ext},k} \leftarrow \mathcal{T}'_{\text{ext},k} \quad \forall k \in \{1, 2, \ldots, n_{\text{ext,slots}}\}$
$V_{\text{cur}} \leftarrow$ empty
$V_{\text{next}} \leftarrow$ empty
$V_{\text{cur}} \leftarrow \text{put}(r, V_{\text{next}}) \quad \forall r \in \text{parents}(j_{\text{diff}})$
for all $v \in V_{\text{cur}}$
    for all $j \in \text{parents}(v) \mid j \notin \mathcal{V}_s$
        $\mathcal{T}_{\text{ext},1} \leftarrow \mathcal{T}_{\text{ext},1} \cup \{(v, j)\}$
    FIX_CONFLICTS$(v, G_s, \mathcal{T}_{\text{int},1}, \mathcal{T}_{\text{int},2}, \ldots, \mathcal{T}_{\text{int},n_{\text{int,slots}}} \mathcal{T}_{\text{ext},1}, \mathcal{T}_{\text{ext},2}, \ldots, \mathcal{T}_{\text{ext}})$
    for all $i \in \text{children}(v) \mid i \in \mathcal{V}_s$
        $\mathcal{T}_{\text{int},1} \leftarrow \mathcal{T}_{\text{int},1} \cup \{(i, v)\}$
        if $\nexists j \in \text{parents}(i) \mid (i, j) \notin \mathcal{T}_{\text{int},p} \wedge (i, j) \notin \mathcal{T}_{\text{ext},p} \quad \forall p$
            $V_{\text{next}} \leftarrow \text{put}(i, V_{\text{next}})$
while numel$(V_{\text{next}}) \neq 0$
    $V_{\text{cur}} \leftarrow V_{\text{next}}$
    $V_{\text{next}} \leftarrow$ empty
    for all $v \in V_{\text{cur}}$
        FIX_CONFLICTS$(v, G_s, \mathcal{T}_{\text{int},1}, \mathcal{T}_{\text{int},2}, \ldots, \mathcal{T}_{\text{int},n_{\text{int,slots}}}, \mathcal{T}_{\text{ext},1}, \mathcal{T}_{\text{ext},2}, \ldots, \mathcal{T}_{\text{ext},n_{\text{ext,slots}}})$
        $t \leftarrow 1 + \max\{q \in \mathbb{N} \mid \exists j \in \text{parents}(v) \wedge (v, j) \in \mathcal{T}_{\text{int},q}\}$
        for all $i \in \text{children}(v) \mid i \in \mathcal{V}_s$
            $\mathcal{T}_{\text{int},t} \leftarrow \mathcal{T}_{\text{int},t} \cup \{(i, v)\}$
            if $\nexists k \in \text{parents}(i) \mid (i, j) \notin \mathcal{T}_{\text{int},p} \wedge (i, j) \notin \mathcal{T}_{\text{ext},p} \quad \forall p$
                $V_{\text{next}} \leftarrow \text{put}(i, V_{\text{next}})$
terminate with success

Algorithm 6.3: Algorithm that updates the assignment of edges to time slots from a sub-graph $G'_s := (\mathcal{V}'_s, \mathcal{E}'_s)$ of graph G' to the sub-graph $G_s := (\mathcal{V}_s, \mathcal{E}_s)$ in the graph G. The graph G follows the graph G' in a sequence. The algorithm makes sure no read-write conflict occurs by using the function FIX_CONFLICTS in Algorithm 4.3 (page 110). The algorithm uses the containers of vertices V_{cur} and V_{next}, on which the *put* and *numel* operations are defined. In the pseudocode, $\mathcal{T}_{\text{int},p}$ and $\mathcal{T}_{\text{ext},p}$ are the p-th time slot of internal and external edges respectively, for sub-graph G_s. The sets parents(v) and children(v) are respectively the sets of the parents and children of vertex v in the graph G.

6.3 EXPERIMENTAL RESULTS

To evaluate the performance of the algorithms discussed in this chapter, some tests were carried out on a computer equipped with an Intel i3-2130 CPU [152],

which has two cores and a L3 cache memory of 3 MB, and an Nvidia Quadro K4000 GPU, with a peak double-precision performance of 1.27 TFLOPS [132].

Since the algorithms discussed in this chapter are meant to be used in numerical methods where the matrix changes at each iteration, like the simplex method, the test case selected for the experiments belonged to the MILP2010 collection of mixed-integer programming problems [13]. In particular, the test cases selected where the largest test cases in terms of number of non-zero entries, among those categorised as the most difficult to solve. For each of these test cases, the solver Scip [139] was executed. Scip solves mixed-integer optimisation problems by first dividing the original mixed-integer problem into linear programming sub-problems, and then executing the simplex method for each one of these sub-problems. This technique is also called *branch-and-bound*.

During the execution of Scip on each of the test cases, the number of rows and columns of the basic matrix was recorded. When these parameters stopped varying during the execution of Scip, the software was halted and the basic matrix B being used at the time was stored. Then, B was factorised with the LU decomposition [151] and the lower triangular factor L was taken, as it is done in the simplex method. Using the L factor, a sequence of 10 matrices was generated by randomly removing and adding non-zero entries in a different column of L such that the number of entries was unchanged. This was done to limit the number of parameters changing from one matrix in the sequence to the next, so that the changes in the execution time of the analysis and the solve phases were only due to the change in the sparsity pattern. Table 6.2 lists the average number of vertices and edges in each of the sequences obtained.

The experimental results are summarised in Table 6.3. As expected, the execution time of the analysis phase was reduced in all of the test cases considered when Algorithm 6.2 and Algorithm 6.3 were used. A further test was carried out for the test case *ivu52*, which was the test case with the most edges per vertex among those with the largest reduction in excution time. The aim of the test was to observe whether the execution time of the analysis phase was reduced in the case of more than one column changing from one matrix in the sequence to the next. In this test, denoted with *ivu52_multi*, a random number

Table 6.2: Summary of the test cases used for experimental evaluation. These are the largest test cases in the MIPLIB2010 collection [13] among those categorised as the most difficult to solve. For each of the test cases, the MIP solver Scip [139] was executed and the parameters of the basic matrices were recorded. In the table, n_{vertices} and n_{edges} represent the average number of vertices and edges in the graphs associated to the lower triangular part in the basic matrices.

Name	n_{vertices}	n_{edges}	$\frac{n_{\text{edges}}}{n_{\text{vertices}}}$
ivu52	1906	6841	3.589
neos-1140050	470	13 664	2.907×10^{1}
opm2-z12-s14	3575	3804	1.064
opm2-z12-s7	10 116	10 852	1.073
rail02	1369	1210	8.839×10^{-1}
rail03	1378	1258	9.129×10^{-1}
shs1023	171	149	8.713×10^{-1}
wnq-n100-mw99-14	6144	8006	1.303

Table 6.3: Summary of the experimental results. For each of the test cases considered the following are shown: the ratio of the execution time of the update to that of the analysis, and the ratio of the execution time of the solve phase after the update to that of the solve phase after the analysis.

Name	$\frac{t_{\text{analysis,update}}}{t_{\text{analysis}}}$	$\frac{t_{\text{solve,update}}}{t_{\text{solve}}}$
ivu52	0.400	1.012
neos-1140050	0.572	1.011
opm2-z12-s14	0.249	1.014
opm2-z12-s7	0.405	1.034
rail02	0.376	1.035
rail03	0.471	1.007
shs1023	0.105	1.007
wnq-n100-mw99-14	0.214	1.024
ivu52_multi	0.837	1.013

of entries was changed in each matrix in the sequence instead of only those in one column. For *ivu52_multi*, it was observed an increase of the analysis time because it was necessary to re-partition the graph more frequently. On average, the execution time of the update was 40 % of that of the analysis phase without

update. Moreover, for all the test cases, it was observed that the execution time of the solve phase increased on average of 1.74 %, because of the effect of the removal of edges explained in Section 6.2.1.

6.4 CONCLUSIONS FROM THIS CHAPTER

Solving a sequence of STLs on GPUs requires two elements: fast algorithms to update the output of the analysis phase and techniques to overcome the slowdown of the solve phase. The two elements are not always compatible because fast updates can cause an increase in the slowdown and a complete elimination of the slowdown can only be obtained if the analysis phase is repeated.

A good balance between the two elements can only be achieved by considering the sparsity pattern of each matrix in the sequence. If the sequence of STLs is solved during the LL-LU or SM algorithms, one expects the first few matrices to have few non-zero entries. As the sequence progresses, the number of non-zero entries increases but the repetition of the analysis phase is expected to be needed only rarely. Algorithm 6.1 and Algorithm 6.2 describe how to update the output of the analysis case under these assumptions, and their use can reduce the execution time of the analysis phase of more than a half, with respect to solving only one STL.

In the next chapter, it is described how TASSL's ability to deal with the case of sequences of STLs with different matrices, which was described in this chapter, can be used to implement the simplex method (SM). This method is largely used in optimisation, but in the scientific literature there is no parallel implementation that can be used to solve practical optimisation problems. Using the performance models described throughout the thesis, the next chapter also shows the limitations of any parallel implementation of the SM on practical test cases.

7

ACCELERATION OF THE SIMPLEX METHOD

This chapter discusses the acceleration of the simplex method (SM) [15, 153, ch. 13, 154, ch. 3] with GPUs and its limitations, using TASSL in particular. This application is of interest in the context of this thesis because it requires solving a sequence of sparse linear systems and is used both to solve linear programs (LPs) and mixed-integer programs (MIPs).

Section 7.1 illustrates the principles and implementations of the SM. This section is composed of three sections: Section 7.1.1 and Section 7.1.2 summarise some results known in the literature and describe how the SM is implemented, and Section 7.1.3 shows with experiments that solving sequences of sparse linear systems of equations is in practice the most time consuming part of the SM. Then, Section 7.2 discusses the acceleration of the SM by deriving a mathematical model of the execution time of SM iterations. Section 7.4 is a survey of the literature on the topic of the parallelisation of the simplex method. Finally, Section 7.3 summarises the experimental results and draws conclusions from the work in chapter.

The research contributions in this chapter are the following:

- the description of a GPU accelerator for the SM that was implemented with the Scip optimisation suite [139].
- A mathematical model that describes the execution time of the accelerator when the matrices in the sequence of linear systems are obtained with the Forrest-Tomlin algorithm [155, 156]. In this case, the model shows that accelerator performance is limited by the data transfer from the CPU to the GPU.
- A mathematical model that describes the execution time of the accelerator when the matrices in the sequence of linear systems are obtained with the elementary transform matrices algorithm [157, 158, ch. 2]. In this case, accelerator performance is limited by the number of SM iterations

between two re-factorisations of the matrices in the sequence of linear systems

7.1 THE SIMPLEX METHOD

The SM [15] is one of the most commonly used algorithms that solves linear programs (LPs), which are optimisation problems that can be written in the form

$$\min_{\boldsymbol{x}} \boldsymbol{c}^\intercal \boldsymbol{x} \tag{7.1a}$$

$$\text{subject to:} \quad A\boldsymbol{x} = \boldsymbol{b} \tag{7.1b}$$

$$\boldsymbol{x} \geq \boldsymbol{0} \tag{7.1c}$$

where A is a matrix in $\mathbb{R}^{m \times n}$ with full row rank and with $m < n$. Also, $\boldsymbol{x} = [x_1, x_2, \ldots, x_n]^\intercal \in \mathbb{R}^n$ is the *vector of decision variables*. LPs occur for example in network design and task scheduling [154].

This section explains background information on the SM that is necessary to understand the discussion in the following sections. Of this section, only Section 7.1.3 contains original work that was carried out for this thesis.

7.1.1 *Algorithm of the simplex method*

To solve (7.1), the SM uses the geometry of the constraints (7.1b) and (7.1c), and some results on optimisation [153, 158] that are recalled in this section.

Since in (7.1) matrix A has full row rank, it is possible to select m linearly independent columns $\boldsymbol{a}_{k_1}, \boldsymbol{a}_{k_2}, \ldots, \boldsymbol{a}_{k_m}$ and to define the *basic matrix* $B \in \mathbb{R}^{n \times n}$ as

$$B := \begin{bmatrix} | & | & & | \\ \boldsymbol{a}_{k_1} & \boldsymbol{a}_{k_2} & \cdots & \boldsymbol{a}_{k_m} \\ | & | & & | \end{bmatrix}.$$

The columns of A can then be permuted to obtain

$$A' \coloneqq [B \mid N], \tag{7.2}$$

where N is a matrix in $\mathbb{R}^{m\times(n-m)}$, and is composed of all the columns of A that are not in B. Also, one can define the index sets

$$\mathcal{B} \coloneqq \{k_1, k_2, \ldots, k_m\} \tag{7.3}$$
$$\mathcal{N} \coloneqq \{1, 2, \ldots, n\} \setminus \mathcal{B}, \tag{7.4}$$

where $\mathcal{B}$ is called the *basis*. By permuting the columns of A according to (7.2), (7.1) becomes

$$\begin{aligned} \min_{\boldsymbol{x'}} \ & (\boldsymbol{c'})^\intercal \boldsymbol{x'} \\ \text{subject to:} \quad & A'\boldsymbol{x'} = \boldsymbol{b} \\ & \boldsymbol{x'} \geq \boldsymbol{0}, \end{aligned}$$

from which, applying (7.2) and (7.3), and defining $\boldsymbol{x'} \coloneqq \left[\boldsymbol{x}_{\mathcal{B}}^\intercal, \boldsymbol{x}_{\mathcal{N}}^\intercal\right]^\intercal$ and $\boldsymbol{c'} \coloneqq \left[\boldsymbol{c}_{\mathcal{B}}^\intercal, \boldsymbol{c}_{\mathcal{N}}^\intercal\right]^\intercal$ we obtain

$$\min_{\left[\boldsymbol{x}_{\mathcal{B}}^\intercal, \boldsymbol{x}_{\mathcal{N}}^\intercal\right]^\intercal} \boldsymbol{c}_{\mathcal{B}}^\intercal \boldsymbol{x}_{\mathcal{B}} + \boldsymbol{c}_{\mathcal{N}}^\intercal \boldsymbol{x}_{\mathcal{N}} \tag{7.5a}$$
$$\text{subject to:} \quad B\boldsymbol{x}_{\mathcal{B}} + N\boldsymbol{x}_{\mathcal{N}} = \boldsymbol{b} \tag{7.5b}$$
$$\boldsymbol{x}_{\mathcal{B}} \geq \boldsymbol{0} \tag{7.5c}$$
$$\boldsymbol{x}_{\mathcal{N}} \geq \boldsymbol{0}. \tag{7.5d}$$

In (7.5), the vector $\boldsymbol{x}_{\mathcal{B}}$ is defined in $\mathbb{R}^m$ and is composed of the variables x_j such that $j \in \mathcal{B}$, which are called *basic variables*. Similarly, $\boldsymbol{x}_{\mathcal{N}}$ is defined in $\mathbb{R}^{(n-m)}$ and is composed of the variables x_j such that $j \in \mathcal{N}$, which are called *non-basic variables*. The vectors $\boldsymbol{c}_{\mathcal{B}}$ and $\boldsymbol{c}_{\mathcal{N}}$ are also defined in $\mathbb{R}^m$ and $\mathbb{R}^{(n-m)}$ respectively.

Since in (7.5b) the matrix B is square and has full rank, it is also invertible and so

$$\boldsymbol{x}_{\mathcal{B}} = B^{-1}\left(\boldsymbol{b} - N\boldsymbol{x}_{\mathcal{N}}\right). \tag{7.6}$$

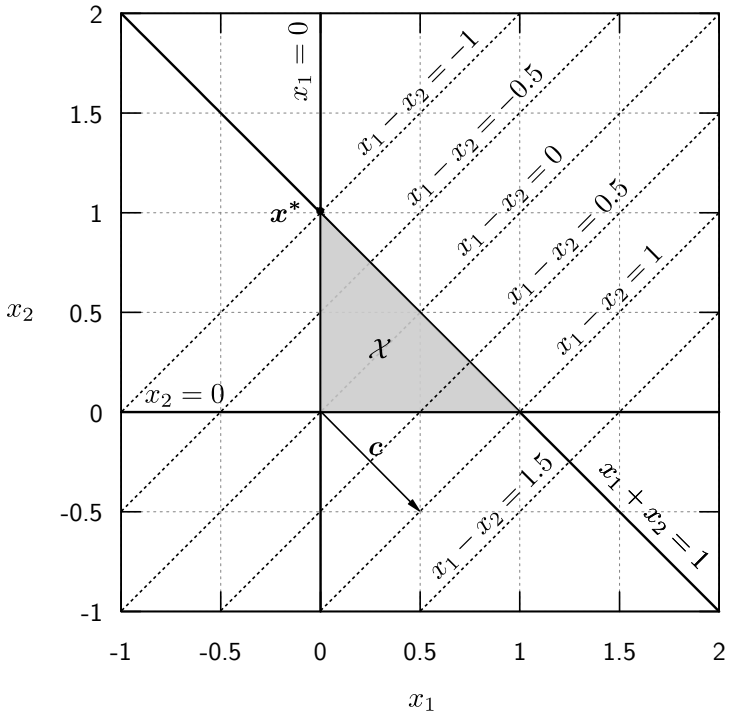


Figure 7.1: Representation of the feasible region for the example problem in (7.7). The optimal solution $\boldsymbol{x}^* = [1, 0]^\intercal$ to the LP falls at one of the vertices of the polyhedron described by $\mathcal{X}$.

If the vector $\boldsymbol{x}'$ is a feasible solution to the LP and also $\boldsymbol{x}_\mathcal{N} = \boldsymbol{0}$, then $\boldsymbol{x}'$ is called a *basic feasible solution*.

Basic feasible solutions have an important geometric meaning [154, ch. 3, 153, ch. 13, 158, ch. 1]. Consider the example LP

$$\begin{aligned} \min_{[x_1, x_2]} \; & x_1 - x_2 \\ \text{subject to:} \quad & x_1 + x_2 = 1 \\ & x_1 \geq 0 \\ & x_2 \geq 0, \end{aligned} \tag{7.7}$$

with feasible region represented in Figure 7.1. Each of the three basic feasible solutions corresponds to a vertex of the feasible region $\mathcal{X}$ and the optimal solution $\boldsymbol{x}^*$ is the one that minimises the goal function. The vector $\boldsymbol{c}$, which is the gradient of the goal function, points to the direction towards which the goal function grows.

The SM is initialised with a basic feasible solution and at each iteration a new basic feasible solution is computed by solving

$$\boldsymbol{x}_{\mathcal{B}} = B^{-1}\boldsymbol{b}, \tag{7.8}$$

until the optimal solution is reached. In terms of geometry, the SM moves from one vertex of the feasible region to the other until the optimal solution is reached.

A base version of the SM is shown in Algorithm 7.1. Another version of the algorithm, called the *dual simplex method* has a different mathematical derivation [154, ch. 6, 153, ch. 13, 158, ch. 2] but consists of the same numerical operations. In particular, the solution of the systems of linear equations with the basic matrix (lines 3 and 12) and its transpose (lines 8 and 27) can be computationally heavy. This issue is addressed with techniques that are based on LU factorisation.

7.1.2 *Use of LU factorisation*

Usually, a sparse LU factorisation of the basic matrix B is computed at the beginning of the SM and is updated at each iteration. There exists more than one algorithm to carry out each task.

The sparse LU factorisation with Markowitz pivoting [157, 158, ch. 8] is the factorisation algorithm implemented in the solver Soplex [159]. This algorithm aims to reduce the *fill-in*, which is the presence, in the factors, of entries that are not in the initial matrix, and estimates the fill-in with a merit number. Since the sparsity in the factors needs to be balanced with numerical stability, in some cases the LU factorisation is computed multiple times before achieving a satisfactory compromise.

In Soplex, the update of the factorisation can be carried out either with elementary transform matrices (ETM) [157, 158, ch. 8] or with the Forrest-Tomlin algorithm [155, 156].

Input: A, $\boldsymbol{b}$, $\boldsymbol{c}$, $\mathcal{B} \coloneqq \{k_1, k_2, \ldots, k_m\}$, $\mathcal{N} \coloneqq \{j_1, j_2, \ldots, j_{n-m}\}$
Output: $\boldsymbol{x}^*$

$B \leftarrow \left[\boldsymbol{a}_{\boldsymbol{k_1}}, \boldsymbol{a}_{\boldsymbol{k_2}}, \ldots, \boldsymbol{a}_{\boldsymbol{k_m}}\right]$
$N \leftarrow \left[\boldsymbol{a}_{\boldsymbol{j_1}}, \boldsymbol{a}_{\boldsymbol{j_2}}, \ldots, \boldsymbol{a}_{\boldsymbol{j_{n-m}}}\right]$
$\boldsymbol{x}_{\mathcal{B}} \leftarrow B^{-1}\boldsymbol{b}$
$\boldsymbol{x}_{\mathcal{N}} \leftarrow \boldsymbol{0}$
$\boldsymbol{c}_{\mathcal{B}} \leftarrow \left[c_{k_1}, c_{k_2}, \ldots, c_{k_m}\right]^\mathsf{T}$
$\boldsymbol{c}_{\mathcal{N}} \leftarrow \left[c_{j_1}, c_{j_2}, \ldots, c_{j_{n-m}}\right]^\mathsf{T}$
$z \leftarrow \boldsymbol{c}_{\mathcal{B}}^\mathsf{T}\boldsymbol{x}_{\mathcal{B}}$
$\boldsymbol{\pi} \leftarrow \left(\boldsymbol{c}_{\mathcal{B}}^\mathsf{T}B^{-1}\right)^\mathsf{T}$
$\boldsymbol{d} \leftarrow \boldsymbol{c}_{\mathcal{N}} - \boldsymbol{\pi}^\mathsf{T}N$
while $\boldsymbol{d} \geq \boldsymbol{0}$
    $q \leftarrow \text{price}(d)$
    $\boldsymbol{\alpha} \leftarrow B^{-1}\boldsymbol{a}_{\boldsymbol{q}}$
    $\boldsymbol{\beta} \leftarrow B^{-1}\boldsymbol{b}$
    $\mathcal{I} \leftarrow \{i \in \mathcal{B} \mid \alpha_i > 0\}$
    if $|\mathcal{I}| = 0$
        terminate with unbounded
    else
        $\theta \leftarrow \min_{i \in \mathcal{I}} \left\{\beta_{k_i}/\alpha_{k_i}\right\}$
        $k_p \leftarrow \operatorname{argmin}_{i \in \mathcal{I}} \left\{\beta_{k_i}/\alpha_{k_i}\right\}$
        $\boldsymbol{x}_{\mathcal{B}} \leftarrow \boldsymbol{x}_{\mathcal{B}} - \theta\boldsymbol{\alpha}$
        $\mathcal{B} \leftarrow \mathcal{B} \setminus \left\{k_p\right\} \cup \{q\}$
        $\mathcal{N} \leftarrow \mathcal{N} \setminus \{q\} \cup \left\{k_p\right\}$
        $B \leftarrow \left[\boldsymbol{a}_{\boldsymbol{k_1}}, \boldsymbol{a}_{\boldsymbol{k_2}}, \ldots, \boldsymbol{a}_{\boldsymbol{k_m}}\right]$
        $N \leftarrow \left[\boldsymbol{a}_{\boldsymbol{j_1}}, \boldsymbol{a}_{\boldsymbol{j_2}}, \ldots, \boldsymbol{a}_{\boldsymbol{j_{n-m}}}\right]$
        $\boldsymbol{c}_{\mathcal{B}} \leftarrow \left[c_{k_1}, c_{k_2}, \ldots, c_{k_m}\right]^\mathsf{T}$
        $\boldsymbol{c}_{\mathcal{N}} \leftarrow \left[c_{j_1}, c_{j_2}, \ldots, c_{j_{n-m}}\right]^\mathsf{T}$
        $\boldsymbol{\pi} \leftarrow \left(\boldsymbol{c}_{\mathcal{B}}^\mathsf{T}B^{-1}\right)^\mathsf{T}$
        $\boldsymbol{d} \leftarrow \boldsymbol{c}_{\mathcal{N}} - \boldsymbol{\pi}^\mathsf{T}N$
$\boldsymbol{x}^* \leftarrow \boldsymbol{c}_{\mathcal{B}}^\mathsf{T}B^{-1}$
terminate with success

Algorithm 7.1: A simplified version of the primal simplex method (SM). In the pseudocode, the matrix $A \in \mathbb{R}^{m \times n}$ with $n > m$ has full row rank, and the vectors $\boldsymbol{b}$ and $\boldsymbol{c}$ are defined in $\mathbb{R}^m$ and $\mathbb{R}^n$ respectively. The index set $\mathcal{B} \coloneqq \{k_1, k_2, \ldots, k_m\} \subset \{1, 2, \ldots, n\}$ is called the *basis*. The function price: $\mathbb{R}^{(n-m)} \to \mathcal{N}$ selects an element of the input vector $\boldsymbol{d}$ such that $d_j < 0$ and outputs its index j.

THE ETM ALGORITHM. To derive the ETM algorithm, consider a system of linear equations $B\boldsymbol{x} = \boldsymbol{b}$ arising during the SM, where B is the basic matrix. Applying a LU factorisation, one obtains

$$LU\boldsymbol{x} = \boldsymbol{b}, \tag{7.9}$$

where both matrices L and U are defined in $\mathbb{R}^{m\times m}$. At the following SM iteration, the basic matrix B' has the same columns as B except for the k_p-th, because k_p is the index of the leaving variable. This column is replaced with the column $\boldsymbol{a_q}$ of matrix A in (7.1), where q is the index of the entering variable. Hence, one obtains

$$B' \coloneqq \begin{bmatrix} | & | & & | & | & | & & | \\ \boldsymbol{a}_{k_1} & \boldsymbol{a}_{k_2} & \cdots & \boldsymbol{a}_{k_{p-1}} & \boldsymbol{a_q} & \boldsymbol{a}_{k_{p+1}} & \cdots & \boldsymbol{a}_{k_m} \\ | & | & & | & | & | & & | \end{bmatrix}. \tag{7.10}$$

The ETM algorithm [157, 158, ch. 2] consists of writing

$$B' = BE',$$

where the matrix $E' \in \mathbb{R}^{m\times m}$ is called an *elementary transform matrix*. Since all the columns in B are linearly independent by hypothesis, because B has full row rank, $\boldsymbol{a_q}$ can be written as

$$\boldsymbol{a_q} = \sum_{j=1}^{m} v_j \boldsymbol{a}_{k_j}\,, \tag{7.11}$$

where the coefficients v_j are real numbers. Inverting (7.11), we obtain

$$\boldsymbol{v} = B^{-1}\boldsymbol{a_q},$$

which, using (7.10), gives

$$B' = B \underbrace{\begin{bmatrix} | & | & & | & | & | & & | \\ \boldsymbol{e_1} & \boldsymbol{e_2} & \cdots & \boldsymbol{e_{p-1}} & \boldsymbol{v} & \boldsymbol{e_{p+1}} & \cdots & \boldsymbol{e_m} \\ | & | & & | & | & | & & | \end{bmatrix}}_{E'}, \tag{7.12}$$

where the entries in the generic vector $\boldsymbol{e_j}$ are all equal to zero apart from that of index j, which is equal to one.

In general, if $L^{(1)}$ and $U^{(1)}$ are the matrices obtained with the LU factorisation at the first SM iteration, after k iterations the ETM algorithm gives

$$B^{(k)} = L^{(1)}U^{(1)} \prod_{w=2}^{k} E^{(w)}. \tag{7.13}$$

Using (7.13), solving a system of linear equations with the basic matrix at the current iteration $B^{(k)}$ requires the execution of the following steps:

$$\boldsymbol{x} \leftarrow L^{-1}\boldsymbol{b'} \tag{7.14a}$$
$$\boldsymbol{x} \leftarrow U^{-1}\boldsymbol{x} \tag{7.14b}$$
$$\textbf{for } w \leftarrow 2, 3, \ldots, k$$
$$\boldsymbol{x} \leftarrow \left(E^{(w)}\right)^{-1}\boldsymbol{x}. \tag{7.14c}$$

While (7.14a) and (7.14b) are the solution of a generic STL, (7.14c) is the solution of a system whose matrix has the following sparsity pattern

$$E^{(w)} = \begin{bmatrix} 1 & & & v_1 & & \\ & 1 & & v_2 & & \\ & & \ddots & \vdots & & \\ & & & 1 \quad v_{p-1} & & \\ & & & v_p & & \\ & & & v_{p+1} \quad 1 & & \\ & & & \vdots & \ddots & \\ & & & v_m & & 1 \end{bmatrix}. \tag{7.15}$$

For example, (7.14c) can be solved by executing

$$x_{k_p} \leftarrow x_{k_p}/v_p$$
$$x_{k_j} \leftarrow x_{k_j} - v_j x_{k_p} \quad \forall j \neq p.$$

Using (7.13), $k-1$ systems of the type in (7.15) need to be solved for each system of equations of the type $B^{(k)}x = b$. The ETM algorithm is applied until the complexity of solving these $k-1$ systems is comparable to that of computing a new LU factorisation or until a very small v_{k_p} is found, which would cause numerical problems.

THE FORREST-TOMLIN ALGORITHM. The main disadvantage of the ETM algorithm is in the fact that the vector $\boldsymbol{v}$ in (7.12) can be dense, which limits the applicability of the algorithm to small LPs. Given the basic matrix $B = LU$, the

Forrest-Tomlin algorithm [155, 160] represents the basic matrix at the following SM iteration B' as

$$B' = LU + \left(\boldsymbol{a_q} - B\boldsymbol{e_p}\right)\boldsymbol{e_p}^\intercal$$

where q is the index of the entering variable and p is the index of the exiting variable in the basis. From this, requiring $L' = L$, we obtain

$$L^{-1}B' = U + \left(L^{-1}\boldsymbol{a_q} - U\boldsymbol{e_p}\right)\boldsymbol{e_p}^\intercal. \tag{7.16}$$

As shown in Figure 7.2a, the term $L^{-1}\boldsymbol{a_q}$ in (7.16) is a *column spike* in the sparsity pattern of $L^{-1}B'$. The Forrest-Tomlin algorithm first applies a transformation R that removes all the entries in the row k_p except the diagonal one such that the matrix $\hat{U}$ is obtained. By defining

$$\tilde{U} \coloneqq L^{-1}B = \begin{bmatrix} — & \tilde{\boldsymbol{u}}_{\text{row},\boldsymbol{1}} & — \\ — & \tilde{\boldsymbol{u}}_{\text{row},\boldsymbol{2}} & — \\ & \vdots & \\ — & \tilde{\boldsymbol{u}}_{\text{row},\boldsymbol{p-1}} & — \\ — & \tilde{\boldsymbol{u}}_{\text{row},\boldsymbol{p}} & — \\ — & \tilde{\boldsymbol{u}}_{\text{row},\boldsymbol{p+1}} & — \\ & \vdots & \\ — & \tilde{\boldsymbol{u}}_{\text{row},\boldsymbol{m}} & — \end{bmatrix}$$

then row p of matrix $\hat{U}$ can be written as

$$\left[0,\, 0,\, \ldots,\, 0,\, \hat{u}_{p,p},\, 0,\, \ldots,\, 0\right] = \sum_{i=1}^{m} r_i \tilde{\boldsymbol{u}}_{\text{row},\boldsymbol{i}}$$

since all the rows of $\tilde{U}$ are linearly independent by hypothesis. Then, we obtain

$$\hat{U} = \underbrace{\begin{bmatrix} — & \boldsymbol{e}_{\text{row},\mathbf{1}} & — \\ — & \boldsymbol{e}_{\text{row},\mathbf{2}} & — \\ & \vdots & \\ — & \boldsymbol{e}_{\text{row},\boldsymbol{p}-\mathbf{1}} & — \\ — & \boldsymbol{r} & — \\ — & \boldsymbol{e}_{\text{row},\boldsymbol{p}+\mathbf{1}} & — \\ & \vdots & \\ — & \boldsymbol{e}_{\text{row},\boldsymbol{m}} & — \end{bmatrix}}_{R} \tilde{U}$$

where the entries in the generic vector $\boldsymbol{e}_{\text{row},\boldsymbol{i}}$ are all equal to zero apart from that of index i, which is equal to one.

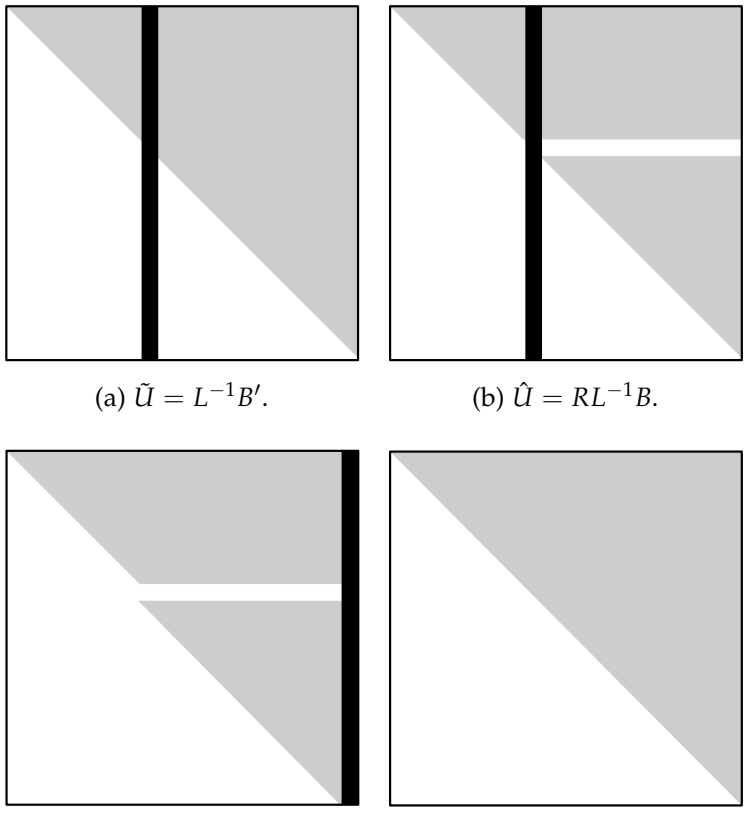
After applying the transformation R, a permutation represented by the matrix Q is applied to the columns of $L^{-1}B'$, moving the column spike to the last column. Finally, the same permutation is applied to the rows of the resulting matrix. The result at the end of this process is the upper triangular matrix $U' = Q^{-1}RL^{-1}B'Q$.

Soplex implements a version of the Forrest-Tomlin algorithm first introduced by Suhl [156]. Given the matrices obtained by the LU factorisation at the first SM iteration $L^{(1)}$ and $U^{(1)}$, Soplex represents the basic matrix $B^{(k)}$ at step k as

$$B^{(k)} = L^{(1)} \left(\prod_{w=2}^{k} R^{(w)} \right) P_r{}^{(1)} \hat{U}^{(k)} P_c{}^{(1)}, \tag{7.17}$$

where the matrix $\hat{U}^{(w)}$ is a *triangularisable* matrix because it can be permuted to a triangular matrix. For example, if $w = 2$ this matrix has a sparsity pattern as shown in Figure 7.2b. The column permutation $Q^{(w)}$ is used explicitly only when $B^{(w)}$ is accessed by column, for example when solving systems of the type $B^{(w)}\boldsymbol{x} = \boldsymbol{b}$. The row permutation $P_r{}^{(1)}$ and the column permutation $P_c{}^{(1)}$ are computed during the LU factorisation.

To solve a system of equations with the basic matrix $B^{(w)}$ using the representation in (7.17) requires the execution of the following steps:



(a) $\tilde{U} = L^{-1}B'$. (b) $\hat{U} = RL^{-1}B$.

(c) $RL^{-1}B'Q$. (d) $U' = Q^{-1}RL^{-1}B'Q$.

Figure 7.2: Working principle of the Forrest-Tomlin algorithm for updating the LU factorisation [155]. The algorithm implicitly builds the LU factorisation of matrix $B' = L'U'$ from that of matrix $B = LU$, requiring $L' = L$. In the figures at the top, the black column represents the spike $L^{-1}\boldsymbol{a_q}$. The Forrest-Tomlin algorithm first applies a transformation R that removes all the entries in the row k_p but the diagonal one. Then, a permutation represented by the matrix Q is applied to the columns of $L^{-1}B'$, moving the column spike to the last column. Finally, the same permutation is applied to the rows of the resulting matrix. The result at the end of this process is the upper triangular matrix $U' = Q^{-1}RL^{-1}B'Q$.

$$\boldsymbol{z} \leftarrow \left(L^{(1)}\right)^{-1}\boldsymbol{b'} \tag{7.18a}$$

for $w \leftarrow 2, 3, \ldots, k$

$$\quad \boldsymbol{z} \leftarrow \left(R^{(w)}\right)^{-1}\boldsymbol{z} \tag{7.18b}$$

$$\boldsymbol{x} \leftarrow P_c{}^{\intercal}\boldsymbol{z} \tag{7.18c}$$

$$\boldsymbol{x} \leftarrow \left(\hat{U}^{(k)}\right)^{-1}\boldsymbol{x} \tag{7.18d}$$

$$\boldsymbol{x} \leftarrow P_r{}^{\intercal}\boldsymbol{x}. \tag{7.18e}$$

While (7.18a) is the solution of a generic STL, each of (7.18b) is solution of a system whose matrix has the following sparsity pattern

$$R^{(w)} = \begin{bmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & \ddots & & & & & \\ & & & 1 & & & & \\ r_1 & r_2 & \dots & r_{p-1} & r_p & r_{p+1} & \dots & r_m \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix}.$$

For example, each of the (7.18b) can be solved by executing

$$z_{k_p} \leftarrow \frac{1}{r_p}\Big(z_{k_p} - \sum_{j \neq p} r_j z_{k_j}\Big). \tag{7.19}$$

Similarly to The ETM algorithm, the Forrest-Tomlin algorithm is applied until the complexity of solving these $k-1$ systems is comparable to that of computing a new LU factorisation or until a very small r_p is found.

Compared to the ETM algorithm, the Forrest-Tomlin algorithm is more numerically stable, but requires changing two matrices at each iteration: the matrix $R^{(k)}$ and the matrix $\hat{U}^{(k)}$. In practice the Forrest-Tomlin algorithm gives matrices R^w that are sparser than the matrices $E^{(w)}$ given by the ETM algorithm, so it is always the preferred update algorithm. Nevertheless, the time required to solve linear systems of equations with the basic matrix $B^{(k)}$ or with its transpose $\left(B^{(w)}\right)^\intercal$ is a bottleneck in the execution of the simplex method.

7.1.3 *The most time-consuming parts of the simplex method*

This section is original work that was carried out for this thesis.

To measure which parts of the SM are the most time-consuming, the bechmark LPs in the NETLIB library [12] were solved using Soplex 2.2.1, which was executed on a computer with an Intel i7 CPU [144], and 16 GB of RAM. A breakdown of the execution time was obtained with gprof [161]. Since the

resolution of gprof is of 10^{-2} seconds, only those test cases whose execution time t_{total} was more than one second were considered.

The experimental results, which are summarised in Table 7.1, show that in 10 of the 13 test cases selected, solving systems of equations with B (lines 3 and 12 of Algorithm 7.1) is usually more time consuming than solving systems of equations with B^{T} (lines 8 and 27 of Algorithm 7.1). The first, represented with t_B, accounted for between 21.5 % and 46.9 % of the execution time, while the second, represented with $t_{B^{\mathsf{T}}}$, accounted for between 5.7 % and 20.4 %. However, in the test case *stocfor3*, $t_{B^{\mathsf{T}}}$ accounted for 48.3 % of the execution time. Hence, if one wants to accelerate the solution of these systems of equations with a GPU, the solution of one type of system of equations cannot be preferred with respect to the other.

Also, if t_B and $t_{B^{\mathsf{T}}}$ are considered together, between 38.3 % (test case *pilot*) and 87.2 % (test case *stocfor3*) of the execution time was spent solving systems of linear equations. These relative execution times are higher than those for the factorisation of the basic matrix, represented with t_{factor}, and its update, represented with t_{update}. For example, although factorising the basic matrix can take up to 33.6 % of the execution time (test case *qap15*), on average this takes about 6 times less than solving linear systems with B. In fact, the geometric mean is 5.02 % for the LU factorisation and 28.7 % for solving linear systems with B. For this reason, just by accelerating the solution of systems of equations in the SM, one can expect the maximum possible speedup to be between 1.6 and 7.8, depending on the LP. The next section describes a model that links the characteristics of an LP to the speedup.

7.2 MODELING THE EXECUTION OF THE SIMPLEX METHOD ON THE GPU

The software Soplex 2.2.1 was modified to use either TASSL, CUSPARSE or CSPARSE to solve systems of linear equations with either the matrix B or its transpose. All the changes to the numerical code were applied within the part of the software which carries out all numerical operations on the basic matrix. Because of this, the interface remained unaltered, but different implementa-

Table 7.1: Breakdown of the execution time of Soplex 2.2.1 on the NETLIB test cases. The software was executed on a computer equipped with an Intel i7 CPU [144] and the measurements were taken with the software gprof [161]. Since the resolution of gprof is of 10^{-2} s, only those test cases whose execution time t_{total} was more than one second are considered. In the table, t_B and $t_{B^\intercal}$ are, respectively, the percentage of execution time spent solving systems of linear equations with the basic matrix B (lines 3 and 12 of Algorithm 7.1), and its transpose (lines 8 and 27 of Algorithm 7.1). Also, t_{update} and t_{factor} are, respectively, the percentage of execution time that is spent updating the basic matrix, and repeating the factorisation of the basic matrix.

Name	m	n	n_{entries}	t_{total} (s)	t_B (%)	$t_{B^\intercal}$ (%)	t_{update} (%)	t_{factor} (%)
qap15	6330	22 275	94 950	2.022×10^4	15.1	35.2	6.0	33.6
qap12	3192	8856	38 304	2.278×10^2	17.5	39.6	6.1	18.3
dfl001	5927	11 165	34 068	1.450×10^1	12.1	40.2	3.2	1.7
stocfor3	16 105	15 151	63 543	1.097×10^1	48.5	34.2	2.9	1.6
pilot87	1967	4587	70 372	6.223	18.2	25.0	3.4	19.4
greenbea	1858	3886	23 425	3.310	35.3	22.4	1.9	1.6
fit2p	3000	10 525	47 284	2.858	5.7	46.9	2.6	8.5
pilot	1373	3337	40 653	2.741	20.4	17.9	2.7	14.2
d2q06c	2020	4763	30 877	2.364	20.4	22.8	2.2	2.2
truss	1000	8806	27 836	2.345	10.4	23.8	2.6	1.7
greenbeb	1855	3875	23 324	1.640	18.0	25.9	1.3	2.5
maros-r7	2156	6620	80 480	1.507	17.4	21.5	5.9	12.2
qap8	912	1632	7296	1.226	11.6	33.1	3.2	1.8

tions could be selected at compilation time. The changes to the numerical code can be divided into three categories: loading of a new basic matrix, update of the basic matrix, and solution of systems of equations.

7.2.1 *Loading of a new basic matrix*

In Soplex, when a new basic matrix is loaded, memory is allocated if necessary and the sparse LU factorisation of the basic matrix is carried out on B using Markowitz pivoting [157, 158]. Given that on average six times less execution time is spent on the LU factorisation than on solving linear systems,

the numerical code for the LU factorisation was not modified. Instead, it was made sure that the output of the LU factorisation was read into either TASSL or CUSPARSE to carry out the analysis as described in Chapter 4.

To derive a model for the speedup of the TASSL implementation, let the positive integers $n_{\text{edges},\hat{U}}$ and $n_{\text{edges},L}$ be respectively the number of edges in the graphs of the factors $\hat{U}$ and L. The execution time of the analysis phase of each matrix is linear in n_{edges} and n_{vertices} because the complexity of the partitioning step is $O(n_{\text{vertices}} + n_{\text{edges}})$ and that of the scheduling step is $O(2n_{\text{edges}})$ as discussed in Chapter 4 (page 103 and page 111). If the number of vertices n_{vertices} is the same in both graphs, the execution time of the analysis of each matrix is

$$t_{\text{analysis},\hat{U}} \approx a_1 n_{\text{edges},\hat{U}} + a_2 n_{\text{vertices}} + o_{\text{analysis}} \tag{7.20}$$

$$t_{\text{analysis},L} \approx a_1 n_{\text{edges},L} + a_2 n_{\text{vertices}} + o_{\text{analysis}} \tag{7.21}$$

$$t_{\text{analysis},\hat{U}^\intercal} \approx a_1 n_{\text{edges},\hat{U}} + a_2 n_{\text{vertices}} + o_{\text{analysis}} \tag{7.22}$$

$$t_{\text{analysis},L^\intercal} \approx a_1 n_{\text{edges},L} + a_2 n_{\text{vertices}} + o_{\text{analysis}}, \tag{7.23}$$

where the positive real numbers a_1, a_2, and o_{analysis} vary depending on the structure of the graph, and consequently from LP to LP.

This model was evaluated on a collection of parametric optimisation problems called *smart-building-optimisation* [162, 163]. These are mixed-integer linear optimisation problems that describe a building composed of a number of flats, each one equipped with a battery and photovoltaic panels. Each of the flats can share the energy stored in the battery with the others, the battery can be recharged, and each of the flats can also produce energy with the photovoltaic panels. The objective of the optimisation problem is to minimise the most expensive electricity bill paid by any of the flats. For the purpose of this chapter, the two parameters that were changed were the number of flats and the number of hours of activity. The modified version of Soplex was executed on a server-class computer [146] equipped with an Intel Xeon E5-2640 v3 CPU, which has a L3 cache of 20 MB [140], and an Nvidia K80 GPU with a peak double-precision performance of 2.91 TFLOPS [136].

Since all the problems in *smart-building-optimisation* have the same structure, by varying the parameters it was possible to obtain the values of the coefficients in the performance model excluding all effects due to the change in the structure of the graphs. The parameters of the model in (7.20) were obtained using the non-negative least squares algorithm [134]. Because Soplex implements the Markowitz LU factorisation, it is assumed that $n_{\text{edges},\hat{U}} \gg n_{\text{edges},L}$. Substituting into (7.20), one gets

$$\begin{aligned} t_{\text{analysis}} &= t_{\text{analysis},\hat{U}} + t_{\text{analysis},L} + t_{\text{analysis},\hat{U}^\intercal} + t_{\text{analysis},L^\intercal} \\ &= 2a_1 n_{\text{edges},\hat{U}} + 4a_2 n_{\text{vertices}} + 4o_{\text{analysis}} \end{aligned} \tag{7.24}$$

After the analysis, the data is transferred onto the GPU. Each matrix contains n_{edges} edges and a data structure of n_{vertices} elements which represents a permutation of the rows and columns. Proceeding as in (7.24), the time to transfer each factor is

$$t_{\text{transfer},\hat{U}} \approx b_1 n_{\text{edges},\hat{U}} + b_2 n_{\text{vertices}} + o_{\text{transfer,mat}} \tag{7.25}$$

$$t_{\text{transfer},L} \approx b_1 n_{\text{edges},L} + b_2 n_{\text{vertices}} + o_{\text{transfer,mat}} \tag{7.26}$$

$$t_{\text{transfer},\hat{U}^\intercal} \approx b_1 n_{\text{edges},\hat{U}} + b_2 n_{\text{vertices}} + o_{\text{transfer,mat}} \tag{7.27}$$

$$t_{\text{transfer},L^\intercal} \approx b_1 n_{\text{edges},L} + b_2 n_{\text{vertices}} + o_{\text{transfer,mat}} \tag{7.28}$$

where b_1, b_2, and $o_{\text{transfer,mat}}$ are non-negative real numbers. In the modified numerical code, the row permutation P_r in (7.17) is also transferred to the GPU, while the column permutation P_c is resolved when reading the data onto either TASSL, CUSPARSE or CSPARSE. This is because only one column changes from one SM iteration to the next and resolving P_c while reading the data reduces the execution time of each SM iterations. The permutation P_r instead depends on the numerical stability of the factorisation and can cause many rows to be permuted at the same time, hence permuting these rows concurrently on the GPU can be advantageous. For P_c, the model is

$$t_{\text{transfer},P_r} \approx b_3 n_{\text{vertices}} + o_{\text{transfer,perm}} \tag{7.29}$$

where b_3 and $o_{\text{transfer,perm}}$ are non-negative real numbers. Summing the terms in (7.25) and (7.29), and considering that $n_{\text{edges},\hat{U}} \gg n_{\text{edges},L}$, the total time to transfer all the data is approximately

$$t_{\text{transfer,initial}} = 2b_1 n_{\text{edges},\hat{U}} + (4b_2 + b_3)\, n_{\text{vertices}} + o_{\text{transfer,perm}} + 4o_{\text{transfer,mat}}. \tag{7.30}$$

7.2.2 *Update of the basic matrix*

In Soplex, the matices $\hat{U}^{(k)}$ and $R^{(k)}$ are updated as in (7.17) when the Forrest-Tomlin algorithm is used, while the matrix $E^{(w)}$ is updated as in (7.13) if the ETM algorithm is used.

When the Forrest-Tomlin algorithm is used, if the matrix $\hat{U}^{(k)}$ is not re-factorised but just updated, the analysis can be updated too as described in Chapter 6. The execution time of the update depends on the sparsity pattern of each matrix and the edges that change in each graph during the update. For example, depending on whether an output edge is added to the vertex i_1 or to the vertex i_2 in Figure 7.3, respectively, five or only one edge have to be traversed by the update algorithm. In the first case, this is 45 % of all the edges in the graph, in the latter this is 9 %. In the worst case, a complete analysis has to be carried out instead of an update.

In practice, as the experimental data shows, the time to update the analysis of the factor $\hat{U}$, which is $t_{\text{update},\hat{U}}$, is between 35 % and 58 % of $t_{\text{analysis},\hat{U}}$. Because of the Forrest-Tomlin algorithm, only the analysis $\hat{U}$ and $\hat{U}^{\intercal}$ are updated, so the terms to consider are

$$\begin{aligned} t_{\text{update},\hat{U}} &\approx c_1 n_{\text{edges},\hat{U}} + c_2 n_{\text{vertices}} + o_{\text{update}} \\ t_{\text{update},\hat{U}^{\intercal}} &\approx c_1 n_{\text{edges},\hat{U}} + c_2 n_{\text{vertices}} + o_{\text{update}}, \end{aligned}$$

where c_1, c_2, and $o_{\text{update,mat}}$ are non-negative real number. The total update time is given by

$$t_{\text{update}} = 2c_1 n_{\text{edges},\hat{U}} + 2c_2 n_{\text{vertices}} + 2o_{\text{update}}. \tag{7.31}$$

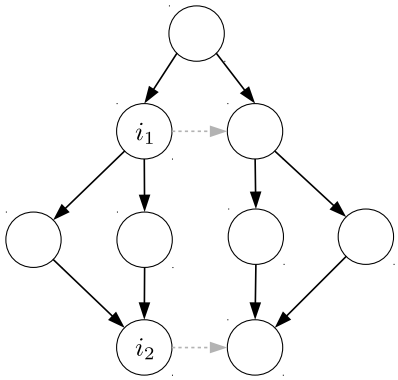


Figure 7.3: An example of how the update time t_{update} changes depending on which edges change. In this figure, if an output edge is added to vertex i_1, that edge and four more edges have to be traversed by the update algorithm. If the output edge is added to vertex i_2, only that edge has to be traversed by the update algorithm.

Also, at the end of the update, the data is to be moved to the GPU again, hence, using (7.25)

$$t_{\text{transfer,update}} = 2b_1 n_{\text{edges},\hat{U}} + 2b_2 n_{\text{vertices}} + 2o_{\text{transfer,mat}}. \tag{7.32}$$

7.2.3 *Solution of systems of equations*

We shall consider the solution of a system of linear equations with the basic matrix expressed in the form in (7.17). The solution can be carried out with the steps in (7.18). Of these steps, in the modified version of Soplex that uses TASSL or CUSPARSE only (7.18b) is executed on the CPU as shown in Figure 7.4. Since from (7.19) all the numerical operations with the $R^{(w)}$ matrices are to be executed in sequence, there is no opportunity for concurrency and using a GPU would not improve performance in this case. Using a GPU would, however, reduce the number of data transfers back and forth between GPU and CPU, although it would require transferring the $R^{(w)}$ matrices to the GPU after each update.

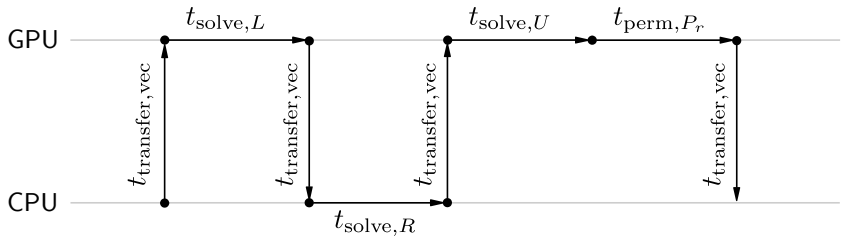


Figure 7.4: Solution of a system of linear equations with the basic matrix B, using the TASSL implementation. The representation of B is as in (7.17) except in that the column permutation P_c is resolved when the data is read to TASSL. In this diagram, the transfer time of a vector $t_{\text{transfer,vec}}$ is approximately the same whether the data is transferred from the CPU to the GPU or vice-versa.

Using the diagram in Figure 7.4, the time to solve a system of equations of the type $B\boldsymbol{x} = \boldsymbol{b}$ has the following components: the time required to transfer data from and to the GPU is equal to $t_{\text{transfer,vec}}$; the time required to solve STLs with the L and $\hat{U}$ factors, which are $t_{\text{solve},L}$ and $t_{\text{solve},\hat{U}}$ respectively; the time to solve all the systems of equations with the $R^{(w)}$ matrices, which is $t_{\text{solve},R}$; and the time to permute an array using P_r, which is t_{perm}. This gives

$$t_{\text{solve},B} \approx 4t_{\text{transfer,vec}} + t_{\text{solve},L} + t_{\text{solve},\hat{U}} + t_{\text{solve},R} + t_{\text{perm}}. \tag{7.33}$$

The term in t_{perm} in (7.33) depends on the size of the array being permuted as follows

$$t_{\text{perm}} \approx d_1 n_{\text{vertices}} + o_{\text{perm}}, \tag{7.34}$$

where d_1 and o_{perm} are non-negative real numbers.

The terms $t_{\text{solve},L}$ and $t_{\text{solve},\hat{U}}$ can be written using the performance model (3.30) (page 83) and including the time to permute the input array to the format used by TASSL (page 76), which gives

$$\begin{aligned} t_{\text{solve},L} &\approx (1 - C_L)\tau_L n_{\text{edges},L} + d_2 n_{\text{vertices}} + o_{\text{solve}} \\ t_{\text{solve},\hat{U}} &\approx (1 - C_{\hat{U}})\tau_{\hat{U}} n_{\text{edges},\hat{U}} + d_2 n_{\text{vertices}} + o_{\text{solve}} \end{aligned} \tag{7.35}$$

where d_2 is the rate of permutation of array elements on the GPU and o_{solve} is a positive real constant. In (7.35), $C_{\hat{U}}$ is the concurrency factor of matrix $\hat{U}$

with respect to the solve phase of TASSL, which is defined in (3.25) (page 82). Similarly, C_L is the concurrency factor of matrix L. The concurrency factor is a value between zero and one. Given a matrix M, if C_M is zero, the TASSL analysis algorithm assigned each edge to a separate time slot, which means the execution of the TASSL solution algorithm shows no concurrency. Instead, if C_M is close to one, the TASSL analysis algorithm assigned many edges to the same time slot, which means the execution of the TASSL solution algorithm shown high concurrency. In (7.35), $\tau_{\hat{U}}$ and τ_L are the effective operation time of the solve phase of TASSL on the matrices $\hat{U}$ and L respectively. The effective operation time is defined in (3.30) (page 82).

The term $t_{\text{transfer,vec}}$ in (7.33) represents the time necessary to transfer an array of n_{vertices} elements from the CPU to the GPU or vice versa. Similarly to (7.29), it is

$$t_{\text{transfer,vec}} \approx d_3 n_{\text{vertices}} + o_{\text{transfer,vec}}, \tag{7.36}$$

where d_3 and $o_{\text{transfer,vec}}$ are non-negative real numbers.

As represented in the diagram in Figure 7.4, the terms $t_{\text{solve},R}$ in (7.33) is the execution time of (7.18b) on the CPU. The Forrest-Tomlin algorithm gives matrices $R^{(w)}$ that have few entries, hence executing the numerical operations in (7.19) on the GPU woule be little profitable because it would require transferring the matrix to the GPU. For this reason, (7.18b) is solved on the CPU. The experimental results show that $t_{\text{solve},R}$ is between 2.640×10^{-7} s and 3.328×10^{-5} s, which is at least an order of magnitude less than $t_{\text{transfer,vec}}$. Because of this, the term $t_{\text{solve},R}$ in (7.33) shall be neglected.

Substituting (7.34), (7.36) in (7.33), given $n_{\text{edges},\hat{U}} \gg n_{\text{edges},L}$, it results

$$\begin{aligned} t_{\text{solve},B} = {} & \tau_{\hat{U}} \left(1 - C_{\hat{U}}\right) n_{\text{edges},\hat{U}} \\ & + \left(d_1 + 2d_2 + 16d_3\right) n_{\text{vertices}} \\ & + 2o_{\text{solve}} + 4o_{\text{transfer,vec}} + o_{\text{perm}}, \end{aligned} \tag{7.37}$$

and, for the matrix $B^{\intercal}$

$$\begin{aligned} t_{\text{solve},B^\intercal} = {} & \tau_{\hat{U}^\intercal} \left(1 - C_{\hat{U}^\intercal}\right) n_{\text{edges},\hat{U}} \\ & + \left(d_1 + 2d_2 + 16d_3\right) n_{\text{vertices}} \\ & + 2o_{\text{solve}} + 4o_{\text{transfer,vec}} + o_{\text{perm}} \end{aligned} \tag{7.38}$$

7.2.4 *Conditions for speedup with the Forrest-Tomlin algorithm*

With the unmodified Soplex, if $n_{\text{edges},\hat{U}} \gg n_{\text{edges},L}$ the execution time for the solution of a system of linear equations with matrix B or $B^\intercal$ is given by

$$\tilde{t}_{\text{solve},B} \approx \tilde{d}_1 n_{\text{edges},\hat{U}} + \tilde{o}_{\text{solve}}, \tag{7.39}$$

where $\tilde{d}_1$ and $\tilde{o}_{\text{solve}}$ are non-negative real numbers. The total execution time for the unmodified Soplex is given by the sum of $2\tilde{t}_{\text{solve},B}$ over all the iterations of the SM, because at each iteration two systems of linear equations are solved: one with B and one with $B^\intercal$. If the number of edges of $\hat{U}^{(k)}$ does not change much from one iteration to the other, one gets

$$t_{\text{soplex}} = 2n_{\text{iterations}}\tilde{t}_{\text{solve},B} = 2n_{\text{iterations}}\tilde{d}_1 n_{\text{edges},\hat{U}} + 2n_{\text{iterations}}\tilde{o}_{\text{solve}}. \tag{7.40}$$

With the modified version of Soplex that uses TASSL, under the same assumptions as those in (7.40), the total execution time over $n_{\text{iterations}}$ is given by

$$\begin{aligned} t_{\text{tassl}} = {} & t_{\text{analysis}} + t_{\text{transfer,initial}} \\ & + \left(n_{\text{iterations}} - 1\right) \left(t_{\text{update}} + t_{\text{transfer,update}}\right) \\ & + n_{\text{iterations}} \left(t_{\text{solve},B} + t_{\text{solve},B^\intercal}\right) \end{aligned}$$

and substituting (7.24), (7.30), (7.31), (7.32), (7.37), and (7.38) one obtains

$$\begin{aligned} t_{\text{tassl}} = &\left[2(d_1 + 16d_3 + 2d_2 + b_2 + c_2)n_{\text{iterations}} - 2c_2 + b_3 + 2b_2 + 4a_2\right] n_{\text{vertices}} \\ &+ \left\{ n_{\text{iterations}} \left[2\left(b_1 + c_1\right) - \left(C_{\hat{U}^\intercal} - 1\right)\tau_{\hat{U}^\intercal} - \left(C_{\hat{U}} - 1\right)\tau_{\hat{U}}\right] - 2\left(c_1 - a_1\right)\right\} n_{\text{edges},\hat{U}} \\ &+ 2n_{\text{iterations}}(o_{\text{update}} + 4o_{\text{transfer,vec}} + o_{\text{transfer,mat}} + 2o_{\text{solve}} + o_{\text{perm}}) \\ &- 2o_{\text{update}} + o_{\text{transfer,perm}} + 2o_{\text{transfer,mat}} + 4o_{\text{analysis}}. \end{aligned} \tag{7.41}$$

The speedup of the TASSL implementation over the unmodified soplex is

$$S_{\text{tassl}} = \frac{t_{\text{soplex}}}{t_{\text{tassl}}}. \tag{7.42}$$

To study the speedup, we shall observe that the number of edges in the matrix $\hat{U}$ can be written as $n_{\text{edges},\hat{U}} = \delta n_{\text{vertices}}^2$, where δ is the density of the entries in the matrix, equal to 1.531×10^{-3} in the experimental data but strongly dependent on the LP problem. Substituting (7.40) and (7.41) in (7.42) and considering very large values of n_{vertices}, one gets

$$S_{\text{tassl}} = \frac{2n_{\text{iterations}}\tilde{d_1}}{n_{\text{iterations}}\left[2\left(c_1 + b_1\right) + \left(1 - C_{\hat{U}^\intercal}\right)\tau_{\hat{U}^\intercal} + \left(1 - C_{\hat{U}}\right)\tau_{\hat{U}}\right] - 2\left(c_1 - a_1\right)},$$

which becomes, assuming the execution time of both the analysis and the update to be mainly given by the partitioning of graphs,

$$S_{\text{tassl}} \approx \frac{2\tilde{d_1}}{2b_1 + \left(1 - C_{\hat{U}^\intercal}\right)\tau_{\hat{U}^\intercal} + \left(1 - C_{\hat{U}}\right)\tau_{\hat{U}}}. \tag{7.43}$$

Although the speedup is higher when the concurrency factors of $\hat{U}$ and its transpose are close to one, (7.43) shows that the speedup is limited by the bandwidth of data transfer to the GPU, since the term b_1 in (7.43) arises from the transfer of $\hat{U}$ after the update. In the ideal case $C_{\hat{U}} = 1$ and $C_{\hat{U}^\intercal} = 1$ and $S_{\text{tassl}} = \tilde{d_1}/b_1$, which, using the experimental data, is about 5.549×10^{-2}. Also, with the Forrest-Tomlin algorithm, the speedup does not depend on the number of iterations.

7.2.5 *Conditions for speedup with the ETM algorithm*

If instead of the Forrest-Tomlin algorithm one considers the ETM algorithm in (7.13), no matrix needs to be updated or transferred to the GPU at each SM iteration, hence

$$t'_{\text{tassl}} = t_{\text{analysis}} + t_{\text{transfer,initial}} + n_{\text{iterations}} \left(t_{\text{solve},B} + t_{\text{solve},B^\intercal} \right),$$

and by proceeding as for (7.43) one finally gets

$$S'_{\text{tassl}} = \frac{2\tilde{d}_1 n_{\text{iterations}}}{2b_1 + \left[\left(1 - C_{U^\intercal}\right) \tau_{U^\intercal} + \left(1 - C_U\right) \tau_U \right] n_{\text{iterations}}}. \tag{7.44}$$

Note that (7.44) shows that if the number of iterations is high enough, the cost of the analysis is amortised and speedup can be achieved. In particular, in the ideal case $C_U = 1$ and $C_{U^\intercal} = 1$ and $S'_{\text{tassl}} = \tilde{d}_1 n_{\text{iterations}} / b_1$ hence, using the experimental data, speedup can be achieved if $n_{\text{iterations}}$ is more than 19. If the number of iterations is very high, (7.44) gives $S'_{\text{tassl}} \approx \frac{2\tilde{d}_1}{\left(1-C_{U^\intercal}\right)\tau_{U^\intercal} + \left(1-C_U\right)\tau_U}$, which shows that in order to obtain speedup it is necessary to both amortise the cost of the analysis with many iterations and have enough concurrency in the execution of the solution algorithm.

Efficient implementations of the simplex method, such as Soplex [159] do not use the ETM algorithm by default. In fact, the Forrest-Tomlin algorithm, whose speedup or slowdown is shown in (7.43), is preferred to the ETM algorithm because it tends to achieve a sparser LU factorisation and can be applied to larger problems. For this reason, compared to using the ETM algorithm, discussing the Forrest-Tomlin algorithm can give a more representative picture of the problems in accelerating the simplex method in practical use cases. The next section examines these problems focusing on the Forrest-Tomlin algorithm by means of an experimental evaluation.

Table 7.2: Model coefficients obtained from the experimental results over a collection of parametric optimisation problems called *smart-building-optimisation* [162, 163], which describe large smart buildings in operation for a few days up to two weeks. The data was obtained by executing a modified version of Scip [139, 159] that uses the TASSL algorithms on a computer [146] equipped with an Intel Xeon E5-2640 v3 CPU [140], and an Nvidia K80 GPU [136].

Model	Parameter	Value
t_{analysis} (7.24)	a_1	2.519×10^{-6} s/edge
	a_2	4.702×10^{-7} s/vertex
	o_{analysis}	0 s
$t_{\text{transfer,initial}}$ (7.30) $t_{\text{transfer,update}}$ (7.32)	b_1	1.690×10^{-8} s/edge
	b_2	2.870×10^{-8} s/vertex
	b_3	5.964×10^{-8} s/vertex
	$o_{\text{transfer,mat}}$	8.3863×10^{-5} s
	$o_{\text{transfer,perm}}$	2.785×10^{-5} s
t_{update} (7.31)	c_1	0 s/edge
	c_2	1.014×10^{-6} s/vertex
	o_{update}	0 s
$t_{\text{solve},B}$ (7.37) $t_{\text{solve},B^\intercal}$ (7.38)	d_1	1.879×10^{-8} s/vertex
	d_2	5.713×10^{-9} s/vertex
	d_3	6.216×10^{-9} s/vertex
	$\tau_{\hat{U}}$	7.022×10^{-8} s/time slot
	$\tau_{\hat{U}^\intercal}$	7.0781×10^{-8} s/time slot
	o_{perm}	1.266×10^{-4} s
	o_{solve}	2.817×10^{-4} s
	$o_{\text{transfer,vec}}$	4.287×10^{-4} s
t_{soplex} (7.40)	$\tilde{d}_1$	2.286×10^{-12} s/edge
	$\tilde{o}_{\text{solve}}$	1.048×10^{-5} s

7.3 EXPERIMENTAL RESULTS IN DETAIL

Table 7.2 summarises the values of the speedup model in (7.43) obtained using a collection of parametric optimisation problems called *smart-building-optimisation* [162, 163] and the non-negative least squares algorithm implementation in Matlab [85, 134]. To obtain these values, a modified version of Soplex that uses the TASSL algorithms was executed on a computer [146] equipped with an Intel

Table 7.3: Mean values of the problem parameters for the experimental results. These optimisation problems were taken from the benchmark *smart-building-optimisation* [162, 163] considering models of large smart buildings in operation for a few days up to two weeks. The data was obtained by executing a modified version of Scip [139, 159] that uses the TASSL algorithms on a computer [146] equipped with an Intel Xeon E5-2640 v3 CPU [140], and an Nvidia K80 GPU [136].

Problem parameter	Mean value
$n_{\text{edges},L}$	0
$n_{\text{edges},\hat{U}}$	10195
n_{vertices}	7489.7
$C_{\hat{U}}$	9.731×10^{-1}
$C_{\hat{U}^\intercal}$	9.504×10^{-1}
$n_{\text{iterations}}$	100
δ	1.500×10^{-7}

Xeon E5-2640 v3 CPU [140], and an Nvidia K80 GPU [136]. Two parameters of the problems in *smart-building-optimisation* were changed during the tests: the number of hours of activity of the smart building and the number of flats in the smart building. The first parameter was set to 72, 168 and 336, while the second was set to 64, 80 and 160. These parameters describe optimisation problems for quite large-sized smart buildings in activity from a few days up to two weeks. In total, twelve optimisation problems were solved and for each of these optimisation problems, the Scip optimisation suite was executed with all the pre-solving routines disabled. Although these routines reduce problem size using algebraic transformations and by fixing variables [164], they optimise the execution time of Soplex on single-core CPUs without considering other architectures.

Table 7.3 shows the mean values of the parameters of interest for the speedup model (7.43) that were measured during the tests. The experiments confirmed that the number of edges of the L factor, $n_{\text{edges},L}$ is usually negligible compared to that of the $\hat{U}$ factor, $n_{\text{edges},\hat{U}}$. Also, the density δ of the matrices has an order of magnitude of -7.

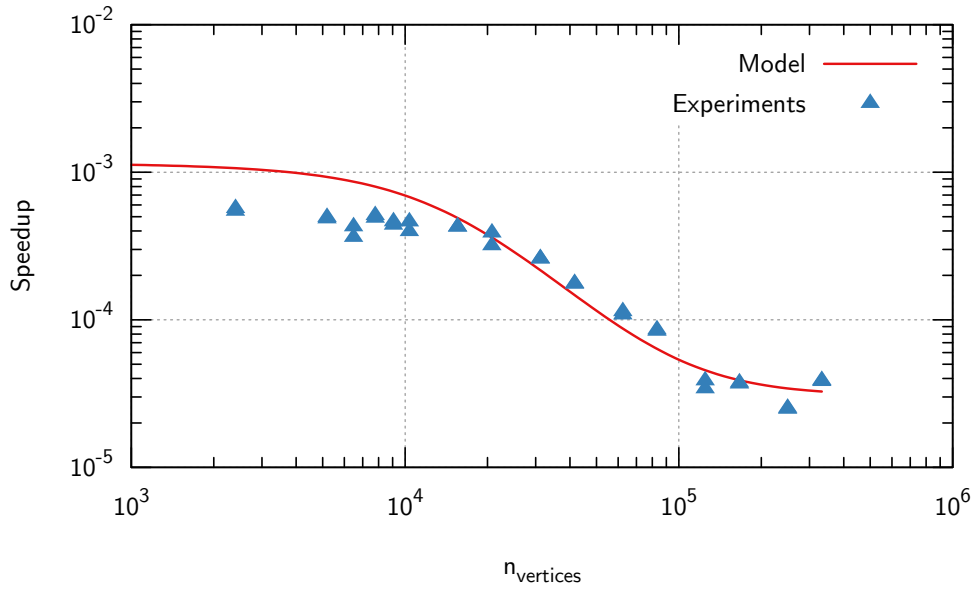


Figure 7.5: Comparison between the speedup model in (7.43) and the experimental results obtained using a collection of parametric optimisation problems called *smart-building-optimisation* [162, 163]. The model used is that in and the data used is in Table 7.2 and in Table 7.3. Also, the speedup measured experimentally is also shown.

The speedup model (7.42) is represented in Figure 7.5, using the data in Table 7.2 and in Table 7.3 for the model, together with the speedup measured experimentally. The model correctly predicts a slowdown compared to the original Scip implementation. The experiments show that before n_{vertices} is about equal to 2×10^4, the model overestimates the speedup, which is approximately given by the ratio of the constant terms at the numerator and denumerator of (7.42). Between 2×10^4 and 1×10^5 vertices, the overheads of partitioning update and data transfer in the TASSL implementation cause a reduction of the speedup which is predicted by the model and captured by the coefficients of n_{vertices} at the denumerator of (7.42). After 1×10^5 vertices, the speedup is given by the ratio of the coefficients of $n_{\text{edges}} \approx \delta n_{\text{vertices}}^2$ at the numerator and denumerator of (7.42), as also illustrated in (7.43).

Table 7.2 shows that the parameter $\tilde{d}_1$, which describes the time required to execute a numerical operation on the single-core CPU, is several orders of magnitude smaller than the parameter $\tau_{\mathcal{U}}$, which has a similar meaning for the TASSL implementation. This is because Soplex uses a specialised version

of the forward substitution algorithm when the right-hand side vector of a STL is sparse [25], for example a column of a very sparse matrix as in lines 3 and 12 of Algorithm 7.1. This version of the algorithm carries out only a partial traversal of the DAG associated with the system matrix, thus greatly reducing the execution time of the solve.

If a fraction r of edges is traversed by this specialised algorithm, (7.39) can be rewritten as

$$\tilde{t}_{\text{solve},B} \approx \tilde{d}_{1,r} r n_{\text{edges},\hat{U}} + \tilde{o}_{\text{solve}}, \tag{7.45}$$

where $\tilde{d}_{1,r} := \tilde{d}_1 / r$. A TASSL version of this algorithm would traverse this fraction of edges in parallel, in a time given by

$$t_{\text{solve},\hat{U},r} \approx \tau_{\hat{U}} \left(1 - C_{\hat{U},r}\right) r n_{\text{edges},U} + d_2 n_{\text{vertices}} + o_{\text{solve}}. \tag{7.46}$$

If the external and internal time slots of sub-graph g that are traversed by this TASSL version are $l'_{g,r}$ and $l_{g,r}$ respectively, then the concurrency factor of the matrix $\hat{U}$ with respect to the TASSL version of the specialised algorithm can be defined similarly to (3.25) (page 82)

$$C_{\hat{U},r} := 1 - \frac{\sum_{q=1}^{n_{\text{levels},\hat{U}}} \left(l'_{\tilde{q},r} + l_{\tilde{q},r}\right)}{r n_{\text{edges},\hat{U}}} \tag{7.47}$$

having denoted with $G_{\tilde{q}}$ the sub-graph in level $\mathcal{L}_q$ with the longest processing time during the solve phase. Note that (7.47) shows that, with respect to the algorithm discussed in Chapter 3 (page 78), a lower concurrency is to be expected for the specialised algorithm. This can be explained intuitively by observing that in practice if few edges are to be traversed it is less likely that these can be traversed concurrently. For example, if only two edges are to be traversed, the concurrency factor cannot be greater than $1 - 1/2 = 0.5$. If three edges are to be traversed the concurrency factor cannot be greater than $1 - 1/3 \approx 0.667$. However, if a hundred edges are to be traversed, then the maximum value of the concurrency factor is 0.99. Specifically, the concurrency factor $C_{\hat{U},r}$ is less than the concurrency factor $C_{\hat{U}}$ if

$$r < \frac{\sum_{q=1}^{n_{\text{levels},\hat{U}}} \left(l'_{\tilde{q},r} + l_{\tilde{q},r} \right)}{\sum_{q=1}^{n_{\text{levels},\hat{U}}} \left(l'_{\tilde{q}} + l_{\tilde{q}} \right)}.$$

Using (7.46) and (7.45) to derive the execution times $t_{\text{tassl},r}$ and t_{soplex}, the speedup for large values of $n_{\text{edges},\hat{U}}$ is

$$S_{\text{tassl},r} = \frac{2 n_{\text{iterations}} \tilde{d}_{1,r}}{n_{\text{iterations}} \left[\tau_{\hat{U}} \left(1 - C_{\hat{U},r} \right) + \tau_{\hat{U}^\intercal} \left(1 - C_{\hat{U}^\intercal,r} \right) + 2 r b_1 \right] + 2 r a_2},$$

which, if r is very small, can be approximated with

$$S_{\text{tassl},r} \approx \frac{2 \tilde{d}_{1,r}}{\tau_{\hat{U}} \left(1 - C_{\hat{U},r} \right) + \tau_{\hat{U}^\intercal} \left(1 - C_{\hat{U}^\intercal,r} \right)}. \tag{7.48}$$

Although discussed only in theory in this thesis, (7.48) shows that the limitations of (7.43), due to the data transfer, and of (7.44), due to the number of iterations required to amortise the cost of the analysis, can both be overcome in a particular case. However, in (7.48) the parameters $\tau_{\hat{U}}$ and $\tilde{d}_{1,r}$ have at least two orders of magnitude of difference because of the difference in the clock frequencies of GPUs and CPUs. Hence, to achieve speedup, the concurrency factor in (7.47) must be high, which means about a hundred edges have to be traversed by the specialised algorithm.

7.4 RELATED WORK

In this section, an overview of the existing implementations of optimisation algorithms on accelerators is given, as summarised in Table 7.4 and Table 7.5. Table 7.6 illustrates some of the features of GPU implementations in particular.

While Koch *et al.* discussed the parallelisation of optimisation algorithms for LP and MIP in general [186], Boyer and El Baz's survey focused on GPU implementations of optimisation algorithms but covered different types of problems and algorithms [187]. Some of the most relevant work on GPU solvers was developed by Lalami *et al.* who introduced a software implementation of the dense SM, also called the tableau simplex method for single-GPU [172] and

Table 7.4: Classification of some implementations of optimisation algorithms in the literature based on their features.

Paper	Year	Algorithm	Type	Sparsity
[165]	2006	SM	Active set	Dense
[166]	2009	SM	Active set	Dense
[167]	2010	SM	Active set	Dense
[168]	2011	Branch-and-bound		
[169–172]	2011	SM	Active set	Dense
[173]	2012	SM	Active set	Sparse
[174–176]	2012	Branch-and-bound, simplex method		
[177]	2012	Mehrotra	Interior point	Sparse
[178–180]	2013	Branch-and-bound		
[181]	2013	SM	Active set	Sparse
[182]	2013	SM	Active set	Dense
[183]	2015	SM	Active set	Dense

Table 7.5: Classification of the implementations in the state of the art based on the target device. The *sCPU* and *mCPU* columns represent single core and multi core CPU implementations respectively. *PDIP* stands for for Primal-Dual Interior Point method.

Algorithm	GPU	FPGA	mCPU	CPU–GPU
SM	[166, 167, 169, 170, 172, 181, 182]	[165]	[173, 182, 183]	[170, 171]
PDIP		[177]		
Branch-and-bound	[168, 174, 178, 179]		[176, 179, 180]	[175, 179]

multi-GPU systems [171]. This variation of the SM algorithm performs the steps listed in Algorithm 7.1 as row and column operations on a matrix called the *tableau*. In the paper, this was split into blocks and each block was associated with a work group, which copied the data onto its local memory, performed the mathematical operations and wrote the results back to the global memory.

A similar splitting of the tableau matrix into blocks had been implicitly used by Spampinato and Elster [166], Bieling *et al.* [167], and Hamzić *et al.* [169] to perform linear algebra operations. Although efficient for the dense SM, this

Table 7.6: Classifications of GPU implementations of algorithms that solve LPs and MIPs in the literature based on their performance. M and N are respectively the number of constraints and columns of the largest problem solved, which belonged to the test set in column *Set*. *R* stands for randomly generated instances, *N* for NETLIB [12], *CPU* stands for comparison on the same algorithm implemented both on GPU and CPU while *mCPU* stands for comparison against the same algorithm on multicore CPU. $S_{\max}$ is the maximum speedup achieved. The precision is specified whenever it was stated in the corresponding publication.

Implementation	M	N	Set	Comparison	$S_{\max}$
Spampinato and Elster [166]	2000	2000	R	CPU	2.5
Bieling *et al.* [167]	2700	2700	R	GLPK [184]	18
Carneiro *et al.* [168]		81	R	mCPU	10.69
Lalami *et al.* [171]	4000	4000	R	CPU	12.5
Lalami *et al.* [172]	4000	4000	R	CPU	12.5
Meyer *et al.* [170]	5000	25000	N	CLP	≈ 2
Hamzić *et al.* [169]	4096	8192	R	CPU	28.93
Boukedjar *et al.* [174]		500	R	CPU	8.27
Lalami and El-Baz [175]	500	500	R	CPU	20.48
Meyer *et al.* [181]	1904	2857	N	CLP	< 1
Chakroun *et al.* [178]	200	20	R	CPU	74.20
Chakroun *et al.* [179]	500	20	R	CPU	170.69
Smith [182]	10000	10000	R	Gurobi [185]	0.18
Ploskas and Samaras [183]	6000	6000	R	MATLAB [85]	18.5

technique implies a large overhead if the tableau is a sparse matrix, which is the case of most practical applications.

Another implementation of the dense SM was the work of Meyer *et al.* [170], in which all dense linear algebra kernels were implemented by using the highly optimized CUBLAS library [10]. When tested against NETLIB test cases [12], the software architecture achieved a speedup of about 2 with respect to an open source solver running on a single core CPU. Considering the peak performance that can be achieved by a GPU, which is in the order of the teraflops, this results proves the poor computational efficiency of the dense SM on GPUs for practical cases.

An FPGA implementation of the dense SM was designed by Bayliss *et al.* [165]. The FPGA architecture could process several LPs at the same time and targeted an application to MIP and optimisation solvers based on the branch-and-bound algorithm [188, ch. 18]. In spite of that, it was limited by the available on-chip memory, which is in the order of megabytes even for a large device. Consequently, only very small problems could be solved by the FPGA architecture.

The dense SM does not exploit sparsity but processing sparse data structures leads to great reduction of memory footprint and execution time [25, ch. 1]. For this reason, Meyer *et al.* implemented the sparse SM was implemented on GPU [181] and showed a large reduction of execution time with respect to the tableau SM on the largest problems in the NETLIB benchmark [12]. Nonetheless, no speedup was obtained with respect to the solver CLP [189] on a single-core CPU. The CUSPARSE library [17] was deployed to perform all of the matrix operations. Similarly, no speedup with respect to any C++ software was achieved by Ploskas and Samaras's implementation of the sparse SM [183]. However this implementation, which uses MATLAB's Parallel Computing Toolbox [85], was shown to be faster than MATLAB's built-in interior point method algorithm in some problems. Ploskas and Samaras also studied basis update algorithms for the dense SM on GPUs, focusing on the ETM algorithm and not considering the Forrest-Tomlin algorithm [190], and pivoting rules for the dense SM on GPUs [191]. In both cases, it was shown that the best techniques for the dense case minimise the communication overhead.

Hall observed that little success has been achieved in implementing the sparse SM on multi-core CPUs unless very specific matrix structures were considered [182, 192]. For example, Hall and Huangfu's solver ParISS [173] is an implementation of SM for multi-core CPUs that partially exploits sparsity but does not achieve any speedup with respect to the solver CLP [189]. The same conclusion was reached by Smith, who implemented the dense SM for both multi-core CPUs and GPUs and the sparse SM for multi-core CPUs [182]. This shows that parallelising the sparse SM is an open problem in the literature independently of the particular computer architecture used.

Exploiting the structure of problems is a possible way to overcome the limitations of generic linear algebra packages such as CUSPARSE and to improve performance. Jerez *et al.* introduced a formulation based on block-banded matrices of a particular type of optimisation problem that is recurring in the field of model predictive control [177]. By taking into account the recurrences in the structure of the matrices, an implementation of an interior-point method was designed for FPGAs. Although this technique seems particularly profitable for FPGA implementations, solving MIPs requires high accuracy because the result of each LP determines the execution of the branch-and-bound algorithm [188, ch. 18]. Hence, floating-point arithmetic is more suitable in the context of MIPs compared to fixed-point arithmetic, which currently dominates in FPGA implementations because it can achieve performance comparable to the peak of GPUs [177].

In the specific case of MIPs and the branch-and-bound algorithm, parallelisation with GPUs was shown in principle to achieve high speedup with respect to single-core CPU implementations [175, 179]. However, the implementations in the literature do not consider application to a realistic MIP solver. For example none of them consider the case in which the basic matrix is sparse and stored in factorised form [168, 174, 175, 178, 179, 194, 195], which is always done in practice for computational and numerical reasons [158, ch. 8]. Although not on GPUs, only the work of Shinano *et al.* [176, 180] discussed the parallelisation of the branch-and-bound algorithm for the solver Scip [139, 159], showing that even if many sub-problems can be processed at the same time, the execution time of the single sub-problem is the limiting factor for performance.

7.5 CONCLUSIONS FROM THIS CHAPTER

The SM is a numerical application that requires the solution of sequences of systems of linear equations, where each of the matrices in the sequence differs from the previous matrix in one column. Solving these systems of linear equations is the most time-consuming part in the SM, as discussed in Section 7.1.3.

How to accelerate the solution of these systems of linear equations depends on the type of algorithm used to obtain the matrices in the sequence.

The Forrest-Tomlin algorithm is both the most numerically stable algorithm and the one that obtains the sparsest factorised version of the system matrix, called the *basic matrix*. However, to obtain speedup on the GPU, it would be necessary to transfer the updated version of the matrix from the CPU to the GPU almost ten times faster than possible with current GPU technologies, which is not realistic.

The ETM algorithm usually obtains a factorised version of the basic matrix that is less sparse than the one obtained with the Forrest-Tomlin algorithm, but requires less frequent transfers from the CPU to the GPU. For this algorithm, a theoretical model (7.44) shows that speedup can be achieved if the re-factorisation of the basic matrix infrequent and if the graph associated with the factors of the basic matrix can be traversed with high concurrency.

The theoretical models derived in this chapter suggest that in order to obtain speedup compared with a CPU implementation on this type of sparse linear algebra applications, it is necessary for the data to be reused as much as possible. This corresponds to a high value of $n_{\text{iterations}}$ in (7.43) and (7.44), and amortises the cost of the data transfer which limits GPU performance. However, even if a high value of $n_{\text{iterations}}$ can be achieved, the speedup can be limited if the concurrency of the graph traversal, given by the concurrency factor, is not high.

These limitations could be overcome with a computer architecture where a multi-core CPU and the GPU are on the same chip. This would reduce the data transfer overhead and it could allow for better performance with a low value of $n_{\text{iterations}}$. Also, this could make it possible to use the GPU only for those matrices that can achieve a high concurrency factor. Intel's Gen9 Processor Graphics [196] is an example of this type of computer architecture, which is used in desktop and laptop computers but not computer clusters.

8

CONCLUSIONS AND FUTURE WORK

This thesis has examined the implementation of sparse linear algebra algorithms on GPUs and, in particular, the solution of sparse triangular systems of linear equations (STLs). This algorithm occurs in more complex algorithms such as the preconditioned conjugate gradient (PCG) and the simplex method (SM).

The solution of STLs, like all sparse linear algebra algorithms, shows opportunity for parallelisation depending on the sparsity pattern of the system matrix. The sparsity pattern can be represented as a directed acyclic graph (DAG) which can have a particular structure. For example, can be composed of very connected parts or can be composed of disjoint components.

Sparse matrices are stored with specialised data structures, which cause sparse linear algebra algorithms to have an irregular pattern of memory accesses. This, in turn, can cause a loss of performance of the implementation of sparse linear algebra algorithms on GPUs and other parallel computer architectures.

8.1 INSIGHT FROM THIS THESIS

This thesis has discussed an approach to implement high-performance sparse linear algebra algorithms on GPUs. The discussion has given insight both on sparse linear algebra algorithms, which are a type of irregular application, and GPU architectures.

8.1.1 *Sparse linear algebra and irregular applications*

This thesis showed that the implementation of sparse linear algebra algorithms, and in general that of irregular applications, should not aim to achieve the high-

est possible level of concurrency, which is the aim of the level-set approach implemented by CUSPARSE. In fact, if concurrency is high, the GPU's memory sub-system is less capable of quickly coalescing memory accesses and, as a consequence, a part of the execution time is spent waiting for the irregular memory accesses to be completed. On the contrary, the approach in this thesis is to partition the sparse matrix, while considering the architecture of the GPU's memory sub-system and the structure of the associated graph at the same time.

This concept of balancing concurrency and data locality is similar to that of balancing sparsity and stability in numerical analysis, of which one example is Markowitz's LU decomposition algorithm [157, 158, ch. 2] mentioned in Section 7.1.2 on page 163. Markowitz's LU decomposition computes an approximation of the triangular factors L and U, denoted as $\tilde{L}$ and $\tilde{U}$. aiming to obtain the sparsest possible $\tilde{L}$ and $\tilde{U}$, because solving the corresponding STLs on a single-core CPU is faster than with denser factors. However, since the factors obtained by Markowitz's algorithm are an approximation, the product $\tilde{L}\tilde{U}$ can have a much higher condition number than that of the product LU. The condition number of the first of these two products can be improved by reducing the sparsity of $\tilde{L}$ and $\tilde{U}$.

Markowitz's algorithm, like most of classical sparse linear algebra algorithms, was designed to be executed on single-core CPUs, in which the solution of an STL is obtained by traversing in sequence each edge of the graph associated with the system matrix. For modern processors, which can traverse more than one edge at the same time, sparsity alone is not sufficient to capture the performance of a sparse linear algebra algorithm. By introducing the concept of concurrency factor, this thesis has addressed this problem both theoretically and pratically.

In general, the fastest approach to solving a sparse linear algebra problem would require balancing all four elements: sparsity, stability, concurrency and data locality. The point of balance depends on the application, which determines both the structure of data, represented by the shape of a graph, and the accuracy requirements on the solution. Usually, this four-dimensional design

space is too large to be explored exhaustively so it is necessary to develop techniques that are capable of informing the decision.

One of these techniques, implemented by both TASSL and CUSPARSE, extracts information from the structure of data with a preliminary analysis. However, this approach can be costly in terms of execution time. The approach opposite to this would be to require an application expert to manually provide all the information about the structure of the data. More research could be carried out in hybrid approaches that reduce the execution time of a preliminary analysis by incorporating information from an expert user.

8.1.2 *GPU architectures*

This thesis has also provided an insight into GPU architectures. Depending on the size of a GPU's local memories, the GPU can be more suited to solving one particular type of STL with respect to another. For example, a GPU with many compute units (CUs) and small local memories is more suited than a GPU with few CUs and large local memories to solving STLs in which the graph is composed of many, small disjoint components.

In general, there is no single GPU architecture that is ideal for all types of STL, contrarily to FPGA architectures, which can be adapted to any STL. However, any accelerator for sparse linear algebra algorithms can be slower than a single-core CPU implementation of the same algorithm if the data is to be transferred frequently from and to the accelerator. This poses a fundamental limit to the performance of accelerators for sparse linear algebra algorithms using current computer technology.

8.2 GENERALISATION OF THE APPROACH

There are many ways in which the work in this thesis could be expanded. This section examines some of these possibilities.

8.2.1 *Other sparse linear algebra algorithms*

The approach devised in this thesis for the solution of STLs can be generalised to other sparse linear algebra algorithms like the sparse matrix-vector multiplication (SPMVM). In fact, SPMVM can be considered a generalisation of this approach to rectangular matrices or equivalently bipartite graphs [94, ch. 1]. In a bipartite graph, vertices are of two types and edges can only be incident from a vertex of one type to a vertex of the other type. In the case of SPMVM, one vertex type represents the set of rows and the other the set of columns. Either of the two sets can be partitioned depending on what is more profitable in terms of data locality.

This approach could be used in theory to implement other graph-based algorithms and irregular applications on GPUs since increasing data locality is particularly beneficial for these applications. For example, the depth-first traversal of a graph could require a generalisation of the partitioning algorithms devised in this thesis to achieve high data locality.

More research should be done specifically on partitioning techniques, since the overhead of analysing the sparse matrix can limit the performance of the approach described in this thesis if the matrix is very large. Also, the possibility of implementing this first phase on GPU could be considered for much larger matrices than those examined in this thesis, for which there could be more opportunity for concurrency.

8.2.2 *Other computer architectures*

Another direction for future research on the topic discussed in this thesis is that of other computer architectures. Although in this thesis the concepts of data locality and concurrency were discussed for GPU implementations, they also apply to multi-core CPUs, computer clusters, and Intel Xeon Phi accelerators among others. For example, in multi-core CPUs and Intel Xeon Phi accelerators the L1 cache memory can be considered the equivalent of the local memory. Also, the explicit data transfer carried out on GPUs could be left to the memory

sub-system in these architectures. Similarly, the shared memory in computer clusters can be considered the equivalent of the global memory discussed in this thesis and the memory available for each cluster node can be considered the equivalent of the local memory.

Generalising the approach to computer architectures other than GPUs would also provide the opportunity to include the tradeoff between synchronous and asynchronous computation in the discussion. For example, it could be possible to have implementations in which all the processing inside sub-graphs is carried out asynchronously and the sub-graph levels are processed synchronously. In this case, it could be investigated how asynchronous computation in the processing of a sub-graph affects memory access patterns and therefore whether different partitioning techniques can improve data locality.

8.2.3 *A specialised computer architecture*

Since certain GPU architectures are more suitable to solving particular STLs than others, it could be of interest to design a GPU-type architecture that can be adapted to different types of problems. Such a GPU architecture would allow the programmer to allocate a varying quantity of local memory space so to process efficiently both graphs composed of many small disjoint components and graphs composed of few large disjoint components. Moreover, such a computer architecture would include a CPU-type core to implement those parts of sparse linear algebra algorithms that do not show any opportunity for concurrency and would instead benefit from executing instructions at a higher rate.

In conclusion, implementing sparse linear algebra algorithms on modern computer architectures is a very active research topic and each approach largely depends on the type of matrix being processed. Although this thesis has examined a number of problems related to this topic, there are many other problems that require further research.

LIST OF ACRONYMS

BSR	Block compressed sparse row format.
CG	Conjugate gradient method.
COO	Coordinate format.
CSC	Compressed sparse column format.
CSR	Compressed sparse row format.
CU	Compute unit.
DAG	Directed acyclic graph.
DSL	Domain-specific language.
ETM	Elementary transform matrix.
FLOPS	Floating-point operations per second.
FPGA	Field-programmable gate array.
GPU	Graphics processing unit.
ILU	Incomplete LU factorisation.
ILU0	Incomplete LU factorisation with zero fill-in.
ILUTP	Incomplete LU factorisation with threshold and pivoting.

LL-LU	Left-looking LU factorisation.
LP	Linear programming.
MIMD	Multiple-instruction-multiple-data [26].
MIP	Mixed-integer programming.
PCG	Preconidtioned conjugate gradient method.
PE	Processing element.
PGMRES	Preconidtioned generalised minimal residuals method.
SIMD	Single-instruction-multiple-data [26].
SM	Simplex method.
SPMVM	Sparse matrix-vector multiplication.
STL	Sparse triangular system of linear equations.
TASSL	Time-slot sub-graph level algorithm.

GLOSSARY

ACTIVE ELEMENT. In this thesis, an active element is either a vertex or an edge in the graph associated with a sparse matrix. Numerical operations associated with the same active element cannot be executed concurrently, so they are either executed in sequence or as atomic operations. In CUSPARSE, an active element is a vertex in the graph and one or more numerical operations are associated with it, each corresponding to one of its entering edges. In TASSL, an active element is an edge in the graph, which has only one numerical operation corresponding to itself. The concept of active element was first introduced by Pingali *et al.* [50].

ADDRESS COALESCING. The merging of one or more memory access requests into a single memory access request in case of good data locality. Multithreaded SIMD processors in GPUs are equipped with sophisticated address coalescing units that analyse memory access requests coming from many SIMD threads [31, ch. 4].

ATOMIC OPERATION. An atomic operation is a special instruction that appears to be uninterruptible to the rest of the system. On GPUs, atomic operations are managed by specialised hardware in the L2 cache memory [31, ch. 4]

BANK CONFLICT. In both Nvidia and AMD GPU architectures, local memory is organised in memory banks and the best performance is achieved only if as many memory banks as possible are accessed at the same time. A bank conflict happens if two or more SIMD lanes access the same memory bank at the same time, because these accesses are sequentialised and performance is reduced.

BARRIER. On GPUs, a barrier is implemented using either a global or a local memory location that acts as a counter. Whenever a work item encoun-

ters a barrier, it increments the counter and it remains idle until the execution of all the other work items in the same work group reaches the same point.

COMPUTE UNIT. In the OpenCL model [28], a compute unit (CU) is the hardware that executes a work group. It contains a local memory and a certain number of processing elements, each one with its private memory. In a GPU, it is a multithreaded SIMD processor [31, ch. 4].

CONCURRENCY FACTOR. In this thesis, the concurency factor C of a sparse triangular matrix L with respect to a given algorithm is a measure of how much concurrency can be used in carrying out the algorithm. In the most general case it is defined as

$$C_L := 1 - \frac{\sum_{q=1}^{n_{\text{levels},L}} \left(\sum_{t' \in \mathcal{T'}_{\hat{q}}} o'_{t'} + \sum_{t \in \mathcal{T}_{\hat{q}}} o_t \right)}{n_{\text{edges},L}},$$

where n_{levels}, L is the number of sub-graph levels and g is a graph in sub-graph level $\mathcal{L}_q$, and $G_{\hat{q}}$ the sub-graph in level $\mathcal{L}_q$ with the longest processing time during the solve phase. Also, $\mathcal{T'}_g$ is the set of external time slots of graph g and $o'_{t'}$ is the maximum number of numerical operations in the external time slot t'. Similarly, $\mathcal{T}_g$ is the set of internal time slots of graph g and o_t is the maximum number of numerical operations in the internal time slot t.

The concurrency factor has a value between zero and one. When the numerator of the fraction in the definition of C_L is equal to 1, the execution of the algorithm requires only requires SIMD threads of length one, with one instruction per work item and only one time slot in total. This is the case of maximum concurrency and C_L is closer to 1 the more the edges in the graph. If the numerator of the fraction in the definition of C_L is equal to $n_{\text{edges},L}$, all the numerical operations are executed in sequence one after the other. This is the case of a graph whose structure is that of a *chain*, which does not allow concurrency. This is captured by a concurrency factor that is closer to zero the more the edges in the graph. Hence,

the concurrency factor C_L captures the scheduling of the instructions in the graph but it does not capture data locality, nor memory access latency hiding, nor thread divergence.

DISJOINT COMPONENT. Also called *weakly connected graph component* [94, ch. 1]. It is a part of the graph that is not connected to any other part of the graph.

EFFECTIVE OPERATION TIME. In this thesis, the effective operation time τ is defined as

$$\tau := \frac{1}{T\eta}(1 + \sigma + \lambda)$$

where T is the ideal throughput of the type of SIMD instructions considered, η is the efficiency factor, σ is the synchronisation overhead, and λ is the local transfer overhead. The effective operation time captures at the same time all the effects captured by the efficiency factor, the synchronisation overhead, and the local transfer overhead.

The physical meaning of τ is given by its inverse, which has the dimensions of a throughput. In fact, $1/\tau$ is the apparent rate of SIMD operations during the execution of the algorithm.

EFFICIENCY FACTOR. In this thesis, the efficiency factor η of a sparse triangular matrix L with respect to a given algorithm is a measure of the effects of data locality, memory access latency hiding, and thread divergence. In the most general case it is defined as

$$\eta := \frac{\sum_{q=1}^{n_{\text{levels},L}} \left(\sum_{t' \in \mathcal{T}'_{\hat{q}}} o'_{t'} + \sum_{t \in \mathcal{T}_{\hat{q}}} o_t \right)}{\sum_{q=1}^{n_{\text{levels},L}} \left(\sum_{t' \in \mathcal{T}'_{\hat{q}}} \frac{o'_{t'}}{\eta'_{t'}} + \sum_{t \in \mathcal{T}_{\hat{q}}} \frac{o_t}{\eta_t} \right)}$$

where n_{levels}, L is the number of sub-graph levels and g is a graph in sub-graph level $\mathcal{L}_q$, and $G_{\hat{q}}$ the sub-graph in level $\mathcal{L}_q$ with the longest processing time during the solve phase. Also, $\mathcal{T}'_g$ is the set of external

time slots of graph g and $o'_{t'}$ is the maximum number of numerical operations in the external time slot t' and $\eta'_{t'}$ is the operation efficiency of t'. Similarly, $\mathcal{T}_g$ is the set of internal time slots of graph g and o_t is the maximum number of numerical operations in the internal time slot t and η_t is the operation efficiency of t.

The efficiency factor captures the effects of data locality, memory access latency hiding, and thread divergence. Like the operation efficiency, η can be equal to one only in the ideal case, in which the operation throughput is equal to the theoretical maximum.

EXTERNAL EDGE. Given a partition of the vertices of a graph $G \coloneqq (\mathcal{V}, \mathcal{E})$ into $n_s + 1$ disjoint subsets $\mathcal{V}_0, \mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_{n_s}$, an edge (i, j) is an external edge if the vertices i and j are not in the same sub-set $\mathcal{V}_k$. The set of all external edges for sub-graph $G_k \coloneqq (\mathcal{V}_k, \mathcal{E}_k)$ is

$$\mathcal{E}_{\text{ext},k} \coloneqq \{(i, j) \in \mathcal{E} \mid i \in \mathcal{V}_k \wedge j \in \mathcal{V} \wedge j \in \mathcal{V}_k\}.$$

FEASIBLE PARTITION. A partition of the vertices of a graph $G \coloneqq (\mathcal{V}, \mathcal{E})$ into $n_s + 1$ disjoint subsets $\mathcal{V}_0, \mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_{n_s}$ is feasible if two conditions are met:

CONDITION I: if an edge goes from a vertex in $\mathcal{V}_k$ to a vertex in $\mathcal{V}_p$, then $p \geq k$.

CONDITION II: the number of vertices in the set $\mathcal{V}_k$, n_k, must not be greater than a positive integer $n_{\max}$.

GLOBAL MEMORY. A memory shared by all the compute units in the OpenCL model [28]. On a GPU, it is an off-chip DRAM [31, ch. 4].

INTERNAL EDGE. Given a partition of the vertices of a graph $G \coloneqq (\mathcal{V}, \mathcal{E})$ into $n_s + 1$ disjoint subsets $\mathcal{V}_0, \mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_{n_s}$, an edge (i, j) is an internal edge if both vertex i and vertex j are in the same sub-set $\mathcal{V}_k$. The set of all internal edges for sub-graph $G_k \coloneqq (\mathcal{V}_k, \mathcal{E}_k)$ is

$$\mathcal{E}_{\text{int},k} \coloneqq \{(i, j) \in \mathcal{E} \mid i \in \mathcal{V}_k \wedge j \in \mathcal{V}_k\}.$$

KERNEL. In the OpenCL model [28], it is the set of all work groups being executed on a device.

LOCAL MEMORY. In the OpenCL model [28], a memory contained in a compute unit and shared by all the work items in a work group. On GPUs, it is a SRAM in a multithreaded SIMD processor [31, ch. 4]. On Nvidia GPUs in particular, this SRAM can be partitioned into local memory and L1 cache memory [34].

LOCAL TRANSFER OVERHEAD. In this thesis, the local transfer overhead λ of a sparse triangular matrix L with respect to the TASSL algorithm is defined as

$$\lambda := \frac{\sum_{p=1}^{n_{\text{levels}}} \beta n_{\hat{p}}}{\frac{1-C}{\eta} \iota n_{\text{edges}}}$$

where n_{levels} is the number of sub-graph levels, $\hat{p}$ is defined in (3.23) at page 81, C is the concurrency factor, η is the efficiency factor, and n_{edges} is the number of edges in the graph G associated with L.
The local transfer overhead represents how much time is spent transferring data to and from the local memory with respect to the time spent carrying out numerical operations.

MEMORY ACCESS LATENCY HIDING. On GPUs, good memory access latency hiding is achieved when there are enough SIMD threads ready for execution to keep arithmetic-logic units busy at all times, while other SIMD threads are waiting for memory operations [31, ch. 4].

MULTITHREADED SIMD PROCESSOR. It is the GPU equivalent of a CPU core. It contains a number of SIMD lanes, a register file, a SIMD thread scheduler, an address coalescing unit, a local memory or L1 cache, and other cache memories for read-only and constant data [31, ch. 4].

OPERATION EFFICIENCY. In this thesis, the operation efficiency $\eta_{g,q}$ is a value between 0 and 1 and captures the effects of data locality, memory access coalescing, and thread divergence in the execution of time slot $\mathcal{T}_{g,q}$,

while excluding synchronisation overheads. When $\eta_{g,q}$ is equal to 1, the throughput achieved by the GPU software is equal to that in case of no thread divergence and a null memory access latency.

PER-THREAD CONCURRENCY. In this thesis, the per-thread concurrency metric W estimates how many work items are used to execute the algorithm on the GPU. The per-thread concurrency of matrix L with respect to an algorithm is defined in the general case as:

$$W := \frac{\max_{l \in \{\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_{n_{\text{levels}}}\}} \left\{ \sum_{g \in l} \max_{q,q'} \left\{ |\mathcal{T}_{g,q}|, |\mathcal{T}'_{g,q'}| \right\} \right\}}{n_{\text{edges}}},$$

where the numeration represents the maximum number of work items used during the execution of the algorithm. If W is close to one, there is almost one work item per vertex in the graph, and consequently also C is close to one because the longest SIMD is composed of few SIMD instructions.

The per-thread concurrency metric is similar to the *degree of parallelism* metric first defined by Kulkarni *et al.* [51]. Given the number of compute units in the GPU n_{CU} and the maximum number of work items per compute unit n_{w}, the models developed in this thesis are limited to the case $W \leq \frac{n_{\text{CU}} n_{\text{w}}}{n_{\text{edges}}}$.

PRIVATE MEMORY. In the OpenCL model [28], a memory that is private to each processing element. In a GPU it is implemented as a set of registers private to each SIMD lane or, if the data does not fit into these registers, part of the off-chip DRAM [31, ch. 4].

PROCESSING ELEMENT. The processing element (PE) is the most elementary architectural element in the OpenCL model [28]. A PE executes one work item and in practice corresponds to a SIMD lane on GPUs [31, ch. 4].

REGISTER SPILLING. GPU compilers try to store all data belonging to private memory in the OpenCL model [28] in the register file of multithreaded SIMD processors. However, if this is not possible, a part of the data is

stored in the DRAM. This fact is called register spilling and causes a reduction in performance because it increases the number of accesses in the L2 cache memory [31, ch. 4].

SIMD LANE. On a GPU, it is the hardware that executes each part of a SIMD instruction. Each multithreaded SIMD processor usually has 16 SIMD lanes. Two SIMD lanes cannot execute different SIMD instructions simultaneously [31, ch. 4].

SIMD THREAD. Also called *warp*. On a GPU, it is a sequence of SIMD instructions being processed by a multithreaded SIMD processor and it corresponds to 32 work items [31, ch. 4]. Usually, many SIMD threads are processed by a multithreaded SIMD processor at the same time and their execution is managed by the SIMD thread scheduler.

SIMD THREAD DIVERGENCE. The effect of conditional branching in GPU software, for example with the `if-else` and `case` statements. Since many work items are mapped to one SIMD thread, every branching statement causes the execution of more SIMD instructions, each one operating on only some of the of the operands [31, ch. 4].

SUB-GRAPH. Given a partition of the vertices of a graph $G \coloneqq (\mathcal{V}, \mathcal{E})$ into $n_s + 1$ disjoint subsets $\mathcal{V}_0, \mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_{n_s}$, a sub-graph $G_k \coloneqq (\mathcal{V}_k, \mathcal{E}_k)$ of G is composed of the vertices in the sub-set $\mathcal{V}_k$, and the union of the set of internal edges and the external edges for $\mathcal{V}_k$. Hence, $\mathcal{E}_k = \mathcal{E}_{\text{int},k} \cup \mathcal{E}_{\text{ext},k}$.

SYNCHRONISATION OVERHEAD. In this thesis, the synchronisation overhead σ of a sparse triangular matrix L with respect to a given algorithm is defined in the general case as

$$\sigma \coloneqq \frac{\kappa n_{\text{levels},L} + \sum_{q=1}^{n_{\text{levels},L}} \left(l'_{\hat{q}} \sigma_{\text{global}} + l_{\hat{q}} \sigma_{\text{local}} \right)}{\frac{1-C}{T\eta} n_{\text{edges},L}}$$

where n_{levels}, L is the number of sub-graph levels and g is a graph in sub-graph level $\mathcal{L}_q$, and $G_{\hat{q}}$ the sub-graph in level $\mathcal{L}_q$ with the longest

processing time during the solve phase. Also, l'_g is the number of of external time slots of graph g and l_g is ithe number of internal time slots, κ is the time required for a GPU kernel to be stopped and for a new one to be started, σ_{global} is the execution time of a global memory barrier, σ_{local} is the execution time of a local memory barrier, C is the concurrency factor, η is the efficiency factor, T is the maximum theoretical throughput for the type of numerical operations executed, and finally $n_{\text{edges},L}$ is the number of edges in the graph associated with L.

The synchronisation overhead is the the ratio of the part of execution time of the algorithm that is spent on synchronisation points and the part that is spent carrying out numerical operations and transferring data.

SYNCHRONISATION POINT. In this thesis, a synchronisation point is either the start of the execution of a GPU kernel, the end of the execution of a GPU kernel, or a barrier.

TIME SLOT. In this thesis, a time slot is the set of numerical operations executed between two synchronisation points. The numerical operations in a time slot are grouped by their active element. The duration of time slot $\mathcal{T}_q$ is determined by the largest number of numerical operations associated with an active element of $\mathcal{T}_q$ and is represented by o_q.

WORK GROUP. In the OpenCL model [28], a group of work items executed in the same compute unit. Work items in the same work group can be syncrhonised, while work items in different work groups cannot be synchronised.

WORK ITEM. An elementary piece of software in the OpenCL model [28], executed by a processing element. Work items executed in the same compute unit form a work group and can be synchronised. Inside a GPU, every 32 work items are grouped together in one SIMD thread and executed in parallel, implementing SIMD parallelism [31, ch. 4].

BIBLIOGRAPHY

[1] M. Feldman and A. Snell, "Accelerated computing: A tipping point for HPC", Intersect360 Research, Tech. Rep., Nov. 2015, Accessed 21 July 2016. [Online]. Available: http://images.nvidia.com/content/pdf/tesla/accelerated-computing-at-a-tipping-point.pdf (cit. on p. 16).

[2] H. Mujtaba. (Feb. 2016). NVIDIA pascal GP100 GPU expected to feature 12 TFLOPs of single precision compute, 4 TFLOPs of double precision compute performance. Accessed 20 July 2016, [Online]. Available: http://wccftech.com/nvidia-pascal-gp100-gpu-compute-performance/ (cit. on pp. 16, 17).

[3] "CUDA 7.0 performance report", NVIDIA Corporation, Tech. Rep., May 2015, Accessed 20 July 2016. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/compute-docs/cuda-performance-report.pdf (cit. on pp. 16, 17).

[4] "NVIDIA tesla K20-K20X GPU accelerators – benchmarks", NVIDIA Corporation, Tech. Rep., 2012, Accessed 20 July 2016. [Online]. Available: http://www.nvidia.com/docs/IO/122874/K20-and-K20X-application-performance-technical-brief.pdf (cit. on pp. 16, 17).

[5] H. Mujtaba. (Nov. 2015). NVIDIA pascal GPU's double precision performance rated at over 4 TFLOPs, 16nm FinFET architecture confirmed – volta GPU peaks at over 7 TFLOPs, 1.2 TB/s HBM2. Accessed 20 July 2016, [Online]. Available: http://wccftech.com/nvidia-pascal-volta-gpus-sc15/ (cit. on p. 17).

[6] "CUDA 6.5 performance report", NVIDIA Corporation, Tech. Rep., Sep. 2014, Accessed 20 July 2016. [Online]. Available: http://developer.

download.nvidia.com/compute/cuda/6_5/rel/docs/CUDA_6.5_Performance_Report.pdf (cit. on p. 17).

[7] K. Rupp. (Mar. 2014). CPU, GPU and MIC hardware characteristics over time. Accessed 21 July 2016, [Online]. Available: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/ (cit. on p. 17).

[8] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley, "An updated set of basic linear algebra subprograms (BLAS)", *ACM Transactions on Mathematical Software*, vol. 28, pp. 135–151, 2001. DOI: 10.1145/567806.567807 (cit. on pp. 17, 51).

[9] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999, ISBN: 0-89871-447-8 (paperback) (cit. on pp. 17, 51).

[10] *CUBLAS library user guide*, 7.5, NVIDIA, Oct. 2015. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf (cit. on pp. 17, 126, 188).

[11] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems", *Parallel Computing*, vol. 36, no. 5–6, pp. 232–240, Jun. 2010, ISSN: 0167-8191. DOI: 10.1016/j.parco.2009.12.005 (cit. on p. 17).

[12] D. M. Gay. (Aug. 2005). Netlib LP benchmark documentation. Accessed on 7/11/2014, [Online]. Available: http://www.netlib.org/lp/data/readme (cit. on pp. 18, 137, 170, 188, 189).

[13] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter, "MIPLIB 2010",

Mathematical Programming Computation, vol. 3, no. 2, pp. 103–163, 2011. DOI: 10.1007/s12532-011-0025-9 (cit. on pp. 18, 156, 157).

[14] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Society for Industrial and Applied Mathematics, 2003. DOI: 10.1137/1.9780898718003 (cit. on pp. 18, 52, 120–126, 134, 137).

[15] G. Dantzig, "Programming of interdependent activities: II mathematical model", English, *Econometrica*, vol. 17, no. 3/4, pp. 200–211, 1949, ISSN: 00129682 (cit. on pp. 18, 137, 159, 160).

[16] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU", NVIDIA Corporation, Technical report 2011–001, Jun. 2011. [Online]. Available: http://research.nvidia.com/sites/default/files/publications/nvr-2011-001.pdf (cit. on pp. 18, 43, 47, 51–54, 56, 59, 63–66, 71, 73, 86, 87, 91, 113, 114, 117, 133, 134).

[17] *CUSPARSE library*, 7.0, Accessed 1 September 2015., NVIDIA corporation, Mar. 2015. [Online]. Available: https://developer.nvidia.com/cuda-toolkit-70 (cit. on pp. 18, 51, 60, 62, 94, 133, 134, 189).

[18] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jüngel, and S. Selberherr, "ViennaCL - linear algebra library for multi- and many-core architectures", *SIAM Journal on Scientific Computing*, vol. 38, no. 5, S412–S439, 2016. DOI: 10.1137/15M1026419 (cit. on pp. 18, 51, 52, 55, 94, 112, 120, 134).

[19] *Paralution - user manual*, 1.1.0, 2016. [Online]. Available: http://www.paralution.com/downloads/paralution-um.pdf (cit. on pp. 18, 51, 52, 55, 94, 112, 120, 134).

[20] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects", *Journal of Physics: Conference Series*, vol. 180, no. 1, p. 012 037, 2009. [Online]. Available: http://stacks.iop.org/1742-6596/180/i=1/a=012037 (cit. on pp. 18, 51, 52, 55, 94, 112, 120, 134).

[21] T. Brandes, E. Schricker, and T. Soddemann, "The LAMA approach for writing portable applications on heterogenous architectures", Fraunhofer Institute for Algorithms and Scientific Computing, Sankt Augustin, Germany, Tech. Rep., 2016. [Online]. Available: http://www.libama.org/assets/lamawhitepaper.pdf (cit. on pp. 18, 51, 55, 94, 112, 120).

[22] M. Kreutzer, J. Thies, M. Röhrig-Zöllner, A. Pieper, F. Shahzad, M. Galgon, A. Basermann, H. Fehske, G. Hager, and G. Wellein, "GHOST: Building blocks for high performance sparse linear algebra on heterogeneous systems", *International Journal of Parallel Programming*, pp. 1–27, 2016, ISSN: 1573-7640. DOI: 10.1007/s10766-016-0464-z (cit. on pp. 18, 50–52, 55, 94, 112, 120, 134).

[23] A. Picciau, G. E. Inggs, J. Wickerson, E. C. Kerrigan, and G. A. Constantinides, "Balancing locality and concurrency: Solving sparse triangular systems on GPUs", in *Proceedings of the 23rd IEEE International Conference on High Performance Computing*, ser. HiPC '16, Hyderabad, IN, Dec. 2016, pp. 183–192. DOI: 10.1109/HiPC.2016.030 (cit. on pp. 22, 62, 63, 97).

[24] J. Feo, O. Villa, A. Tumeo, and S. Secchi, Eds., *Proceedings of the First Workshop on Irregular Applications: Architectures and Algorithms*, IAAA '11, Seattle, WA, USA: ACM, 2011, pp. 1–2, ISBN: 978-1-4503-1121-2. DOI: 10.1145/2089142.2089144 (cit. on p. 23).

[25] T. A. Davis, *Direct Methods for Sparse Linear Systems*, ser. Fundamentals of Algorithms. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2006, vol. 2, ISBN: 0898716136 (cit. on pp. 23, 44, 47, 53, 55, 56, 62, 127, 137, 185, 189).

[26] M. J. Flynn, "Some computer organizations and their effectiveness", *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sep. 1972, ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009071 (cit. on pp. 24, 46, 198).

[27] *X86 assembly language reference manual*, Online, Accessed 4 June 2017, Oracle, 2014. [Online]. Available: http://docs.oracle.com/cd/E36784_01/html/E36859/docinfo.html#scrolltoc (cit. on p. 24).

[28] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems", *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, May 2010, ISSN: 0740-7475. DOI: 10.1109/MCSE.2010.69 (cit. on pp. 24, 27, 200, 202–204, 206).

[29] *Implementing FPGA design with the OpenCL standard*, Whitepaper, Nov. 2013. [Online]. Available: http://www.altera.co.uk/literature/wp/wp-01173-opencl.pdf (cit. on p. 26).

[30] *The Xilinx SDAccel development environment*, 2014. [Online]. Available: http://www.xilinx.com/publications/prod_mktg/sdnet/sdaccel-backgrounder.pdf (cit. on p. 26).

[31] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011, ISBN: 012383872X (cit. on pp. 27–29, 36, 38, 42, 199, 200, 202–206).

[32] N. Whitehead and A. Fit-Florea, "Precision & performance: Floating point and IEEE 754 compliance for nvidia GPUs", NVIDIA Corporation, Tech. Rep. 1, 2011. [Online]. Available: http://docs.nvidia.com/cuda/floating-point/#axzz4gbha7lKQ (cit. on p. 28).

[33] M. Scarpino, *OpenCL in action*. Manning Publications Co., 2011 (cit. on p. 28).

[34] "NVIDIA's next generation CUDA compute architecture: Fermi", NVIDIA Corporation, Tech. Rep., 2009. [Online]. Available: http://www.nvidia.it/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (cit. on pp. 28, 30, 31, 203).

[35] "AMD graphics cores next (GCN) architecture", AMD Corporation, Online, 2012, Accessed on 21 June 2017. [Online]. Available: https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf (cit. on p. 31).

[36] K. Choo, W. Panlener, and B. Jang, "Understanding and optimizing GPU cache memory performance for compute workloads", in *Proceedings of the 13th IEEE International Symposium on Parallel and Distributed Computing*, ser. ISPDC '14, Lymassol, CY: IEEE Computer Society, 2014,

pp. 189–196, ISBN: 978-1-4799-5919-8. DOI: 10.1109/ISPDC.2014.29 (cit. on p. 31).

[37] M. Doggett, "Texture caches", *IEEE Micro*, vol. 32, no. 3, pp. 136–141, May 2012, ISSN: 0272-1732. DOI: 10.1109/MM.2012.44 (cit. on p. 32).

[38] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator", in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS '09, Boston, MA, USA, Apr. 2009, pp. 163–174. DOI: 10.1109/ISPASS.2009.4919648 (cit. on p. 32).

[39] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for CPU-GPU computing", in *21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12, Sep. 2012, pp. 335–344 (cit. on p. 32).

[40] F. Candel, S. Petit, J. Sahuquillo, and Duato, "Accurately modeling the on-chip and off-chip GPU memory subsystem", *Future Generation Computer Systems*, 2017, ISSN: 0167-739X. DOI: 10.1016/j.future.2017.02.012 (cit. on pp. 32, 33).

[41] F. Candel, S. Petit, J. Sahuquillo, and J. Duato, "Accurately modeling the GPU memory subsystem", in *Proceedings of the 2015 International Conference on High Performance Computing Simulation*, ser. HPCS '15, Amsterdam, NL, Jul. 2015, pp. 179–186. DOI: 10.1109/HPCSim.2015.7237038 (cit. on p. 33).

[42] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness", in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09, Austin, TX, USA: ACM, 2009, pp. 152–163, ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555775 (cit. on p. 33).

[43] ——, "An integrated GPU power and performance model", in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, Saint-Malo, France: ACM, 2010, pp. 280–289, ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1815998 (cit. on p. 33).

[44] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with CUDA", *IEEE Micro*, vol. 28, no. 4, pp. 13–27, Jul. 2008, ISSN: 0272-1732. DOI: 10.1109/MM.2008.57 (cit. on p. 33).

[45] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, "Program optimization space pruning for a multithreaded GPU", in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '08, Boston, MA, USA: ACM, 2008, pp. 195–204, ISBN: 978-1-59593-978-4. DOI: 10.1145/1356058.1356084 (cit. on p. 34).

[46] P. Thoman, K. Kofler, H. Studt, J. Thomson, and T. Fahringer, "Automatic OpenCL device characterization: Guiding optimized kernel design", in *Proceedings of the 17th International Conference on Parallel Processing*, ser. Euro-Par '11, Bordeaux, FR: Springer-Verlag, 2011, pp. 438–452, ISBN: 978-3-642-23396-8. DOI: 10.1007/978-3-642-23397-5_43 (cit. on p. 34).

[47] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures", *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 105–118, Jan. 2011, ISSN: 1045–9219. DOI: 10.1109/TPDS.2010.107 (cit. on p. 34).

[48] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for GPGPUs", in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '13, Edinburgh, UK: IEEE Press, 2013, pp. 157–166, ISBN: 978-1-4799-1021-2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2523721.2523745 (cit. on p. 34).

[49] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-driven dynamic GPU cache bypassing", in *Proceedings of the 29th ACM International Conference on Supercomputing*, ser. ICS '15, New-

port Beach, CA, USA: ACM, 2015, pp. 67–77, ISBN: 978-1-4503-3559-1. DOI: 10.1145/2751205.2751237 (cit. on p. 34).

[50] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms", *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 12–25, Jun. 2011, ISSN: 0362-1340. DOI: 10.1145/1993316.1993501 (cit. on pp. 35, 39–41, 65, 199).

[51] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Caşcaval, "How much parallelism is there in irregular applications?", in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '09, Raleigh, NC, USA: ACM, 2009, pp. 3–14, ISBN: 978-1-60558-397-6. DOI: 10.1145/1504176.1504181 (cit. on pp. 35, 71, 204).

[52] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs", in *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS '09, Boston, MA, USA, Apr. 2009, pp. 65–76. DOI: 10.1109/ISPASS.2009.4919639 (cit. on p. 35).

[53] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-free irregular computations on GPUs", in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, ser. GPGPU-6, Houston, Texas, USA: ACM, 2013, pp. 96–107, ISBN: 978-1-4503-2017-7. DOI: 10.1145/2458523.2458533 (cit. on pp. 35, 41, 42).

[54] ——, "Data-driven versus topology-driven irregular computations on GPUs", in *Proceedings of the 27th IEEE International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13, Boston, MA, US: IEEE Computer Society, 2013, pp. 463–474, ISBN: 978-0-7695-4971-2. DOI: 10.1109/IPDPS.2013.28 (cit. on pp. 35, 41–43).

[55] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics", in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, PA, USA: ACM, 2013,

pp. 456–471, ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522739 (cit. on pp. 35, 39, 41).

[56] D. Prountzos, R. Manevich, and K. Pingali, "Synthesizing parallel graph programs via automated planning", in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15, Portland, OR, USA: ACM, 2015, pp. 533–544, ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737953 (cit. on pp. 35, 39, 41).

[57] R. Kaleem, A. Venkat, S. Pai, M. Hall, and K. Pingali, "Synchronization trade-offs in GPU implementations of graph algorithms", in *Proceedings of the 30th International Parallel and Distributed Processing Symposium*, ser. IPDPS '16, Chicago, IL, USA, May 2016, pp. 514–523. DOI: 10.1109/IPDPS.2016.106 (cit. on pp. 35, 42).

[58] M. Sutton, T. Ben-Nun, A. Barak, S. Pai, and K. Pingali, "Adaptive work-efficient connected components on the GPU", The Computing Research Repository, Tech. Rep., 2016. [Online]. Available: http://arxiv.org/abs/1612.01178 (cit. on pp. 35, 42).

[59] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular GPGPU graph applications", in *Proceedings of the IEEE International Symposium on Workload Characterization*, ser. IISWC '13, Portland, OR, USA, Sep. 2013, pp. 185–195. DOI: 10.1109/IISWC.2013.6704684 (cit. on p. 35).

[60] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation", in *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, Budapest, HUN, Sep. 2004, pp. 97–104 (cit. on p. 36).

[61] S. S. Mukherjee, S. D. Sharma, M. D. Hill, J. R. Larus, A. Rogers, and J. Saltz, "Efficient support for irregular applications on distributed-memory machines", in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '95, Santa Barbara,

CA, USA: ACM, 1995, pp. 68–79, ISBN: 0-89791-700-6. DOI: 10.1145/209936.209945 (cit. on p. 36).

[62] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs", *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1999. DOI: 10.1137/S1064827595287997 (cit. on pp. 36, 107).

[63] J. Mellor-Crummey, D. Whalley, and K. Kennedy, "Improving memory hierarchy performance for irregular applications using data and computation reorderings", *International Journal of Parallel Programming*, vol. 29, no. 3, pp. 217–247, Jun. 2001, ISSN: 0885-7458. DOI: 10.1023/A:1011119519789 (cit. on pp. 36, 37).

[64] H. Han and C.-W. Tseng, "Exploiting locality for irregular scientific codes", *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 7, pp. 606–618, Jul. 2006, ISSN: 1045-9219. DOI: 10.1109/TPDS.2006.88 (cit. on pp. 37, 41).

[65] B. B. Fraguela, R. Doallo, and E. L. Zapata, "Modeling set associative caches behavior for irregular computations", in *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '98/PERFORMANCE '98, Madison, Wisconsin, USA: ACM, 1998, pp. 192–201, ISBN: 0-89791-982-3. DOI: 10.1145/277851.277910 (cit. on p. 37).

[66] C. Scholtes, "A method to derive the cache performance of irregular applications on machines with direct mapped caches", *International Journal of Computational Science and Engineering*, vol. 1, no. 2-4, pp. 157–174, May 2005, ISSN: 1742-7185. DOI: 10.1504/IJCSE.2005.009700 (cit. on p. 37).

[67] C. Zhang, C. Ding, M. Ogihara, Y. Zhong, and Y. Wu, "A hierarchical model of data locality", in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06, Charleston, SC, USA: ACM, 2006, pp. 16–29, ISBN: 1-59593-027-2. DOI: 10.1145/1111037.1111040 (cit. on p. 37).

[68] A. Lain and P. Banerjee, "Exploiting spatial regularity in irregular iterative applications", in *Proceedings of the 9th International Symposium on Parallel Processing*, ser. IPPS '95, Santa Barbara, CA, USA: IEEE Computer Society, 1995, pp. 820–826, ISBN: 0-8186-7074-6. [Online]. Available: http://dl.acm.org/citation.cfm?id=645605.663225&preflayout=tabs (cit. on p. 37).

[69] C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at run time", *ACM SIGPLAN Notices*, vol. 34, no. 5, pp. 229–241, May 1999, ISSN: 0362-1340. DOI: 10.1145/301631.301670 (cit. on p. 37).

[70] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis, "Locality-aware task management for unstructured parallelism: A quantitative limit study", in *Proceedings of the 25th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '13, Montréal, QB, Canada: ACM, 2013, pp. 315–325, ISBN: 978-1-4503-1572-2. DOI: 10.1145/2486159.2486175 (cit. on p. 37).

[71] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system", in *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95, Santa Barbara, California, USA: ACM, 1995, pp. 207–216, ISBN: 0-89791-700-6. DOI: 10.1145/209936.209958 (cit. on p. 38).

[72] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen, "Solving large, irregular graph problems using adaptive work-stealing", in *Proceedings of the 37th International Conference on Parallel Processing*, ser. ICPP '08, Portland, OR, USA, Sep. 2008, pp. 536–545. DOI: 10.1109/ICPP.2008.88 (cit. on pp. 38, 39).

[73] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger, "KLA: A new algorithmic paradigm for parallel graph computations", in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14, Edmonton, AB, CA: ACM, 2014, pp. 27–38, ISBN:

978-1-4503-2809-8. DOI: 10.1145/2628071.2628091 (cit. on pp. 38–41, 43).

[74] D. T. Neves, "EPIC: A framework to exploit parallelism in irregular codes", English, *Concurrency and Computation - Practice & Experience*, vol. 29, no. 2, Jan. 2017, ISSN: 1532-0626. DOI: 10.1002/cpe.3842 (cit. on p. 38).

[75] D. Gregor and A. Lumsdaine, "The parallel BGL: A generic library for distributed graph computations", in *Parallel Object-Oriented Scientific Computing*, ser. POOSC '05, Jul. 2005. [Online]. Available: http://www.osl.iu.edu/publications/prints/2005/Gregor:POOSC:2005.pdf (cit. on pp. 38, 39).

[76] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "STAPL: Standard template adaptive parallel library", in *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, ser. SYSTOR '10, Haifa, IL: ACM, 2010, 14:1–14:10, ISBN: 978-1-60558-908-4. DOI: 10.1145/1815695.1815713 (cit. on p. 39).

[77] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing", in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10, Indianapolis, IN, USA: ACM, 2010, pp. 135–146, ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807184 (cit. on pp. 39, 41, 43).

[78] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud", *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012, ISSN: 2150-8097. DOI: 10.14778/2212351.2212354 (cit. on pp. 39, 40).

[79] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Power-Graph: Distributed graph-parallel computation on natural graphs", in *Proceedings of the 10th USENIX Conference on Operating Systems Design*

and Implementation, ser. OSDI '12, Hollywood, CA, USA: USENIX Association, 2012, pp. 17–30, ISBN: 978-1-931971-96-6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880 (cit. on pp. 39, 40).

[80] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory", in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, Shenzhen, CN: ACM, 2013, pp. 135–146, ISBN: 978-1-4503-1922-5. DOI: 10.1145/2442516.2442530 (cit. on pp. 39, 40).

[81] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Grappa: A latency-tolerant runtime for large-scale irregular applications", in *Proceedings of the International Workshop on Rack-Scale Computing*, ser. WRSC '14, Amsterdam, NL, Apr. 2014. [Online]. Available: ftp://trout.cs.washington.edu/tr/2014/02/UW-CSE-14-02-01.PDF (cit. on pp. 39–41, 43).

[82] S. Salihoglu and J. Widom, "HelP: High-level primitives for large-scale graph processing", in *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, ser. GRADES '14, Snowbird, UT, USA: ACM, 2014, 3:1–3:6, ISBN: 978-1-4503-2982-8. DOI: 10.1145/2621934.2621938 (cit. on pp. 39, 40).

[83] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "High-performance graph algorithms from parallel sparse matrices", in *Proceedings of the 8th International Conference on Applied Parallel Computing: State of the Art in Scientific Computing*, ser. PARA '06, Umeå, SE: Springer-Verlag, 2006, pp. 260–269, ISBN: 978-3-540-75754-2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1775097 (cit. on pp. 39, 40).

[84] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-Marl: A DSL for easy and efficient graph analysis", in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, London, UK: ACM, 2012, pp. 349–362, ISBN: 978-1-4503-0759-8. DOI: 10.1145/2150976.2151013 (cit. on pp. 39, 41).

[85] MATLAB, *Version 9.2.0.538062 64-bit (r2017a)*, Natick, Massachusetts, 2017 (cit. on pp. 40, 84, 130, 182, 188, 189).

[86] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal", *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 117–128, Feb. 2012, ISSN: 0362-1340. DOI: 10.1145/2370036.2145832 (cit. on pp. 41–43).

[87] M. Christen, O. Schenk, E. Neufeld, P. Messmer, and H. Burkhart, "Parallel data-locality aware stencil computations on modern micro-architectures", in *Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing*, ser. IPDPS '09, Rome, IT, May 2009, pp. 1–10. DOI: 10.1109/IPDPS.2009.5161031 (cit. on p. 42).

[88] S. Unkule, C. Shaltz, and A. Qasem, "Automatic restructuring of GPU kernels for exploiting inter-thread data locality", in *Proceedings of the 21st International Conference on Compiler Construction*, ser. CC '12, Tallinn, EE: Springer-Verlag, 2012, pp. 21–40, ISBN: 978-3-642-28651-3. DOI: 10.1007/978-3-642-28652-0_2 (cit. on p. 42).

[89] S.-Y. Lee and C.-J. Wu, "CAWS: Criticality-aware warp scheduling for GPGPU workloads", in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14, Edmonton, AB, CA: ACM, 2014, pp. 175–186, ISBN: 978-1-4503-2809-8. DOI: 10.1145/2628071.2628107 (cit. on pp. 42, 54).

[90] J. Hbeika and M. Kulkarni, "Locality-aware task-parallel execution on GPUs", in *Proceedings of the 29th International Workshop on Languages and Compilers for Parallel Computing*, C. Ding, J. Criswell, and P. Wu, Eds., ser. LCPC '16. Rochester, NY, USA: Springer International Publishing, 2017, pp. 250–264, ISBN: 978-3-319-52709-3. DOI: 10.1007/978-3-319-52709-3_19 (cit. on p. 42).

[91] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: Vertex-centric graph processing on GPUs", in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14, Vancouver, BC, CA: ACM, 2014, pp. 239–252, ISBN: 978-1-4503-2749-7. DOI: 10.1145/2600212.2600227 (cit. on pp. 43, 55).

[92] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus", *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, Jun. 2014, ISSN: 1045-9219. DOI: 10.1109/TPDS.2013.111 (cit. on pp. 43, 55).

[93] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU", in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16, Barcelona, ES: ACM, 2016, 11:1–11:12, ISBN: 978-1-4503-4092-2. DOI: 10.1145/2851141.2851145 (cit. on pp. 43, 55).

[94] R. Diestel, *Graph Theory*, 3rd ed., ser. Graduate Texts in Mathematics. Springer–Verlag Heidelberg, Aug. 2005, vol. 173, ISBN: 3540261826 (cit. on pp. 44, 195, 201).

[95] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. Baltimore, MD, USA: Johns Hopkins University Press, 1996, ISBN: 0-8018-5414-8 (cit. on pp. 45, 55).

[96] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on GPGPUs", *ACM Transactions on Mathematical Software*, vol. 43, no. 4, 30:1–30:49, Jan. 2017, ISSN: 0098-3500. DOI: 10.1145/3017994 (cit. on pp. 46, 47).

[97] D. R. Kincaid, T. C. Oppe, and D. M. Young, *ITPACKV 2D user's guide*, May 1989. [Online]. Available: http://www.netlib.no/netlib/itpack/index.html (cit. on pp. 47, 49, 133).

[98] M. R. Hugues and S. G. Petiton, "Sparse matrix formats evaluation and optimization on a GPU", in *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications*, ser. HPCC '10, Melbourne, AUS, Sep. 2010, pp. 122–129. DOI: 10.1109/HPCC.2010.85 (cit. on p. 47).

[99] D. Langr and P. Tvrdik, "Evaluation criteria for sparse matrix storage formats", *IEEETransactions on Parallel and Distributed Systems*, vol. 27,

no. 2, pp. 428–440, Feb. 2016, ISSN: 1045-9219. DOI: 10.1109/TPDS.2015.2401575 (cit. on p. 47).

[100] M. M. Baskaran and R. Bordawekar, "Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies", IBM Research Division, Tech. Rep., 2008. [Online]. Available: http://domino.watson.ibm.com/library/CyberDig.nsf/1e4115aea78b6e7c85256b360066f0d4/1d32f6d23b99f7898525752200618339?OpenDocument (cit. on p. 47).

[101] Y. Zhang, Y. H. Shalabi, R. Jain, K. K. Nagar, and J. D. Bakos, "FPGA vs. GPU for sparse matrix vector multiply", in *Proceedings of the 2009 International Conference on Field-Programmable Technology*, ser. FPT '09, Sydney, NSW, AUS, Dec. 2009, pp. 255–262. DOI: 10.1109/FPT.2009.5377620 (cit. on p. 49).

[102] S. Li, Y. Wang, W. Wen, Y. Wang, Y. Chen, and H. Li, "A data locality-aware design framework for reconfigurable sparse matrix-vector multiplication kernel", in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Austin, TX, US, Nov. 2016, pp. 1–6. DOI: 10.1145/2966986.2966987 (cit. on p. 49).

[103] A. Rafique, G. A. Constantinides, and N. Kapre, "Communication optimization of iterative sparse matrix-vector multiply on GPUs and FPGAs", *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 24–34, Jan. 2015, ISSN: 1045-9219. DOI: 10.1109/TPDS.2014.6 (cit. on p. 49).

[104] I. Reguly and M. Giles, "Efficient sparse matrix-vector multiplication on cache-based GPUs", in *Proceedings of Innovative Parallel Computing*, ser. InPar '12, San Jose, CA, USA, May 2012, pp. 1–12. DOI: 10.1109/InPar.2012.6339602 (cit. on pp. 49, 50).

[105] W. Liu and B. Vinter, "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication", in *Proceedings of the 29th ACM International Conference on Supercomputing*, ser. ICS '15, Newport Beach, CA, USA: ACM, 2015, pp. 339–350, ISBN: 978-1-4503-3559-1. DOI: 10.1145/2751205.2751209 (cit. on p. 50).

[106] M. Daga and J. L. Greathouse, "Structural agnostic SpMV: Adapting CSR-adaptive for irregular matrices", in *Proceedings of the 22nd IEEE International Conference on High Performance Computing*, ser. HiPC '15, Bengaluru, IN, Dec. 2015, pp. 64–74. DOI: 10.1109/HiPC.2015.55 (cit. on p. 50).

[107] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for GPU architectures", in *High Performance Embedded Architectures and Compilers, HiPEAC '10*, Y. N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, Eds., ser. Lecture Notes in Computer Science, Pisa, IT: Springer Berlin Heidelberg, 2010, pp. 111–125, ISBN: 978-3-642-11515-8. DOI: 10.1007/978-3-642-11515-8_10. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11515-8_10 (cit. on p. 50).

[108] M. Kreutzer, G. Hazen, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units", *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014. DOI: 10.1137/130930352 (cit. on pp. 50, 51).

[109] W. T. Tang, W. J. Tan, R. Ray, Y. W. Wong, W. Chen, S.-h. Kuo, R. S. M. Goh, S. J. Turner, and W.-F. Wong, "Accelerating sparse matrix-vector multiplication on GPUs using bit-representation-optimized schemes", in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, Denver, CO, USA: ACM, 2013, 26:1–26:12, ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503234 (cit. on p. 50).

[110] P. Gottschling. (2013). CUDA-MTL4 manual. Accessed on 10 May 2017, SimuNova UG, [Online]. Available: http://www.simunova.com/node/300 (cit. on p. 51).

[111] (2014). The sparse parallel robust algorithms library (SPRAL), Numerical Analysis Group, Rutherford Appleton Laboratory, [Online]. Avail-

able: http://www.numerical.rl.ac.uk/spral/ (cit. on pp. 51, 52, 134).

[112] (2016). clSPARSE documentation. version 0.10.0.0. Accessed on 10 May 2017, [Online]. Available: http://clmathlibraries.github.io/clSPARSE/ (cit. on pp. 51, 52).

[113] D. C.-L. Fong and M. Saunders, "LSMR: An iterative algorithm for sparse least-squares problems", *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2950–2971, 2011. DOI: 10.1137/10079687X. [Online]. Available: https://doi.org/10.1137/10079687X (cit. on p. 52).

[114] J. D. Hogg, E. Ovtchinnikov, and J. A. Scott, "A sparse symmetric indefinite direct solver for GPU architectures", *ACM Transactions on Mathematical Software*, vol. 42, no. 1, 1:1–1:25, Jan. 2016, ISSN: 0098-3500. DOI: 10.1145/2756548 (cit. on pp. 52, 55, 56).

[115] B. W. Peyton, A. Pothen, and X. Yuan, "Partitioning a chordal graph into transitive subgraphs for parallel sparse triangular solution", *Linear Algebra and its Applications*, vol. 192, no. 0, pp. 329–353, 1993, ISSN: 0024-3795. DOI: 10.1016/0024-3795(93)90248-M. [Online]. Available: http://www.sciencedirect.com/science/article/pii/002437959390248M (cit. on pp. 52, 56).

[116] F. L. Alvarado, A. Pothen, and R. Schreiber, "Highly parallel sparse triangular solution", English, in *Graph Theory and Sparse Matrix Computation*, ser. The IMA Volumes in Mathematics and its Applications, A. George, J. R. Gilbert, and J. W. H. Liu, Eds., vol. 56, Springer New York, 1993, pp. 141–157, ISBN: 978-1-4613-8371-0. DOI: 10.1007/978-1-4613-8369-7_7 (cit. on p. 53).

[117] O. Temam and W. Jalby, "Characterizing the behaviour of sparse algorithms on caches", in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '92, Minneapolis, MN, USA, 1992, pp. 578–587 (cit. on p. 53).

[118] P. M. W. Knijnenburg and H. A. G. Wijshoff, "On improving data locality in sparse matrix computations", High Performance Computing

Division, Dept. of Computer Science, Leiden University, Tech. Rep., 1994. [Online]. Available: http://liacs.leidenuniv.nl/assets/PDF/TechRep/tr94-15.pdf (cit. on p. 53).

[119] M. M. Strout, L. Carter, and J. Ferrante, "Rescheduling for locality in sparse matrix computations", in *Proceedings of the 2001 International Conference on Computational Science*, V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. J. K. Tan, Eds., ser. ICCS '01. San Francisco, CA, USA: Springer Berlin Heidelberg, 2001, pp. 137–146, ISBN: 978-3-540-45545-5. DOI: 10.1007/3-540-45545-0_23 (cit. on pp. 53, 56).

[120] R. Vuduc, S. Kamil, J. Hsu, R. Nishtala, J. W. Demmel, and K. A. Yelick, "Automatic performance tuning and analysis of sparse triangular solve", in *Proceeding of the Workshop on Performance Optimization of High-level Languages and Libraries (POHLL) at the ACM International Conference on Supercomputing*, ser. ICS '02, Venice, IT, Jun. 2002. [Online]. Available: http://bebop.cs.berkeley.edu/pubs/vuduc2002-sts-bounds.pdf (cit. on p. 53).

[121] M. M. Wolf, M. A. Heroux, and E. G. Boman, "Factors impacting performance of multithreaded sparse triangular solve", in *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, ser. VECPAR '10, Berkeley, CA, USA: Springer-Verlag, 2011, pp. 32–44, ISBN: 978-3-642-19327-9. [Online]. Available: http://dl.acm.org/citation.cfm?id=1964246 (cit. on pp. 53, 56).

[122] B. Suchoski, C. Severn, M. Shantharam, and P. Raghavan, "Adapting sparse triangular solution to GPUs", in *Proceeding of the 41st International Conference on Parallel Processing Workshops*, ser. ICPPW '12, Los Alamitos, CA, USA, Sep. 2012, pp. 140–148. DOI: 10.1109/ICPPW.2012.23 (cit. on pp. 53–56).

[123] E. Totoni, M. T. Health, and L. V. Kale, "Structure-adaptive parallel solution of sparse triangular linear systems", *Parallel Computing*, vol. 40, no. 9, pp. 454–470, Oct. 2014. DOI: 10.1016/j.parco.2014.06.006 (cit. on pp. 53, 56).

[124] J. Mayer, "Parallel algorithms for solving linear systems with sparse triangular matrices", *Computing*, vol. 86, no. 4, pp. 291–312, Sep. 2009, ISSN: 0010-485X. DOI: 10.1007/s00607-009-0066-3 (cit. on pp. 54, 56).

[125] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU", in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, Saint-Malo, FR: ACM, 2010, pp. 451–460, ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1816021 (cit. on p. 54).

[126] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers", *Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, Feb. 2013, ISSN: 0920-8542. DOI: 10.1007/s11227-012-0825-3 (cit. on pp. 55, 133).

[127] J. Hogg and J. Scott, "New parallel sparse direct solvers for multicore architectures", *Algorithms*, vol. 6, no. 4, pp. 702–725, 2013, ISSN: 1999-4893. DOI: 10.3390/a6040702. [Online]. Available: http://www.mdpi.com/1999-4893/6/4/702 (cit. on p. 55).

[128] Z. Chen, H. Liu, and B. Yang, "Parallel triangular solvers on GPU", The Computing Research Repository, Tech. Rep., 2016. [Online]. Available: http://arxiv.org/abs/1606.00541 (cit. on pp. 55, 56).

[129] *Profiler user's guide*, 7.0, Accessed 12 July 2017., NVIDIA corporation, Mar. 2015. [Online]. Available: http://docs.nvidia.com/cuda/profiler-users-guide/index.html#axzz4mitW5BwQ (cit. on pp. 65, 71, 73, 89, 90).

[130] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking", in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software*, ser. ISPASS '10, Mar. 2010, pp. 235–246. DOI: 10.1109/ISPASS.2010.5452013 (cit. on p. 67).

[131] *CUDA C programming guide*, 7.5, NVIDIA, Oct. 2015. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz4mitW5BwQ (cit. on pp. 67, 68).

[132] *NVIDIA professional graphics solution*, Online, NVIDIA Corporation, Jul. 2013. [Online]. Available: http://www.nvidia.co.uk/content/PDF/line_card/6660-nv-prographicssolutions-linecard-july13-final-lr.pdf (cit. on pp. 67, 68, 72, 84, 88, 92, 156).

[133] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection", *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, Dec. 2011, ISSN: 0098-3500. DOI: 10.1145/2049662.2049663 (cit. on pp. 71–73, 85, 86, 89, 91, 93, 115, 117, 137).

[134] C. Lawson and R. Hanson, *Solving Least Squares Problems*. Society for Industrial and Applied Mathematics, 1995. DOI: 10.1137/1.9781611971217. eprint: http://epubs.siam.org/doi/pdf/10.1137/1.9781611971217 (cit. on pp. 72, 174, 182).

[135] R. H. Byrd, J. C. Gilbert, and J. Nocedal, "A trust region method based on interior point techniques for nonlinear programming", *Mathematical Programming*, vol. 89, no. 1, pp. 149–185, Nov. 2000, ISSN: 1436-4646. DOI: 10.1007/PL00011391 (cit. on p. 81).

[136] *NVIDIA tesla GPU accelerators*, Online, Accessed 13 October 2016, NVIDIA corporation, 2014. [Online]. Available: http://international.download.nvidia.com/pdf/kepler/TeslaK80-datasheet.pdf (cit. on pp. 87, 115, 130, 173, 182, 183).

[137] *AMD Firepro W5000*, Online, AMD Corporation, 2014. [Online]. Available: http://www.amd.com/documents/2795_W5000_DataSheet_R3.pdf (cit. on p. 92).

[138] R. Tarjan, "Depth-first search and linear graph algorithms", *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, Jun. 1972 (cit. on p. 99).

[139] T. Achterberg, "Constraint integer programming", PhD thesis, Technische Universität Berlin, 2008. [Online]. Available: https://opus4.kobv.de/opus4-zib/frontdoor/index/index/docId/1112 (cit. on pp. 107, 137, 156, 157, 159, 182, 183, 190).

[140] *Intel Xeon processor E5-2640 v3*, Online, Accessed 13 October 2016, Intel Corporation, 2014. [Online]. Available: http://ark.intel.com/products/83359/Intel-Xeon-Processor-E5-2640-v3-20M-Cache-2_60-GHz (cit. on pp. 115, 118, 130, 173, 182, 183).

[141] G. Rodriguez, *Algoritmi numerici*. Pitagora Editrice, 2008, p. 117, ISBN: 88-371-1714-0 (cit. on p. 120).

[142] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain", Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep., 1994. [Online]. Available: http://www.cs.cmu.edu/~./quake-papers/painless-conjugate-gradient.pdf (cit. on pp. 122, 125).

[143] J. W. Eaton, D. Bateman, S. Hauberg, and R. Wehbring, *GNU octave version 4.0.0 manual: A high-level interactive language for numerical computations*, Accessed 20 July 2016, 2015. [Online]. Available: http://www.gnu.org/software/octave/doc/interpreter (cit. on pp. 123, 129).

[144] *Intel core i7-3770 processor*, Online, Accessed 13 October 2016, Intel Corporation, 2012. [Online]. Available: http://ark.intel.com/products/65719/Intel-Core-i7-3770-Processor-8M-Cache-up-to-3_90-GHz (cit. on pp. 129, 137, 170, 172).

[145] *NVIDIA GeForce GT 430*, Online, Accessed 13 October 2016, NVIDIA corporation, 2010. [Online]. Available: http://www.nvidia.com/object/product-geforce-gt-430-oem-us.html (cit. on p. 129).

[146] *Imperial college high performance computing service*. [Online]. Available: http://www.imperial.ac.uk/admin-services/ict/self-service/research-support/hpc/ (cit. on pp. 130, 173, 182, 183).

[147] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra, "Matrix market: A web resource for test matrix collections", in *Proceedings of the IFIP TC2/WG2.5 Working Conference on Quality of Numerical Software: Assessment and Enhancement*, Oxford, UK: Chapman & Hall, Ltd., 1997, pp. 125–137, ISBN: 0-412-80530-8. [Online]. Available: http://dl.acm.org/citation.cfm?id=265834.265854 (cit. on p. 132).

[148] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the GPU: Conjugate gradients and multigrid", in *ACM SIGGRAPH 2003 Papers*, ser. SIGGRAPH '03, San Diego, California: ACM, 2003, pp. 917–924, ISBN: 1-58113-709-5. DOI: 10.1145/1201775.882364 (cit. on p. 133).

[149] L. Buatois, G. Caumon, and B. Lévy, "Concurrent number cruncher: A GPU implementation of a general sparse linear solver", *International Journal of Parallel, Emergent and Distributed Systems*, vol. 24, no. 3, pp. 205–223, 2009. DOI: 10.1080/17445760802337010 (cit. on p. 133).

[150] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure, "On the limits of GPU acceleration", in *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism*, ser. HotPar'10, Berkeley, CA, USA: USENIX Association, 2010, pp. 13–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863086.1863099 (cit. on p. 134).

[151] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems", *ACM Transactions on Mathematical Software*, vol. 37, no. 3, pp. 1–17, Sep. 2010, ISSN: 0098-3500. DOI: 10.1145/1824801.1824814 (cit. on pp. 137, 156).

[152] *Intel core i3-2130 processor*, Online, Accessed 13 October 2016, Intel Corporation, 2011. [Online]. Available: http://ark.intel.com/products/53428/Intel-Core-i3-2130-Processor-3M-Cache-3_40-GHz (cit. on p. 155).

[153] J. Nocedal and S. Wright, *Numerical Optimization*, 2nd. New York: Springer, 2006 (cit. on pp. 159, 160, 162, 163).

[154] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali, *Linear Programming and Network Flows*. Wiley, 2010 (cit. on pp. 159, 160, 162, 163).

[155] J. J. H. Forrest and J. A. Tomlin, "Updated triangular factors of the basis to maintain sparsity in the product form simplex method", *Mathematical Programming*, vol. 2, no. 1, pp. 263–278, 1972, ISSN: 1436-4646. DOI: 10.1007/BF01584548 (cit. on pp. 159, 163, 167, 169).

[156] L. M. Suhl and U. H. Suhl, "A fast LU update for linear programming", *Annals of Operations Research*, vol. 43, no. 1, pp. 33–47, 1993, ISSN: 0254-5330. DOI: 10.1007/BF02025534 (cit. on pp. 159, 163, 168).

[157] H. M. Markowitz, "The elimination form of the inverse and its application to linear programming", *Management Science*, vol. 3, no. 3, pp. 255–269, Apr. 1957, ISSN: 0025-1909. DOI: 10.1287/mnsc.3.3.255 (cit. on pp. 159, 163, 165, 172, 193).

[158] I. Maros, *Computational Techniques of the Simplex Method*, Springer, Ed., ser. International Series in Operations Research & Management Science. Kluwer Academic, 2003, vol. 61 (cit. on pp. 159, 160, 162, 163, 165, 172, 190, 193).

[159] R. Wunderling, "Paralleler und objektorientierter Simplex-Algorithmus", German, PhD thesis, Technische Universität Berlin, 1996 (cit. on pp. 163, 181–183, 190).

[160] Q. Huangfu and J. Hall, "Novel update techniques for the revised simplex method", *Computational Optimization and Applications*, vol. 60, no. 3, pp. 587–608, 2015. DOI: 10.1007/s10589-014-9689-1 (cit. on p. 167).

[161] J. Osier, *GNU gprof*, Free Software Foundation, Inc., Jan. 1993. [Online]. Available: https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_chapter/gprof_1.html (cit. on pp. 170, 172).

[162] A. Picciau. (2016). Smart building optimisation problems. Git repository, Accessed 10 May 2017, [Online]. Available: https://github.com/andpic/smart-building-optimisation (cit. on pp. 173, 182–184).

[163] C. Zhao, S. Dong, F. Li, and Y. Song, "Optimal home energy management system with mixed types of loads", *CSEE Journal of Power and Energy Systems*, PP, vol. 1, no. 4, pp. 29–37, Dec. 2015. DOI: 10.17775/CSEEJPES.2015.00045 (cit. on pp. 173, 182–184).

[164] G. Gamrath, T. Koch, A. Martin, M. Miltenberger, and D. Weninger, "Progress in presolving for mixed integer programming", *Mathematical Programming Computation*, vol. 7, no. 4, pp. 367–398, 2015, ISSN: 1867–2957. DOI: 10.1007/s12532-015-0083-5 (cit. on p. 183).

[165] S. Bayliss, C.-S. Bouganis, G. A. Constantinides, and W. Luk, "An FPGA implementation of the simplex algorithm", in *Proceedings of the IEEE International Conference on Field Programmable Technology*, ser. FPT '06, Bangkok, TH, Dec. 2006, pp. 49–56. DOI: 10.1109/FPT.2006.270294 (cit. on pp. 187, 189).

[166] D. D. Spampinato and A. C. Elster, "Linear optimization on modern GPUs", in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, ser. IPDPS '09, Rome, IT, May 2009, pp. 1–8. DOI: 10.1109/IPDPS.2009.5161106 (cit. on pp. 187, 188).

[167] J. Bieling, P. Peschlow, and P. Martini, "An efficient GPU implementation of the revised simplex method", in *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '10, Atlanta, GE, USA, 2010, pp. 1–8. DOI: 10.1109/IPDPSW.2010.5470831 (cit. on pp. 187, 188).

[168] T. Carneiro, A. E. Muritiba, M. Negreiros, and G. A. L. de Campos, "A new parallel schema for branch-and-bound algorithms using GPGPU", in *Proceedings of the 23rd International Symposium on Computer Architecture and High Performance Computing*, ser. SBAC-PAD '11, Vitória, Espírito Santo, BR, Oct. 2011, pp. 41–47. DOI: 10.1109/SBAC-PAD.2011.20 (cit. on pp. 187, 188, 190).

[169] A. Hamzić, A. Huseinović, and N. Nosović, "Implementation and performance analysis of the simplex algorithm adapted to run on commodity OpenCL enabled graphics processors", in *Proceedings of the XXIII International Symposium on Information Communication and Automation Technologies*, ser. ICAT '11, Sarajevo, BIH, Oct. 2011, pp. 1–7. DOI: 10.1109/ICAT.2011.6102135 (cit. on pp. 187, 188).

[170] X. Meyer, P. Albuquerque, and B. Chopard, "A multi-GPU implementation and performance model for the standard simplex method", in *Proceedings of the 1st International Symposium and 10th Balkan Conference on Operational Research*, ser. BALCOR '11, Thessaloniki, GR, Sep. 2011,

pp. 312–319. [Online]. Available: http://spc.unige.ch/lib/exe/fetch.php?media=pub:sgpu_europar2011.pdf (cit. on pp. 187, 188).

[171] M. E. Lalami, D. El-Baz, and V. Boyer, "Multi GPU implementation of the simplex algorithm", in *Proceedings of the 13th IEEE International Conference on High Performance Computing and Communications*, ser. HPCC '11, Banff, AB, CA, Sep. 2011, pp. 179–186. DOI: 10.1109/HPCC.2011.32 (cit. on pp. 187, 188).

[172] M. E. Lalami, V. Boyer, and D. El-Baz, "Efficient implementation of the simplex method on a CPU–GPU system", in *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '11, Anchorage, AL, USA, May 2011, pp. 1999–2006, ISBN: 978-1-61284-425-1. DOI: 10.1109/IPDPS.2011.362 (cit. on pp. 186–188).

[173] J. Hall and Q. Huangfu, "A high performance dual revised simplex solver", English, in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, Eds., vol. 7203, Springer Berlin Heidelberg, 2012, pp. 143–151, ISBN: 978-3-642-31463-6. DOI: 10.1007/978-3-642-31464-3_15 (cit. on pp. 187, 189).

[174] A. Boukedjar, M. E. Lalami, and D. El-Baz, "Parallel branch and bound on a CPU–GPU system", in *Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, ser. PDP '12, Garching, DE, Feb. 2012, pp. 392–398. DOI: 10.1109/PDP.2012.23 (cit. on pp. 187, 188, 190).

[175] M. E. Lalami and D. El-Baz, "GPU implementation of the branch and bound method for knapsack problems", in *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium Workshops and PhD Forum*, ser. IPDPSW '12, Shanghai, CN, 2012, pp. 1769–1777. DOI: 10.1109/IPDPSW.2012.219 (cit. on pp. 187, 188, 190).

[176] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, and T. Koch, "ParaSCIP: A parallel extension of SCIP", in *Competence in High Performance Computing*, C. Bischof, H.-G. Hegering, W. Nagel, and G. Wittum, Eds.,

2012, pp. 135–148. DOI: 10.1007/978-3-642-24025-6_12 (cit. on pp. 187, 190).

[177] J. L. Jerez, G. A. Constantinides, and E. C. Kerrigan, "Towards a fixed point QP solver for predictive control", in *Proceedings of the 51st IEEE Annual Conference on Decision and Control*, ser. CDC '12, Maui, HI, USA, Dec. 2012, pp. 675–680. DOI: 10.1109/CDC.2012.6427015 (cit. on pp. 187, 190).

[178] I. Chakroun, M. Mezmaz, N. Melab, and A. Bendjoudi, "Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm", *Concurrency and Computation: Practice and Experience*, vol. 25, no. 8, pp. 1121–1136, Sep. 2012. DOI: 10.1002/cpe.2931 (cit. on pp. 187, 188, 190).

[179] I. Chakroun, N. Melab, M. Mezmaz, and D. Tuyttens, "Combining multi-core and GPU computing for solving combinatorial optimization problems", *Journal of Parallel and Distributed computing*, vol. 73, no. 12, pp. 1563–1577, Dec. 2013, ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2013.07.023 (cit. on pp. 187, 188, 190).

[180] Y. Shinano, S. Heinz, S. Vigerske, and M. Winkler, "FiberSCIP: A shared memory parallelization of SCIP", ZIB, Takustr.7, 14195 Berlin, Tech. Rep. 13–55, 2013 (cit. on pp. 187, 190).

[181] X. Meyer, B. Chopard, and P. Albuquerque, "Linear programming on a GPU: A case study", in *Designing Scientific Applications on GPUs*, R. Couturier, Ed., ser. Numerical Analysis and Scientific Computing. Chapman & Hall/CRC Press, 2013, ch. 10, pp. 215–249, ISBN: 9781466571624 (cit. on pp. 187–189).

[182] E. Smith, "Parallel solution of linear programs", PhD thesis, University of Edinburgh, 2013. [Online]. Available: http://hdl.handle.net/1842/8833 (cit. on pp. 187–189).

[183] N. Ploskas and N. Samaras, "Efficient GPU-based implementations of simplex type algorithms", *Applied Mathematics and Computation*, vol. 250, pp. 552–570, 2015, ISSN: 0096-3003. DOI: 10.1016/j.amc.2014.10.096 (cit. on pp. 187–189).

[184] *GNU linear programming kit reference manual*, Free Software Foundation, 2010. [Online]. Available: http://kam.mff.cuni.cz/~elias/glpk.pdf (cit. on p. 188).

[185] *Gurobi optimizer reference manual*, version 7.0, Gurobi Optimization, Dec. 2017. [Online]. Available: http://www.gurobi.com/documentation/7.0/refman.pdf (cit. on p. 188).

[186] T. Koch, T. Ralphs, and Y. Shinano, "Could we use a million cores to solve an integer program?", *Mathematical Methods of Operations Research*, vol. 76, no. 1, pp. 67–93, 2012, ISSN: 1432-5217. DOI: 10.1007/s00186-012-0390-9 (cit. on p. 186).

[187] V. Boyer and D. El Baz, "Recent advances on GPU computing in operations research", in *Proceedings of the 27th IEEE International Symposium on Parallel & Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '13, Boston, MA, USA: IEEE Computer Society, May 2013, pp. 1778–1787. DOI: 10.1109/IPDPSW.2013.45 (cit. on p. 186).

[188] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1982, ISBN: 0-13-152462-3 (cit. on pp. 189, 190).

[189] J. Forrest, D. de la Nuez, and R. Lougee-Heimer, *CLP user guide*, IBM Research, 2004. [Online]. Available: http://www.coin-or.org/Clp/userguide/index.html (cit. on p. 189).

[190] N. Ploskas and N. Samaras, "A computational comparison of basis updating schemes for the simplex algorithm on a CPU-GPU system", *American Journal of Operations Research*, vol. 3, no. 6, pp. 497–505, Nov. 2013. DOI: 10.4236/ajor.2013.36048 (cit. on p. 189).

[191] ——, "GPU accelerated pivoting rules for the simplex algorithm", *Journal of Systems and Software*, vol. 96, pp. 1–9, 2014, ISSN: 0164-1212. DOI: 10.1016/j.jss.2014.04.047. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121214001174 (cit. on p. 189).

[192] J. Hall, "Towards a practical parallelisation of the simplex method", English, *Computational Management Science*, vol. 7, no. 2, pp. 139–170, 2010, ISSN: 1619–697X. DOI: 10.1007/s10287-008-0080-5 (cit. on p. 189).

[193] J. L. Jerez, G. A. Constantinides, E. C. Kerrigan, and K.-V. Ling, "Parallel MPC for real-time FPGA-based implementation", in *Proceedings of the 18th IFAC World Congress*, vol. 18, Milan, IT, Sep. 2011, pp. 1338–1345. DOI: 10.3182/20110828-6-IT-1002.01392 (cit. on p. 190).

[194] N. Melab, I. Chakroun, M. Mezmaz, and D. Tuyttens, "A GPU-accelerated branch-and-bound algorithm for the flow-shop scheduling problem", in *IEEE Int. Conf. Cluster Computing*, ser. CLUSTER '12, Beijing, CN, Sep. 2012, pp. 10–17. DOI: 10.1109/CLUSTER.2012.18 (cit. on p. 190).

[195] I. Chakroun and N. Melab, "An adaptive multi-GPU based branch-and-bound. a case study: The flow-shop scheduling problem", in *Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications*, ser. HPCC '12, Liverpool, UK, Jun. 2012, pp. 389–395. DOI: 10.1109/HPCC.2012.59 (cit. on p. 190).

[196] "The compute architecture of intel processor graphics Gen9", Intel Corporation, Tech. Rep., 2015. [Online]. Available: https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf (cit. on p. 191).