

GiMMiK - Generating Bespoke Matrix Multiplication Kernels for Accelerators: Application to High-Order Computational Fluid Dynamics

Bartosz D. Wozniak^a, Freddie D. Witherden^b, Francis P. Russell^a, Peter E. Vincent^b,
Paul H. J. Kelly^a

^a*Department of Computing, Imperial College London*

^b*Department of Aeronautics, Imperial College London*

Abstract

Matrix multiplication is a fundamental linear algebra routine ubiquitous in all areas of science and engineering. Highly optimised BLAS libraries (cuBLAS and clBLAS on GPUs) are the most popular choices for an implementation of the General Matrix Multiply (GEMM) in software. In this paper we present GiMMiK - a generator of bespoke matrix multiplication kernels for the CUDA and OpenCL platforms. GiMMiK exploits a prior knowledge of the operator matrix to generate highly performant code. The performance of GiMMiK's kernels is particularly apparent in a *block-by-panel* type of matrix multiplication, where the block matrix is typically small (e.g. dimensions of 96×64). Such operations are characteristic to our motivating application in PyFR - an implementation of Flux Reconstruction schemes for high-order fluid flow simulations on mixed unstructured meshes. GiMMiK fully unrolls the matrix-vector product and embeds matrix entries directly in the code to benefit from the use of the constant cache and compiler optimisations. Further, it reduces the number of floating-point operations by removing multiplications by zeros. Together with the ability of our kernels to avoid the poorly optimised cleanup code, executed by library GEMM, we are able to outperform cuBLAS on two NVIDIA GPUs: GTX 780 Ti and Tesla K40c. We observe speedups of our kernels over cuBLAS GEMM of up to 9.98 and 63.30 times for a 294×1029 99% sparse PyFR matrix in double precision on the Tesla K40c and GTX 780 Ti correspondingly. In single precision, observed speedups reach 12.20 and 13.07 times for a 4×8 50% sparse PyFR matrix on the two aforementioned cards. Using GiMMiK as the matrix multiplication kernel provider allows us to achieve a speedup of up to 1.70 (2.19) for a simulation of an unsteady flow over a cylinder executed with PyFR in double (single) precision on the Tesla K40c. All results were generated with GiMMiK version 1.0.

Keywords: Matrix Multiplication, GPU, CUDA, Code Generation, GEMM, GiMMiK, cuBLAS

Email addresses: bartosz.wozniak10@imperial.ac.uk (Bartosz D. Wozniak),
freddie.witherden08@imperial.ac.uk (Freddie D. Witherden), francis.russell02@imperial.ac.uk
(Francis P. Russell), p.vincent@imperial.ac.uk (Peter E. Vincent), p.kelly@imperial.ac.uk (Paul H. J. Kelly)

1. Introduction

Matrix multiplication is ubiquitous in all spheres of science and engineering, hence the need for efficient and performant implementations of such operations in software. Significant effort has been expended in building and optimising *Basic Linear Algebra Subprograms* (BLAS) libraries [1, 2]. The *General Matrix Multiplication* (GEMM) subroutine of level-3 BLAS is among the most popular choices for an implementation of the matrix product. However, GEMM is very generic and usually performs best with large problem sizes [3, 4, 5]. In situations where the matrices are known a priori, faster implementations can be achieved. In this paper we are interested in a *block-by-panel* type of matrix multiplication, where the operator matrix is typically small (e.g. dimensions of 96×64). This is motivated by an application in Flux Reconstruction [6] schemes for high-order fluid flow simulations on unstructured grids. However, given the ubiquity of GEMM we envisage that a variety of other applications within various fields of engineering could also benefit from our research. In this paper we present GiMMiK¹ - a generator of matrix multiplication kernels. GiMMiK analyses a given operator matrix and generates optimised and highly performant CUDA and OpenCL kernel code that can run across a variety of hardware accelerators.

In 2007 Huynh [6] presented the Flux Reconstruction (FR) approach, a unifying mathematical framework allowing for an efficient development of high-order schemes for numerical fluid flow simulations. PyFR is an open-source Python implementation of the FR schemes, which in a performance-portable way targets various hardware accelerators [7]. PyFR is capable of solving compressible Navier-Stokes equations on mixed unstructured grids, which has key applications within the field of Computational Fluid Dynamics (CFD). The nature of the Flux Reconstruction approach allows us to cast many of the computation steps into a form of matrix-matrix multiplication, which makes a GPU implementation of these schemes very attractive. The required multiplication is of the form $C = \alpha A \times B + \beta C$, where A is the operator matrix (which may be sparse or dense), B is the operand matrix and C - the output matrix.

There are a number of highly optimised BLAS libraries, which can be employed to solve this problem. NVIDIA cuBLAS GEMM [8] or the OpenCL clBLAS GEMM [9] are the obvious candidates for the GPU platform. However, Castonguay et al. [10] have shown that bespoke matrix multiplication kernels can grant performance benefits to their solver over BLAS. The problem with the available, highly optimised BLAS libraries is not sub-optimal implementation per se but rather their general nature. For sparse matrix operations on the GPU, performance improvements have been achieved through auto-tuning and modifications to storage layout, though this work has mainly considered sparse matrix-vector multiplication (SpMV) [11, 12, 13, 14]. Application of a sparse operator matrix in PyFR is equivalent to a sparse matrix-matrix multiplication (SpMM). Work on SpMM has primarily considered the effect of different matrix storage formats applied to relatively large operator matrices [15, 16]. In contrast, GiMMiK generates code specialised to matrix structure and content, and is concerned with much smaller matrices.

Each time step of a simulation performed by PyFR amounts to a repeated application of the same set of five operator matrices to the data, combined with some element-local operations [7]. Application of the operator matrix is illustrated in Figure 1. These operations share a set of characteristics known prior to the start of the simulation, which opens up an opportunity to analyse them and generate bespoke matrix multiplication kernels capable of outperforming BLAS

¹Software available at <https://github.com/vincentlab/GiMMiK>. GiMMiK version 1.0 as used in this paper is tagged as “v1.0”.

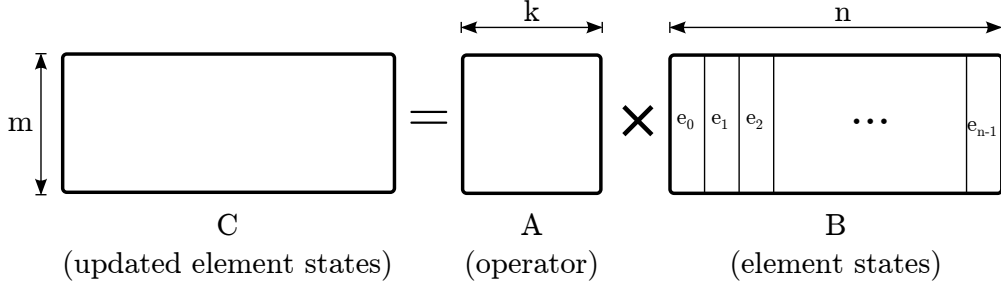


Figure 1: Application of the operator matrix in PyFR ($\alpha = 1$ and $\beta = 0$ for clarity). The block-shaped operator matrix A , which may be dense or sparse, is applied to the dense panel-shaped matrix B containing data for large numbers of elements (n). Typically $n \approx 50,000$, but may range as high as 250,000. The operator matrices remain constant for a given choice of numerical scheme. Typical values of m and k may be found in Appendix A.

library calls. The exact characteristics of these matrices depend on many numerical method choices i.e. the shape and dimensionality of the mesh elements, the desired polynomial order and the type of equations used to solve the problem. The performance of the solver largely depends on the matrix multiplication operations, which constitute a dominant proportion of the total number of operations performed. This allows us to navigate the numerical scheme choices freely, without incurring any performance penalties.

The parameters of the numerical schemes dictate the size of the operator matrices, which are typically small. For hexahedral meshes they range from (4×8) to (96×64) and up to (1029×343) for the first, third and sixth polynomial order correspondingly. The full specification of the characteristics of these matrices is available in Appendix A. The operator matrices stay constant for the duration of the simulation - the same set of matrices is applied to the data at each time step. Further, the types of elements in the mesh directly correspond to the entries in the operator matrix. These entries are also known in advance. It follows that we also know the positions of all zeros in the matrices (zero entries are present only for quadrilateral and hexahedral meshes due to their tensor product formulation of the solution points). Hence, knowing the operator matrices a priori we can allow some time at the beginning of the simulation to analyse them and generate bespoke, highly specialised kernels for each matrix. The time required to perform code generation is negligible compared to the time of the actual simulation. These kernels have the capacity to outperform state-of-the-art BLAS libraries. As the implementation platform we choose GPUs due to their inherently high floating-point performance and memory bandwidth, which make them particularly suitable for matrix multiplication type operations.

The contributions of this paper are:

- We show how, having a prior knowledge of the operator matrix, we are able to generate matrix multiplication kernel code, which performs better than state-of-the-art cuBLAS GEMM.
- We present GiMMiK - an open-source generator of matrix multiplication kernels for CUDA and OpenCL platforms, which utilises the optimisations discussed in this paper.
- We show how, through the use of bespoke matrix multiplication kernels, we are able to grant significant performance benefits to PyFR version 0.2.0.

Sections 2 to 4 discuss our approach to generating matrix multiplication kernels suitable for use in a *block-by-panel* type of matrix multiplication characteristic to PyFR. We have taken a systematic approach to evaluate each of the proposed optimisations in order to incorporate the successful ones into GiMMiK. Section 5 presents empirical results obtained by benchmarking our kernels against cuBLAS GEMM on the CUDA platforms. Lastly, our kernels are incorporated into PyFR to investigate the overall performance improvement achieved for an example real-world application through the use of GiMMiK as the matrix multiplication kernel provider.

2. Methodology

Due to a significant parametrization of the operator matrices it is highly impractical to generate specific implementations for each instance by hand, hence we require the matrix multiplication kernels to be auto-generated. We have developed GiMMiK - an open-source Python library, capable of generating matrix multiplication kernel code for CUDA and OpenCL. Basic loop unrolling serves as a basis for all of the applied optimisations. By design, each of our kernels computes an entire matrix-vector product. As part of the preliminary analysis we have considered a series of software optimisations, which we speculated would bring performance improvements to our kernels and hence enable us to outperform the state-of-the-art cuBLAS library. Subsequently, the successful optimisations were put together to generate final highly performant kernels and incorporated into the PyFR solver. The hypothesised effectiveness of the optimisations proposed in Section 3 was evaluated experimentally on the latest architecture NVIDIA’s GPUs: a consumer-grade Geforce GTX 780 Ti and an industry-grade Tesla K40c. Both cards are based around the Kepler architecture. The GTX 780 Ti has a higher core clock and increased memory bandwidth over the Tesla K40c. However, being a consumer-grade product it lacks support for error correcting code (ECC) memory and has double precision performance restricted to an eighth that of the Tesla K40c. Hence, by comparing the performance of GiMMiK across these two cards we are able to get a broad indication of which kernels are compute bound and which are bandwidth bound.

The memory layout, the operator matrices and the problem size were set up to mirror those in PyFR. Full specification of the experimental setup is available in Appendix A. Experiments were performed in both single and double precision.

We note that the desired product is of the form $C = \alpha A \times B + \beta C$ and values of α and β need to be handled with care. Generating kernels with embedded values of the operator matrix allows us to pre-multiply them with α to reduce the required number of floating-point operations. A special case occurs when $\beta = 0$, where we do not need to load entries of the output matrix at all. For this reason, each of the experiments mentioned above is run twice to investigate the effects of our optimisations in both cases when $\beta = 0$ and $\beta \neq 0$.

3. Strategy Assessment

3.1. Value Embedding

We speculate that embedding all entries of the operator matrix directly into the kernel code (as opposed to loading them from memory) will be beneficial to our fully unrolled kernels. Embedding values in the code is realized through storing them in the constant memory. This is advantageous as the compiler has explicit knowledge about the memory usage pattern. This optimisation also exposes the opportunity for the compiler to store the embedded values in registers for efficient reuse.

To verify our assumption we generate two types of kernels. The first type reads the operator matrix from global memory. On the Kepler architecture global memory loads are not cached in the L1 cache making the *read-only data cache* the only suitable way of caching data. This cache is known as the *texture cache* on older architectures and has to be explicitly managed by the programmer. The CUDA compiler (nvcc) will convert all loads into loads cached in the read-only data cache whenever it can. Hence, to better control experimental variables, we use it explicitly. The second kernel has values embedded in the code and relies on the *constant memory* and the *constant cache* to bring them efficiently into registers.

Experimental results show that embedding matrix values directly in the kernel brings significant performance improvements for a large fraction of the tested matrices; it has no effect for the others. This optimisation gives the compiler knowledge about the access pattern to the entries in the operator matrix. Hence, the compiler is able to find the most efficient memory storage layout to maximise data reuse from the *constant cache* and eliminate any possible memory bank conflicts, which might not be possible when relying on global memory.

3.2. Common Sub-expression Elimination

Values along the rows of the operator matrix occasionally repeat. This exposes the opportunity for us to reduce common sub-expressions and save on the number of floating-point operations required to compute the matrix product. This can be achieved by summing the elements of the operand matrix corresponding to the repeated values prior to multiplication by the common term.

To test the effects of common sub-expression elimination we compare two types of kernels, which embed all values from the operator matrix directly in the code and eliminate all multiplications by zeros. Only the second kernel-type removes common sub-expressions to reduce the number of floating-point multiplications.

Experimental data revealed that common sub-expression elimination did not bring the expected effect of improving performance of our kernels. While the number of floating-point operations has decreased, the number of generated terms increased. Values used to compute more than one term had to be loaded from memory multiple times, which increased memory traffic and decreased the performance of the kernels. If the compiler was able to reorder instructions more effectively or we explicitly used temporaries to reduce the number of global memory loads, we could have potentially overcome this issue.

3.3. Sparsity Elimination

To further reduce the number of floating-point operations we eliminate all multiplications by zeros. In PyFR, quadrilateral and hexahedral element meshes correspond to matrices with large sparsity factors. Further, some of these matrices contain whole columns of zeros, which effectively means that the dimensionality of the matrices decreases and entire rows of the operand matrix do not need to be loaded from memory at all.

To verify the effectiveness of sparsity elimination we compare two types of kernels, which embed all values from the operator matrix directly in the code. All multiplications by zeros were eliminated from the second kernel-type only.

The experiments show that sparsity elimination brings large performance improvements to matrices for quadrilateral and hexahedral meshes; fully dense cases of triangular and tetrahedral matrices are unaffected. Profiling has revealed that through removal of unnecessary floating-point operations our kernels become fully bound by the available memory-bandwidth in both cases of single and double precision.

This represents a considerable improvement over traditional sparse-by-dense matrix multiplication methods whereby the sparse matrix is represented in compressed sparse row (CSR) format. While such representations are effective at eliminating multiplications by zero they do so at the cost of additional memory bandwidth. Specifically, in the CSR format it is also necessary to store row offsets and column indices in addition to the nonzero entries. Retrieving these extra values consumes memory bandwidth. By combining sparsity elimination with value embedding we are able to firstly, avoid the need for such data values — saving bandwidth; secondly, make it possible to avoid reloading entire columns of B while working through the rows of A .

A preliminary investigation determined that for the sparse matrices we are interested in, cuSPARSE [17] was unable to outperform (even in the largest/sparsest cases) dense cuBLAS. We attribute this to the overheads associated with iterating over sparse matrices stored in the CSR format.

3.4. Cleanup Code

GEMM implementations in BLAS libraries often utilise some form of tiling to efficiently reuse data from the cache, registers or - in the case of CUDA - shared memory. Poor tiling choices result in the need to execute cleanup code over the elements of the matrix that fall outside of the tile boundaries. This cleanup code is known to be poorly optimised [1]. For the case of small operator matrices such as those used in PyFR, the performance penalty due to poor tiling choices can be particularly significant. Our fully unrolled kernels do not incur any of these costs.

To examine the effect of cleanup code in cuBLAS on performance, we benchmarked fully unrolled kernels with sparsity eliminated and values of the operator matrix embedded in the code, against the cuBLAS GEMM and against a naïve 3-loop implementation of the matrix product. The naïve implementation serves as a reference to determine the extent to which performance of cuBLAS is dominated by the cleanup code.

The ability of our kernels to avoid the poorly optimised cleanup code brought large performance benefits to the particularly small matrices (low polynomial orders). We develop more certainty in this claim by noticing that for very small operator matrices the performance of cuBLAS is comparable with the naïve 3-loop implementation.

4. GiMMiK

Having performed the experiments described in Section 3 we are in a position to generate highly performant kernels. The successful optimisations have been incorporated into GiMMiK. We conclude that to achieve the best performance of our kernels we will eliminate sparsity from the operator matrix to reduce the number of floating-point operations. Through the preliminary analysis we found that this optimisation was highly beneficial for the matrices corresponding to hexahedral and quadrilateral meshes as they have high sparsity factors. Further, GiMMiK will embed all non-zero values directly in the kernel code to benefit from the fast constant cache and compiler optimisations. This optimisation is expected to increase the performance of kernels for all element types. Lastly, we avoid the need to execute any cleanup code through loop unrolling, which is shown to be beneficial especially for small matrices (low polynomial orders) across all element types. GiMMiK’s kernels will not reduce common sub-expressions, as this optimisation was found to have a negative effect on their performance. This is because of the increased number of memory loads necessary to compute the subterms and an increased register pressure, due to a

$$A = \begin{bmatrix} 0.0 & 0.0 & 0.59097691 \\ 0.63448574 & 0.0 & 0.0 \\ 0.0 & 0.71191878 & 0.95941663 \end{bmatrix}$$

(a) Arbitrary 3×3 operator matrix input to GiMMiK.

```

1  __global__ void
2  gimmik_mm(const double* __restrict__ b,
3            double* __restrict__ c,
4            const int width,
5            const int bstride,
6            const int cstride)
7  {
8      int index = blockDim.x * blockIdx.x + threadIdx.x;
9      if (index < width)
10     {
11         const double *b_local = b + index;
12         double *c_local = c + index;
13
14         const double subterm_0 = b_local[2 * bstride];
15         const double subterm_1 = b_local[0 * bstride];
16         const double subterm_2 = b_local[1 * bstride];
17
18         c_local[0 * cstride] = 0.5909769053580467 * subterm_0;
19         c_local[1 * cstride] = 0.6344857400767476 * subterm_1;
20         c_local[2 * cstride] = 0.9594166286064713 * subterm_0
21                               + 0.7119187815275971 * subterm_2;
22     }
23 }

```

(b) Kernel code generated by GiMMiK.

Figure 2: Sample kernel code generated by GiMMiK for the CUDA platform for an arbitrary operator matrix and $\beta = 0$.

larger number of temporary variables needed in the code. Figure 2 shows sample code generated by GiMMiK for the CUDA platform for an arbitrary 3×3 operator matrix and $\beta = 0$.

5. Performance Analysis

Having decided on the beneficial optimisations and incorporated them into GiMMiK, we have used the same experimental setup to evaluate the final performance of our kernels. Full details on the setup are available in Appendix A.

We find that our bespoke CUDA kernels are able to outperform cuBLAS GEMM in nearly all cases for quadrilateral, hexahedral and triangular element matrices in double and single precision on both the GTX 780 Ti and Tesla K40c with $\beta = 0$ as well as $\beta \neq 0$. These matrices are either small and dense or large and sparse. In the case of large tetrahedral matrices the library implementation proves superior. This is expected as these matrices are dense and sufficiently large for cuBLAS to perform well. In the case of single precision we find a small number of large hexahedral matrices, where cuBLAS is vaguely more performant than our kernels as well, as the

library GEMM implementation can fully utilise the inherently higher single precision floating-point performance of the devices without it being a limiting factor.

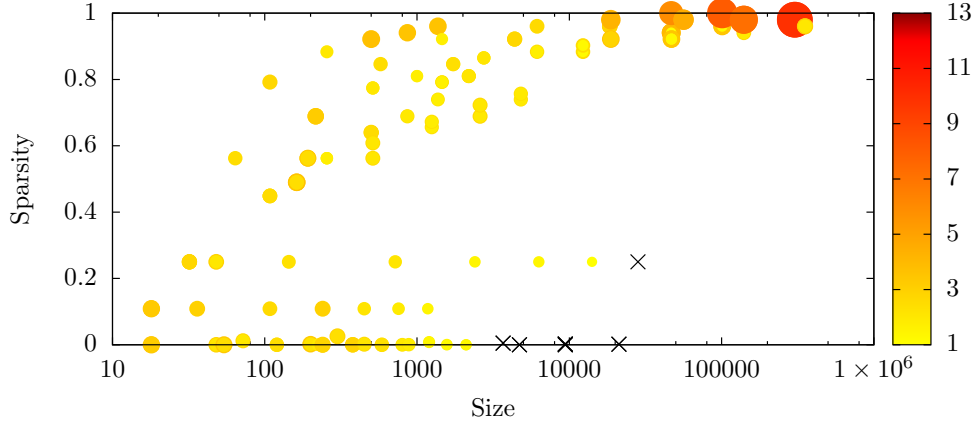
The top plots in Figures 3 to 6 illustrate the achieved speedups across a variety of matrix sizes (number of elements) and sparsity patterns corresponding to the benchmark matrices from PyFR for 1–6 polynomial orders. We conclude that very small matrices and large, sparse matrices benefit the most from optimisations employed by GiMMiK. This is expected since GiMMiK reduces the number of floating-point operations required to compute the product as well as avoids the poorly optimised cleanup code, which dominates performance of cuBLAS for small matrices. We observe cases of matrices with the exact same size (total number of entries) and sparsity factor (fraction of zero-entries), that achieve different speedups. These matrices differ in their width and height, affecting cuBLAS performance due to tiling choices, but has little effect on our kernels. In the case of double precision on the GTX 780 Ti we note particularly impressive speedups due to the restricted double precision floating-point performance on the consumer-grade card. Through the use of GiMMiK we are able to significantly reduce the number of floating-point operations required and hence better utilise the available resources.

Performance of BLAS libraries is typically bound by the floating-point capabilities of the hardware [5], because GEMM routines do not exploit any sparsity or redundancy in the data. Profiling of our kernels has revealed that through the applied optimisations all kernels for sparse matrices (hexahedral and quadrilateral meshes) become bound by the available memory bandwidth. Small dense kernels do not require enough floating-point operations to utilise a large percentage of the available floating-point rate and hence are also bound by the available memory bandwidth. Large dense matrices are bound by the floating-point performance of the device. These results hold in both single and double precision. The bottom two plots in Figures 3 to 6 illustrate the achieved percentage of the peak floating-point rate and memory bandwidth by GiMMiK’s kernels for various size and sparsity patterns corresponding to the benchmarked PyFR matrices. Both floating-point rate and memory bandwidth values are those reported by the profiler (as opposed to being derived algebraically based on operand sizes and speedup factors). Accesses to local memory, which would occur as a consequence of register spilling in GiMMiK-generated kernels, are excluded from the memory bandwidth values.

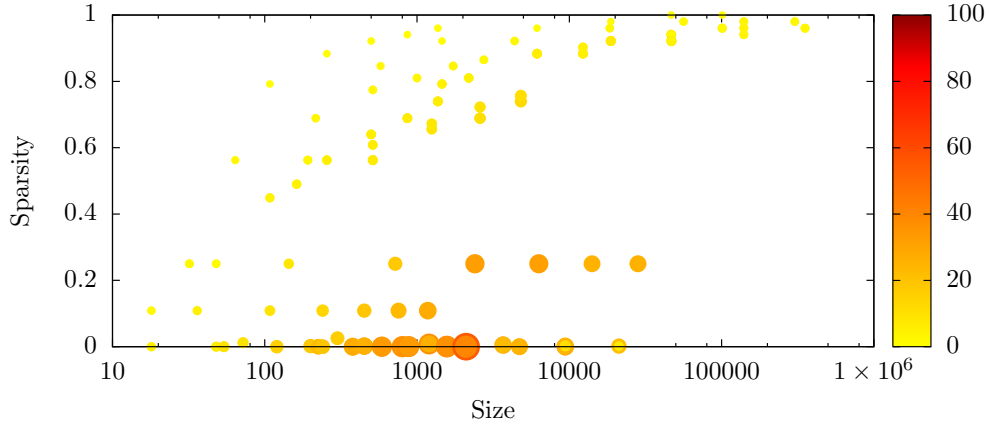
The achieved floating-point rate is defined in terms of the number of floating-point operations needed to perform the matrix product and the time taken to execute the kernels. We use the NVIDIA profiler (nvprof) to count the number of floating-point operations executed by our kernels. This is more accurate than computing this number algebraically, due to the reduction of the number of operations due to compiler optimisations and elimination of sparsity.

The achieved memory bandwidth is computed in terms of the time taken to execute the matrix product and the total amount of memory needed to be read and written by the kernel. Similarly to the floating-point rate, the achieved memory bandwidth was obtained through profiling. We note that the metric reported by the compiler is inclusive of any traffic to and from local memory.

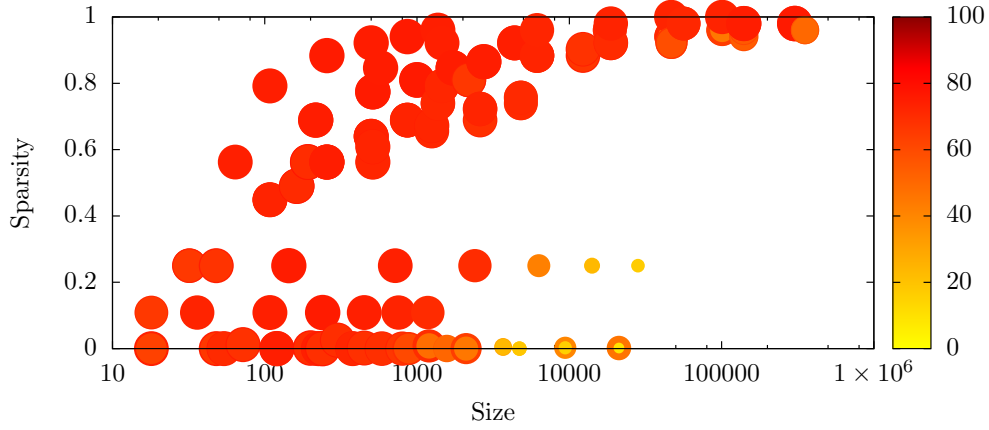
By analysing the aforementioned plots we notice that the utilisation of the available memory bandwidth dramatically drops for large dense matrices (also for very large sparse matrices) and results in performance degradation of GiMMiK’s kernels up to the point where they perform worse than cuBLAS GEMM. Further, not unexpectedly, dense matrices achieve higher utilisation of the available floating-point rate than sparse matrices. We observe a significantly higher utilisation of the double precision floating-point rate on the GTX 780 Ti due to the restricted double precision performance on consumer-grade hardware.



(a) Speedup

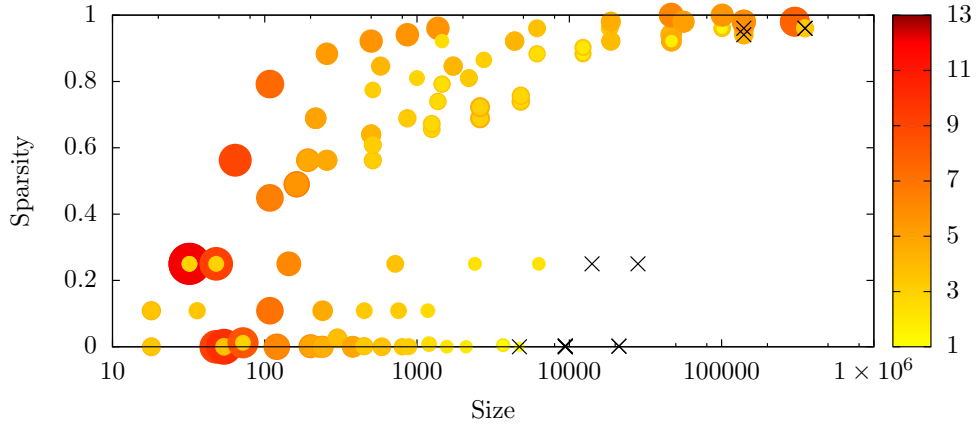


(b) % FLOPS

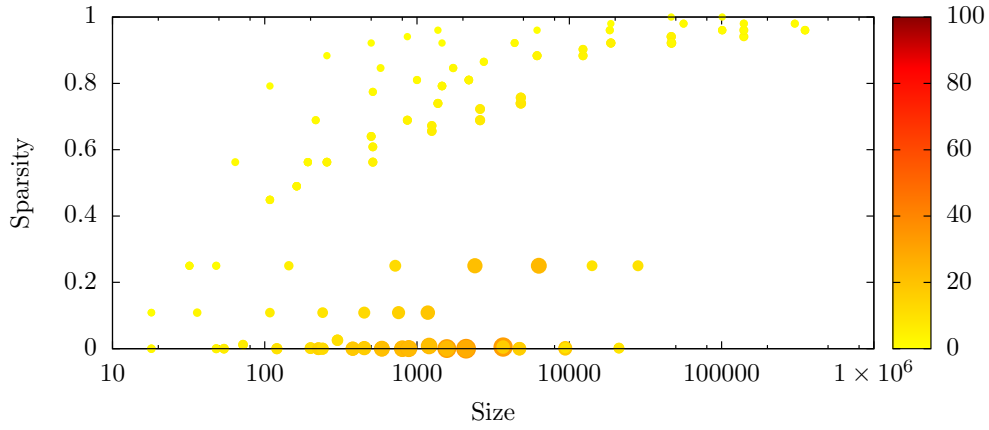


(c) % Memory Bandwidth

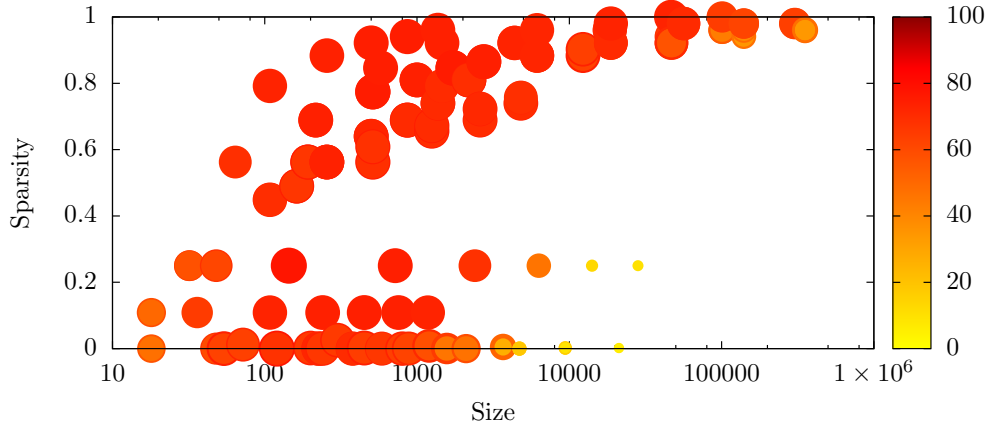
Figure 3: Plots illustrating the speedup of GiMMiK’s kernels over cuBLAS (a), the achieved percentage of the peak floating-point rate (b) and the achieved percentage of the peak memory bandwidth (c). Size refers to the number of elements in the operator matrix and sparsity is the ratio of the number of zero elements to the size. The metric of interest is represented through the size and colour of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for double precision on the Tesla K40c. 9



(a) Speedup

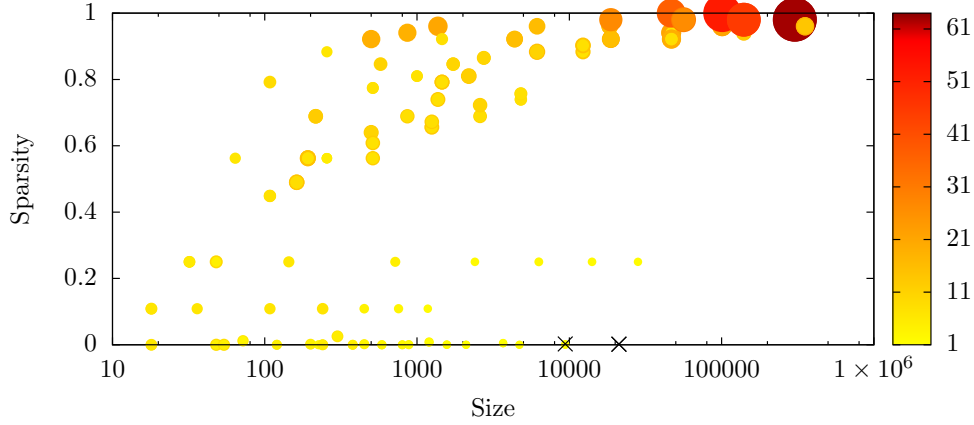


(b) % FLOPS

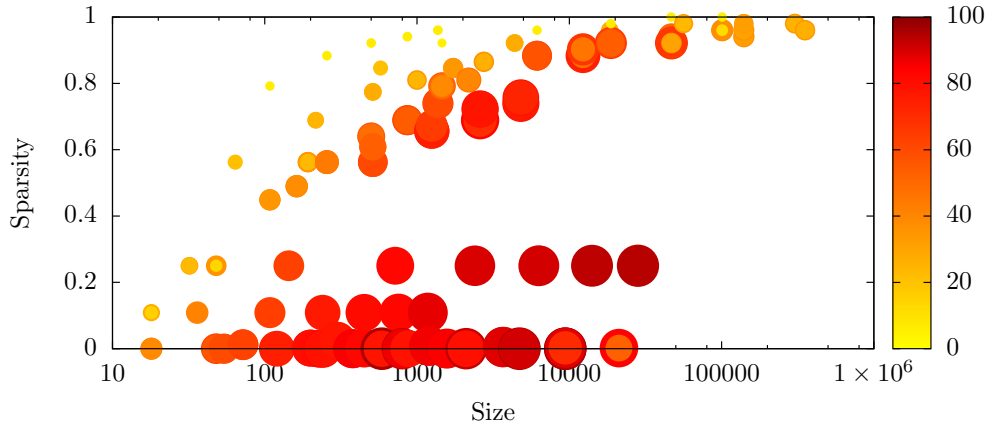


(c) % Memory Bandwidth

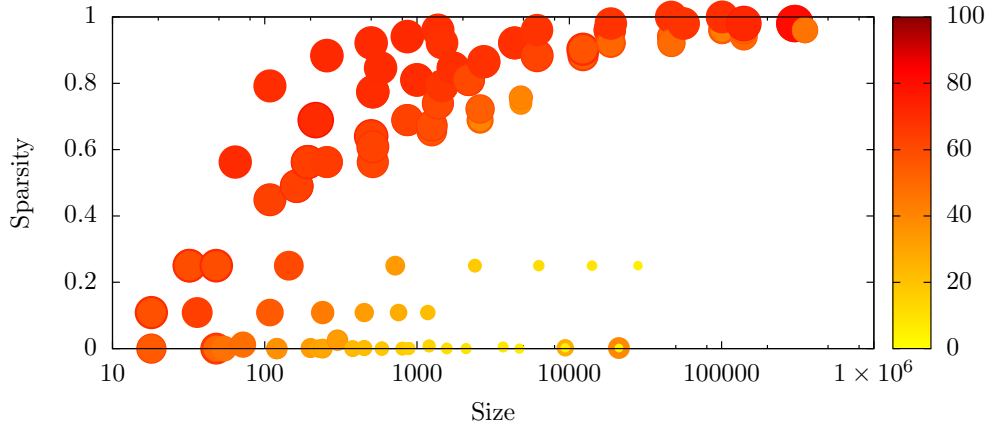
Figure 4: Plots illustrating the speedup of GiMMiK’s kernels over cuBLAS (a), the achieved percentage of the peak floating-point rate (b) and the achieved percentage of the peak memory bandwidth (c). Size refers to the number of elements in the operator matrix and sparsity is the ratio of the number of zero elements to the size. The metric of interest is represented through the size and colour of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for single precision on the Tesla K40c.



(a) Speedup

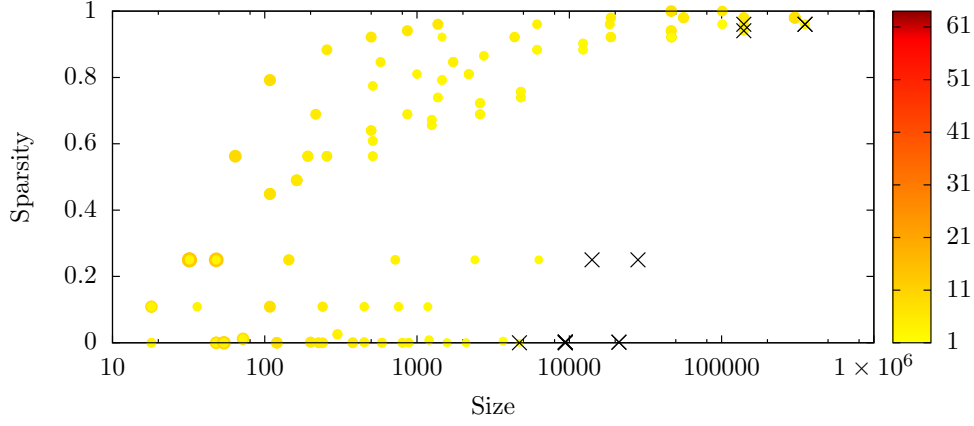


(b) % FLOPS

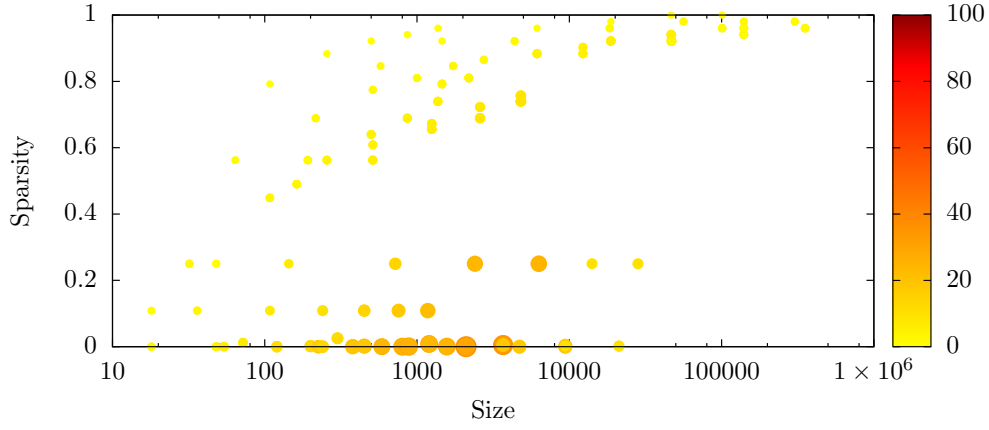


(c) % Memory Bandwidth

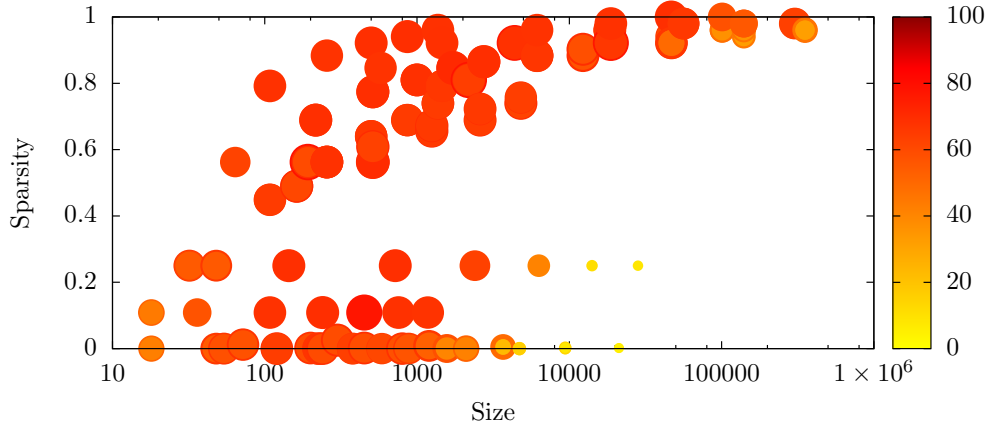
Figure 5: Plots illustrating the speedup of GiMMiK’s kernels over cuBLAS (a), the achieved percentage of the peak floating-point rate (b) and the achieved percentage of the peak memory bandwidth (c). Size refers to the number of elements in the operator matrix and sparsity is the ratio of the number of zero elements to the size. The metric of interest is represented through the size and colour of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for double precision on the GTX 780 Ti. 11



(a) Speedup



(b) % FLOPS



(c) % Memory Bandwidth

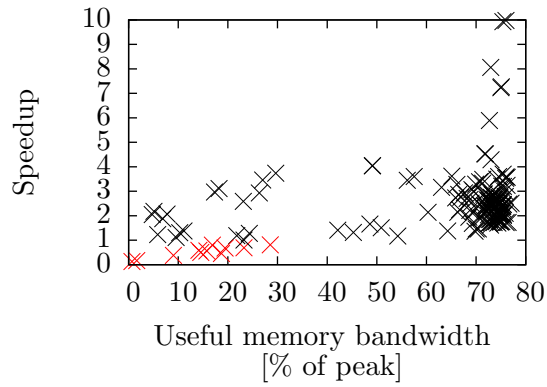
Figure 6: Plots illustrating the speedup of GiMMiK’s kernels over cuBLAS (a), the achieved percentage of the peak floating-point rate (b) and the achieved percentage of the peak memory bandwidth (c). Size refers to the number of elements in the operator matrix and sparsity is the ratio of the number of zero elements to the size. The metric of interest is represented through the size and colour of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for single precision on the GTX 780 Ti. 12

Figures 3 to 6 identify a set of GiMMiK’s kernels which do not benefit from our proposed optimisations and fail to achieve any speedup over cuBLAS GEMM. These kernels correspond to large dense matrices in cases of both single and double precision. Additionally, in single precision we note decreased performance of our kernels for large sparse matrices. Through profiling we observe that kernels which fail to achieve a speedup attempt to utilise a very large (often the maximum available) number of registers, which decreases the kernels’ occupancy on the multiprocessors and increases register pressure due to a large number of temporary variables used. Increased register pressure can lead to a large amount of register spillage into memory. In the case of GiMMiK’s kernels for large matrices this spillage is so severe that the spilled data cannot be retained in the L1 or even L2 caches. As a consequence of this, it needs to be written and read from local memory. This increases memory traffic, decreases occupancy and as a consequence decreases performance of our kernels. It also explains why the achieved fraction of the peak memory bandwidth and floating-point rate of our kernels for large matrices drops. We believe that to overcome this limitation we can tile the matrix product in such a way to bring the register pressure down and hence achieve higher occupancy and reduce spillage of registers into local memory. Figure 7 further illustrates the effects of achieved memory bandwidth and amount of data spillage into local memory on the speedups of GiMMiK over cuBLAS. It shows that the speedups over cuBLAS decreases as the achieved useful memory bandwidth decreases (device memory bandwidth exclusive of traffic due to local memory). In the case of double precision on the GTX 780 Ti we see that the achieved useful memory bandwidth is often lower than in the remaining cases due to the cap placed on the floating-point rate, which becomes a significant factor affecting the performance of our kernels. Cases which failed to achieve a speedup are marked in red. From the plots we see that the smallest speedups are obtained by kernels which utilise a smaller percentage of the available memory bandwidth. All raw data is provided as electronic supplementary material.

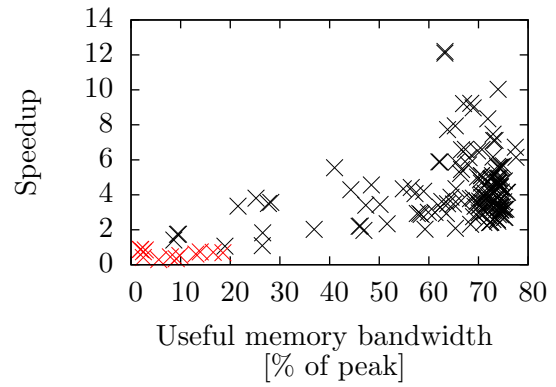
To investigate the effective performance improvement that our bespoke kernels bring to PyFR we have undertaken an example real-world simulation of unsteady flow over a cylinder at Reynolds number 3900. For further details regarding a similar simulation, please refer to Witherden et al. [7]. The simulation was executed on a mesh of 42,030 hexahedral elements. The average time (in ms) per time-step of the simulation is reported in Table 1. We observe an overall speedup of 1.40 for this simulation executed with third polynomial-order solution in double precision on Tesla K40c. Achieved speedups of PyFR for this simulation for other polynomial orders and across our two devices are summarised in Table 2. Isosurfaces of density captured during the PyFR simulation executed using GiMMiK are illustrated in Figure 8.

Order	GiMMiK				cuBLAS			
	GTX 780 Ti		Tesla K40c		GTX 780 Ti		Tesla K40c	
	single	double	single	double	single	double	single	double
1	16.65	29.72	23.50	39.66	41.01	110.76	51.56	61.02
2	43.56	80.60	61.63	107.60	82.33	361.27	104.88	146.97
3	106.34	182.53	145.50	239.52	174.79	927.48	221.74	334.50
4	215.02	379.22	289.63	480.94	393.51	2876.47	490.80	818.23

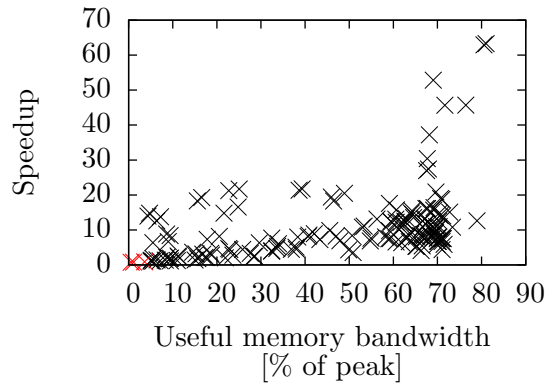
Table 1: Average time in milliseconds of a time-step in a simulation of unsteady flow over a cylinder across 1–4 polynomial orders in single and double precision with PyFR v0.2.0 using GiMMiK v1.0 and PyFR v0.2.0 using cuBLAS. Averages were calculated using 10^7 time-steps.



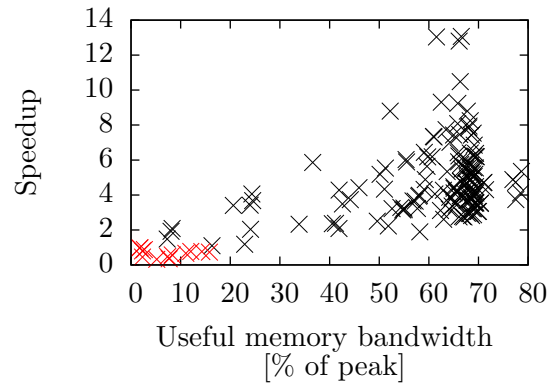
(a) Double precision, Tesla K40c



(b) Single precision, Tesla K40c



(c) Double precision, GTX 780 Ti



(d) Single precision, GTX 780 Ti

Figure 7: Plots of speedup against useful memory bandwidth (exclusive of local memory traffic) expressed as a percentage of the peak memory bandwidth for the set of benchmark matrices. Speedups less than 1 are marked red.

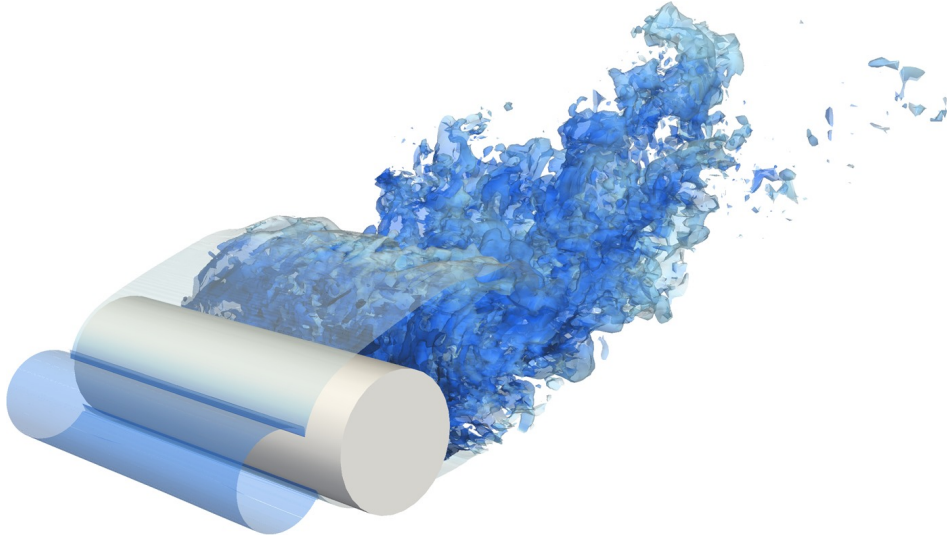


Figure 8: Isosurfaces of density obtained during a simulation of unsteady flow over a cylinder executed with PyFR using GiMMiK as the matrix multiplication kernel provider.

Order	GTX 780 Ti		Tesla K40c	
	single	double	single	double
1	2.46	3.73	2.19	1.54
2	1.89	4.48	1.70	1.37
3	1.64	5.08	1.52	1.40
4	1.83	7.59	1.69	1.70

Table 2: Speedups achieved when running PyFR to simulate unsteady flow over a cylinder across 1–4 polynomial orders in single and double precision with PyFR v0.2.0 using GiMMiK v1.0 for matrix multiplication steps over the same version relying on cuBLAS GEMM.

6. Conclusion

In this paper we have demonstrated that through generation of bespoke matrix multiplication kernels with a prior knowledge of the operator matrix we are able to outperform the highly optimised cuBLAS library on CUDA platforms on NVIDIA GPUs. We have presented GiMMiK - a Python library for generating highly performant matrix multiplication kernels. The generated kernels in our solution consist of a fully unrolled matrix inner product with a reduced number of floating-point operations through the removal of sparsity. Further, our kernels embed the operator matrix directly in the code to benefit from the constant cache and compiler optimisations.

We find that this technique works particularly well for *block-by-panel* type of matrix multiplications. We have benchmarked GiMMiK against a set of matrices extracted from PyFR and observed that our bespoke CUDA kernels are able to outperform cuBLAS in nearly all cases for sparse matrices and small dense matrices, corresponding to quadrilateral, hexahedral and triangular element meshes, in double and single precision for 1–6 polynomial orders. In the case of large tetrahedral matrices the library implementation proves superior, as these matrices are insufficiently small and too dense to benefit from our optimisations.

We note speedups between 1.35 and 1.72 for an example fluid flow PyFR simulation executed in double precision on a hexahedral mesh for 1–6 polynomial orders on a single Tesla K40c. These results can influence the numerical method choices made for various types of fluid simulations, while the performance increase, to some extent, can allow for use of higher order meshes and finer time steps, improving the physical properties and increasing the quality of the results of the simulation. Given the ubiquity of GEMM we envisage that a variety of other applications, possibly outside of the area of CFD, could also benefit from the use of GiMMiK as the kernel provider for matrix multiplication operations performed on the CUDA and OpenCL platforms.

7. Acknowledgements

The authors would like to thank the Engineering and Physical Sciences Research Council for their support through grants EP/K027379/1, EP/L000407/1, EP/I00677X/1 and EP/I006613/1.

- [1] R. C. Whaley, A. Petitet, J. J. Dongarra, Automated empirical optimizations of software and the ATLAS project, *Parallel Computing* 27 (1–2) (2001) 3–35. doi:10.1016/S0167-8191(00)00087-9.
- [2] K. Goto, R. van de Geijn, Anatomy of high-performance matrix multiplication, *ACM Trans. Math. Softw.* 34 (3) (2008) 12:1–12:25. doi:10.1145/1356052.1356053.
- [3] C. Jhurani, P. Mullenney, A GEMM interface and implementation on NVIDIA GPUs for multiple small matrices, *Journal of Parallel and Distributed Computing* 75 (2015) 133–140. doi:10.1016/j.jpdc.2014.09.003.
- [4] M. Georgiou, Developing Optimized Matrix Multiplication Kernels for CPU and GPU Platforms: Application to High Order CFD, Master’s thesis, Imperial College of Science, Technology and Medicine (September 2013).
- [5] R. Nath, S. Tomov, J. Dongarra, An improved MAGMA GEMM for Fermi graphics processing units, *International Journal of High Performance Computing Applications* 24 (4) (2010) 511–515. doi:10.1177/1094342010385729.
- [6] H. Huynh, A Flux Reconstruction Approach to High-Order Schemes Including Discontinuous Galerkin Methods, AIAA paper 4079. doi:10.2514/6.2007-4079.
- [7] F. D. Witherden, A. M. Farrington, P. E. Vincent, PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach, *Computer Physics Communications* 185 (11) (2014) 3028–3040. doi:10.1016/j.cpc.2014.07.011.
- [8] NVIDIA, cuBLAS Library User Guide, <http://docs.nvidia.com/cuda/cublas/index.html>, [Online; accessed 31-August-2015] (Mar. 2015).
- [9] clBLAS: OpenCL BLAS, <http://clmathlibraries.github.io/clBLAS/>, [Online; accessed 31-August-2015] (Aug. 2013).
- [10] P. Castonguay, D. Williams, P. Vincent, M. López, A. Jameson, On the Development of a High-Order, Multi-GPU Enabled, Compressible Viscous Flow Solver for Mixed Unstructured Grids, AIAA paper 3229. doi:10.2514/6.2011-3229.
- [11] J. W. Choi, A. Singh, R. W. Vuduc, Model-driven autotuning of sparse matrix-vector multiply on GPUs, in: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’10*, ACM, New York, NY, USA, 2010, pp. 115–126. doi:10.1145/1693453.1693471.
- [12] A. Monakov, A. Lokhmotov, A. Avetisyan, Automatically tuning sparse matrix-vector multiplication for GPU architectures, in: *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers, HiPEAC’10*, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 111–125. doi:10.1007/978-3-642-11515-8_10.
- [13] I. Regulý, M. Giles, Efficient sparse matrix-vector multiplication on cache-based GPUs, in: *Innovative Parallel Computing (InPar)*, 2012, 2012, pp. 1–12. doi:10.1109/InPar.2012.6339602.
- [14] J. L. Greathouse, M. Daga, Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format, in: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC ’14*, IEEE Press, Piscataway, NJ, USA, 2014, pp. 769–780. doi:10.1109/SC.2014.68.
- [15] F. Vázquez, G. Ortega, J. J. Fernández, I. García, E. M. Garzón, Fast sparse matrix product based on ELLR-T and GPU computing, in: *Parallel and Distributed Processing with Applications (ISPA)*, 2012 IEEE 10th International Symposium on, 2012, pp. 669–674. doi:10.1109/ISPA.2012.99.
- [16] H. M. Aktulga, A. Buluç, S. Williams, C. Yang, Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations, in: *Proceedings of the 2014 IEEE 28th International Parallel*

and Distributed Processing Symposium, IPDPS '14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 1213–1222. doi:10.1109/IPDPS.2014.125.

- [17] NVIDIA, cuSPARSE Library User Guide, <http://docs.nvidia.com/cuda/cusparse/index.html>, [Online; accessed 31-August-2015] (Mar. 2015).

Appendix A. Experimental heights, widths and sparsity factors

All experiments are performed on the following two pieces of NVIDIA hardware. Operator matrices correspond directly to those used in PyFR. The operand and output matrices were selected to be 50,000 elements wide. 30 runs were averaged to produce the final timing result. 30 runs were executed before each timing run to eliminate bias due to idle under-clocking of the CUDA GPUs. All matrices used in benchmarking are stored in a row-major order and are padded to ensure coalesced and aligned memory accesses by all threads. The results are verified against a CPU-based matrix multiplication implementation to ensure correctness. PyFR version 0.2.0 and GiMMiK version 1.0 were used to perform all of the experiments.

Tesla K40c.

Compute Capability 3.5

ECC ON

Peak Memory Bandwidth 288 GB/s

Peak Double-precision 1.43 TFLOPs

Peak Single-precision 4.29 TFLOPs

CUDA Version 6.0^a

Geforce GTX 780 Ti.

Compute Capability 3.5

ECC N/A

Peak Memory Bandwidth 336 GB/s

Peak Double-precision 210 GFLOPs

Peak Single-precision 5.04 TFLOPs

CUDA Version 6.0

^aWe found impressive speedups of up to 80% of cuBLAS GEMM after updating from CUDA 5.5 to 6.0.

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	8	4	0.50
$M132$	4	8	0.50
$M3$	4	8	0.50
$M460$	8	4	0.50
$M6$	8	8	0.75

(a) Order = 1

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	12	9	0.67
$M132$	9	18	0.70
$M3$	9	12	0.67
$M460$	18	9	0.70
$M6$	18	12	0.83

(a) Order = 2

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	16	16	0.75
$M132$	16	32	0.75
$M3$	16	16	0.75
$M460$	32	16	0.75
$M6$	32	16	0.88

(a) Order = 3

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	20	25	0.80
$M132$	25	50	0.81
$M3$	25	20	0.80
$M460$	50	25	0.81
$M6$	50	20	0.90

(a) Order = 4

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	24	36	0.83
$M132$	36	72	0.83
$M3$	36	24	0.83
$M460$	72	36	0.83
$M6$	72	24	0.92

(a) Order = 5

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	28	49	0.86
$M132$	49	98	0.86
$M3$	49	28	0.86
$M460$	98	49	0.86
$M6$	98	28	0.93

(a) Order = 6

Table A.9: Heights (m), widths (k) and sparsity factors of quadrilateral matrices obtained using the Gauss-Legendre method.

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	12	9	0.89
$M132$	9	18	0.70
$M3$	9	12	0.67
$M460$	18	9	0.70
$M6$	18	12	0.83

(a) Order = 2

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	16	16	0.94
$M132$	16	32	0.78
$M3$	16	16	0.75
$M460$	32	16	0.78
$M6$	32	16	0.88

(a) Order = 3

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	20	25	0.96
$M132$	25	50	0.82
$M3$	25	20	0.80
$M460$	50	25	0.82
$M6$	50	20	0.90

(a) Order = 4

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	24	36	0.97
$M132$	36	72	0.85
$M3$	36	24	0.83
$M460$	72	36	0.85
$M6$	72	24	0.92

(a) Order = 5

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	28	49	0.98
$M132$	49	98	0.87
$M3$	49	28	0.86
$M460$	98	49	0.87
$M6$	98	28	0.93

(a) Order = 6

Table A.15: Heights (m), widths (k) and sparsity factors of quadrilateral matrices obtained using the Gauss-Legendre-Lobatto method.

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	24	8	0.75
$M132$	8	24	0.75
$M3$	8	24	0.75
$M460$	24	8	0.75
$M6$	24	24	0.92

(a) Order = 1

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	54	27	0.89
$M132$	27	81	0.90
$M3$	27	54	0.89
$M460$	81	27	0.90
$M6$	81	54	0.96

(a) Order = 2

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	96	64	0.94
$M132$	64	192	0.94
$M3$	64	96	0.94
$M460$	192	64	0.94
$M6$	192	96	0.98

(a) Order = 3

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	150	125	0.96
$M132$	125	375	0.96
$M3$	125	150	0.96
$M460$	375	125	0.96
$M6$	375	150	0.99

(a) Order = 4

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	216	216	0.97
$M132$	216	648	0.97
$M3$	216	216	0.97
$M460$	648	216	0.97
$M6$	648	216	0.99

(a) Order = 5

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	294	343	0.98
$M132$	343	1029	0.98
$M3$	343	294	0.98
$M460$	1029	343	0.98
$M6$	1029	294	0.99

(a) Order = 6

Table A.22: Heights (m), widths (k) and sparsity factors of hexahedral matrices obtained using the Gauss-Legendre method.

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	54	27	0.96
$M132$	27	81	0.90
$M3$	27	54	0.89
$M460$	81	27	0.90
$M6$	81	54	0.96

(a) Order = 2

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	96	64	0.98
$M132$	64	192	0.95
$M3$	64	96	0.94
$M460$	192	64	0.95
$M6$	192	96	0.98

(a) Order = 3

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	150	125	0.99
$M132$	125	375	0.96
$M3$	125	150	0.96
$M460$	375	125	0.96
$M6$	375	150	0.99

(a) Order = 4

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	216	216	0.99
$M132$	216	648	0.98
$M3$	216	216	0.97
$M460$	648	216	0.98
$M6$	648	216	0.99

(a) Order = 5

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	294	343	0.99
$M132$	343	1029	0.98
$M3$	343	294	0.98
$M460$	1029	343	0.98
$M6$	1029	294	0.99

(a) Order = 6

Table A.28: Heights (m), widths (k) and sparsity factors of hexahedral matrices obtained using the Gauss-Legendre-Lobatto method.

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	6	3	0.00
$M132$	3	6	0.33
$M3$	3	6	0.00
$M460$	6	3	0.33
$M6$	6	6	0.33

(a) Order = 1

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	9	6	0.00
$M132$	6	12	0.11
$M3$	6	9	0.00
$M460$	12	6	0.11
$M6$	12	9	0.33

(a) Order = 2

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	12	10	0.00
$M132$	10	20	0.04
$M3$	10	12	0.00
$M460$	20	10	0.04
$M6$	20	12	0.33

(a) Order = 3

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	15	15	0.00
$M132$	15	30	0.04
$M3$	15	15	0.00
$M460$	30	15	0.04
$M6$	30	15	0.33

(a) Order = 4

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	18	21	0.00
$M132$	21	42	0.02
$M3$	21	18	0.00
$M460$	42	21	0.02
$M6$	42	18	0.33

(a) Order = 5

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	21	28	0.00
$M132$	28	56	0.02
$M3$	28	21	0.00
$M460$	56	28	0.02
$M6$	56	21	0.33

(a) Order = 6

Table A.35: Heights (m), widths (k) and sparsity factors of triangular matrices obtained using the Williams-Shunn method.

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	12	4	0.00
$M132$	4	12	0.50
$M3$	4	12	0.00
$M460$	12	4	0.50
$M6$	12	12	0.50

(a) Order = 1

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	24	10	0.00
$M132$	10	30	0.16
$M3$	10	24	0.00
$M460$	30	10	0.16
$M6$	30	24	0.50

(a) Order = 2

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	40	20	0.00
$M132$	20	60	0.09
$M3$	20	40	0.00
$M460$	60	20	0.09
$M6$	60	40	0.50

(a) Order = 3

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	60	35	0.00
$M132$	35	105	0.07
$M3$	35	60	0.00
$M460$	105	35	0.07
$M6$	105	60	0.50

(a) Order = 4

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	84	56	0.00
$M132$	56	168	0.05
$M3$	56	84	0.00
$M460$	168	56	0.05
$M6$	168	84	0.50

(a) Order = 5

Matrix	m	k	$\frac{\text{zeros}}{m \times k}$
$M0$	112	84	0.00
$M132$	84	252	0.04
$M3$	84	112	0.00
$M460$	252	84	0.04
$M6$	252	112	0.50

(a) Order = 6

Table A.42: Heights (m), widths (k) and sparsity factors of tetrahedral matrices obtained using the Shunn-Ham method.