A case for work-stealing on FPGAs with OpenCL atomics

Nadesh Ramanathan
Imperial College London, UK
n.ramanathan14@imperial.ac.uk

John Wickerson
Imperial College London, UK
j.wickerson@imperial.ac.uk

Felix Winterstein
Imperial College London, UK
f.winterstein12@imperial.ac.uk

George A. Constantinides
Imperial College London, UK
g.constantinides@imperial.ac.uk

ABSTRACT

We provide a case study of *work-stealing*, a popular method for run-time load balancing, on FPGAs. Following the Cederman–Tsigas implementation for GPUs, we synchronize work-items not with locks, mutexes or critical sections, but instead with the atomic operations provided by Altera's OpenCL SDK. We evaluate work-stealing for FPGAs by synthesizing a K-means clustering algorithm on an Altera P385 D5 board, both with work-stealing and with a statically-partitioned load. When block RAM utilization is maximized in both cases, we find that work-stealing leads to a 1.5× speedup. This demonstrates that the ability to do load balancing at run-time can outweigh the drawback of using 'expensive' atomics on FPGAs. We hope that our case study will stimulate further research into the high-level synthesis of fine-grained, lock-free, concurrent programs.

Keywords

atomic operations, high-level synthesis, K-means clustering, load balancing, lock-free synchronization, parallelism.

1. INTRODUCTION

The central task for high-level synthesis (HLS) tools is mapping the computation and data accesses described by the source program into the FPGA's execution and memory hierarchies. Some computations can be straightforwardly divided among parallel execution units on the FPGA, but it is often the case that partitioning statically (*i.e.*, at compile-time) either is infeasible or leads to inefficient circuits. In this paper, we explore a different approach that involves balancing dynamically (*i.e.*, at run-time) the computational workload across execution units.

Our approach is based on *work-stealing*, a popular paradigm for programming those algorithms that can be phrased in terms of many small tasks, and that have at least one of the following characteristics: 1) data-dependent task execution time, and 2) dynamic sub-task creation. Each parallel execution unit ('*work-item*') maintains its own task queue, but can steal from another's queue should its own become empty. We present an implementation of work-stealing that builds on an implementation for GPUs, by Cederman and Tsigas [5], of an algorithm due to Arora *et al.* [3]. It is written in OpenCL (a multi-threaded extension of C for programming heterogeneous systems of CPUs, GPUs, and FPGAs [13]) and automatically compiled to hardware using Altera's software development kit for OpenCL (AOCL) [2]. We describe how we have optimized the OpenCL code for performance and compatibility with the restrictions imposed by AOCL.

Our work-stealing implementation is particularly novel in an FPGA context because we avoid the use of locks and barriers, and rely instead on OpenCL's *atomic operations* (atomics) to synchronize threads. Atomics enable fine-grained concurrency whereby threads can execute without blocking other threads. Although atomics have recently been demonstrated empirically to be the fastest synchronization method for conventional multiprocessors [8], their support on FPGAs is lacking. AOCL supports them for 32-bit integers, but discourages their use, warning that they are 'expensive to implement on FPGAs' and 'might decrease kernel performance or require a large amount of hardware' [1], while Xilinx's OpenCL tool, SDAccel [21], does not support them at all. In this work, we demonstrate that despite these misgivings, atomics can in fact be usefully employed on FPGAs to give overall application speedup.

In our case study, we apply our work-stealing implementation to a K-means clustering (KMC) algorithm [11] with data-dependent task execution time and dynamic sub-task creation. On an Altera P385 D5 board, we compare the performance (a) when the workload is statically determined at compile-time, and (b) when work-stealing is enabled. In both cases, we synthesize sufficiently many work-items to maximize block RAM (BRAM) utilization. We show that our use of dynamic data partitioning for KMC yields a 1.5× overall speedup over an earlier implementation by Winterstein *et al.*, which was already optimized for FPGAs but which relied on static data partitioning [20]. We also show best-case speedup of 1.9× when the number of work-items is fixed. We encourage readers to view our codebase at https://github.com/nadeshr/kmeans-stealing.git.

2. MOTIVATING EXAMPLE

Consider a program that traverses, depth first, binary trees with integer-valued nodes with the help of a stack from/to which pointers to sub-trees yet to be traversed can

FPGA'16, February 21-23, 2016, Monterey, CA, USA

 ISBN 978-1-4503-3856-1/16/02…$15.00

DOI: http://dx.doi.org/10.1145/2847263.2847343

be popped/pushed. The TRAVERSE program, shown in Algorithm 1, represents a very common class of software programs, yet most HLS tools cannot synthesize it efficiently. Here, POP updates the tree pointer that it is passed by reference, and it returns a Boolean encoding whether the operation succeeded. Each tree node comprises some data (d) and pointers to left and right sub-trees (l and r).

Algorithm 1 Tree traversal

```
1:  procedure TRAVERSE(tree* t)
2:      stack s ← new stack
3:      s.PUSH(t)
4:      while s.POP(&t) do
5:          if t=NULL continue
6:          PROCESS(t->d)
7:          s.PUSH(t->r)
8:          s.PUSH(t->l)
9:      end while
10: end procedure
```

The first serious attempt to automatically synthesize FPGA implementations of programs like TRAVERSE was made by Winterstein *et al.* [20]. Under the assumption that the order in which tree nodes are PROCESS'd is immaterial, they divide the tree, at a fixed distance from the root, into a small number (say, P) of disjoint sub-trees that can be traversed in parallel. Applied to our TRAVERSE example, their transformation with $P = 2$ would yield Algorithm 2, in which the vertical line separates parallel threads. The chief shortcoming of their approach is that the static distribution of the workload is optimal only in the case that the input tree is perfectly balanced.

Algorithm 2 Parallel tree traversal (static partitioning)

```
1:  procedure TRAVERSE2(tree* t)
2:      if t=NULL return
3:      PROCESS(t->d)
4:      stack s0 ← new stack          | stack s1 ← new stack
5:      s0.PUSH(t->r)                 | s1.PUSH(t->l)
6:      tree* t0                      | tree* t1
7:      while s0.POP(&t0) do          | while s1.POP(&t1) do
8:          if t0=NULL continue       |     if t1=NULL continue
9:          PROCESS(t0->d)            |     PROCESS(t1->d)
10:         s0.PUSH(t0->r)            |     s1.PUSH(t1->r)
11:         s0.PUSH(t0->l)            |     s1.PUSH(t1->l)
12:     end while                     | end while
13: end procedure
```

In this paper, we present an alternative approach to implementing TRAVERSE-like programs, in which the workload is dynamically distributed at run-time via work-stealing. This overcomes the potential unpredictabilities caused by suboptimal partitioning, data-dependent task execution time, and dynamic sub-task creation. When applied to our TRAVERSE example, again setting $P = 2$, we obtain Algorithm 3. Our approach replaces the stacks with double-ended queues (deques) to enable stealing. Each thread seeks to get work by popping from the local deque ($q[0]$ or $q[1]$), or stealing from the other deque ($q[1]$ or $q[0]$) if popping fails. This implementation uses a *done* array of Boolean flags to ascertain when there is no remaining work in any deque.

Algorithm 3 Parallel tree traversal (dynamic partitioning)

```
1:  procedure TRAVERSE3(tree* t)
2:      deque q[2] = {new deque, new deque}
3:      bool done[2] = {0, 0}
4:      while done ≠ {1, 1} do        | while done ≠ {1, 1} do
5:          tree* t0                  |     tree* t1
6:          done[0] ← q[0].POP(t)     |     done[1] ← q[1].POP(t)
7:                  || q[1].STEAL(t)  |             || q[0].STEAL(t)
8:          if t0=NULL continue       |     if t1=NULL continue
9:          PROCESS(t0->d)            |     PROCESS(t1->d)
10:         q[0].PUSH(t0->r)          |     q[1].PUSH(t1->r)
11:         q[0].PUSH(t0->l)          |     q[1].PUSH(t1->l)
12:     end while                     | end while
13: end procedure
```

3. DESIGNING THE BASELINE

This section introduces a K-means clustering algorithm, explains why it stands to benefit from work-stealing, and describes how we produced an OpenCL implementation to serve as a baseline for our case study.

K-means clustering (KMC) refers to the problem of partitioning a set of D-dimensional points $X = \{x_1, \ldots, x_N\}$ into a set of clusters $\mathcal{S} = \{S_1, \ldots, S_K\}$ where K is provided as a parameter. A cluster S_i is represented by the geometrical center μ_i of its points. The goal is to assign each point in X to the cluster with the nearest center. In this paper, we consider an efficient algorithm for KMC [11] that uses a *kd-tree* data structure instead of working directly on the point set X. The algorithm begins by choosing a random initial center-set, say $M_0 = \{\mu_1, \ldots, \mu_K\}$. The set $\mathcal{S}$ is iteratively refined until it no longer changes.

In order to assess the potential of work-stealing on the KMC algorithm, we need a baseline OpenCL design for comparison. OpenCL applications are divided into host code that runs on a CPU, and kernel code that runs on an accelerator device (an FPGA in our case). In our application, the host code builds the input tree and partitions it into P sub-trees, each processed by one of P independent work items. Algorithm 4 shows our OpenCL kernel for the KMC algorithm, following Winterstein *et al.*'s implementation in C that was optimized for Vivado HLS [20]. We also provide a graphical representation of the kernel in Figure 2a. The inputs of the KMC kernel are an array of sub-trees t and a center-set M. The kernel uses a heap h that holds the temporary candidate center-sets, a stack whose entries consist of a tree pointer and a heap pointer, and a shared array of center-sets *Ms* whose elements are reduced to the final center-set result (line 16) after all **while** loops terminate. This result can be then passed to future kernel iterations.

This kernel exhibits some of the features seen in Algorithm 1 that make it a good candidate for acceleration with work-stealing: the processing time of each tree node (line 10) depends on the center-set data, and the decision to traverse a node's children is also data-dependent (line 11 and 12). The effectiveness of parallelization depends on the balancedness of the number of nodes in each sub-tree, which depends on the input data set [20]. It is also known that the heuristics used to generate the *kd-tree* are not optimal [11], which means that even in the best-case the tree is not guaranteed to be perfectly balanced.

Algorithm 4 OpenCL Baseline KMC algorithm

```
1:  attribute(REQD_WORK_GROUP_SIZE(P,1,1))
2:  kernel KMC1(global tree *t[P], local centerset *M)
3:      local stack[P] s
4:      global heap[P] h
5:      local centerset[P] Ms
6:      i ← GET_LOCAL_ID(0)
7:      Ms[i] ← M
8:      s[i].PUSH(t[i], h[i])
9:      while s[i].POP(&t[i], &h[i]) do
10:         if PROCESS(t[i], &h[i], &Ms[i]) then
11:             s[i].PUSH(t[i]->r, h[i])
12:             s[i].PUSH(t[i]->l, h[i])
13:         end if
14:     end while
15:     barrier
16:     if i = 0 then M ← REDUCE(Ms)
17: end kernel
```

We found that OpenCL's explicit parallelism simplified our design entry considerably. Where the original C design was a sequential program annotated with special HLS directives to eliminate inferred dependencies between parallel execution units, our OpenCL design could simply define each work-item with a piece of sequential code.

In order to efficiently implement our OpenCL kernel for FPGAs, we need to consider restrictions imposed by AOCL. Firstly, we used OpenCL's `reqd_work_group_size` attribute to ensure that AOCL synthesize only the required number of work-items [1]. Secondly, we have to consider the size of arrays when deciding whether they should be declared in OpenCL private, local or global memory. AOCL implements private memory as registers, local memory as BRAMs, and global memory as DDR memory [2]. Although some arrays could technically be declared private, we actually declare them as local or global memory to save FPGA resources. Based on the size constraints presented by Winterstein *et al.*, we declare the stacks and center-sets in local memory, and the heaps and sub-trees in global memory.

4. ADDING WORK-STEALING

We now describe how we use work-stealing, which we have implemented using OpenCL's atomics, to add dynamic load balancing to the KMC algorithm.

Our implementation of work-stealing follows Cederman and Tsigas [5], who give an implementation for GPUs of an algorithm due to Arora et al. [3]. The implementation is based around a collection of double-ended queues (deques), one per OpenCL work-item in this work. A deque comprises *head* and *tail* pointers to opposite ends of a task array. Each deque provides three main operations: push, pop and steal. Each OpenCL work-item owns a deque to which it has exclusive push and pop access via the *tail* pointer, as seen in Figure 1a and 1c. Since only one work-item can push or pop a deque, the *tail* accesses can be non-atomic. On the other hand, all work-items can steal tasks from any non-empty deque via the *head* pointer (as seen in Figure 1b). Since any work-item can update the *head* pointer, access to the *head* must be atomic (specifically, via OpenCL's `atomic_cmpxchg` function [13]) to eliminate data races. The use of atomics during stealing arbitrates access to the deque

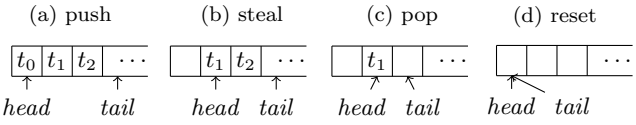


Figure 1: Deque Chronological Sample Execution.

in a fine-grained manner, because multiple work-items can attempt an `atomic_cmpxchg` operation at the same time but only one will succeed. This policy is non-blocking and guarantees that at least one work-item makes progress.

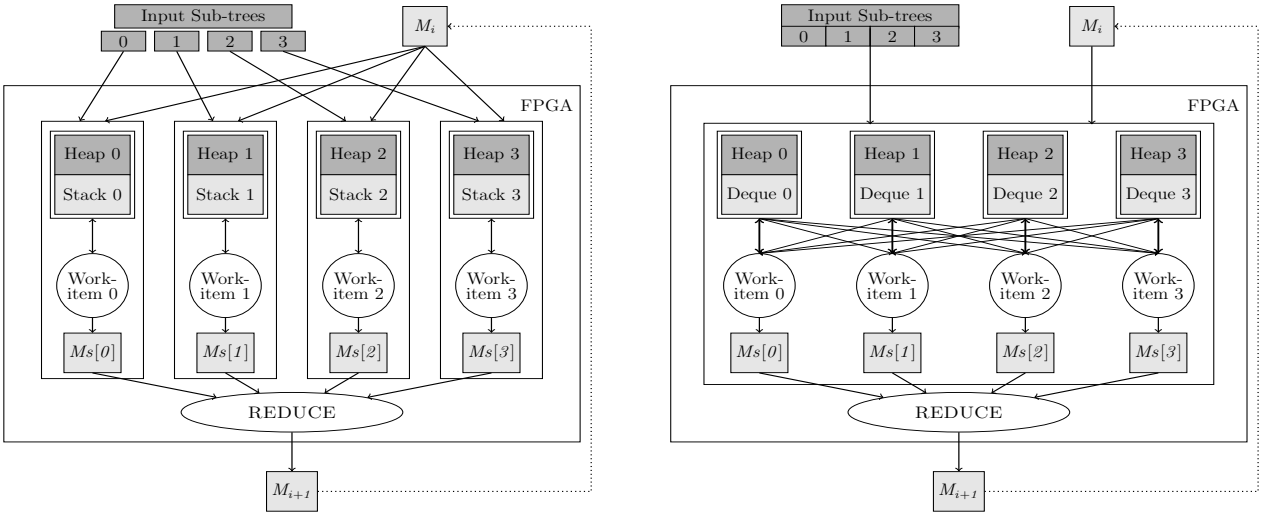
Algorithm 5 presents the KMC algorithm with work-stealing and Figure 2b provides a graphical representation. Compared to the baseline (Algorithm 4), a key change is that the stacks have become deques. We also replaced the POP function with the GET function (line 11), which updates $t[i]$ and $h[i]$ if successful. GET first attempts to pop from the local deque $q[i]$, and then steals from the next deque $q[sid]$ if popping fails. *sid* initially refers to the immediately-following deque, but is updated in a round-robin style if stealing fails. We use a Boolean field *finish* in the deque to keep track of the system's workload: if GET fails to obtain work (line 12), then the work-item is deemed to have finished. When all work-items have finished (line 10), the loop terminates since there is no possibility of new tasks being pushed.

Algorithm 5 OpenCL Work-stealing KMC algorithm

```
1:  attribute(REQD_WORK_GROUP_SIZE(P,1,1))
2:  kernel KMC2(global tree *t[P], local centerset *M)
3:      local deque[P] q
4:      global heap[P] h
5:      local centerset[P] Ms
6:      i ← GET_LOCAL_ID(0)
7:      sid ← (i + 1) mod P
8:      Ms[i] ← M
9:      q[i].PUSH(t[i], h[i])
10:     while ¬(q[0].finish && ... && q[P − 1].finish) do
11:         success ← GET(&t[i], &h[i], q, i, &sid)
12:         q[i].finish ← ¬success
13:         if success then
14:             if PROCESS(t[i], &h[i], &Ms[i]) then
15:                 q[i].PUSH(t[i]->r, h[i])
16:                 q[i].PUSH(t[i]->l, h[i])
17:             end if
18:         end if
19:     end while
20:     barrier
21:     if i = 0 then M ← REDUCE(Ms)
22: end kernel
```

We use a single OpenCL work-group with P work-items to allow our workload to be shared across all work-items. This way, we can characterize the benefits of work-stealing *on-chip* since otherwise work-stealing among multiple work-groups would have to be performed on off-chip global memory.

In adding work-stealing to our design, we expect some resource overhead and clock frequency penalty due to atomics [1]. Figure 2 highlights a further architectural overhead caused by work-stealing. In work-stealing, all the deques,



(a) Baseline implementation

(b) Work-stealing implementation

Figure 2: KMC algorithm for $P = 4$ (dark gray is global memory and light gray is local memory).

heaps and sub-trees are shared between all work-items (as suggested by the crossbar in Figure 2b), where in the baseline these were private to each work-item. Although we introduce sharing in work-stealing, the cycle times of these shared data structures remain comparable since we already have to synthesize larger arrays in OpenCL local or global memory for the baseline.

In addition, we noticed that AOCL replicates local data structures that are accessed many times in a kernel [1]. This replication policy penalizes large structures, since a structure is accessed every time one of its fields is accessed, so we split such structures into individual fields wherever possible. We further reduce the replication effect by manually partitioning the task arrays.

5. EVALUATION

This section evaluates how effectively work-stealing balances the processing load across work-items in the KMC algorithm (Section 5.1). We also evaluate the effect of work-stealing on wall-clock time (Section 5.2) and resource consumption (Section 5.3). We used AOCL (version 15.0.0) for HLS and Quartus (version 15.0.0) for RTL synthesis. All results are taken from fully placed-and-routed designs running on a Nallatech P385 D5 board that includes an Altera Stratix V D5 FPGA and 8GB DDR3 memory.

Our goal is to have as many work-items within a single OpenCL work-group as resource utilization on the FPGA permits. At full capacity of the FPGA, we are able to synthesize 64 work-items for our baseline implementation and 32 work-items for our work-stealing implementation.

The inputs to the KMC algorithm are a tree built from 2^{20} data-points and a 128-element center-set. Each stack and deque holds 128 elements. We run the KMC algorithm for 16 iterations.

5.1 Load Balancing

The goal of our work-stealing approach is to minimize the variation in workload across OpenCL work-items, regardless of the shape of the input tree, to avoid situations where

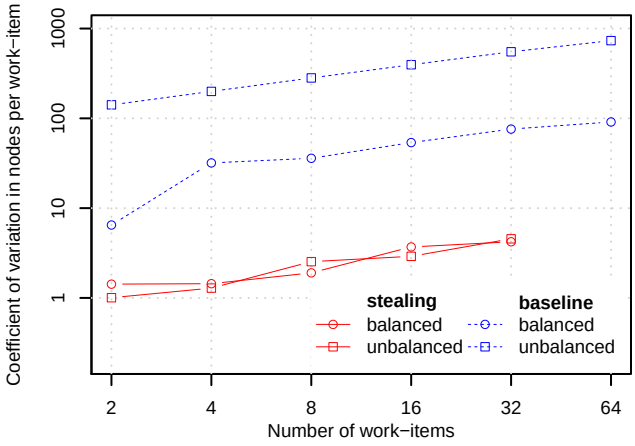


Figure 3: Processing load variation over work items.

the entire kernel has to wait for a single, heavily loaded work-item to complete. The processing load of the KMC algorithm is proportional to the aggregate number of *node-center-pairs*, *i.e.* the cumulative number of candidate centers processed at the visited tree nodes to compute the clustering result. This number depends on the initial choice of centers and, more importantly, on the shape of the input tree. The host code uses heuristics to produce trees that are *fairly balanced*. To push our implementations to their extremes, we also compare with trees that are *perfectly unbalanced*, i.e. trees whose nodes form a single chain with no branching.

Fig. 3 quantifies the variation in the processing load across work-items for each P. It uses the *coefficient of variation*, which is the standard deviation divided by the mean. We make two observations about this graph. Firstly, the variations for the work-stealing cases are much smaller than either of the baseline cases. Work-stealing is able to improve even the best-case input tree because the routine that builds the

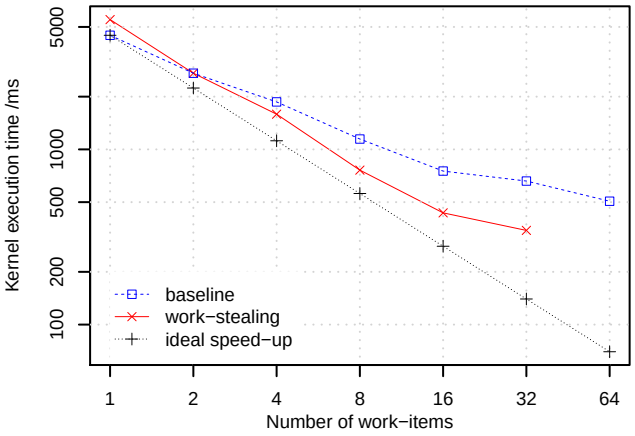


Figure 4: Kernel execution time for the filtering algorithm.

tree does not guarantee perfect balance and and the decision to recurse on the children of a node, which in turn spawns more subtasks, is data-dependent. Secondly, work-stealing is immune to the shape of the tree and produce very similar workload variations in both scenarios.

5.2 Execution time

Next, we demonstrate the effect of load balancing on the overall kernel execution time (wall-clock time). The timing results are obtained from the number of clock cycles divided by the minimum achievable clock frequency. The achievable clock frequencies range from 237 MHz ($P = 1$) to 163 MHz ($P = 64$) for the baseline, and from 214 MHz ($P = 1$) to 201 MHz ($P = 32$) with an outlier of 184 MHz at $P = 4$ for the work-stealing implementation.

Fig. 4 shows the kernel execution time (excluding host preprocessing) for both implementations in logarithmic scale with respect to P. We also include the theoretically achievable speed-up, which we obtain by dividing the single work-item baseline by P, to give an indicate to how these implementations could scale. At $P = 1$ in Figure 4, work-stealing performs worse than the baseline implementation because it lowers the clock frequency, as seen in Table 1, and is not applicable to a single work-item. From $P = 4$ onwards, work-stealing is closer to the ideal speed-up trend. The speed-ups for fixed P are shown in the last row of Table 1. At full BRAM capacity on the FPGA, work-stealing achieves an overall speed-up of 1.5× over the baseline. This comparison is made between $P = 64$ for the baseline and $P = 32$ for the work-stealing implementation, showing that the load-balanced version only requires half the number of work-items to achieve better performance.

5.3 Resources

The acceleration gained from load balancing comes at the cost of additional FPGA resources. The total logic utilization (as reported by Quartus II Fitter) for both implementations varies between 47% and 57%. The utilization of DSP blocks is below 3% and remains the same for both the baseline and work-stealing implementations. Table 1 quantifies the resource overheads of work-stealing relative to the baseline for each P in terms of logic utilization, BRAM and clock frequency degradation. Ultimately, BRAM is the resource that limits the scaling of P that is synthesizable for the

Table 1: Relative resource overhead and relative clock rate penalty due to the work-stealing implementation (negative values indicate increase in clock frequency).

P	1	2	4	8	16	32
Logic overhead	15%	2%	4%	6%	6%	8%
RAM overhead	57%	45%	68%	58%	58%	39%
Clock penalty	10%	1%	16%	7%	0%	−1%
Speedup	0.8×	1.0×	1.2×	1.5×	1.7×	1.9×

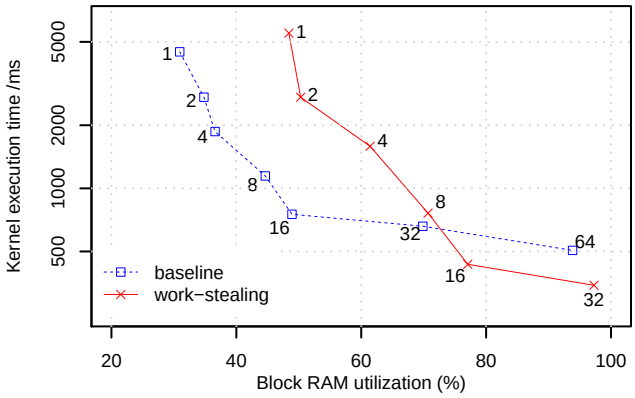


Figure 5: BRAM utilization vs. kernel execution time, for various values of P

FPGA. The work-stealing implementations requires 350 to 550 additional BRAMs for the same P. The BRAM overhead is due to the replication performed by AOCL as discussed in Section 4, which limits work-stealing to $P = 32$ work-items and the baseline to $P = 64$.

Fig. 5 compares the BRAM-time graphs of both implementations. The break-even point occurs at 1500 BRAMs. The work-stealing implementation provides enough speed-up such that, despite its BRAM overhead, it becomes more efficient in terms of BRAM-time product beyond this point. For the two end-points of the BRAM-time graphs ($P = 64$ for the baseline, $P = 32$ for work-stealing), we spend 3.6% more BRAMs (5.6% more logic utilization, no DSP blocks overhead) for a 1.5× improvement of execution time.

6. RELATED WORK

Several authors have investigated the capabilities of OpenCL for FPGAs [6, 14, 16, 10, 17], but their focus has been on highly-uniform, FPGA-friendly benchmarks such as vector addition and matrix multiplication.

Wang *et al.* have studied the problem of data partitioning within an FPGA, but consider only static partitioning [19]; in our work we move towards dynamic partitioning. Wang *et al.*'s work provides the only other case study of OpenCL's atomics of which we are aware; they use atomics to implement locks where we use them in lock-free context. They report that atomics are 'expensive' and that their use is a 'disadvantage'. While our work does not contradict their claims in general, it does provide evidence that atomics can play a vital role in highly-efficient FPGA implementations.

Principles of work-stealing appear in Kestur *et al.*'s FPGA implementation of matrix–vector multiplication [12], but their design is at the register transfer level (RTL). We believe our work to be the first FPGA work-stealing implementation programmed in a high-level language, in the form of OpenCL. Similarly, Nahill *et al.* have investigated imple-

menting non-blocking synchronization on FPGAs [18], but they also work at the RTL level.

The compilation of parallel software threads to hardware has been studied in the context of the Kiwi [9] and LegUp [4] HLS tools, and George *et al.* have looked into mapping parallel computation patterns, such `map`, `reduce`, `zipWith` and `foreach`, into efficient hardware models for FPGAs [7]. All of these tools are limited to lock-based concurrency; our work explores the lock-free case.

Finally, Kumar *et al.* have devised a task-based parallel programming model that involves work-stealing between CPUs and digital signal processors (DSPs) [15]. Our work, on the other hand, implements work-stealing within a single FPGA device.

7. CONCLUSION

We have demonstrated an effective use case of the OpenCL programming language's explicit parallelism constructs to achieve an implementation of work-stealing on FPGAs. Our work is particularly interesting because we are able to describe a work-stealing implementation in a lock-free manner using a state-of-the-art HLS tool (AOCL). Lock-freedom properties require concurrent programs to be non-blocking and synchronized in a fine-grained manner using atomics. Atomics have been deemed 'expensive' in terms of performance and resources by the Altera OpenCL Programming Guide [1]. However, we show an overall speedup 1.5× for the KMC algorithm that incorporates work-stealing (using atomics) to perform dynamic load balancing, compared to a baseline with static partitioning. Our work thereby provides demonstrates that the dynamic load-balancing advantages of work-stealing far outweigh the performance and resource overhead penalties that atomics introduce.

Acknowledgements. We thank David Thomas, Gordon Inggs and our reviewers for their feedback and encouragement. The support of the EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS, Grant Reference EP/L016796/1) and grants EP/I020357/1 and EP/K015168/1, the Royal Academy of Engineering and Imagination Technologies is gratefully acknowledged.

8. REFERENCES

[1] Altera. *Altera SDK for OpenCL - Best Practices Guide*. OCL003-14.1.0, 2014.

[2] Altera. *Altera SDK for OpenCL - Programming Guide*. OCL002-14.1.0, 2014.

[3] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA*, 1998.

[4] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *FPGA*, 2011.

[5] D. Cederman and P. Tsigas. Dynamic load balancing using work-stealing. In *GPU Computing Gems*. Elsevier, 2012.

[6] T. Czajkowski, U. Aydonat, D. Denisenko, and J. Freeman. From OpenCL to high-performance hardware on FPGAs. In *FPL*, 2012.

[7] N. George, H. Lee, D. Novo, M. Owaida, D. Andrews, K. Olukotun, and P. Ienne. Automatic support for multi-module parallelism from computational patterns. In *FPL*, 2015.

[8] V. Gramoli. More than you ever wanted to know about synchronization. In *PPoPP*, 2015.

[9] D. Greaves and S. Singh. Kiwi: Synthesis of FPGA circuits from parallel programs. In *FCCM*, 2008.

[10] M. Hosseinabady and J. L. Nunez-Yanez. Optimised OpenCL workgroup synthesis for hybrid ARM-FPGA devices. In *FPL*, 2015.

[11] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *Pattern Matching and Machine Intelligence*, 24(7):881–892, July 2002.

[12] S. Kestur, J. D. Davis, and E. S. Chung. Towards a universal FPGA matrix–vector multiplication architecture. In *FCCM*, 2012.

[13] Khronos Group. *The OpenCL 1.0 Specification*. 2009.

[14] H.-S. Kim, M. Ahn, J. A. Stratton, and W.-m. W. Hwu. Design evaluation of OpenCL compiler framework for coarse-grained reconfigurable arrays. In *FPT*, 2012.

[15] V. Kumar, A. Sbîrlea, A. Jayaraj, Z. Budimlić, D. Majeti, and V. Sarkar. Heterogeneous work-stealing across CPU and DSP cores. In *HPEC*, 2015.

[16] V. Mirian and P. Chow. Using an OpenCL framework to evaluate interconnect implementations on FPGAs. In *FPL*, 2014.

[17] T. T. Mutlugün and S.-D. Wang. OpenCL computing on FPGA using multiported shared memory. In *FPL*, 2015.

[18] B. Nahill, A. Ramdial, H. Zeng, M. Di Natale, and Z. Zilic. An FPGA implementation of wait-free data synchronization protocols. In *ETFA*, 2013.

[19] Z. Wang, B. He, and W. Zhang. A study of data partitioning on OpenCL-based FPGAs. In *FPL*, 2015.

[20] F. Winterstein, S. Bayliss, and G. A. Constantinides. High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *FPT*, 2013.

[21] Xilinx. *SDAccel Development Environment*. UG1023 (v2015.1), 2015.