Performance Evaluation of Compiler Optimizations in FPGA Accelerators

Gustavo Leite[1], Alexandro Baldassin[1], Guido Araújo[2], José Nelson Amaral[3]

[1]Departamento de Estatística, Matemática Aplicada e Computação
Universidade Estadual Paulista (UNESP)
Rio Claro – SP – Brazil

[2]Instituto de Computação
Universidade Estadual de Campinas (UNICAMP)
Campinas – SP – Brazil

[3]Computing Science Department
University of Alberta
Edmonton – AB – Canada

{gustavo.leite,alexandro.baldassin}@unesp.br,

guido@ic.unicamp.br, jamaral@ualberta.ca

Abstract. *With the increasing power wall in microprocessor design, engineers shifted their attention to heterogeneous architectures, wherein several classes of devices are used for computation. Among them are FPGAs which offer comparable performance to CPUs while consuming only a fraction of energy. Despite the increasing interest in these devices, programmability and performance engineering in FPGAs remain hard. This work presents an evaluation of the most prominent code transformations targeting FPGAs. More specifically, it studies the performance effect of unrolling loops, replicating compute units and transferring data using DMA in a matrix multiplication OpenCL kernel through an Intel® FPGA. The results indicate that these optimizations can achieve speedups up to 3.78× for a matrix multiplication application, and 412.5× speedup in data transfer.*

1. Introduction

With Moore's Law [Moore 1965] and Dennard Scaling [Dennard et al. 1974] approaching their end, the high-performance market shifted its focus towards heterogeneous computing systems, where each device is specialized to accelerate domain-specific applications [Hennessy and Patterson 2019a]. Among the many classes of devices, the most noteworthy are graphical processing units (GPUs), tensor processing units (TPUs), and field-programmable gate arrays (FPGAs) [Hennessy and Patterson 2019b]. In particular, FPGAs have existed since the mid-1980s and have been used to create logic circuits for embedded systems. More recently, FPGAs also have been commercialized as accelerators, integrating multicore ARM-processors, DRAM, digital signal processors (DSPs), storage onto a single board usually referred to as "System on Chip" (SoC). Due to their reconfigurable nature, FPGAs offer comparable performance compared to CPUs and usually higher performance per watt compared to GPUs.

Despite the clear benefits, FPGA devices are still not as widespread as GPUs, for instance. This is due to three main reasons: (i) the long time to perform hardware synthesis; (ii) the lack of a high-level programming model; and (iii) lack of portability. The process of translating a circuit written in hardware description language into a bitstream—a hardware configuration file—is called hardware synthesis. Synthesizing hardware can last from minutes to a few days [Bacon et al. 2013]. This characteristic limits the ability to write incremental code and fast prototyping. Although the programmability of these devices has improved since their first appearance, current software development kits adopted OpenCL [Khronos Group 2019]. For novices and non-experts, even that can be fairly low-level. Before that, reconfigurable hardware was mainly programmed using hardware description languages (VHDL, Verilog), which was even worse since these languages work in the register transfer level (RTL), a paradigm most software developers are unfamiliar with. Lastly, porting a design from one device to another requires a considerable amount of work to be redone. The challenge is aggravated when porting across devices from different vendors. Performance tuning is not trivial and usually requires several iterations. With other classes of devices, it is straightforward to experiment with different implementations. With FPGAs, however, the long synthesis time severely delays the design cycle. With this scenario in mind, it is important that we better understand how to tune applications and evaluate trade-offs.

This work makes the following contributions: (i) it compiles and explains a subset of code transformations found in the literature and the state of research about FPGAs for high-performance computing; and (ii) it presents a performance evaluation of these transformations on a matrix multiplication application. This paper is organized as follows: Section 2 provides background on FPGAs; Section 3 introduces code transformations that can be applied to OpenCL kernels; Section 4 presents our experimental evaluation of the transformations; and finally Section 5 concludes the work.

2. Background and Related Work

Field-Programmable Gate Arrays are silicon devices that can be reconfigured to emulate any combinational and sequential logic the user desires. Its ability to be reprogrammed after manufacturing grants its name "field-programmable". The FPGA chip is composed of hundreds of thousands of logic blocks organized as a mesh—hence the name "gate array"—and a programmable interconnect that links these elements. The basic building blocks of FPGAs are logic elements. These elements are usually composed of LUTs (lookup tables) for combinational logic, registers for sequential logic and adders for arithmetic.

For many years, the use of FPGAs has been restricted to engineers with hardware expertise. This is attributed to the lack of high-level abstractions to program such devices. Formerly, engineers would design the circuit in a hardware description language (HDL) and obtain the concrete representation of the circuit through a process called synthesis. These representations are commonly referred to as *bitstreams*, and they inform how bits in the logic elements and programmable switches should be set to compose the circuit. With modern SDKs, however, programmers are able to implement the algorithm as OpenCL kernels as one would normally do for other types of accelerators. The vendor tool chain gets the OpenCL source and translates it to HDL via high-level synthesis. The HDL description, in turn, goes through the process of synthesis. At the end of this process, a

bitstream file is produced that can be used to program the circuit onto the device.

Although functionally portable, executing code targeted for GPUs on FPGAs is unlikely to yield any benefits both in terms of speedup and power efficiency. The code is said to lack performance portability [Lee et al. 2016]. This is due to architectural differences: while GPUs are massively parallel devices with thousands of computing cores suited for data-parallel computation, FPGAs are more suitable for executing pipelined computations. In this context, Zohouri *et al.* [Zohouri et al. 2016] evaluate the impact optimizations have on power consumption and performance when comparing FPGAs against CPUs and GPUs. The authors apply optimizations manually on Rodinia [Che et al. 2009] applications and execute them on an Intel® FPGA. The work shows that FPGAs present better power efficiency in every application except for one (cfd). In terms of application speedup, FPGAs stay behind GPUs while offering comparable performance to CPUs.

Lloyd *et al.* [Lloyd et al. 2017] automate the process of optimizing kernels using compiler passes. The authors describe a method for integrating host and device compilers by propagating information from one another thus revealing new optimization opportunities. Their method optimizes code on the intermediate representation level and are thus restricted to Intel® FPGAs. Although lower speedups were measured when compared to Zohouri *et al.*, it has the advantage of being automatic. Lee *et al.* [Lee et al. 2016] propose the use of a directive-based approach for high performance computing with FPGAs. The authors extended the OpenARC [Lee and Vetter 2014] compiler to produce OpenCL kernel source from a C code annotated with OpenACC [OpenACC 2019] directives. The kernel generated from the annotated region is already optimized. A follow-up publication [Lambert et al. 2018], by the same group, adds new optimizations to the compiler. Unlike the work of Lloyd *et al.*, the solution does not require host and device compiler integration because the compiler has control of the device code that is produced.

The aim of this work is to apply some optimizations and analyze their performance impact in FPGAs. This is of extreme importance due to the popularization of FPGA devices and their use in the upcoming years [Amazon Web Services 2019, Barr 2017, Fowers et al. 2018]. Unlike GPUs which are widespread and have a broad set of open-source tools for profiling and characterization, FPGA development is still in its infancy and the tools are mainly proprietary. There is also the difficulty of running experiments due to the long compilation time, which poses a barrier to the thorough characterization of performance. Therefore, this work serves as a first step towards understanding the performance of code optimizations for FPGAs with the aim of automating them through a compiler infrastructure.

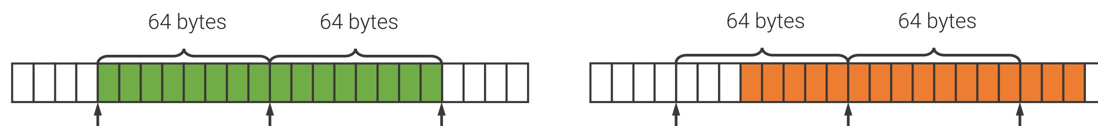
3. Code Transformations

This section presents some prominent code transformations found in the literature: Section 3.1 introduces data transfers using DMA; Section 3.2 shows loop unrolling; Section 3.3 presents compute unit replication; Section 3.4 introduces the `restrict` keyword. These transformations are targeted at Intel® FPGAs.

3.1. DMA Alignment

When transferring data between host and device memory, it is possible to increase efficiency using direct memory access (DMA). However, in order to Intel® FPGAs make

use of DMA hardware, the data is required to be 64 bytes aligned, as depicted in Figure 1. If this condition is not met, the CPU will stay busy transferring the data byte-to-byte. This way, it is recommended that the user allocates aligned memory by calling `_aligned_malloc` (C99, Windows), `posix_memalign` (C99, Linux) or `std::aligned_alloc` (C++17) instead of `malloc`.



(a) Data is aligned to 64 bytes in main memory **(b) Data is offset from the 64 bytes aligned position in main memory.**

Figure 1. Aligned versus non-aligned memory.

3.2. Loop Unrolling

Loops inside the kernel can be optimized using the loop unrolling transformation [Aho et al. 2006]. Intel® FPGA SDK for OpenCL provides the `#pragma unroll X` directive, where X is the unroll factor, as the example shown in Listing 1. The compiler will try to unroll loops even if the user did not insert this pragma. However, it is possible to force no unrolling by using a factor of 1. According to the best practices guide [Intel Corporation 2018], the FPGA compiler arranges the operations inside a loop in a pipeline fashion respecting the data flow semantics. Normally, one loop iteration is issued per cycle; therefore, if the loop is unrolled, more iterations can be finished each cycle, increasing its performance. Besides making the pipeline wider, the compiler will also coalesce memory accesses, further improving performance. The downside of this transformation is that it increases area resource utilization.

```
1 #pragma unroll 2
2 for (int k = 0; k < n; k++) {
3   acc += A[i * n + k] * B[k * n + j];
4 }
```

Listing 1. Example of loop unrolling in OpenCL.

3.3. Compute Unit Replication

It is possible to increase throughput by replicating the pipeline or creating SIMD (Single Instruction Multiple Data) units [Intel Corporation 2018]. The user can control these parameters by inserting attributes in the source code. Placing the attribute `num_compute_units(N)` before the function definition instructs the compiler to replicate the pipeline N times, so that multiple work-items can be computed in parallel. This process is demonstrated in Listing 2. This attribute works best when the computation is embarrassingly parallel, *i.e.*, no synchronization is needed. Replicating the pipeline can occupy more area because load/store units are also replicated besides the pipeline. Having excessively many pipelines can harm performance by putting too much pressure in the memory subsystem.

```
1  __attribute__((num_compute_units(4)))
2  __kernel void kernel_replication() { /* ... */ }
```

Listing 2. Example of replication and SIMD.

3.4. Restrict Parameters

The Intel® optimization guide [Intel Corporation 2018] encourages users to mark pointers as `restrict` in kernel parameters. This keyword tells the compiler that the pointer does not alias with other pointers in the program. Without this information, the compiler must be conservative and create data dependencies to ensure correctness. These dependencies are potentially useless and will prevent the compiler to issue loads and stores in parallel, degrading throughput. Of course, the user herself must guarantee that the pointers do not alias otherwise undefined behavior can be expected.

4. Performance Characterization

This experimental analysis was performed on a system powered by an Intel® Core™ i7-4770 @ 3.40GHz with 32GB of RAM running on Ubuntu 16.04.5 LTS Codename Xenial. This machine has also a Terasic DE5-Net board that holds an Intel® Stratix® V FPGA (5SGXA7) and 4GB 1600MHz DDR3 RAM connected to the CPU through a x16 PCIe 2.0 bus capable of transferring 8000MB per second. The FPGA has 234720 ALMs, 938880 registers, 2560 memory blocks, and 256 DSP blocks [Intel Corporation 2015]. The host C and C++ compilers used were the ones available in the GNU Compiler Collection (GCC) version 7.4.0. Our benchmarks were written in C++17. For device code compilation we used the Altera Offline Compiler bundled in the Intel® FPGA SDK for OpenCL version 18.0.0.614.

In order to evaluate the impact of code transformations, we selected the naive matrix multiplication algorithm as the baseline. Let A and B be matrices with sizes $n \times m$ and $m \times o$, respectively. The multiplication of A and B results in a matrix C of size $n \times o$. Element $c_{i,j}$ of C equals the dot product between i-th row in A and j-th column in B. This algorithm's pseudo-code is shown in Listing 3. It can be noted that this algorithm has three nested loops, therefore, if $n = m = o$, its time complexity can be expressed as $O(n^3)$.

```
1  for (int i = 0; i < n; ++i) {
2    for (int j = 0; j < o; ++j) {
3      C[i,j] = 0;
4      for (int k = 0; k < m; ++k) {
5        C[i,j] += A[i,k] * B[k,j];
6      }
7    }
8  }
```

Listing 3. Algorithm of the naive matrix multiplication algorithm.

The naive matrix multiplication algorithm implemented as an OpenCL kernel is shown in Listing 4. For simplicity, this implementation only multiplies square matrices of size $n \times n$. Upon execution, multiple work-items will be launched with different IDs to

perform the computation. This is the reason this code does not have the familiar structure with three nested loops.

```
1  __kernel void matrix_multiply(
2          __global const int *A,
3          __global const int *B,
4          __global int *C,
5          const unsigned int n)
6  {
7    int i = get_global_id(0);
8    int j = get_global_id(1);
9    int acc = 0;
10
11   for (int k = 0; k < n; k++) {
12     acc += A[i * n + k] * B[k * n + j];
13   }
14
15   C[i * n + j] = acc;
16 }
```

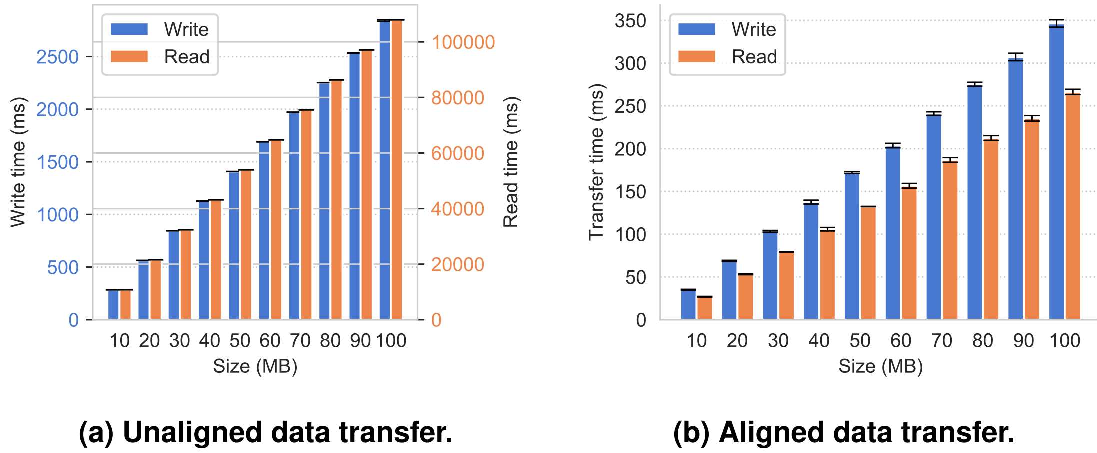
Listing 4. OpenCL kernel for computing matrix multiplication.

We use the code in Listing 4 as our baseline and apply optimizations incrementally. The code optimizations evaluated are: (i) DMA alignment; (ii) loop unrolling; (iii) compute unit replication. The time measurement was done through the OpenCL profiling API that allows the programmer to measure how long command in the queue took to execute. For each execution, we collect 16 samples and discard the first one. This is important because the bitstream is programmed onto the FPGA when the kernel is actually run for the first time, incurring overhead. For subsequent runs of the same kernel, the FPGA does not need to be reprogrammed. With the 15 remaining samples, we calculate the arithmetic mean and standard deviation.

4.1. Memory transfer

Our first benchmark compares transfer speeds of aligned vs unaligned data. We start by allocating unaligned memory, fill it with random data, and finally write and read a buffer stored in the FPGA memory. This is achieved by calling the OpenCL functions `clEnqueueWriteBuffer` and `clEnqueueReadBuffer`, respectively. The initial buffer is 10MiB and it is increased in steps of 10MiB until 100MiB. The data is presented graphically in Figures 2a (unaligned) and 2b (aligned).

As can be observed from Figure 2a, time to read data from the FPGA is one order of magnitude larger compared to writing when the data is not aligned. The PCIe interface bandwidth is symmetrical, meaning that data is transferred through the bus with the same speed in both directions. We would need access to the concrete hardware design of the FPGA in order to find out the causes of this anomaly. Of course, the chip design is the intellectual property of Intel®. As shown in Figure 2b, data transfer times are indeed reduced and both reading and writing to FPGA memory show similar magnitude in case of aligned data transfer. The speedup for transferring data from FPGA to host (read) goes up to 412.5x whereas when transferring data from host to the device (write) a speedup of 8.3x is achieved.



(a) Unaligned data transfer. **(b) Aligned data transfer.**

Figure 2. Aligned versus unaligned data transfer.

4.2. Loop Unrolling

The loop unrolling optimization was evaluated in the matrix multiplication kernel. In order to transform the source, it suffices to add the unroll pragma shown in Section 3.2 to the line preceding the loop. The kernel was recompiled with factors 1 (no unrolling), 2, 4 and 8 and tested with matrices containing 1024×1024 elements. The computation was divided into 32 work-items per work group[1]. The kernel execution time measurements can be observed in Figure 3.

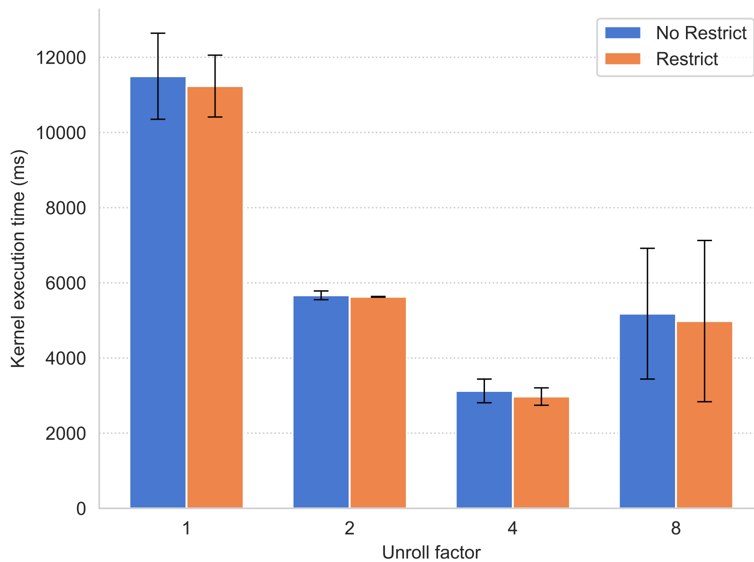


Figure 3. Matrix multiplication kernel execution time with loop unrolling.

Without unrolling the loop the execution time measured is 11.23 seconds on average. Unrolling the loop two times reduces the execution time by half, resulting in 5.63 seconds on average (1.99x speedup). With loop unrolling factor 4, the average execution time is further reduced to 2.97 seconds (3.78x speedup). Unrolling 8 times show

[1]The OpenCL work group size was determined empirically.

Table 1. Area usage of matrix multiplication kernels with loop unrolling.

Kernel	ALUTs	FFs	RAMs	DSPs
Unroll 1	9229 (2%)	12877 (1%)	102 (4%)	8 (3%)
Unroll 2	13153 (3%)	16877 (2%)	145 (6%)	14 (5%)
Unroll 4	20593 (4%)	26494 (3%)	231 (9%)	26 (10%)
Unroll 8	35846 (8%)	45740 (5%)	403 (16%)	50 (20%)

a speedup of 2.25x, however, apart from yielding a lower speedup when compared to unrolling 4 times, the measurements indicate the high deviation from the mean.

Table 1 contains the area information for the loop unrolling kernel with parameters marked as `restrict`. This table presents how many components were necessary to assemble the circuit. The components are: arithmetic and logic units (ALUTs); Flip-Flops (FFs); RAM blocks (RAMs); and DSPs. It is possible to observe that unrolling the matrix multiplication kernel does not impose a great penalty in area utilization. The critical component in this circuit is DSPs. These blocks are capable of doing multiplication and, since there is a multiplication inside the loop, these blocks tend to grow very fast. Unrolling the loop eight times uses 20% of the DSPs in the board. The second most used component is the memory blocks that reach 16% in the worst case.

4.3. Compute Unit Replication

As introduced in Section 3.3, compute unit replication make an identical copy of the pipeline so that more computation can be done in parallel. The matrix multiplication kernel was replicated with 1, 2, 4, 8 and 16 compute units and executed on matrices of size 1024×1024 elements with 16 work-items per workgroup. The data is shown in Figure 4.

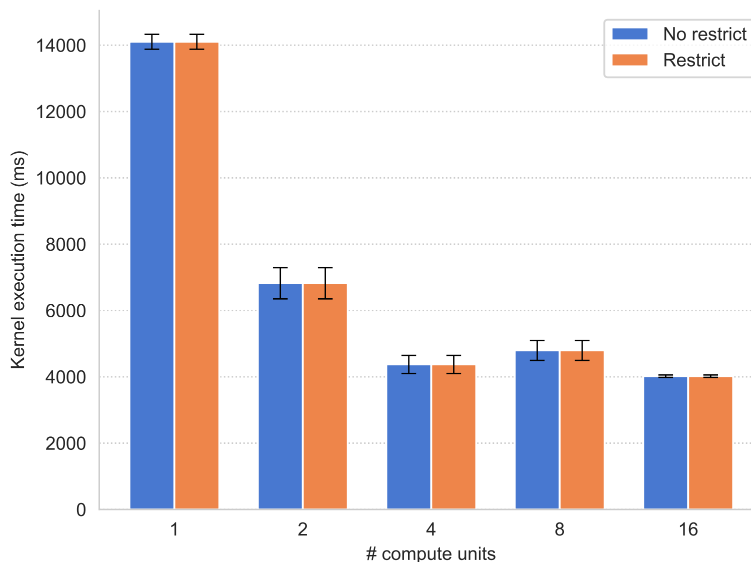


Figure 4. Matrix multiplication kernel execution time with compute unit replication.

Using a single compute unit, the matrix multiplication is computed in 14.1 seconds on average. With 2 compute units, however, kernel execution time is halved to 6.73 seconds (2.09x speedup). With 4 compute units, execution time drops to 4.44 seconds (3.17x). Above 4 compute units, returns start to diminish—4.5 seconds (3.13x) with 8 compute units and 4.02 seconds (3.51x) with 16 units. Again, no significant difference in execution time was noticed when using `restrict` keyword.

The area utilization data is presented in Table 2. Memory (RAM) blocks are critical when replicating compute units. The reason is that not only the pipeline is replicated but also the memory subsystem. Apart from using more area, having multiple load and store units can put too much pressure on the board memory. This is made evident when comparing the best execution time with loop unrolling (2.97s) against the best execution time when replicating compute units (4.02s). Also, observe that replicating the pipeline 4 times uses more DSPs (64) than unrolling the loop 4 times (50).

Table 2. Area usage of matrix multiplication kernels with compute unit replication.

Kernel	ALUTs	FFs	RAMs	DSPs
1 units	9245 (2%)	12909 (1%)	102 (4%)	8 (3%)
2 units	17629 (4%)	24670 (3%)	204 (8%)	16 (6%)
4 units	34333 (7%)	48064 (5%)	408 (16%)	32 (13%)
8 units	67805 (14%)	94980 (10%)	816 (32%)	64 (25%)
16 units	135005 (29%)	189324 (20%)	1632 (64%)	128 (50%)

4.4. Loop Unrolling + Compute Unit Replication

The previous two optimizations were combined: we unroll the loops two times for every kernel and also replicate compute units 1, 2, 4 and 8 times, respectively. We test this code with matrices of 1024×1024 elements and 16 work-items per work group. The data is represented in Figure 5.

The baseline kernel with 1 unit executes, on average, in 6.5 seconds. No significant improvement is observed when executing the kernel with 2 units. Using 4 compute units a speedup of 1.30x is measured (4.98s on average). Finally, replicating the pipeline 8 times produces a speedup of 1.27x (5.14s). One more time, the restrict parameter optimization seems to be irrelevant to performance.

Table 3 compiles the area utilization for this benchmark. Memory blocks and DSPs are even more critical with this benchmark. As expected, the kernel unrolled and replicated 8 times uses 1160 (45%) memory blocks and 112 (44%) DSPs compared to 816 (32%) RAMs and 64 (25%) DSPs for the kernel only replicated 8 times.

4.5. Discussion

The experimental results presented in this paper indicate that, among the code transformations studied and for the benchmarks used, the most important one has aligned memory allocation for data transfers. By allocating aligned memory, it was observed up to 412.5x speedup for transfer from the device to the host and 8.3x from the host to the device. The

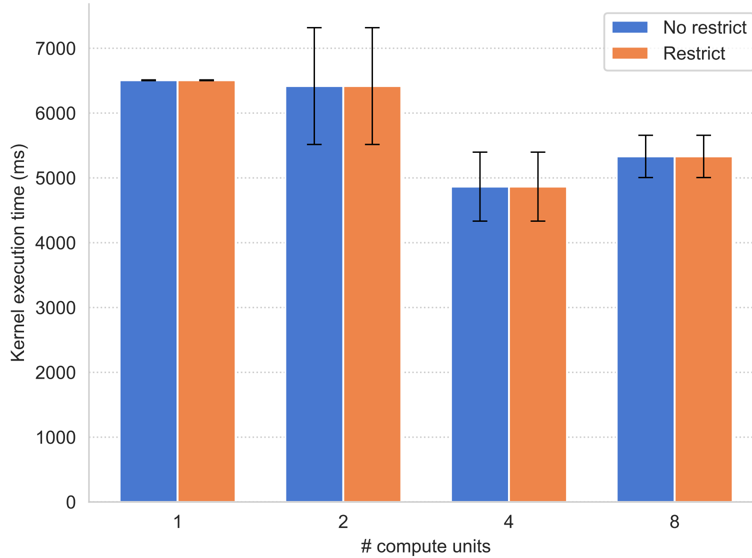


Figure 5. Matrix multiplication kernel execution time with compute unit replication and loop unrolling.

Table 3. Area usage of matrix multiplication kernels with compute unit replication and loop unrolling (factor 2).

Kernel	ALUTs	FFs	RAMs	DSPs
1 units/unroll 2	13153 (3%)	16877 (2%)	145 (6%)	14 (5%)
2 units/unroll 2	25445 (5%)	32606 (3%)	290 (11%)	28 (11%)
4 units/unroll 2	50029 (11%)	64064 (7%)	580 (23%)	56 (22%)
8 units/unroll 2	99325 (21%)	127236 (14%)	1160 (45%)	112 (44%)

experimental results also indicate that it is possible to speedup a matrix multiplication kernel by 3.78x by unrolling loops and 3.17x by replicating compute units. It should be noted that both of these optimizations aims at increasing kernel throughput, however, by unrolling loops a higher performance is achieved with less penalty in circuit area when compared to compute unit replication. These results supply evidence to support a claim that, for kernels that have loops, the loop unrolling optimization should be favored over replication. Nonetheless, for kernels that do not have loops, replicating compute units should provide a good enough speedup. We also observed that the performance gain was much lower (1.30x) and the area usage was much higher when combining both optimizations. Thus the results also lead to a caution that over-optimizing code can cause side effects like excessive memory access, causing performance to decrease when inserting more optimizations. Finally, no improvement was observed when using the restrict parameter optimization, contradicting the Intel® optimization guide. A more thorough experimental analysis is needed to characterize program performance when applying this optimization.

5. Conclusion

The challenge of increasing performance-per-watt in high-performance computing systems led engineers to adopt reconfigurable accelerators. However, most programmers are unfamiliar with this class of devices, therefore the task of optimizing applications for FPGAs is not trivial. One of the reasons for this difficulty is the lack of performance portability in OpenCL kernels. In other words, a kernel written for GPUs may show low performance when executed on FPGAs. In that sense, this work evaluates the performance of code transformations for OpenCL kernels targeted at FPGAs. This work provides the first step toward a full performance characterization for FPGAs that could later be used to build compiler-based tools for automatic optimization.

Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful comments. This work was supported by FAPESP (Grants 2017/09065-9, 2018/08116-1, 2018/15519-5) and PROCAD (Grant 88887.124141/2014-00).

References

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition.

Amazon Web Services (2019). Amazon EC2 F1 Instances. [Online]. Available: `https://aws.amazon.com/ec2/instance-types/f1/`. (Acessed Feb. 11, 2019).

Bacon, D., Rabbah, R., and Shukla, S. (2013). FPGA programming for the masses. *Queue*, 11(2):40:40–40:52.

Barr, J. (2017). EC2 F1 Instances with FPGAs – Now Generally Available. [Online]. Available: `https://aws.amazon.com/blogs/aws/ec2-f1-instances-with-fpgas-now-generally-available/`. (Acessed Feb. 11, 2019).

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54.

Dennard, R. H., Gaensslen, F. H., Rideout, V. L., Bassous, E., and LeBlanc, A. R. (1974). Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268.

Fowers, J., Ovtcharov, K., Papamichael, M., Massengill, T., Liu, M., Lo, D., Alkalay, S., Haselman, M., Adams, L., Ghandi, M., Heil, S., Patel, P., Sapek, A., Weisz, G., Woods, L., Lanka, S., Reinhardt, S. K., Caulfield, A. M., Chung, E. S., and Burger, D. (2018). A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 1–14, Piscataway, NJ, USA. IEEE Press.

Hennessy, J. and Patterson, D. (2019a). *Computer architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Cambridge, MA.

Hennessy, J. L. and Patterson, D. A. (2019b). A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60.

Intel Corporation (2015). *Intel(R) Stratix(R) V Device Overview*.

Intel Corporation (2018). *Intel(R) FPGA SDK for OpenCL(TM) Pro Edition: Best Practices Guide*.

Khronos Group (2019). Open Computing Language (OpenCL). [Online]. Available: `https://www.khronos.org/opencl/`. (Acessed Feb. 15, 2019).

Lambert, J., Lee, S., Kim, J., Vetter, J. S., and Malony, A. D. (2018). Directive-based, high-level programming and optimizations for high-performance computing with FPGAs. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, pages 160–171, New York, NY, USA. ACM.

Lee, S., Kim, J., and Vetter, J. S. (2016). OpenACC to FPGA: A framework for directive-based high-performance reconfigurable computing. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 544–554.

Lee, S. and Vetter, J. S. (2014). Openarc: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 115–120, New York, NY, USA. ACM.

Lloyd, T., Chikin, A., Ochoa, E., Ali, K., and Amaral, J. N. (2017). A case for better integration of host and target compilation when using OpenCL for FPGAs. In *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers*, pages 1–9.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8):56–59.

OpenACC (2019). OpenACC: Directives for Accelerators. [Online]. Available: `https://www.openacc.org/`. (Acessed Feb. 15, 2019).

Zohouri, H. R., Maruyama, N., Smith, A., Matsuda, M., and Matsuoka, S. (2016). Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 409–420.