On Coding Techniques for Targeting FPGAs via OpenCL

Nuno PAULINO [a,1], Luís REIS [a] João M.P. CARDOSO [a]
[a] *INESC TEC and Faculty of Engineering of the University of Porto, Porto, Portugal*
Emails: {nmcp, luis.cubal, jmpc}@fe.up.pt

Abstract. Software developers have always found it difficult to adopt Field-Programmable-Gate-Arrays (FPGAs) as parallel computing targets. Recent advances in HLS tools aim to bridge this gap by abstracting the hardware design and FPGA programming effort via a standard OpenCL interface and execution model. However, OpenCL is a low-level programming language and requires that developers master the target architecture in order to achieve efficient results. Thus, efforts addressing the generation of OpenCL from high-level languages are of paramount importance to increase design productivity and to help software developers.

Existing approaches bridge this by translating MATLAB/Octave code into *C*, or similar languages, in order to achieve a higher performance by efficient compilation for the target hardware. One example is the MATISSE source-to-source compiler, which translates MATLAB code into standard-compliant C and/or OpenCL code.

In this paper, we analyse the viability of combining both flows so that sections of MATLAB code can be translated to specialized hardware with a small amount of effort, and test a few code optimizations and their effect on performance. We present preliminary results relative to execution times, and resource and power consumption, for two OpenCL kernels generated by MATISSE, and manual optimizations of each kernel based on different coding techniques.

Keywords. FPGA, OpenCL, High-Level Synthesis, MATLAB, source-to-source compilers

1. Introduction

In the scope of parallel programming and parallel-oriented architectures, the most commonly used devices are either GPUs or multi-core processors. The appeal of these devices is the high-level abstraction from the hardware itself by mature flows and programming models, e.g. OpenCL [Khr16], for GPUs, and OpenMP for multi-core CPUs.

In contrast, Field-Programmable Gate Arrays (FPGAs) allow for fine-grain configuration of their circuitry in order to implement specialized hardware. Thus, beyond being prototyping devices, FPGAs can be used as application-specific accelerators in embedded or High-Performance Computing systems. Unlike the homogeneous hardware archi-

[1]This is work has been partially supported by the TEC4Growth project, "NORTE-01-0145-FEDER-000020", financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF). Reis acknowledges the support of the Portuguese Science Foundation (FCT), through the grant SFRH/BPD/118211/2016.

tecture of GPUs, which is oriented towards SIMT or SPMD, FPGAs can host multiple programming models and several parallel instances of specialized pipelined circuits.

However, FPGAs are unfamiliar platforms to software programmers, due to the harsh deviation from familiar flows. Significant hardware/software co-design effort is required, including hardware design knowledge, understanding the target device, modifying the software application, and integration steps. This is a lengthy and error-prone process, unlike the fast iterative design provided by existing OpenCL platforms.

To address these issues, FPGA vendors have developed design flows with a high-level of abstraction. For instance, Xilinx relies on their Vivado HLS tool [Xil17b]. Either *C* or OpenCL code can be fed into Xilinx's SDAccel [Xil17a] design environment, which generates all FPGA programming files, making both the process of programming the FPGA, as well as dispatching execution onto it, completely transparent to the programer.

This paper explores the impact of some OpenCL kernel coding styles when targeting an FPGA device via the SDAccel toolflow, in the context of a MATLAB to OpenCL compiler. Namely, the kernels used were generated with MATISSE, which compiles MATLAB code to standard-compliant OpenCL or C code [BRC15]. Different modifications to the MATISSE-generated code are explored, in order to increase kernel execution performance, providing an indication on how to improve MATISSE for FPGA targets.

Similar approaches include [KScF16], which focuses on designer guided hints, expressed in a visual programming model, to generate synthesizable OpenCL. The MATCH approach [B$^+$00] adresses MATLAB to FPGA compilation, via the generation of HDL from MATLAB code. The many other existing works on HLS tools (see [N$^+$16,CDW10] for an overview) mostly focus on HLS from C. To the best of our knowledge, there is no recent research work focused on MATLAB code for HLS onto FPGAs.

This paper is organized as follows. Section 2 describes the MATISSE compiler and Xilinx's SDAccel flow. Section 3 describes the experimental setup and the two code kernels used to evaluate the effects of different coding techniques. Section 6 provides hints regarding OpenCL code for FPGA targets, and Section 7 concludes the paper.

2. From MATLAB code to FPGAs via OpenCL

MATISSE [BRC15] is a compiler able to translate a subset[2] of MATLAB code into standard-compliant OpenCL code and C code, as shown in Figure 1. The generated code can target either CPU or GPU devices. In this paper, MATISSE is the tool used to generate the OpenCL code to execute on our FPGA target. MATISSE not only generates the OpenCL kernel code, but also the host-side code to setup kernel execution. It is also capable of tuning its code generation depending on the target, via designer guided hints. Determining suitable optimizations for FPGA targets is the focus of this work.

Our previous experiments have shown that MATISSE-generated OpenCL achieves a geometric mean speedup of 11.0$\times$ on a R9 Radeon 280X over sequential *C* code on an AMD A10-7850K CPU, for a set of 7 kernel functions. Note however that the primary purpose is to provide a tool to quickly migrate MATLAB code into functional OpenCL code, thus reducing developer effort. The MATISSE tool and documentation is available in the SPeCS Lab homepage [SPe17].

[2]The subset of MATLAB currently supported by MATISSE does allow MATISSE to compile an extensive set of MATLAB code.

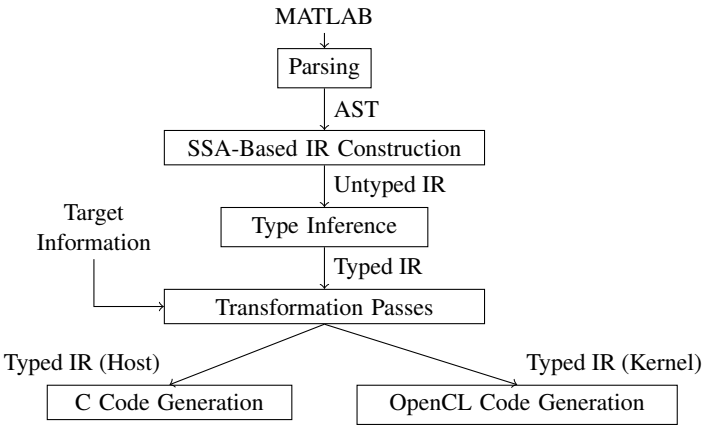

Figure 1. MATISSE code transformation flow (adapted from [BRC15])

The SDAccel flow [Xil17a] is part of the most recent release of Xilinx'x toolkit and framework, SDx. The compilation flow allows for the designer to choose a number of kernels to accelerate, and how many Compute Units (CUs) to implement for each. The compiler then automatically generates the corresponding Hardware Description Language (HDL) and bitstreams, using Vivado HLS as a backend. SDAccel's relies on its own OpenCL compiler, *xocc* [Xil17a], which supports FPGA-specific compilation hints.

At a low level, Dynamic Partial Reconfiguration is used to support multiple kernels by time-multiplexing. The static side of the FPGA circuitry hosts the interface between host machine and FPGA, and the reconfigurable areas host the kernel circuits. All interface related circuitry between the generated CUs and host processor is also generated, without manual software modification or system integration.

In short, this flow is mostly aimed at software designers who want to exploit the customization potential of FPGAs, without having to concern themselves with manual hardware design or in-depth knowledge of FPGAs. This paper shows, besides exploring some coding techniques, that MATISSE can be used in conjunction with SDAccel, exposing this specialization potential to existing MATLAB code and to MATLAB programmers.

3. Experimental Setup

The experimental setup consists of one desktop machine running the host code, which sets up the kernel for execution on the FPGA and retrieves execution times given standard OpenCL profiling functions. The target board used is an Alpha Data ADM-PCIE-KU3 [Dat17], a PCI-Express based board with a Kintex-6 XCKU060 FPGA. The OS used was Ubuntu 16.04 64-bit, and the version of SDAccel was 2016.3. The desktop (i.e., host) machine has an *Intel Core i7-6700K CPU @ 4.00GHz*, and 32GB of DDR4 RAM.

We do not focus on the performance of the host machine, as we measure only the kernel execution time on the FPGA, so as to compare the performance of different code versions. To test a few coding styles, we used two kernels, explained in the next section, which were generated by MATISSE from MATLAB code; the different coding techniques for each kernel are shown in Tables 1 and 2. Although simple, these kernels allow us to understand, relatively easily, the impact of the code style and of some decisions.

Table 1. Description of RGB-to-YUV kernel versions[3]

Kernel	Description
Baseline (BL)	Translation of the MATLAB RGB kernel by MATISSE
Opt1	BL + read, compute, and write stages placed into directives to pipeline workitems
Opt2	BL + 2-element vectorization
Opt3	BL + 4-element vectorization
Opt4	BL + separate innermost loops with pipelining hints for read, compute, and write stages
Opt5	Opt4 + local memory partitioning and OpenCL unroll hints on innermost loops

Table 2. Description of Monte-Carlo kernel versions[3]

Kernel	Description
Baseline (BL)	Translation of the MATLAB Monte-Carlo kernel by MATISSE
A1	BL + *always_inline* attributed applied to all functions
A2	BL + Manually inlined all functions
A3	A2 + larger workgroup size (1024)
T1	A3 + calls to *cos* replaced by 1024-entry table
T2	BL + larger workgroup size (1024) + *cos* replaced by 1024-entry table + *log* and *exp* replaced by Taylor series (order 10)
T3	BL + larger workgroup size (1024) + *cos*, *exp*, and *log* replaced by tables
N1	BL + *cos*, *exp*, and *log* replaced by *native_* equivalents
N2	BL + *cos*, *exp*, and *log* replaced by *half_* equivalents
V1	BL + 4-element Vectorization

We first executed the MATISSE-generated OpenCL on the board, and then a number of modified versions of the same kernel. Each version is not necessarily an incremental version of the previous, although some do rely on multiple modifications. For each kernel version, we tested the performance by varying the number of CUs used. For each case we retreived the execution times, resource usage, and power consumption.

For all cases, *xocc* is called using -O2 as optimization level and with multi-threading enabled, launching up to 4 concurrent compilation jobs. The *xocc* compiler defines its own optimization level flags to control the hardware synthesis, the mapping, and placement and routing processes. The chosen level, according to the documentation, attempts to balance hardware implementation efficiency with the required compilation time.

3.1. Use Case 1 - Color Space Conversion Kernel

The first kernel used to test the coding strategies is an RGB-to-YUV color space conversion[3]. The kernel processes three 2D arrays of 8-bit data, which represent the RGB components of each pixel in an image. The nested loop converts this to the YUV color space, and saves the output in 2D arrays of the same dimension as the input. Therefore, every iteration of this nested loop could be computed entirely in parallel, assuming appropriate hardware support. The feasibilty of this depends on the total number of iterations (i.e. $N \times M$), and therefore the hardware required. Realistically, an OpenCL runtime would divide the total workload into as many CUs as possible, maximizing the number of parallel iterations. Alternatively, a single CU capable of agressive loop pipelining could lead to good results. This later strategy is only viable on an FPGA target, due to the possibility of custom circuit design. Coupling this with the potential of parallelizing the loop, i.e., instantiating several CUs, increases the performance improvement potential.

[3] All code can be found in `http://specs.fe.up.pt/rgb_and_montecarlo_opencl_src.zip`

3.2. Use Case 2 - Monte-Carlo Kernel

The second kernel is an adapted Monte-Carlo simulation example, available at the MathWorks code repository [Mat17]. Most inputs are floating-point numbers, and all are scalar. Likewise, each kernel call produces two scalars. That is, there are two output values per workitem, which cannot be produced in parallel due to a direct data dependency.

Each call to the kernel receives a seed value, and values which control the number of iterations of stock pricing and asian option calculation loops. The kernel calls two sequential functions, each containing a main loop (with a variable iteration count) with intra-iteration data dependencies. Within these loops are calls to the transcendental math functions *cos*, *log*, and *exp*. This kernel has a more unpredictable behaviour than the previous, because input variables control the number of iterations, the order of magnitude of the results and intermediate calculations, which may also vary the execution time of the transcendental functions, depending on their implementation.

Unlike the previous example, this kernel is not so easily expressed as a single pipelineable loop, since there are other inner loops which cannot be easily unrolled, as well as math function calls. In this instance, it seems reasonable to attempt to apply a more standard OpenCL-type approach, which is to express the outer-most loop level in terms of workgroup size, and to attempt to use explicit vectorization combined with multiple instances of an efficient CU design. The advantage of this kernel is the small amount of data transfers required between host and FPGA.

4. Results - RGB-to-YUV Conversion Kernel

This section analyzes the execution time, resource requirements, power and energy consumption of 24 implementations of the color conversion kernel, as well as required compilation times. The results were measured from executions on the target board.

All cases process the same amout of data: a 128×128 matrix of elements of type *uchar*. Depending on the kernel, the global and local work group sizes vary. For the first two cases (MATISSE, and *Opt1*), the global workgroup size is $(128, d2, 1)$, for the following two it is $(64, d2, 1)$ and $(32, d2, 1)$, respectively. For the last two, it is $(1, d2, 1)$. For all cases $d2 = \#workitems/128$. Each local workgroup size is given by setting $d2 = 1$.

4.1. Execution Times

Figure 2 shows the execution time in milliseconds. The most influential parameter is the number of CUs, regardless of the kernel version. Between kernels the optimization that leads to the best improvements is explicit vectorization, which is applied in *Opt2* and *Opt3*. When comparing implementations with the same number of CUs, the remaining kernel versions show negligible performance variations.

Increasing the number of CUs improves the performance, to a point of diminishing returns. The geometric mean speedup for cases with 2 CUs over the equivalent cases with 1 CU is $1.94\times$. This decreases to $1.78\times$ when comparing the 4 CUs implementations with 2 CUs; and finally, using 8 CUs speeds up execution by $1.28\times$ over using 4 CUs.

These gains must be weighed against the respective compilation times, which are shown in Figure 3. We find that the increase in compilation time is $1.07\times$ between 2

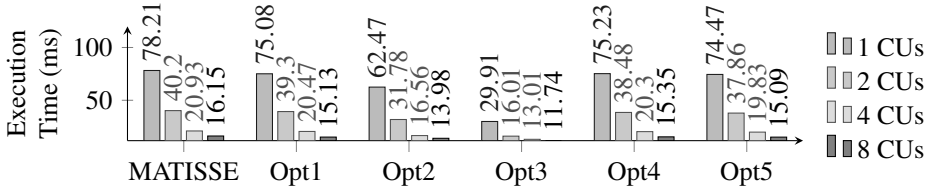

Figure 2. Kernel Execution Times for an Input Matrix of 128x128 Elements

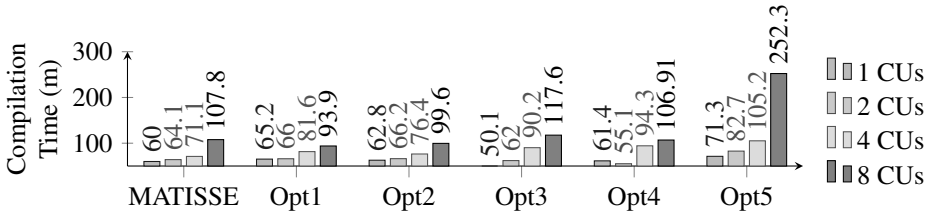

Figure 3. Kernel Compilation Times for RGB-to-YUV Kernel

and 1 CUs, 1.32× between 4 and 2 CUs, and 1.47× between 8 and 4 CUs. That is, the compilation time increases more than the improvement in execution time when comparing 8 CUs to 4. Note that these values hold for this particular kernel, and for the FPGA in question. However, the general conclusion is applicable to other cases: increasing the number of CUs over a certain point is too costly and provides little to no benefit.

The number of CUs however is not the only factor. The global and local workgroup sizes also affect the speedups. However, modifying the required workgroup size of a kernel requires re-compilation of the kernel. As a quick test, we varied the global workgroup size, and the size of the input matrix to 2048×128, leaving the local sizes per kernel unchanged. The decrease in execution time between 1 and 2, and 2 and 4 CUs, remains the same as before, but the implementations with 8 CUs now provide a greater improvement over their equivalents with 4 CUs, decreasing the execution time by 1.43× on average.

4.2. Compilation Times and Resource Requirements

Regarding compilation times, the behavior is as expected when increasing the number of CUs: a non-linear increase in required time. As the amount of logic to implement increases past a certain point, the total synthesis, placement and routing time increases sharply. This happens noticeably for the *Opt5* implementation with 8 CUs, since the cyclic memory partitioning uses physical Block RAMs (BRAMs) on the device. Because BRAM placement is fixed, routing becomes more difficult as more BRAMs are used.

The resource requirements per kernel, for the implementations with one CU, are shown in Table 3. Comparing these cases alone, we find that most require approximately the same amount of Flip-Flops (FFs) and Lookup Tables (LUTs), except for *Opt5*, which requires the most out of any type of FPGA resource. We confirm here that this case requires the most BRAMs. Interestingly, *Opt2* requires less Digital Signal Processor (DSP) blocks than the baseline, despite speedup increase of 1.26×.

For these evaluations, increasing the number of CUs increases the required resources near-linearly. Doubling the number of CUs causes the number of required DSPs and

Table 3. Resource Requirements of Implementations of the RGB-to-YUV kernel with 1 Compute Unit (% are relative to the total amount of FPGA resources)

Kernel	FFs	LUTs	DSPs	BRAMs
Baseline	16154 (2.44 %)	10836 (3.28 %)	88 (3.19 %)	**1** (0.09 %)
Opt1	15569 (2.35 %)	**10327** (3.12 %)	63 (2.28 %)	6 (0.56 %)
Opt2	**15462** (2.34 %)	10451 (3.16 %)	**60** (2.17 %)	**1** (0.09 %)
Opt3	19109 (2.89 %)	12753 (3.86 %)	100 (3.62 %)	**1** (0.09 %)
Opt4	16200 (2.45 %)	10436 (3.16 %)	69 (2.50 %)	5 (0.46 %)
Opt5	26467 (4.00 %)	18200 (5.50 %)	198 (7.17 %)	19 (1.76 %)

BRAMs to increase by the same factor, as these are physical hardware blocks. In terms of FFs and LUTs, the increase is not as linear. Placement and routing processes remove hierarchy, so these resources can be shared between modules, and may also be replicated to increase the maximum clock frequency. However, the trend is similar for all cases. Using 2, 4, and 8 CUs requires 2.07×, 3.87×, and 7.52× the number of FFs that one CU requires; for LUTs, the respective increases are 2.11×, 4.15×, and 8.19×. In terms of clock frequency, the number of CUs does not cause a decrease relative to implementing a single instance, according to the reports generated by *xocc*. For all cases, the operating frequency of each CU is 274 MHz, with the exception of *Opt2*, for which it is 244 MHz.

4.3. Power and Energy Consumption

Analyzing the power consumption according to post-route reports, we find that implementations with 1 CUs require an average of 9.34 W, 2 CUs require 9.75 W, 4 CUs require 10.54 W, and 8 CUs require 12.14 W. This indicates a non-linear increase in power consumption. The implementations of *Opt2* require the least power, relative to other implementations with the same number of CUs. Finally, *Opt5* requires the most power, for all of its implementations, likely due to the higher number of BRAMs and DSPs.

However, since the execution time decreases much more drastically, relative to the increase in power consumption, the end result is a decrease in total energy consumed. Despite *Opt2* resulting in the lowest power consumption, it is *Opt3* that results in the least energy consumed, regardless of the number of instantiated cores. This holds when processing an input matrix of 1680×1050, where *Opt3* requires 3.54 J with 8 cores, versus 8.38 J for the equivalent *Opt2* case. The baseline for this case requires 11.18 J.

5. Results - Monte-Carlo Kernel

This section analyzes 37 implementations of the Monte-Carlo kernel. The variations shown in Table 2 are implemented with 1, 2, 3, or 4 CUs, with the exception of *V1*, which is only implemented with 1 CU. Like the previous section, we retrieve execution times from actual on-board implementations, and study power and resource consumption.

The coding styles applied to this kernel can be divived into 3 categories. Cases *A1*, *A2*, and *A3* focus on manual or hint-based inlining of functions; *T1*, *T2*, and *T3* replace the transcendental math functions with approximations for faster execution and less resource consumption; *N1* and *N2* replace these functions with lower precision implementations; *V1* is a simple vectorization of the baseline. All kernel versions process 131,072 workitems, with a one dimentional workgroup size. The local workgroup size is 128 for the baseline, *A1*, *A2*, *N1*, and *N2*; 1204 for *A3*, *T1*, *T2*, and *T3*; and 32 for *V1*.

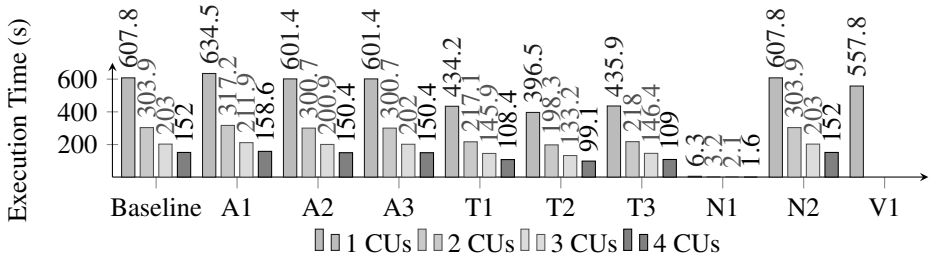

Figure 4. Monte-Carlo kernel execution times for 131,072 workitems

5.1. Execution Times

The execution times for all kernels are shown in Figure 4, for different numbers of CUs. For nearly all cases, except *N1*, there is marginal variation in performance, relative to the baseline, when comparing cases with the same number of CUs. Only *T1*, *T2* and *T3* achieve noticeable improvements, and *N1* is clearly the optimal solution. The execution time decreases linearly with the number of CUs, at least for the number of CUs tested. The following paragraphs briefly analyse each type of optimization.

A1, A2, and A3 Inlining the called functions, either via hints or manually, yielded virtually no change, and even a minor increase for *A1*. Modifying the workgroup size between *A2* and *A3* (both manually inlined) resulted in no change. Unlike the previous kernel, increasing the local workgroup size did not affect the performance noticeably. We suspect this is due to: a lack of large amounts of data to transfer to the FPGA, and a shorter execution time per workitem, relative to a single workitem of the RGB-to-YUV kernel.

T1, T2, and T3 For these cases there is a noticable improvement in execution time. The speedup for each case is the same regardless of the number of CUs: 1.40× for *T1*, 1.53× for *T2*, and 1.39× for *T3*. Resorting to Taylor series approximations for *log* and *exp* leads to better performance over lookup tables, for the order of approximation used.

Despite the speedups, the approximation techniques used introduce precision errors (i.e., insufficient table entries or low approximation order), or produce correct results only in certain conditions. For instance, in *T1* the *cos* function is replaced with a table. Since an unbound input argument (i.e., angle), can be reduced to a value between 0 and 2π, any precision errors are due to the number of entries in the table.

In contrast, for *log* and *exp*, both a table based approach and a Taylor series can only approximate the target functions within a limited range of input values, around a central point. In this kernel, the inputs to these functions are dependant on the kernel call inputs. In other words, it is not a generalized implementation, although it is still usable in cases where some input parameters are fixed. Another strategy would be to implement several hardware versions, each tuned to a calling context.

V1 Despite noticable performance improvements for the RGB-to-YUV kernel when relying on 4-element vectorization, the speedup for *V1* is only of 1.09× over the MATISSE baseline. This is not unexpected since the previous tests (i.e., *T1* to *T3*) already indicated that the bottleneck was due to the transcendental functions. Also, *V1* was not tested with more CUs, since the compilation failed with 2 or more CUs.

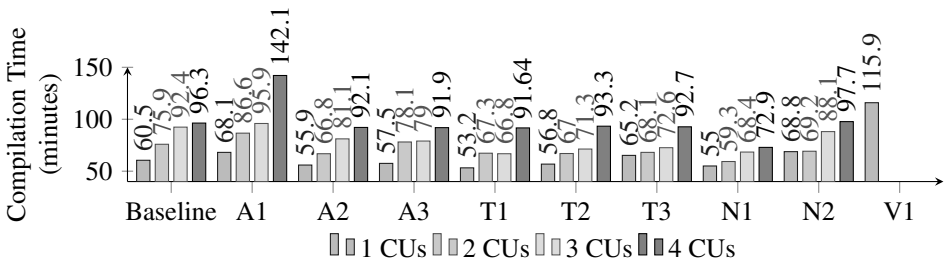

Figure 5. Kernel Compilation Times for Monte-Carlo Kernel

Table 4. Resource Requirements of Implementations of the Monte-Carlo kernel with 1 Compute Unit; (% are relative to the total amount of FPGA resources)

Kernel	FFs	LUTs	DSPs	BRAMs
Baseline	27741 (4.19 %)	29623 (8.96 %)	142 (5.14 %)	7 (0.65 %)
A1	31380 (4.74 %)	35338 (10.69 %)	233 (8.44 %)	7 (0.65 %)
A2	26639 (4.03 %)	28991 (8.77 %)	128 (4.64 %)	7 (0.65 %)
A3	26645 (4.03 %)	29067 (8.79 %)	128 (4.64 %)	7 (0.65 %)
T1	24207 (3.66 %)	24316 (7.35 %)	**38** (1.38 %)	**5** (0.46 %)
T2	28240 (4.27 %)	22984 (6.95 %)	57 (2.07 %)	5 (0.46 %)
T3	25113 (3.80 %)	25110 (7.59 %)	52 (1.88 %)	7 (0.65 %)
N1	**21096** (3.19 %)	**16580** (5.01 %)	60 (2.17 %)	0 (N/A)
N2	27741 (4.19 %)	29623 (8.96 %)	142 (5.14 %)	7 (0.65 %)
V1	79039 (11.95 %)	107974 (32.65 %)	852 (30.87 %)	9 (0.83 %)

N1 and N2 The transcendental functions are the performance bottleneck, as *N1* achieves a speedup of 96.22× by relying on SDAccel's built-in *native_* implementations of these functions. This holds for all numbers of CUs tested. The half-precision implementations used in *N2* do not decrease execution time. Over an implementatiom with one CU of the baseline code, a 4 CU implementation of *N1* achieves a speedup of 384×, which is still 21× slower than the baseline code running on the CPU, and 29× slower than the *N1* version of the code running on the CPU.

5.2. Compilation Times and Resources

Figure 5 shows the compilation times for all cases. Despite the increased complexity of the Monte-Carlo kernel, relative to the RGB-to-YUV kernel, compilation times do not increase dramatically. For the same number of cores, the average time is similar. For 2, 3, and 4 CUs, *A1* requires the most time; for one CU, *N2* just barely requires more time than *A1*. As expected, the compilation times correlate with the amount of logic required.

Table 4 shows the resource requirements for the implementations with one CU. Again, the amount of resources required scales linearly with the number of CUs for all code versions. In general, there is little variation in the number of required FFs or LUTs, with the exception of *V1*, which is discussed shortly. The number of required DSPs and BRAMs varies more noticeably, and *A1* requires the most LUTs, FFs, and DSPs.

Although the same code modification was used, the differences in resource requirements for *A1* and *A2* (and by extension *A3*), are noticable. This seems to indicate that a local context is created per function call, when resorting to the *inlining* attribute in *A1*,

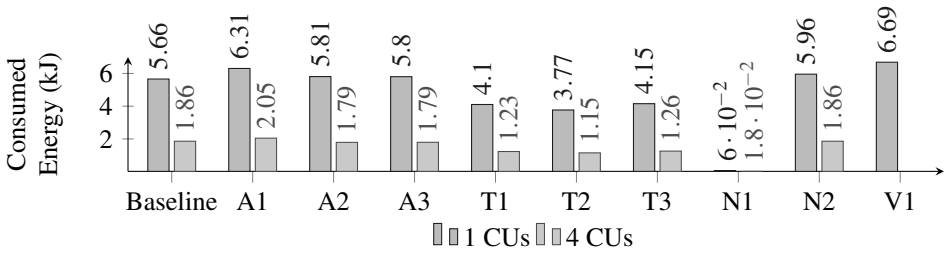

Figure 6. Energy consumption for all cases, when processing 131072 workitems

and that variables therein are replicated, and not shared. Since *A2* contains manually inlined function calls, we conclude that logic is reutilized when two calls are inlined in this way in the same parent. The result is a considerable difference in resources despite nearly the same performance: *A2* requires about 80 % of the LUTs and FFs that *A1* requires, and barely over half as many DSPs, for any number of CUs.

We can also observe the effect of resorting to pre-calculated tables for the transcendental functions, or computing them via approximations. Between *T2* and *T3*, the former requires more DSPs, since *exp* and *log* are computed by the standard function calls; *T3* requires more BRAMs, since the two functions are implemented as tables, although a greater decrease in the number of DSPs was expected. Curiously, it is for *T1* that the least amount of both DSPs and BRAMs are used (for a 1024-entry table).

When comparing implementations with the same number of CUs there is no significant variation in the amount of BRAMs required. The exception is *N1* which does not require BRAMs for one CU. For 4 CUs, only 2 BRAMs are needed by *N1*; 30 are needed by the baseline, *V1*, *A1*, *A2*, and *A3*; 22 by *T1* and *T2*; and finally, 28 by *T3*.

Implementing *V1*, with one CU, requires approximately the same resources as 4 CU implementations of most other versions. That is, it is similar to either increase the number of CUs, or to explicitly use vector data types. Since the SDAccel flow imposes a limit of 16 CUs, instantiating this maximum number while also using vector data types essentialy maximizes paralellism. The compilation time of *V1* with one CU is also comparable to the average compilation times of all other cases with 4 CUs: 115 min versus 98 min.

5.3. Power and Energy Consumption

The energy consumption for one and four CUs is shown in Figure 6. As expected, *N1* consumes the least energy, due to the significant decrease in execution time, but it also achieves the lowest power consumption for all cases. It requires 9.4 W, 10.0 W, 10.6 W, and 11.2 W for 1, 2, 3, and 4 CUs, respectively. In contrast, *A1* requires the most power, for any number of CUs. For 1 CU the power consumed is 9.87 W, 3 % above average; for 4 CUs it is 11.87 W, 8.7 % above average. These averages exclude *V1*, which consumes 11.99 W. This is the second highest power consumption amongst all cases, after the 4 CU implementation of *N2*, which consumes 12.24 W.

As with the RGB-to-YUV kernel, the power consumption increases only marginally with the number of CUs. However, since the execution time decreases significantly, it is preferable to instantiate more units if possible, for performance and energy benefits.

6. Coding Style Hints for OpenCL Targeting FPGAs

Given the results presented, and the experiments conducted, we list herein a number of observations which provide hints towards generation of OpenCL when targeting FPGAs.

- Explicit vectorization leads to performance improvements, which is similar to instantiating more CUs of a non-vectorized code version. However, instantiating more CUs replicates all the logic generated from the source code, whereas manual vectorization may aid in reutilization of local variables and arithmetic blocks.
- Maximum parallelism is achieved by instantiating the most CUs, up to 16, and employing explicit vectorization, up to 8 data; this requires both vectorizing the code and separating execution into workgroups.
- Use of the *native_* implementations for math functions is highly recommended, when the loss of precision is acceptable.
- A static code analysis of all calling contexts of a given kernel would aid in generating context-specific circuit implementations.
- We observed no effect on the operating frequency when increasing the number of CUs; the frequency is also nearly equal when comparing both tested kernels.
- Explicit function inlining leads to fewer required resources relative to automatic inlining, most likely due to reutilization of constants and intermediate results.
- Since SDAccel allows for global constants, a possible optimization would be to move all constants in function bodies to a global scope, avoiding logic duplication when relying on inlining attributes, thereby avoiding manual inlining efforts. Resorting to global variables compromises code portability however.
- Code with many function calls in inner loops, is difficult to target to an FPGA, since loop pipelining is difficult at the outer most loop level.

It is clear that to achieve the best implementations for specific requirements such as execution time, power and/or energy consumption, it is necessary to make code transformations to the OpenCL kernels, to tune some parameters, and to select the best possible OpenCL computing architecture (e.g., in terms of the number of CUs). While some of these aspects can be explored by using autotuning and design space exploration schemes, it is very important to have approaches tuning and searching the design space without requiring actual implementations, and thus avoiding the long overall compilation times.

Thus, a generator of OpenCL code targeting FPGAs must not only help developers to decide about the selection of the functions to be offloaded to the OpenCL computing FPGA platform, but also to generate target-aware OpenCL,and to suggest the OpenCL architecture instance to be implemented. This may require performance models, high-level estimators, developer expertize, and the use of best practices possibly specified as strategies, to guide the source to source compiler.

7. Conclusion

This paper evaluated the performance of Xilinx's OpenCL-based HLS flow for FPGAs, by relying on two code kernels. The kernel code was automatically generated from MATLAB code, using the MATISSE source-to-source compiler. We evaluated how performance, power/energy consumption, and resource usage varied, when modifing the code of the two kernels in several ways.

There are coding techniques which are easier to integrate into a source-to-source compiler like MATISSE, such as explicit vectorization or loop unrolling, while others require more developer intervention or application profile data, such as memory partitioning or rewritting a single kernel function into several inter-communicating kernels.

As expected, the most straightforward way to increase performance is to increase the number of Compute Units, until a certain point. However, depending on the code, this may entail a laborous redesign as to adapt the code to the workgroup model of OpenCL.

Some kernels, particularly larger kernels which are stream oriented, may benefit from agressive loop pipelining, which is, in addition to hardware customization, possibly the most notable difference of an FPGA versus GPUs or CPUs, when considering hardware acceleration. Depending on data dependencies between loop iterations, combining both methods is not mutually exclusive. In general, loop pipelining is better suited for very large kernels, where it would be more costly to increase performance by directly replicating hardware in order to implement parallelism at a *work-item* level.

Future work plans include to extend MATISSE to generate code with these FPGA-oriented aspects in mind, and also to analyse data transfer characteristics, both from/to the kernel code as well as between processing blocks of the kernel, so as to also automate the partitioning of a single MATLAB kernel into read/compute/write sections which can communicate efficiently with inter-kernel channels and on-chip memory.

References

[B+00] P. Banerjee et al. A MATLAB Compiler for Distributed, Heterogeneous, Reconfigurable Computing Systems. In *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 39–, Washington, DC, USA, 2000. IEEE Computer Society.

[BRC15] João Bispo, Luís Reis, and João M. P. Cardoso. C and OpenCL Generation from MATLAB. In *Proc. of the 30th Annual ACM Symp. on Applied Computing*, SAC '15, pages 1315–1320, 2015.

[CDW10] João M. P. Cardoso, Pedro C. Diniz, and Markus Weinhardt. Compiling for reconfigurable computing: A survey. *ACM Comput. Surv.*, 42(4):13:1–13:65, June 2010.

[Dat17] Alpha Data. ADM-PCIE-KU3 User Manual 1.11. `http://www.alpha-data.com/pdfs/adm-pcie-ku3usermanual.pdf`, 2017. Accessed: 20-03-2017.

[Khr16] Khronos OpenCL Working Group. The OpenCL C Specification, Version 2.0 - Revision 33. `https://www.khronos.org/registry/OpenCL/specs/opencl-2.0-openclc.pdf`, 2016. Accessed: 16-05-2016.

[KScF16] K. Krommydas, R. Sasanka, and W. c. Feng. Bridging the FPGA Programmability-Portability Gap via Automatic OpenCL Code Generation and Tuning. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 213–218, July 2016.

[Mat17] MathWorks. Using GPU ARRAYFUN for Monte-Carlo Simulations. `https://www.mathworks.com/examples/parallel-computing/mw/distcomp-ex56222285-using-gpu-arrayfun-for-monte-carlo-simulations`, 2017. Accessed: 20-03-2017.

[N+16] R. Nane et al. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, Oct 2016.

[SPe17] SPeCS. SPeCS - Special-Purpose Computing Systems, Languages and Tools. `https://sites.google.com/site/specsfeup/`, 2017. Accessed: 09-02-2017.

[Xil17a] Xilinx. SDAccel Development Environment. `https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`, 2017. Accessed: 05-06-2017.

[Xil17b] Xilinx. Vivado High-Level Synthesis. `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`, 2017. Accessed: 05-06-2017.