Impact of Compiler Phase Ordering When Targeting GPUs

Ricardo Nobre[1,2](✉), Luís Reis[1,2], and João M. P. Cardoso[1,2]

[1] Faculty of Engineering of the University of Porto, Porto, Portugal
{ricardo.nobre,luis.cubal}@fe.up.pt, jmpc@acm.org
[2] INESC TEC, Porto, Portugal

Abstract. Research in compiler pass phase ordering (i.e., selection of compiler analysis/transformation passes and their order of execution) has been mostly performed in the context of CPUs and, in a small number of cases, FPGAs. In this paper we present experiments regarding compiler pass phase ordering specialization of OpenCL kernels targeting NVIDIA GPUs using Clang/LLVM 3.9 and the libclc OpenCL library. More specifically, we analyze the impact of using specialized compiler phase orders on the performance of 15 PolyBench/GPU OpenCL benchmarks. In addition, we analyze the final NVIDIA PTX assembly code generated by the different compilation flows in order to identify the main reasons for the cases with significant performance improvements. Using specialized compiler phase orders, we were able to achieve performance improvements over the CUDA version and OpenCL compiled with the NVIDIA driver. Compared to CUDA, we were able to achieve geometric mean improvements of 1.54× (up to 5.48×). Compared to the OpenCL driver version, we were able to achieve geometric mean improvements of 1.65× (up to 5.70×).

Keywords: GPU · Phase ordering · Optimization

1 Introduction

High Performance Computing (HPC) can offer Petaflops of performance by relying on increasingly more heterogeneous systems, such as the combination of Central Processing Units (CPUs) with accelerators in the form of Graphics Processing Units (GPUs) programmed with languages such as OpenCL [1] or CUDA [2]. Heterogeneous systems are widespread as a way to achieve energy efficiency and/or performance levels that are not achievable by a single device/architecture (e.g., matrix multiplication is much faster on GPUs than on CPUs for the same power/energy budget [3]). These accelerators offer a large number of specialized cores that the CPUs can use to offload computation that exhibits data-parallelism and often other types of parallelism as well (e.g., task-level parallelism). This adds an extra layer of complexity if one wants to target these systems efficiently, which in the case of HPC systems such as supercomputers is of

© Springer International Publishing AG, part of Springer Nature 2018
D. B. Heras and L. Bougé (Eds.): Euro-Par 2017 Workshops, LNCS 10659, pp. 427–438, 2018.
https://doi.org/10.1007/978-3-319-75178-8_35

utmost importance. An inefficient use of the hardware is amplified by the magnitude of such systems (hundreds/thousands of CPU cores and accelerators), with increasing utilization/power bill and/or cooling challenges as a consequence. In order to efficiently utilize the hardware resources, programmers need advanced compilers and they also need high levels of expertise. The programmer(s) and the compiler(s) have to be able to target different computing devices (CPU, GPU, and/or FPGA) and/or architectures (e.g., system with ARM or x86 CPUs) in a manner that achieves suitable results for certain metrics, such as execution time and energy efficiency.

Compiler users tend to rely on the standard compiler optimization levels, typically represented by flags such as GCC's `-O2` or `-O3`. These flags represent fixed sequences of analysis and transformation compiler passes, also referred to as compiler phase orders. Programs compiled with these flags tend to outperform the unoptimized equivalent. However, there are often other assembly/binary representations of the source application in the solution space with higher performance than the ones achieved through the use of the standard optimization levels [4–8]. However, we can often achieve further performance, energy or power improvements by using specialized optimizing compiler sequences. Domains such as embedded systems or HPC tend to prioritize metrics such as energy efficiency that typically receive less attention from the compiler developers, so these domains benefit further from these specialized sequences [9].

Ideally, the standard compiler optimization levels would already correspond to the use of the best compiler phase selection/order for a given metric. However, there appears to be no single best phase order that applies to all programs. This is caused by the complex interactions between compiler passes. Some compiler passes negatively or positively interact with other compiler pases, resulting in the creation/destruction of optimization opportunities when executing the latter [11]. As such, a customized approach that produces different phase selections/orders for different functions/programs can lead to better performance.

Heterogeneous systems typically include a number of sub-devices with substantial differences. For this reason, different optimization strategies are needed for each computing component. With phase ordering, we can achieve closer-to-optimal optimization for these sub-devices, by specifying custom compiler sequences for each of them. This approach is orthogonal to other optimization strategies. For instance, it does not interfere with user and hardware optimizations. The use of compiler phase order specialization can reduce engineering costs. In a number of cases the same source code can be used when targeting architecturally different computing devices and/or different metrics through the use of different compiler phase orders. This reduces or mitigates the need to develop and maintain multiple versions of the same function/application.

The contributions of this paper are the following:

1. Compare performance between OpenCL and CUDA kernels implementing the same freely available and representative benchmarks (PolyBench/GPU) using recent NVIDIA drivers and CUDA toolchain, on an NVIDIA GPU with an up-to-date architecture (NVIDIA Pascal).

2. Assess the performance improvement that can be achieved using compiler pass phase ordering specialization with LLVM 3.9, in comparison with both use of that same LLVM compiler version without the use of phase ordering specialization and in comparison with the default OpenCL and CUDA kernel compilation strategies to NVIDIA GPUs.
3. Explain why the versions produced by phase selection/ordering specialization outperform the remaining ones, by analyzing the generated NVIDIA PTX assembly. We compare the specialized versions with CUDA's NVCC and OpenCL LLVM outputs.

Additionally, to the best of our knowledge this is the first work to present results of compiler pass phase ordering specialization targeting GPUs and considering OpenCL kernels.

The rest of this paper is organized as follows. Section 2 describes the methodology for the experiments presented in this paper. Section 3 presents the experimental results. Final remarks about the presented work and ongoing work are presented in Sect. 4.

2 Experimental Setup

We extended our compiler phase ordering Design Space Exploration (DSE) system [8] to support exploring compiler sequences targeting NVIDIA GPUs using Clang/LLVM and the libclc OpenCL library.

We used a workstation with an Intel Xeon E5-1650 v4 CPU, running at 3.6 GHz (4.0 GHz Turbo) and 64 GB of Quad-channel ECC DDR4 at 2133 MHz. For the experiments we relied on Ubuntu 16.04 64-bit with the NVIDIA CUDA 8.0 toolchain (released in Sept. 28, 2016) and the NVIDIA 378.13 Linux Display Driver (released in Feb. 14, 2017).

The GPU used for the experiments is a variant of the NVIDIA GP104 GPU in the form of an EVGA NVIDIA GeForce GTX 1070 graphics card (08G-P4-6276-KR) with a 1607 MHz/1797 MHz base/boost graphics clock and 8 GB of 256 bit GDDR5 memory with a transfer rate of 8008 MHz (256.3 GB/s memory bandwidth). The graphics card is connected to a PCI-Express 3.0 16x interface.

The GPU is set to persistence mode with the command `nvidia-smi -i <target gpu> -pm ENABLE`. This forces the kernel mode driver to keep the GPU initialized at all instances, avoiding the overhead caused by triggering GPU initialization at application start. The preferred performance mode is set to *Prefer Maximum Performance* under the *PowerMizer settings* tab in the *NVIDIA X Server Settings*, in order to reduce the occurrence of extreme GPU and memory frequency variation during execution of the GPU kernels.

In order to reduce DSE overhead, and given the fact that we found experimentally that multiple executions of the same compiled kernel on the GTX1070 GPU had a small standard deviation in relation to registered wall time, each generated code is only tested a single time during DSE. Only in a final phase on the DSE process are the top solutions executed 30 times and averaged in order to

select a single compiler phase order. All execution time metrics reported (baseline CUDA/OpenCL and OpenCL optimized with phase ordering) in this paper correspond to the average over 30 executions.

2.1 Kernels and Objective Metric

In this paper we use the Polybench/GPU benchmark suite [10] kernels to assess the potential for improvement with phase ordering when targeting NVIDIA GPUs. We selected this particular benchmark as it is freely available and thus contributes to making the results presented in this paper reproducible.

We modified the benchmarks to ensure that the CUDA and OpenCL versions use the same floating-point precision. For instance, the OpenCL implementation of the original `MVT` kernel uses double floating point precision, while the CUDA implementation uses single precision. We performed the minimum of changes to ensure a fair comparison.

Polybench/GPU is a collection of codes implemented for GPUs using CUDA, OpenCL, and HMPP. This benchmark suite includes kernels from 15 benchmarks from different domains which represent computations that would be performed on GPUs in the context of HPC, including convolution kernels (`2DCONV`, `3DCONV`), linear algebra (`2MM`, `3MM`, `ATAX`, `BICG`, `GEMM`, `GESUMMV`, `GRAMSCH`, `MVT`, `SYR2K`, `SYRK`), datamining (`CORR`, `COVAR`), and stencil computations (`FDTD-2D`).

For our experiments we use both the CUDA and the OpenCL implementations available for each PolyBench/GPU benchmark. We rely on the default dataset shape so that reproducibility of our results (e.g., performance improvement using the specialized phase orders presented in this paper) is more straightforward.

2.2 Compilation and Execution Flow with Specialized Phase Ordering

We use Clang compiler's OpenCL frontend with the libclc library to generate an LLVM IR representation of a given input OpenCL kernel. The libclc library is an open source library with support for AMDGCN and NVPTX targets that implements functions as specified in OpenCL 1.1.

Then, we use the LLVM Optimizer tool (`opt`) to optimize the IR using a specific optimization strategy represented by a compiler phase order, and we link this optimized IR with the libclc OpenCL functions for our target using `llvm-link`. Finally, using Clang, we generate the NVIDIA PTX representation of the kernel from the LLVM bytecode resulting from the previous step, using the `nvptx64-nvidia-nvcl` target. PTX is NVIDIA's intermediate representation for GPU computations, and is used by NVIDIA's OpenCL and CUDA implementations. Although PTX is itself an IR and not a direct match to the code that is executed on the GPU, it is the closest we can get without direct access to the internals of NVIDIA's drivers.

Normally, programs that use OpenCL load the kernels and pass it to the `clCreateProgramWithSource`, which compiles them (*online compilation*). For

specialized phase ordering, we instead compile the source code to PTX using Clang/LLVM and pass the PTX to the `clCreateProgramWithBinary` (*offline compilation*).

A compiler phase order represents not only the compiler passes to execute in the compiler pipeline, which can be in order of the hundreds, but also their order of execution. The fact that compiler passes are interdependent and interfere with each other's execution in ways that are difficult to predict can make it extremely hard to manually generate suitable compiler sequences. For the experiments presented in this paper, the OpenCL kernels from each of the PolyBench/GPU benchmarks were compiled/tested with a set of $10,000$ randomly generated compiler phase orders (the same set was used with all OpenCL codes) composed of 256 LLVM pass instances (can include repeated calls to the same pass). Passes were selected from a list with all LLVM 3.9 passes except the ones that resulted in compilation and/or execution problems when used individually to compile the PolyBench/GPU OpenCL kernels.

2.3 Validation of the Code Generated After Phase Ordering

Each PolyBench/GPU benchmark has verification embedded in its code that consists in executing the OpenCL GPU kernel(s) followed by a functionally equivalent sequential C version on the CPU, and comparing the two. This alone poses a challenge, as CPU executing using the same parameters as the ones used for GPU execution takes a long time for a considerable number of the PolyBench/GPU benchmarks. This would have an unreasonable impact on the phase ordering exploration time.

To reduce the time for each DSE iteration, we separate the validation from the measurement phases. We validate the programs by executing on the CPU and GPU (as in the original PolyBench/GPU) with inputs that can be processed quickly. However, we also execute the same GPU code using the original inputs (without CPU validation) in order to measure the execution time.

We further reduce exploration time by checking whether an identical PTX file was previously generated. If so, we reuse the results (i.e., correctness and performance) from that previous execution.

At the end of phase ordering exploration, all compiler pass sequences that were iteratively tested during DSE are ordered by their resulting objective metrics. For the experiments presented in this paper, sequence/metric pairs are ordered from the one resulting in the fastest execution time to the one resulting in least performance. Then, as a final validation process, the optimized version that resulted in highest performance is executed with the original inputs on both the non-optimized CPU version and the optimized GPU version, and also with 30 randomly generated inputs that result in the same number of operations. We choose the fastest optimized version that passes validation.

This is performed to eliminate possible situations where a compiled PTX kernel gives correct results using a small input set but gives wrong results with the original input set.

The PolyBench/GPU kernels are mostly composed of floating-point operations and the result of floating-point operations can be affected by reordering operations and rounding. Because of this we allow for up to 1% difference between the outputs of CPU and GPU executions when testing if a given compiler phase order results in code that generates valid output.

3 Results

For each of the benchmarks, we measured the execution times for the CUDA version, the original OpenCL (from source), an offline compiled OpenCL without optimization, an offline compiled OpenCL with standard LLVM optimization levels (i.e., the best of `-O1`, `-O2`, `-O3` and `-Os` for each benchmark, which we will refer to as `-OX`) and an offline compiled OpenCL with our custom compiler optimization phase orders resulting from DSE.

3.1 Performance Evaluation

We compared the results for the various versions of the benchmarks (offline OpenCL versions, OpenCL from source and CUDA) to determine how they perform. Using custom phase orders found by iterative compilation produced code that consistently outperforms the other OpenCL variants, and nearly always outperforms the CUDA version.

Figure 1 depicts the performance improvements with phase ordering over the OpenCL compiled online and CUDA baselines and the other OpenCL baselines (compiled with Clang/LLVM). With phase ordering specialization we were able to achieve a geometric mean performance improvement of 1.54× over the CUDA version and a performance improvement of 1.65× over the execution of the OpenCL kernels compiled from source. Additionally, code compiled with specialized phase ordering can be up to 5.48× and up to 5.70× faster than the respective CUDA implementation and the OpenCL compiled from source.

For the tested benchmarks, there were mostly no significant performance difference between the offline compilation model using Clang/LLVM without custom phase ordering and the OpenCL versions from source. There were exceptions, such as `GESUMMV` and `SYR2K`, that were 1.18× and 1.15× slower when using Clang/LLVM (with no optimization) to compile the kernels offline.

Using the LLVM standard optimization level flags did not result in noticeable improvements in terms of the performance of the generated code for most benchmarks. We believe this is because the PTX code is further aggressively optimized by the NVIDIA driver before generating the final assembly code for the target NVIDIA GPU [12], so effectively we are using LLVM only as a pre-optimizer.

For `2DCONV`, `FDTD-2D` and `SYR2K` all of the standard optimization level flags (including `-O0`) resulted in the same code being generated. For benchmarks `2MM`, `3DCONV`, `3MM`, `ATAX`, `BICG`, `GEMM`, `GESUMMV`, `GRAMSCHM`, `MVT` and `SYRK`, the optimization level flags lead to code that is different from the code without optimizations. `CORR` and `COVAR` are the only benchmarks for which different optimization level

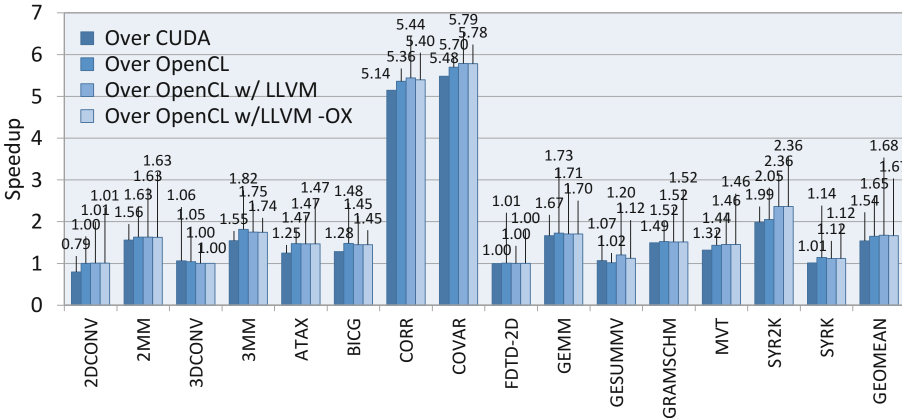

Fig. 1. Performance improvements from phase ordering with LLVM over CUDA implementations and OpenCL using the default online compilation pipeline for the NVIDIA GTX1070 GPU and over OpenCL to PTX compilation using Clang/LLVM without (OpenCL w/LLVM) and with standard optimization levels (OpenCL w/LLVM -OX).

flags produce different code. However, even in these benchmarks, the performance impact was minimal (within 1%).

For the `GESUMMV` and `GRAMSCHM`, there were significant performance improvements associated with the use of standard optimization levels. In the case of `GESUMMV`, the use the standard optimization levels resulted in 1.07× performance improvement over the non-optimized version. For `GRAMSCHM`, the non-optimized version was 1.04× faster than the versions produced by the optimization level flags.

The difference between the OpenCL baselines is that one represents the de facto OpenCL compilation flow (with compile from source) and the others represent the compilation using LLVM (with compile from binary) using the standard optimization level that results in the generation of code with highest performance on a kernel-by-kernel basis, and compilation using LLVM but with no optimization. Finally, on these benchmarks, performance with CUDA tends to be better than with OpenCL, if no specialized phase ordering is considered. The geometric mean (considering all 15 PolyBench/GPU benchmarks) of the performance improvement with CUDA (over OpenCL from source) is 1.07×. The `2DCONV`, `3MM`, `ATAX`, `BICG` and `SYRK` benchmarks are at least 1.1× faster in CUDA than with OpenCL. All other benchmarks with exception for `3DCONV` and `GESUMMV` are still faster in CUDA than in OpenCL, although by a smaller margin.

Table 1 depicts LLVM 3.9 compiler phase orders found to have better performance than the OpenCL baseline that relies on Clang/LLVM and the libclc OpenCL library.

Table 1. Compiler phase orders that resulted in compiled kernels with highest performance. Compiler passes that resulted in no performance improvement were eliminated from the compiler phase orders. No compiler phase orders resulted in improving the performance of 2DCONV, 3DCONV or FDTD-2D.

Benchmark	Compiler phase order
2MM	`-cfl-anders-aa -dse -loop-reduce -licm -instcombine`
3MM	`-loop-reduce -gvn-hoist -reg2mem -cfl-anders-aa -sroa -licm`
ATAX	`-bb-vectorize -loop-reduce -licm -cfl-anders-aa`
BICG	`-gvn -loop-reduce -cfl-anders-aa -licm`
CORR	`-cfl-anders-aa -loop-reduce -gvn -loop-extract-single -loop-unswitch -loop-unswitch -ipsccp -reg2mem -licm -nvptx-lower-alloca`
COVAR	`-cfl-anders-aa -loop-unswitch -sink -loop-unswitch -loop-reduce -jump-threading -reg2mem -licm -nvptx-lower-alloca`
GEMM	`-cfl-anders-aa -print-memdeps -loop-reduce -licm`
GESUMMV	`-instcombine -reg2mem -instcombine -mem2reg -cfl-anders-aa -loop-reduce -nvptx-lower-alloca -gvn-hoist -licm`
GRAMSCHM	`-sink -reg2mem -licm -cfl-anders-aa -sroa`
MVT	`-gvn -loop-reduce -cfl-anders-aa -licm`
SYR2K	`-loop-reduce -loop-unroll -instcombine -loop-reduce -licm -cfl-anders-aa`
SYRK	`-licm -cfl-anders-aa -reg2mem -licm -sroa`

3.2 Analysis of the Results

We explain for each PolyBench/GPU benchmark what are the reasons behind the performance improvement achieved with phase ordering, comparing with the performance achieved with the OpenCL and CUDA baselines compiled with the NVIDIA driver. More specifically, we compare the PTX output resulting from OpenCL offline compilation with specialized phase ordering with PTX generated from OpenCL offline compilation without phase ordering and with PTX generated from the CUDA versions.

For `2DCONV`, CUDA is 1.26× faster than the OpenCL version optimized with phase ordering. The compiler pass phase ordering DSE process was not able to find an LLVM sequence capable to optimize this benchmark. The main improvement of CUDA over OpenCL seems to be the generation of more efficient code for loads from global memory. Figure 2 shows the difference between the two approaches. Whereas load operations typically result in a single CUDA operation, the equivalent for OpenCL typically results in 5 PTX instructions. We believe this difference is the primary reason for CUDA's advantage over OpenCL.

For `2MM`, the OpenCL version optimized with phase ordering is 1.63× and 1.56× faster than the OpenCL and CUDA baselines, respectively. The main

```
ld.global.f32   %f2, [%rd6+4]
```

(a) PTX load code generated from CUDA.

```
add.s32         %r17, %r14, %r1;
cvt.s64.s32     %rd16, %r17;
shl.b64         %rd17, %rd16, 2;
add.s64         %rd18, %rd1, %rd17;
ld.global.f32   %f2, [%rd18];
```

(b) PTX load code generated from OpenCL (`-O3`).

Fig. 2. PTX code for equivalent load operations, for CUDA and OpenCL with offline compilation (2DCONV benchmark)

reason for this speedup is the removal of store operations within the kernel loop. Both the OpenCL and the CUDA baseline versions of this kernel repeatedly overwrite the same element and this has a negative impact on performance. The phase ordered version instead uses an accumulator register and performs the store only after all the loop computations are complete, which substantially reduces the number of costly memory accesses. It is unclear why the baseline OpenCL and CUDA versions do not perform this optimization. One possibility is that they are unable to determine that there are no aliasing issues. In the context of this benchmark, this assumption is correct in OpenCL 2.0, as any aliasing would result in a data race, which is undefined behavior [1]. We do not know if the optimization was applied because LLVM correctly discovered this fact, or if there is a bug that happened to result in correct code by accident. Even if the optimization turns out to be the result of a bug, we believe this speedup represents an opportunity for approaches based on *Loop Versioning* transformations. Although this benchmark uses two kernels, both are equivalent (the only difference being kernel and variable names), and thus the same analysis applies to both. There are two differences between the baseline CUDA and OpenCL compiled versions that can explain the different execution times. The first being the aforementioned issue with load instructions (see Fig. 2), the second being a different loop unroll factor as the phase ordered version based on OpenCL uses efficient load instructions, but also uses a loop unroll factor of 2 (while the CUDA version uses an unrolling factor of 8).

For `3DCONV`, we were unable to achieve a speedup on this benchmark using any of the tested compiler phase orders, when compared with LLVM w/ or w/o the optimization level flags. We believe this happens because most of the time spent on the benchmark is due to global memory loads that are not removed or improved by any LLVM pass. Any optimization will only modify the rest of the code, which takes a negligible amount of time compared to the memory operations. There is a speedup from the use of the LLVM PTX backend compared with the OpenCL from source compilation path ($1.05\times$) and the compilation from CUDA ($1.06\times$).

On the 3MM benchmark, we were able to achieve speedups of 1.55× and 1.82× over the baseline CUDA and OpenCL version compiled from source, respectively. The main reason for the performance improvement is the removal of the memory store operation from the computation loop.

The OpenCL version of ATAX optimized with phase ordering achieves a speedup of 1.47× and 1.25× over the baseline OpenCL and CUDA versions, respectively. Once again, the phase ordered version is able to move memory stores out of the innermost loops of the kernels, which explains the speedups. The difference between the CUDA and the baseline OpenCL versions can be explained by a different loop unroll factor (2 for OpenCL, 8 for CUDA). The CUDA version uses the previously described simpler code pattern for memory loads compared to these baseline OpenCL versions, but the phase ordering version also uses an efficient memory load pattern.

On the BICG benchmark, we were able to achieve a speedup of 1.48× over OpenCL, and 1.28× over CUDA. The main differences between the versions are the memory stores in the kernel loop, the unroll factor and the inefficient memory access patterns in the baseline offline OpenCL versions.

The CORR benchmark is one of the benchmarks that benefits the most from phase ordering (5.36× and 5.14× over baseline OpenCL from source and CUDA versions, respectively). Phase ordering is capable of moving global memory stores out of loops, which neither the CUDA version nor the baseline OpenCL versions do. In general, for this benchmark, the CUDA version tends to produce more compact load instructions and use higher loop unroll factors than the OpenCL versions.

COVAR and CORR use the same mean_kernel and reduce_kernel functions. However, this represents only a fragment of the total execution code, so the compiler sequences for the two benchmarks are different. Regardless, the same conclusions from CORR apply to COVAR: phase ordering removes global stores from the loop. COVAR improved by 5.7× and 5.48× with phase ordering specialization, compared with the OpenCL compiled from source and the CUDA version.

The functions of the FDTD-2D benchmark are very straightforward, with little potential for optimization. As such, phase ordering had no impact.

The performance differences for the GEMM benchmark (1.73× and 1.67× over the OpenCL from source and the CUDA baselines) can be explained by the removal of the memory store operation from the kernel loop and the different pattern of memory load instructions.

There was only a small performance improvement for the GESUMMV benchmark (1.07× over CUDA and 1.02× over the baseline OpenCL from source). The phase ordering sequence is able to extract the memory stores out of the main computation loop, but uses a smaller loop unroll factor (2) than the baseline OpenCL and CUDA versions (4 and 16, respectively).

We were able to obtain speedups of 1.49× and 1.52× over the baseline CUDA and OpenCL versions on the GRAMSCHM, respectively. Phase ordering is able to move the memory storage operations out of the loop. Aside from that, it uses the same load from memory instruction pattern and unroll factor as the baseline OpenCL versions.

The MVT benchmark benefits from phase ordering by a factor of 1.32× and 1.44× over the baseline CUDA and OpenCL versions. The main reason for this improvement is the extraction of the store operation from the computation loop.

The SYR2K benchmarks benefits from phase ordering by a factor of 1.99× and 2.05× over the baseline CUDA and OpenCL versions, respectively. In general, the same memory load pattern, loop unroll factor and loop invariant memory storage code motion conclusions apply to this benchmark. Phase ordering also seems to outline the segment of the code containing the kernel loop, but this does not seem to be the reason for the performance difference.

For the SYRK benchmark, phase ordering improves performance by 1.14× over the OpenCL baseline compile from source. We could not achieve significant speedups over the CUDA version. Once again, the main reason for this improvement is the extraction of the store from the loop.

4 Conclusion

This paper showed that compiler pass phase ordering specialization allows achieving considerable performance improvements when compiling OpenCL kernels to NVIDIA GPUs. Using Clang/LLVM 3.9 and libclc we were able to improve the performance of code compiled from PolyBench/GPU OpenCL kernels to up to 5.70× and 1.65× on average over the default NVIDIA OpenCL compilation flow. The performance of OpenCL compiled with specialized compiler pass phase orders also tends to surpass the performance of CUDA implementations of the same kernels compiled with NVCC (from NVIDIA CUDA 8.0 toolchain). The use of phase ordering on top of the OpenCL versions of the kernels resulted in a maximum speedup of 5.48× and a geometric mean speedup of 1.54× when compared with the performance of the equivalent CUDA kernels compiled with NVCC.

We gave insights explaining why the OpenCL kernels compiled with LLVM specialized compiler pass phase orders tend to have considerably higher performance than both the kernels compiled with the traditional OpenCL compilation from source and the CUDA equivalent kernels. One of the optimizations with most impact in performance of the compiled OpenCL kernels over the performance resulting from OpenCL online compilation from source and the CUDA versions consists of moving memory writes out of inner loops of GPU kernels by using of an accumulator register. This avoids the overhead caused by repeated expensive global memory writes. The optimization can be performed even in cases where its correctness can not be proven at compilation time. This can be achieved with *Loop Versioning*, which consists of adding runtime checks that will result in the selection of what loop version (i.e., optimized or non-optimized) to execute at runtime.

We are currently evaluating the potential of compiler phase ordering for GPU energy consumption reduction, and how it correlates with performance as we previously did in the context of C code targeting x86 and ARM based systems [9]. Given the fact that GPUs are used in domains with energy (and power)

concerns (e.g., HPC, embedded), there may be scenarios where it is acceptable to sacrifice performance for less total energy use.

We extended our DSE system to be able to target AMD GPUs, and we are currently exploring software optimization leveraged by compiler phase ordering specialization on these devices.

Acknowledgments. This work was 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 by FCT through PD/BD/105804/2014.

References

1. Khronos OpenCL Working Group. The OpenCL C Specification, Version 2.0 (2015)
2. Nickolls, J., et al.: Scalable parallel programming with CUDA. Queue **6**(2), 40–53 (2008)
3. Betkaoui, B., Thomas, D.B., Luk, W.: Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing. In: 2010 International Conference on Field-Programmable Technology, Beijing, pp. 94–101 (2010)
4. Kulkarni, S., Cavazos, J.: Mitigating the compiler optimization phase-ordering problem using machine learning. In: Proceedings of ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2012, pp. 147–162. ACM, New York (2012)
5. Purini, S., Jain, L.: Finding good optimization sequences covering program space. ACM Trans. Archit. Code Optim. (TACO) **9**(4), 56:1–56:23 (2013)
6. Martins, L.G.A., et al.: Clustering-based selection for the exploration of compiler optimization sequences. ACM Trans. Archit. Code Optim. (TACO) **13**(1), 8:1–8:28 (2016)
7. Nobre, R., Martins, L.G.A., Cardoso, J.M.P.: Use of previously acquired positioning of optimizations for phase ordering exploration. In: Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2015), pp. 58–67. ACM, New York (2015)
8. Nobre, R., Martins, L.G.A., Cardoso, J.M.P.: A graph-based iterative compiler pass selection and phase ordering approach. In: Proceedings of 17th ACM Conference on Languages, Compilers, Tools, and Theory for Embedded Systems, LCTES 2016, pp. 21–30. ACM, New York (2016)
9. Nobre, R., Reis, L., Cardoso, J.M.P.: Compiler phase ordering as an orthogonal approach for reducing energy consumption. In: Proceedings of the 19th Workshop on Compilers for Parallel Computing, CPC 2016 (2016)
10. Grauer-Gray, S., et al.: Auto-tuning a high-level language targeted to GPU codes. In: Proceedings of Innovative Parallel Computing (InPar 2012) (2012)
11. Purini, S., Jain, L.: Finding good optimization sequences covering program space. ACM Trans. Archit. Code Optim. **9**(4), 23 (2013). Article 56
12. Parallel Thread Execution ISA Version 5.0. CUDA toolkit documentation. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html