Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

© 2020 The authors and IOS Press.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/APC200089

Performance and Energy Efficiency of CUDA and OpenCL for GPU Computing Using Python

Håvard H. HOLM [a,c,1] André R. BRODTKORB [a,d], and Martin L. SÆTRA [b,d]

[a] *SINTEF Digital, Department of Mathematics and Cybernetics.*
[b] *Norwegian Meteorological Institute, Information Technology Department.*
[c] *Norwegian University of Science and Technology, Department of Mathematics.*
[d] *Oslo Metropolitan University, Department of Computer Science.*

Abstract. In this work, we examine the performance and energy efficiency when using Python for developing HPC codes running on the GPU. We investigate the portability of performance and energy efficiency between CUDA and OpenCL; between GPU generations; and between low-end, mid-range and high-end GPUs. Our findings show that for some combinations of GPU and GPU code, there is a significant speedup for CUDA over OpenCL, but that this does not hold in general. Our experiments show that performance in general varies more between different GPUs, than between using CUDA and OpenCL. Finally, we show that tuning for performance is a good way of tuning for energy efficiency.

Keywords. GPU Computing, CUDA, OpenCL, High Performance Computing, Shallow-Water Simulation, Power Efficiency

1. Introduction

GPU computing was introduced in the early 2000s, and has since become a popular concept. The first examples were acceleration of simple algorithms such as matrix-matrix multiplication by rephrasing the algorithm as operations on graphical primitives (see e.g., [1]). This was cumbersome and there existed no development tools for general-purpose computing. However, many algorithms were implemented on the GPU as proof-of-concepts, showing large speedups over the CPU [2]. Today, the development environment for GPU computing has evolved tremendously and is both mature and stable: Advanced debuggers and profilers are available, making debugging, profile-driven development, and performance optimization easier than ever.

The GPU has traditionally been accessed using compiled languages such as C/C++ or Fortran for the CPU code, and a specialized programming language for the GPU. The rationale is often that performance is paramount, and that compiled languages are therefore required. However, for many GPU codes, most of the time is spent in the numerical code running on the GPU. In these cases, we can possibly use a higher-level

[1] Corresponding Author

language such as Python for the program flow without significantly affecting the performance.[2] The benefit is that using higher-level languages can significantly increase productivity [3].

We study PyCUDA and PyOpenCL [4] in this work, which offer access to CUDA and OpenCL from Python. They have become mature and popular packages since their initial release nearly ten years ago. We compare the performance and energy efficiency of PyCUDA and PyOpenCL for three different explicit numerical stencils for simulating shallow-water flow. This represents a class of problems that are particularly well suited for GPU computing [5,6]. We demonstrate how profile-driven development in Python is essential for increasing performance (up to 5 times) for GPU code, and show that the energy efficiency increases proportionally with performance tuning. Finally, we investigate the portability of the improvements and power efficiency both between CUDA and OpenCL and between different GPUs.

2. Related work

There are several high-level programming languages and libraries that offer access to the GPU for certain sets of problems and algorithms. OpenACC [7] is one example which is pragma-based and offers a set of directives to execute code in parallel on the GPU. However, such high-level abstractions are typically only efficient for certain classes of problems and are often unsuitable for non-trivial parallelization or data movement. CUDA [8] and OpenCL [9] are two programming languages that offer full access to the GPU hardware, including the whole memory subsystem. This is an especially important point, since memory movement and data transfers are often key bottlenecks in today's numerical algorithms [10] and therefore have significant impact on energy consumption.

The performance of GPUs has been reported extensively [11], and several authors have previously shown that GPUs are efficient in terms of energy-to-solution. Huang et al. [12] demonstrated early on that GPUs could not only speed up computational performance, but also increase power efficiency dramatically using Nvidia CUDA. Qi et al. [13] show how OpenCL on a mobile GPU can increase performance of the discrete Fourier transform by 1.4 times and decrease the energy by 37%. Dong et al. [14] analyze the energy efficiency of GPU BLAST which simulates compressible hydrodynamics using finite elements with CUDA and report a 2.5 times speedup and a 42% increase in energy efficiency. Klôh [15] report that there is a wide spread in terms of energy efficiency and performance when comparing 3D wave propagation and full waveform inversion on two different architectures. They compare an Intel Xeon coupled with an ARM-based Nvidia Jetson TX2 GPU module, and find that the Xeon platform performs best in terms of computational speed, whilst the Jetson platform is most energy efficient. Memeti et al. [16] compare the programming productivity, performance, and energy use of CUDA, OpenACC, OpenCL and OpenMP for programming a system consisting of a CPU and GPU or a CPU and an Intel Xeon Phi coprocessor. They report that CUDA, OpenCL and OpenMP have similar performance and energy consumption in one benchmark, and that OpenCL performs better than OpenACC for another benchmark. In terms of productivity, the actual person writing the code is important, but OpenACC and OpenMP require

[2] Kernel launch overhead is on the same order of magnitude as in C++ in our experiments.

less effort than CUDA and OpenCL, and CUDA can require significantly less effort than OpenCL.

Previous studies have also shown that CUDA and OpenCL can compete in terms of performance as long as the comparison is fair [17,18,19,20] and there have also been proposed automatic source to source compilers that report similar results [21,22].

3. GPU Computing in Python

In this work we focus on using Python to access the GPU through CUDA and OpenCL. These two GPU programming models are conceptually very similar, and both offer the same kind of parallelism primitives. The main idea is that the computational domain is partitioned into equally sized subdomains that are executed independently and in parallel. Even though the programming models are similar, their terminology differs slightly, and in this paper we will use that of CUDA. A full review is outside the scope of this work, but can be found in [23,24]. The following sections give an overview of important parts of CUDA and OpenCL, and discuss their respective Python wrappers.

3.1. CUDA

CUDA [8] (Compute Unified Device Architecture) was first released in 2007, and is available on all Nvidia GPUs as Nvidia's proprietary GPU computing platform. It includes third-party libraries and integrations, the directive-based OpenACC [7] compiler, and the CUDA C/C++ programming language. Today, five of the ten fastest supercomputers (including number one) use Nvidia GPUs, as well as nine out of the ten most energy-efficient [25].

CUDA is implemented in the Nvidia device driver, but the compiler (nvcc) and libraries are packaged in the CUDA toolkit and SDK.[3] The toolkit and SDK contain a plethora of examples and libraries. In addition, the toolkit contains Nvidia Nsight, which is an extension for Microsoft Visual Studio (Windows) and Eclipse (Linux) for interactive GPU debugging and profiling. Nsight offers code highlighting, unified CPU and GPU trace of the application, and automatic identification of GPU bottlenecks. The Nvidia Visual Profiler is a stand-alone cross-platform application for profiling of CUDA programs, and CUDA versions for debugging (cuda-gdb) and memory checking (cuda-memcheck) also exist.

3.2. OpenCL

OpenCL [9] (Open Compute Language) is a free and open heterogeneous computing platform that was initiated by Apple in 2009, and today the OpenCL standard is maintained and developed by the Khronos group. Whilst CUDA is made specifically for Nvidia GPUs, OpenCL can run on a number of heterogeneous computing architectures including GPUs, CPUs, FPGAs, and DSPs. Contrary to CUDA, there is no official toolkit for OpenCL, but there are several third-party libraries.

[3] Available at `https://developer.nvidia.com/cuda-zone`.

Profiling an OpenCL application can be challenging, and the available tools vary depending on your operating system and hardware vendor.[4] It is possible to get timing information on kernel and memory transfer operations by adding counters and enabling event profiling information explicitly in your source code. This requires extra work and makes the code more complex. Visual Studio can measure the amount of run time spent on the GPU, and CodeXL [26] can be used to get more information on AMD GPUs. CodeXL is a successor to gDebugger which offers features similar to those found in Nsight in addition to power profiling, and is available both as a stand-alone cross-platform application and as a Visual Studio extension. While it is possible to use Visual Profiler for OpenCL, this requires the use of the command-line profiling functionality in the Nvidia driver, which needs to be enabled through environment variables and a configuration file. After running the program with the profiling functionality in effect, the profiling data can be imported into Visual Profiler.

One disadvantage of OpenCL is that there are large differences between the OpenCL implementations from different vendors, and good performance might rely on vendor-specific extensions. One example is that OpenCL 2.2 is required for using C++ templates in the GPU code, but vendors such as Nvidia only support OpenCL version 1.2. It should also be mentioned that OpenCL has been deprecated in favour of Metal [27] by Apple in their most recent versions of Mac OS X.

3.3. GPU computing from Python

Researchers spend a large portion of their time writing computer programs [28], and compiled languages such as C/C++ and Fortran have been the de facto standard within scientific computing for decades. These languages are well established, well documented, and give access to a plethora of native and third-party libraries. C++ is the standard way of accessing CUDA and OpenCL today, but developing code with these languages is time consuming and requires great care. Using higher-level languages such as Python can significantly increase development productivity [3,29]. However, it should be mentioned that the OpenCL and CUDA *kernels* themselves are not necessarily made neither simpler nor shorter by using Python: The productivity gain comes instead from Python's less verbose code style for the CPU part of the code. This influences every part of the host code, from boilerplate initialization code and data pre-processing, to CUDA/OpenCL API calls, post-processing, and visualization of results.

PyCUDA and PyOpenCL [4] are Python packages that offer access to CUDA and OpenCL, respectively. Both libraries expose the API of the underlying programming models, and try to minimize the performance impact. The GPU kernels, which are crucial for the inner loop performance, are written in native low-level CUDA or OpenCL, and memory transfers and kernel launches are made explicit through Python functions. The result is an environment suitable for rapid prototyping of high-performing GPU code in a Python environment.

[4] An extensive list of OpenCL debugging and profiling tools can be found at `https://www.khronos.org/opengl/wiki/Debugging_Tools`.

Table 1. A list of the GPUs used in this work. The profile-driven development was carried out on the 840M and GTX780, and all three GPUs were used to benchmark computational performance and power efficiency. Note that the performance in gigaFLOPS is for single precision.

Model	Class	Architecture (year)	Memory	GigaFLOPS	Bandwidth	Power device
GeForce 840M	Laptop	Maxwell (2014)	4 GiB	790	16 GB/s	Watt meter
GeForce GTX780	Desktop	Kepler (2013)	3 GiB	3977	288 GB/s	Watt meter
Tesla V100	Server	Volta (2017)	16 GiB	14899	900 GB/s	nvidia-smi

4. Benchmarking performance and power efficiency

Throughout this section, we consider simulation of the shallow-water equations using three different numerical schemes:

- a linear finite difference scheme,
- a nonlinear finite difference scheme, and
- a high-resolution finite volume scheme.

We benchmark different versions for each of the three codes with respect to computational performance and power efficiency on three different GPUs. The schemes are used for simulating real-world ocean currents, and two of them have been used operationally. All three schemes are essentially stencil operations with an increasing level of complexity, and their details are summarized in Holm et al. [30].

4.1. Profile-driven optimization and porting

Our starting point is OpenCL implementations of the three numerical schemes [30,31], and even though the code is algorithmically well suited for the GPU, little effort has been made to thoroughly optimize its performance on a specific GPU. It is well known in the GPU computing community that performance is not portable between GPUs, neither for CUDA nor OpenCL, and automatically generating good kernel configurations is an active research area (see e.g., [32,33,34]). Using the available profiling tools for CUDA, we perform profile-driven optimization. This is an iterative process that starts by profiling the code to identify the main performance bottleneck. The next step is then to optimize this bottleneck, before running tests that ensure that the optimization did not introduce bugs to the code. Thereafter the code is analysed in the profiler again in search for a new bottleneck [35]. The profiling and tuning is carried out mainly on a laptop with a dedicated GeForce 840M GPU, representing the low-end part of the GPU performance scale, and some final tuning was performed on a desktop with a GeForce GTX780 GPU, representing a typical mid-range GPU. We compare the performance of the original and optimized implementations with PyCUDA and PyOpenCL using these two GPUs, as well as on a recent high-end Tesla V100 GPU commonly found in servers and super computers. Information on these three GPUs is listed in Table 1. Together they represent the different types of GPUs a typical researcher might have access to in a research environment: a laptop and/or a desktop with local access and an expensive powerful server GPU in a remote location.

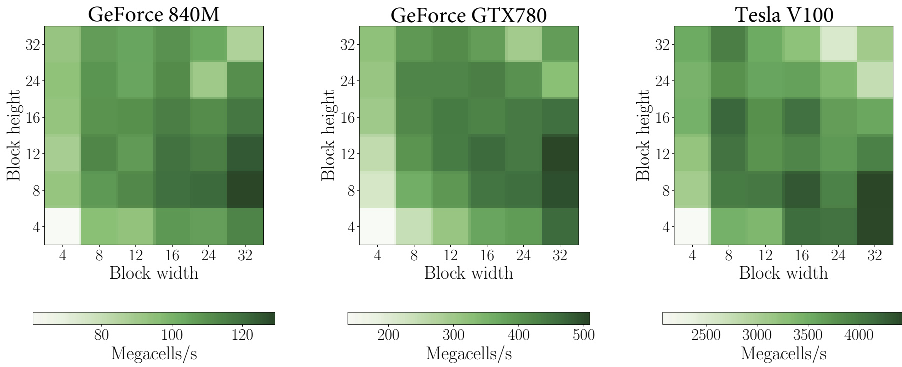

Figure 1. Heat map of performance as a function of block width and block height for selected sizes for the high-resolution scheme on three different GPUs. Notice that even though the performance patterns have similarities, the performance on the V100 would be suboptimal if the optimal configuration from the GTX780 is used. The performance increase is $2-5\times$ for all three GPUs from the slowest to the fastest block size.

An important performance parameter for GPUs is the domain decomposition determined by the block size. CUDA decomposes the work into a grid with equally sized blocks, and all blocks are executed independently. At runtime, the GPU takes the set of blocks and schedules them to the different cores within the GPU. Using a too small block size will under-utilize the GPU, and using a too large block size will similarly exhaust the GPU's resources. Figure 1 shows how the block size has a major impact on performance, and also illustrates that finding the best block size can be difficult. Because of this, we experimentally obtain the optimal configuration for each scheme before starting profiling, and again after the code has been optimized.

The porting process between PyOpenCL and PyCUDA requires changes in both the kernel code that runs on the GPU, and the API calls in the CPU code. Most of the changes in the kernels are straight-forward and consist of changing keywords due to different terminologies [21,22]. The CPU APIs are however quite different between OpenCL and CUDA, but their Python wrappers reduce the differences somewhat. It still requires correct handling of devices, contexts and streams, compiling and linking kernels, setting kernel arguments, grid and block dimensions, and memory transfers. Furthermore, there is also a large difference in availability of native and third-party libraries, built-in and specialized functions, and data types that need to be handled.

In our case, the linear and nonlinear schemes were found to be very memory-bound. Both schemes were originally implemented with three separate kernels (one for each of the three variables in the shallow-water equations), with a large overlap in the data dependency. The first tuning step consisted of fusing the three kernels into one, and thus reduced the overall memory bandwidth requirement of both schemes.

Profiling the high-resolution scheme revealed that the occupancy was very low due to excessive use of shared memory. The optimizations consisted of largely reducing the amount of shared memory used, and instead relying more on re-computations. When shared memory usage was no longer the performance bottleneck, we reduced the number of registers used per thread. All three schemes were further optimized through the use of compiler flags.

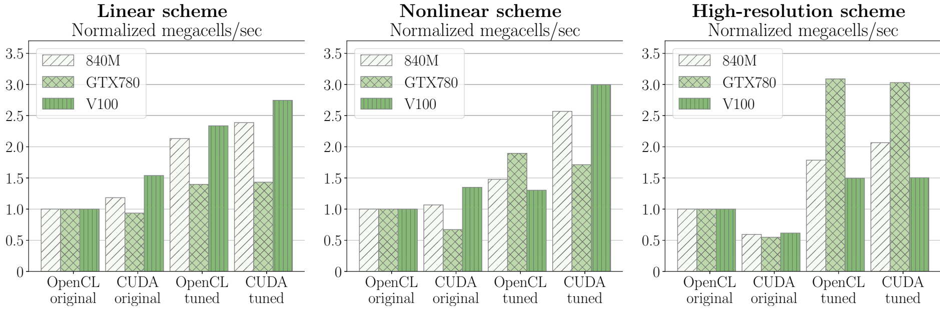

Figure 2. Performance of original, ported, and optimized codes measured in megacells per second, normalized with respect to performance of the original OpenCL implementation. Notice that there is relatively little difference between CUDA and OpenCL, except for the nonlinear scheme on the 840M and V100, whilst there is a significant difference in how effective the tuning is for the different architectures. Furthermore, there is a significant loss of performance when porting from OpenCL to CUDA in our original approach for the high-resolution scheme. From our experience, this relates to how the two languages optimize mathematical expressions with and without the fast-math compilation flags.

4.2. *Comparing performance*

The overall performance gain of our optimization is shown in Figure 2, where all results are given in megacells per second normalized with respect to the original PyOpenCL implementation. The original porting from PyOpenCL to PyCUDA gave a noticeable reduction in performance for the high-resolution scheme on all GPUs. After careful examination, we attribute this to different default compilation flags in PyCUDA and PyOpenCL: In PyCUDA, the fast-math flag was shown to double the performance for the high-resolution scheme, while we found that it gave less than 5% performance gain with PyOpenCL. Note that this affects the linear and nonlinear schemes to less extent, as these schemes contain fewer complex mathematical operations, and we instead observe a varying effect on performance of porting the original OpenCL code to CUDA. When examining the numerical schemes one by one, we see that the optimizations performed for the high-resolution scheme appears to be highly portable when back-ported to PyOpenCL for all GPUs. For the tuned nonlinear scheme, however, we see that the 840M and V100 GPUs give significantly higher performance using CUDA than OpenCL, but similar performance on the GTX780. Finally, for the linear scheme, the tuning gives a similar increase in performance both on CUDA and OpenCL, relative to the original versions. In total, we see that certain scheme and GPU combinations result in a significant speedup for CUDA over OpenCL, but we cannot conclude whether this is caused by differences in driver versions or from other factors. We are therefore not able to claim that CUDA performs better than OpenCL in general.

4.3. *Measuring power consumption*

We measure power consumption in two ways. The first method is by using the nvidia-smi application, which can be used to monitor GPU state parameters such as utilization, temperature, power draw, etc. By programmatically running nvidia-smi in the background during benchmark experiments, we can obtain a log containing a high-resolution power

draw profile for the runtime of the benchmark. The downside of using nvidia-smi is that information about power draw is only supported on recent high-performance GPUs. From our selection of GPUs, this applies only to the Tesla V100 GPU. Further, nvidia-smi monitors the energy consumption of the GPU only, meaning that we do not have any information about energy consumed by the CPU. For each experiment, nvidia-smi is started in the background exactly 3 s before the benchmark, and is configured to log the power draw every 20 ms. This background process is stopped again exactly 3 s after the end of the simulation. This approach allows us to measure the energy consumption of the idle GPU both before and after each benchmark, and we ignore the idle sections when computing the mean and total power consumption for each experiment. All results presented here are with the idle load subtracted from the experimental results.

The second method is to measure the total amount of energy used by the entire computer through a watt meter. The use of the watt meter requires physical access to the computer, and we are therefore restricted to do measurements on the laptop and desktop, containing the GeForce 840M and GeForce GTX780 GPUs, respectively. The watt meter offers no automatic logging or reading, but displays the total power used with an accuracy of 1 Wh. To get sufficiently accurate readings we need to run each benchmark long enough to keep the GPU busy for approximately one hour, after which we read the total and mean power consumption for each experiment. Before and after each benchmark, we also record the background power of the idle system, and the maximum recorded power during the experiment, to monitor whether the operating system is putting any non-related background load on the computer. It should also be noted that the battery was removed from the laptop during these experiments. Similarly to the first method, we subtract the idle loads from the result of each experiment, but we will here have the increased power load on the CPU also included.

4.4. Comparing power consumption

Figure 3 shows the results from the power efficiency experiments using the watt meter on the laptop (840M) and desktop (GTX780). The top row repeats the results for computational performance also shown in Figure 2 for the relevant GPUs, whereas the second row show the normalized mean power consumption with respect to the original OpenCL versions. The first thing we notice is that CUDA seems to require less power on the 840M compared to OpenCL for all versions of the three schemes. On the GTX780, however, there are no differences between the two programming models for equivalent versions. In fact, only the tuned high-resolution scheme seems to be different from the others (using about 30% more power), and this behavior can also be seen on the 840M. The power efficiency of the three schemes is shown in the bottom row in Figure 3, and we see that on the 840M the tuned CUDA versions are the most power efficient. This is because CUDA is both more efficient and uses less power on this system. On the GTX780, CUDA and OpenCL have equivalent power efficiency for all tuned schemes.

The results for the Tesla V100 are shown in Figure 4. The top row shows the computational performance in megacells per second, repeating the results from Figure 2. The second row shows the mean power used by each version of the three schemes. Note here that both CUDA versions for the nonlinear scheme use 60-90% more power than the OpenCL versions, which is the opposite result compared to the 840M results. For both the linear and high-resolution schemes, the results do not differ significantly in favor of

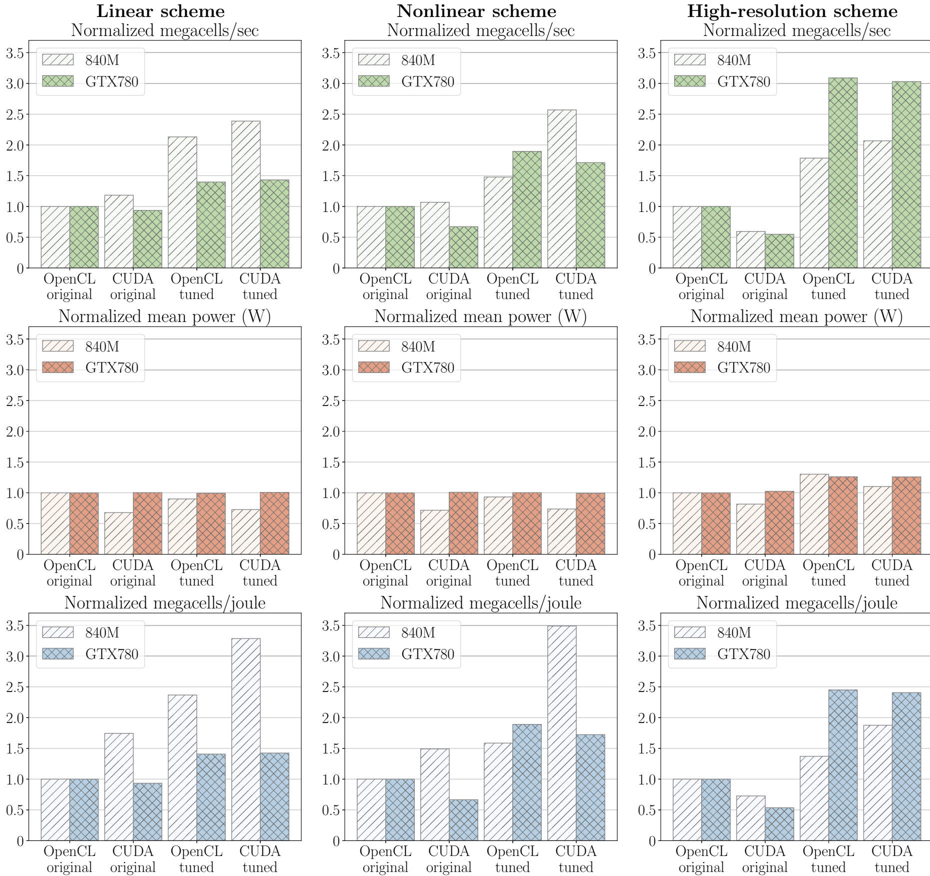

Figure 3. Comparison of original, ported and optimized codes measured in megacells per second (top row), mean power usage (mid row), and megacells per joule (bottom row), normalized with respect to the original OpenCL implementation, for the laptop (840M) and desktop (GTX780) GPUs. The power is measured through a watt meter, and represents the power consumed by the entire computer. Note that the CUDA versions requires less power than the OpenCL versions on the 840M, whereas there are no differences between equivalent versions on the GTX780. In terms of power efficiency, CUDA is more efficient than OpenCL on the M840, whereas the GTX780 gives the same power efficiency.

either CUDA or OpenCL, but the tuned OpenCL version uses slightly more power for the high-resolution scheme. When we consider the power efficiency in the bottom row, we see that the tuned CUDA versions are the best versions for all schemes. In particular for the nonlinear scheme, this is mostly due to the large difference in computational efficiency between CUDA and OpenCL for this particular scheme on this particular GPU.

In general, we observe that all experiments show a mean power usage within about 30% of the original OpenCL versions, with the exception of the nonlinear scheme on the V100. On the other side, the computational performance increases up to 5 times (the high-resolution scheme on the GTX780). This shows that the most important factor for improving power efficiency is to increase computational performance.

Figure 4. Comparison of original, ported and optimized codes measured in megacells per second (top row), mean power usage (mid row), and megacells per joule (bottom row), normalized with respect to the original OpenCL implementation, for the Tesla V100 GPU. The power is measured through nvidia-smi, and represents the power consumed by the GPU only. There are only minor differences in mean power consumption between different versions of the linear and high-resolution scheme, but CUDA uses more power than OpenCL for the nonlinear scheme. CUDA is however more power efficient than OpenCL for all three tuned schemes.

5. Summary

In this work, we have benchmarked three different OpenCL codes, our ported code in CUDA, our optimized CUDA code, and finally our OpenCL code with optimizations found using the CUDA tools. Our results are shown for three different GPUs, thus representing many of the GPU architectures in use today. Finally, we have looked at the power consumption for all versions of the code.

The original motivation for using OpenCL was to support GPUs and similar architectures from multiple vendors. Our motivation for changing from OpenCL to CUDA was because of the better software ecosystem for CUDA, and we have been very happy with our choice. CUDA appears to be a much more stable and mature development ecosystem with better tools for development, debugging and profiling for our hardware.

We found it interesting that our initial port from OpenCL to CUDA imposed a performance penalty, due to different default compiler optimizations. Even though some authors have reported OpenCL to be slower than CUDA, we find no conclusive results that support this in general. The performance gain varied much more with the GPU being used than whether we used CUDA or OpenCL. However, we do observe that for certain combinations of scheme and GPU, we get a significant speedup for CUDA over OpenCL. Additionally, we found that the performance gain of tuning the numerical schemes have vastly different effects on the run time for different GPUs. Even though we profiled and optimized mainly using a laptop GPU, the highest relative performance gain was for a server class and a desktop class GPU.

There does not seem to be any clear relations between the power consumption when comparing different schemes, optimization levels, GPUs, and programming models. When we look at power efficiency, we see that CUDA performs better than OpenCL for all tuned schemes on the Tesla V100 and GeForce 840M GPUs, whereas there are no differences on the GeForce GTX780. When we examine the impact of performance tuning on power efficiency, there appears to be a systematic and clear relationship: A fast code is a power efficient code.

6. Acknowledgments

This work is supported by the Research Council of Norway through grant number 250935 (GPU Ocean). The GPU Ocean project acknowledges the support from UNINETT Sigma2 - the National Infrastructure for High Performance Computing and Data Storage in Norway under project number nn9550k.

References

[1] E. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 55–55. ACM, 2001.

[2] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[3] S. Nanz and C. Furia. A comparative study of programming languages in rosetta code. In *IEEE International Conference on Software Engineering*, volume 1, pages 778–788, 2015.

[4] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

[5] T. Hagen, M. Henriksen, J. Hjelmervik, and K-A. Lie. *How to Solve Systems of Conservation Laws Numerically Using the Graphics Processor as a High-Performance Computational Engine*, pages 211–264. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[6] A. Brodtkorb, M. Sætra, and M. Altinakar. Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. *Computers & Fluids*, 55(0):1–12, 2012.

[7] OpenACC-Standard.org. The OpenACC application programming interface version 2.7, 2018.

[8] NVIDIA. NVIDIA CUDA C programming guide version 10.1, 2019.

[9] Khronos OpenCL Working Group. The OpenCL specification v. 2.2, 2018.

[10] K. Asanović, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, EECS Department, University of California, Berkeley, 2006.

[11] A. Brodtkorb, C. Dyken, T. Hagen, J. Hjelmervik, and O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, 2010.

[12] S. Huang, S. Xiao, and W-C. Feng. On the energy efficiency of graphics processing units for scientific computing. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8. IEEE, 2009.

[13] Z. Qi, W. Wen, W. Meng, Y. Zhang, and L. Shi. An energy efficient opencl implementation of a fingerprint verification system on heterogeneous mobile device. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–8. IEEE, 2014.

[14] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra. A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 972–981. IEEE, 2014.

[15] V. Klôh, D. Yokoyama, A. Yokoyama, G. Silva, M. Ferro, and B. Schulze. Performance and energy efficiency evaluation for HPC applications in heterogeneous architectures. 2018.

[16] S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, pages 1–6. ACM, 2017.

[17] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391–407, 2012.

[18] J. Fang, A. Varbanescu, and H. Sips. A comprehensive performance comparison of CUDA and OpenCL. In *2011 International Conference on Parallel Processing*, pages 216–225, 2011.

[19] T. Gimenes, F. Pisani, and E. Borin. Evaluating the performance and cost of accelerating seismic processing with CUDA, OpenCL, OpenACC, and OpenMP. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 399–408, 2018.

[20] K. Karimi, N. Dickson, and F. Hamze. A performance comparison of CUDA and OpenCL. Technical report, D-Wave Systems Inc., 2011.

[21] G. Martinez, M. Gardner, and W c. Feng. CU2CL: A CUDA-to-OpenCL translator for multi- and many-core architectures. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 300–307, 2011.

[22] J. Kim, T. Dao, J. Jung, J. Joo, and J. Lee. Bridging OpenCL and CUDA: a comparative analysis and translation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[23] D. Kaeli, P. Mistry, D. Schaa, and D. Zhang. *Heterogeneous Computing with OpenCL 2.0*. 2015.

[24] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[25] University of Mannheim, University of Tennessee, and NERSC/LBNL. Top 500 supercomputer sites. http://www.top500.org, June 2019.

[26] AMD Developer Tools Team. CodeXL quick start guide, 2018.

[27] Apple Inc. Metal programming guide, 2018.

[28] G. Wilson, D. Aruliah, C. Brown, N. Chue Hong, M Davis, R. Guy, S. Haddock, K. Huff, I. Mitchell, M. Plumbley, B. Waugh, E. White, and P. Wilson. Best practices for scientific computing. *PLOS Biology*, 12(1):1–7, 2014.

[29] L. Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program. Technical report, Karlsruhe Institute of Technology, 2000.

[30] H. Holm, A. Brodtkorb, K. Christensen, G. Broström, and M. Sætra. Evaluation of selected finite-difference and finite-volume approaches to rotational shallow-water flow. *Communications in Computational Physics*, 2019. [to appear].

[31] A. Brodtkorb. Simplified ocean models on the GPU. In *2018: Norsk Informatikkonferanse*, 2018.

[32] R. Singh, P. Wood, R. Gupta, S. Bagchi, and I. Laguna. Snowpack: Efficient parameter choice for GPU kernels via static analysis and statistical prediction. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 8:1–8:8, New York, NY, USA, 2017. ACM.

[33] J Price and S. McIntosh-Smith. Analyzing and improving performance portability of OpenCL applications via auto-tuning. In *Proceedings of the 5th International Workshop on OpenCL*, pages 14:1–14:4, New York, NY, USA, 2017. ACM.

[34] T. Falch and A. Elster. Machine learning-based auto-tuning for enhanced performance portability of OpenCL applications. *Concurrency and Computation: Practice and Experience*, 29(8), 2017.

[35] A. Brodtkorb, T. Hagen, and M. Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013.