One OpenCL to Rule Them All?

Romain Dolbeau
CAPS entreprise
Rennes, France
Email: romain.dolbeau@caps-entreprise.com

François Bodin
IRISA
Rennes, France
Email: bodin@irisa.fr

Guillaume Colin de Verdière
CEA, DAM, DIF
Bruyères-le-Châtel, France
Email: guillaume.colin-de-verdiere@cea.fr

***Abstract*—OpenCL is now available on a very large set of processors. This makes this language an attractive layer to address multiple targets with a single code base. The question on how sensitive to the underlying hardware is the OpenCL code in practice remains to be better understood. [1].**

This paper studies how realistic it is to use a unique OpenCL code for a set of hardware co-processors with different underlying micro-architectures. In this work, we target Intel® Xeon Phi™, NVIDIA® K20C and AMD® 7970™GPU. All these accelerators provide at least support for OpenCL 1.1 and fit in the same high-end version of accelerator technology.

To assess performance, we use OpenACC CAPS compiler to generate OpenCL code and use a moderately complex mini-application, Hydro. This code uses 22 OpenCL kernels and was tailored to limit data transfers between the host and the accelerator device.

To study how stable are the performance, we performed many experiments to determine the best OpenCL code for each hardware platform. This paper shows that, if well chosen, a single version of the code can be executed on multiple platforms without significant performance losses (less than 12%). This study confirms the need for auto-tuning technology to look for performance tradeoffs but also shows that deploying self-tuning/adaptive code is not always necessary if the ultimate performance is not the goal.

***Keywords*—GPGPU, Code tuning, Performance, Portability**

I. Introduction

OpenCL is a portable syntax to program many different hardware. Performance portability is a critical issue in many cases and it has not been addressed by OpenCL standard yet. This paper studies how critical this issue is and when do we need to use auto-tuning techniques and self-adapting codes. To answer this question, it is important to analyze the impact of relying on a single version of code for multiple targets of the same class of architectures. Performance portability is defined here by how close to the best performance is a given code. In other words, how much performance are we loosing when using a single code version for multiple hardware.

In this work we use three architectures: Intel Xeon Phi, Nvidia K20C, and AMD 7970 GPU. These three architectures are in the same class of high-end co-processor / accelerator technology. They all support OpenCL 1.1 or higher. The performance is analyzed using a moderately complex mini-application, Hydro. This application is complex enough to highlight many performance issues. Simple kernel benchmarks (e.g. matrix-multiply) were discarded because they tend to dramatize or amortize performance variation effects.

By using a realistic case, the contribution of the paper is to highlight the performance impact of choosing a unique code version which is intended to run on many different architectures. We try to understand if deploying a unique code base is realistic when dealing with very different architectures. In other words, we want to determine what is the cost in performance of having a unique code base and how to reach a tradeoff between performance and portability. Answering this question directly impacts application development complexity. Many users are ready to leave a few percents of efficiency on the table if they gain a lot in maintenance and code writing simplicity.

In this work we use CAPS Compiler OpenACC technology [1] to generate many OpenCL versions of the code. This compiler provides optimizing and code generation directives that allow to explore several variants of codes.

This paper is organized as follows. Section II provides an overview of the state of the art. Section III presents the main findings of this study.

II. Related Work

Applications developers are very keen to identify a portable language across parallel processors and accelerators technology. OpenCL [2] and OpenACC [3] have been shown to be available on many hardware and are considered as good alternatives on many architectures. However, portability cannot be completely traded over performance that motivates the use of parallel processors. For many developers, identifying the right tradeoff between performance and portability is the key engineering decision that strongly impacts on the cost and risk of a project. Most research on code tuning have been focusing on getting the best efficiency.

In [4] Komatsu et al. mainly compare OpenCL with CUDA on various GPUs. Their work illustrates the need for target specific optimization to obtain the best performance. The paper [5] analyzes performance portability of OpenCL on a set of platforms: CPU, FirePro, Tesla and Cell. This paper concludes that code tuning must be performed for each architecture and that auto-tuning techniques [6], [7], [8] may help to achieve some level of performance portability. Auto-tuning techniques come at a high development complexity and may induce a significant code size increase due to code versioning [9]. In many cases, the increase in code size is an issue as it depends on the size of the optimization space and the number of

[1]This work has been partially funded by the H4H-PerfCloud ITEA2 project http://www.h4h-itea2.org.

kernels. This problem is especially acute for large legacy codes with a lifespan of over 15 years (5 generations of computers).

In [10] S. Zhou et al. study the portability of the SOLAR code (used in climate and weather researches). Their results show that their OpenCL code can run on various platforms (e.g. CPU and GPU). They obtain speedup on all platforms. However, contrary to this study they do not address the issue of optimality of the code on all platforms. Performance portability is only stated as having speedup on all targets.

In [11] J. Shen et al. show, using 3 codes from the Rodinia benchmark, that OpenCL can reach similar levels of performance than OpenMP on multi-core CPUs. In this study we have left out CPUs to focus only on the use of accelerators.

[12], [4], [13] and [14] all propose that auto-tuning can be used as an appropriate method to achieve portability across platforms. However, these studies do not analyze the tradeoffs when considering only one version of the code only (to minimize code maintenance).

III. Performance Portability Tradeoff Analysis

This section reports the main experiments conducted to study performance and portability. To assess portability, we use three co-processors architectures exhibiting very different characteristics. We chose them in the same class of high-end hardware to avoid dramatizing the tuning effects. Furthermore, in practice it is not usual that the same application is developed for low-end hardware (e.g. embedded devices, tablets) and high-end devices (e.g. supercomputers). The platforms used are presented in Section III-A.

To perform the experiments, we use a moderately complex application, Hydro [15]. It is presented in Section III-B. Section III-C describes the code generation scheme and the various code tuning strategies used for this study. Section III-D reports the performance analysis performed in the paper.

A. Target Architectures

The three target architectures used for this study are described in the next paragraph. They belong to the same class of high-end accelerator technology and are expected to all be the target of a given HPC application. We chose not to expand to a wider set of device categories. Smaller devices such as the ARM- and GPU-based Carma[2] or CPU with integrated GPU devices such as AMD APUs might have overstated the experiment results, or lacked the memory to run our test case. OpenCL is available for CPU devices with SDK from Intel[3], AMD[4] or IBM, but the implicit separation of memory between CPU and accelerator in OpenACC and OpenCL does not suit CPU very well. It is possible to construct a fully zero-copy implementation of the Hydro code in OpenCL, but it would be quite different from the CPU/accelerator implementation and therefore not what we set out to study. Finally, a CPU version would have to be compared to a much more mature and conventional MPI/OpenMP implementation, an aspect also outside the scope of this paper.

[2]http://www.nvidia.com/object/seco-dev-kit.html

[3]http://software.intel.com/en-us/vcsource/tools/opencl-sdk-xe

[4]http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/

AMD Radeon HD7970 GPU
: This is the current top-of-the line consumer GPU from AMD, based on the Graphics Core Next architecture. The card offers 947 GFlops in double precision floating-point, and 264 GB/s of theoretical memory bandwidth. The computational resources are split among 32 "compute units", each with 4 "SIMDs", themselves with 16 scalar FPUs. The "Local Data Share" (i.e. the OpenCL local memory) is shared by all four SIMDs inside a compute unit, which is comparable to a CUDA multiprocessor. Double-precision performance is one fourth of the single-precision performance.

NVidia K20C accelerator
: The current compute-oriented NVidia chip code-named "Kepler". Peak performance in double precision is 1170 GFlops, with 208 GB/s of theoretical memory bandwidth (without ECC, which was activated on our system). Such a card uses 13 "SMX" for a total of 2496 "CUDA cores". Double-precision performance is one third of single-precision performance.

Intel Xeon Phi SE10P co-processor
: The newer accelerator architecture on the market, the Xeon Phi is essentially 61 conventional superscalar in-order x86_64 cores, each capable of running 4 threads via HyperThreading, with an extra 512 bits vector unit in each core. With a peak performance of 1070 GFlops double-precision, and an impressive 352 GB/s of bandwidth, the Xeon Phi also has an on-board Linux operating system that enables native execution of C and Fortran codes, a feature missing on the other two architectures. Double-precision performance is one half single-precision of performance.

Note that unlike the previous generations of AMD devices, the 7970 does not require vectorization of the code by the compiler. Both AMD7970 and NVidia K20C execute single-operation instructions, yet replicated several times on many small cores.The Xeon Phi, on the other hand, requires vectorization of the code to be able to exploit its wide SIMD registers. Fortunately, unlike previous Intel SIMD instruction sets such as SSE and AVX, the Xeon Phi has the ability to mask some of the lanes of the vector operations, simplifying the compilation of conditionals in vector mode.

The current OpenCL implementations (still a beta version for the Xeon Phi) on the three architectures each have specific requirements. The most obvious is the maximum work-group size; the AMD 7970 is limited to 256 work-items, where the Xeon Phi and K20C are both limited to 1024. The number of work-groups required is also very different. The K20C requires at least one work-group per SMX, or a minimum of 13, and can run up to 16 work-groups on each SMX (subject to many restrictions), i.e. 208. The AMD 7970 requires a work-group per compute unit, with a maximum of 10 (again with limitations), so between 32 and 320. The Xeon Phi executes each block in a single thread, requiring 240 work-group to fill all available hardware (one core is usually reserved for the on-board operating system). Further restrictions not detailed here exist for each system. On all hardware, if more work-group are available than the hardware can handle, they are dynamically scheduled when resources become available. Such scheduling is heavily hardware-assisted on two GPU-based architectures, but implemented in software on the Xeon Phi, leading to higher overheads for smaller kernels (at least in the beta release of Intel's OpenCL).

Finally, the OpenCL compiler currently available for the Xeon Phi vectorizes the code by grouping 16 consecutive work-items in the first local dimension in one (single-precision) or two (double-precision) vector registers. This constrains the local dimensions of work-groups for efficient vectorization.

B. Hydro Code

The Hydro code [15][5] is a mini-application built from the RAMSES [16][17][18][6] used to study large-scale structures and galaxy formation. It includes classical algorithms we can find in many applications codes for Tier-0 systems. This version preserves the original algorithms that solve compressible Euler equations of hydrodynamics, based on finite volume numerical method using an explicit second order Godunov scheme for Euler equations.

From a computational science perspective, the Hydro code exhibits several patterns that are interesting in a benchmark. First, the different functions are lightweight and fall in the following categories: data movement (gather / scatter of information) [e.g `gatherConservativeVars`], lightweight computations where the memory bandwidth will be a limiting factor [e.g. `constoprim`], heavyweight computations where the compute capabilities of the hardware are exercised [e.g. `riemann`]. Second, the code is not complicated by features like dealing with multiple materials. Even if this simplicity prevents us from studying the impact of indirections, it offers the potential for more in depth analysis of the different generated binaries. Finally, routines being simple, it allows fast transformations of the source code either to compare languages or to study different architectures while keeping the results invariant.

The code is about 5000 lines of code, and exists in several versions, including pure C (with MPI and OpenMP support) and OpenACC, the version we considered for this study. The accelerated version encompasses 22 OpenACC loop nests, generated into 22 OpenCL kernels using the CAPS compiler. The speedup on usual accelerators is quite high, being up to 4 times faster on the K20C compared to a dual-socket "Westmere" system using MPI and OpenMP. Communication is an insignificant part of the execution time in this version, as all computations are moved to the accelerator, with data transfers occurring only at the very beginning and very end of the computations. While communication is an integral part (and quite often an issue) for many applications, the required transfers will be the same for all accelerators with a non-integrated memory. Therefore the communication cost will depend mostly upon the PCIexpress performance and not the OpenCL generated code. Hence, we consider that communication is outside of the scope of this study.

[5]Hydro is available at https://github.com/HydroBench/Hydro.

[6]RAMSES is available at http://irfu.cea.fr/Phocea/Vie_des_labos/Ast/ast_sstechnique.php?id_ast=904

This application is particularly suited to our study since it contains a realistic workload of computations for an accelerator. We did not consider simple kernel benchmarks since their characteristics are not very balanced and they tend to dramatize the effect of tuning while not representing the complexity and diversity of real applications. By choosing a mini-application we were able to explore extensively the tuning of the code and so made sure we understood what is the highest performance to expect. Having such an extensive view on the performance profile of a code is usually impossible on most real applications due to their complexity and numerous execution profile.

The diverse kernels in the Hydro application have different characteristics, and have different optimal tuning values, even when considering only the work-group size. Some kernels are compute-intensive, others contain almost exclusively memory accesses. While most are fully data-parallel, there is also a reduction kernel taking up a significant amount of time.

The problem size chosen was a 4091×4091 grid, with a 1000 blocking size. The reason for using a prime number for the grid is to break any size-specific behavior that might bias the results for or against any hardware feature, such as unrealistically good memory alignment or exceedingly bad behavior of direct-mapped caches (the real code is run on arbitrary sized problems). This particular prime was chosen for being the largest that could fit inside the AMD 7970 memory restrictions using the chosen blocking size. The other two architectures are able to use larger test cases. The blocking size itself allows for larger problems to be cut in pieces that can be fitted in limited memory space; smaller size helps with larger problem sizes, but larger sizes help with performance on data-parallel hardware. A value of 1000 is a compromise that works reasonably well on all tested architectures.

C. OpenCL Code Generation Scheme

The OpenCL code is obtained using the CAPS OpenACC compiler [1] that is a source-to-source technology able to generate OpenCL codes from annotated C and Fortran codes. This compiler supports OpenACC directives [3] and a set of additional directives and compiler options for code tuning. These directives provide a powerful way to explore various OpenCL code. In the generator we used, the iteration space is projected on the OpenCL "NDRange" index space (see §3.2 in [2]), thus matching a single iteration of the innermost loop to a single OpenCL work-item. The work-items are grouped into work-groups whose "local size" can be set by either specialized directives, or by a default value that can be changed with a compiler option. This "local size" determines how many work-groups will be created for each iteration space, and is a very influential factor on performance in Hydro. In the CAPS compiler, the resulting kernel configuration is called a **"thread grid"**, a terminology inherited from CUDA.

Specifically, most loop nests are made by two or three perfectly nested loops, computing over a horizontal "slice" of the 2D space (i.e. the full width of the 2D domain but a dense subdivision of the height). The size of those slices is defined by the input parameters, but has no influence on the computations: slices are used to allow for optimizing memory and cache usage. Temporary arrays are the size of a slice rather than the size of the entire domain, so smaller slices use less memory. And because all computations on a single slice are done before moving to the next slice, the slice size can be optimized to block the data in a certain level of cache. When loop nests are converted into OpenCL kernels, the iteration space is projected on a two dimensional iteration space. The innermost loop, usually over the full width of a slice, is projected on the first dimension. The outermost loops are all projected on the second dimension. Those dimensions are used as the "global size" in OpenCL kernels. This is then subdivided into "work-groups" of configurable "local size". Each "work-item" is therefore responsible for a single innermost iteration of the original iteration space. This is a very basic generation scheme called "gridification", which is the simplest conversion from a perfectly nested loop nest to OpenCL. However, for code that do not share input data between iterations (where "local memory" could be used for explicit caching), it is quite efficient. The "gridification" scheme used ensure that the memory accesses executed by work-items will be "coalesced" (using the CUDA semantic of the word) on NVidia & ATI hardware, and "vectorizable" by the OpenCL compiler on the Xeon Phi, provided the "local size" is a multiple of 16.

To ensure that the simple code generation did not have a strong influence on the study, we compared the performance of the generated OpenCL codes to the native OpenCL version that was previously developed. The OpenCL generated code is as fast as the hand-written version on the Xeon Phi and slightly faster on the K20C. It should be noted that using CUDA code instead of OpenCL for the K20C does not improve performance on this code.

For this experiment, we have selected a set of thread grid configurations (See Table I). The entire set can be used on the Intel and Nvidia devices while some of them lead to non-functioning code on the AMD device. The work-group local sizes considered in this study where chosen amongst the one performing reasonably well. We discarded the ones leading to bad performance on all hardware or non-functional on the AMD7970.

D. Evaluation

1) Single local size: The first evaluation was the simple case where the user did not tweak the code generation, and simply used the default behavior of the tool. In this case, all loop nests are transformed into OpenCL kernels using the same "gridification" process, and use the same "local size". The default value for the compiler we used is 32×4 for the local size, labeled in bold in table I. Then the simplest of tuning was applied: change the default value from the compiler command line, but using the same value for all file (i.e. a trivial change in the build system).

Performance is reported as relative to the best performance obtained by this tuning process[7]. It is denoted *EfficiencyLoss*, or **E.L.** More formally we define the **E.L.** of a version of the code as $\frac{ExecTime - BestExecTime}{BestExecTime}$ where $BestExecTime$ is the best execution time of Hydro for the target and $ExecTime$ is the execution time for the given thread grid. Table I reports the **E.L.** for each version of the code. The WG column indicates the thread grid configuration used for all OCL kernels. Columns $AMD7970$, $K20C$, and $SE10P$ lists the **E.L.** of the code on the respective hardware. In these columns the boldface numbers show the best configuration for the corresponding hardware. Due to hardware constraints (see Section III-A for an explanation), the $AMD7970$ cannot run the 16×32 configuration. It is however displayed in the table because this is the best configuration for the $K20C$.

[7]Performance absolute values are not given here due to non-disclosure agreements

The $Avg.$ indicates the arithmetic mean of **E.L.** on the three targets. The $Max.$ column reports the maximum loss of performance for the given work-group configuration.

WG	$AMD7970$	$K20C$	$SE10P$	$Avg.$	$Max.$
16x4	6,70%	12,08%	23,56%	14,12%	23,56%
16x8	5,00%	5,75%	34,71%	15,16%	34,71%
16x16	7,54%	2,34%	38,44%	16,11%	38,44%
32x2	4,57%	22,20%	8,14%	11,64%	22,20%
32x4	2,95%	12,44%	10,96%	8,78%	**12,44%**
32x8	**0,00%**	5,89%	18,07%	7,99%	18,07%
64x1	9,64%	33,73%	3,54%	15,64%	33,73%
64x2	9,27%	20,38%	4,52%	11,39%	20,38%
64x4	4,64%	13,24%	5,78%	7,89%	13,24%
128x1	13,22%	27,31%	0,72%	13,75%	27,31%
128x2	7,53%	20,51%	2,15%	10,07%	20,51%
256x1	9,51%	28,56%	**0,00%**	12,69%	28,56%
16x32	N/A	**0,00%**	43,47%	21,74%	**43,47%**

TABLE I. **E.L.** OF THE CODE VARIANTS. LOWER THE BETTER. VALUE **0,00%** INDICATES THAT THE VARIANT REACHES THE BEST PERFORMANCE.

To choose the optimizing configuration, one may want to minimize the maximum penalty. The data in Table I shows that 32×4 gives the best tradeoff. It is not the best configuration for all hardware. On the contrary, using the best configuration for a given hardware produces a large performance degradation on the other ones. For instance, choosing the best configuration (256×1) for the Intel Xeon Phi SE10P will lead to a 29% loss of performance on the K20C. Using the K20C best configuration (16×32) is even worse since it leads to a loss of **43,47%** on the Xeon.

Figure 1 illustrates the correlation between the K20C and SE10P performance (lower the better). As it can be seen by increasing the first dimension of the work-groups the performance of the K20C decreases when the performance of the Xeon Phi improves. As a consequence, none of the target can be used as an "oracle" for the behavior of the others.

2) Per-kernel local size: Obviously, most users looking for performance will try to optimize each kernel independently, so as to achieve maximum overall performance for the application. We therefore measured the cost for each kernel independently, and then validated the overall performance by running a variant of the code where each kernel was set to its ideal "local size".

These variations are closely related to the type of workload: for example `gatherConservativeVars` does only a data movement using a gather operation whereas `riemann` does the bulk of the computations using a limited number of main memory accesses (favoring register usage). On a K20C, a larger configuration will benefit the first kernel by hiding memory access latencies while the high pressure on the registers in the second kernel favors smaller configurations. For instance, a high register usage per thread limits the number of threads that can run concurrently on a single SMX. As those threads are grouped into blocks and not independently run, and only an

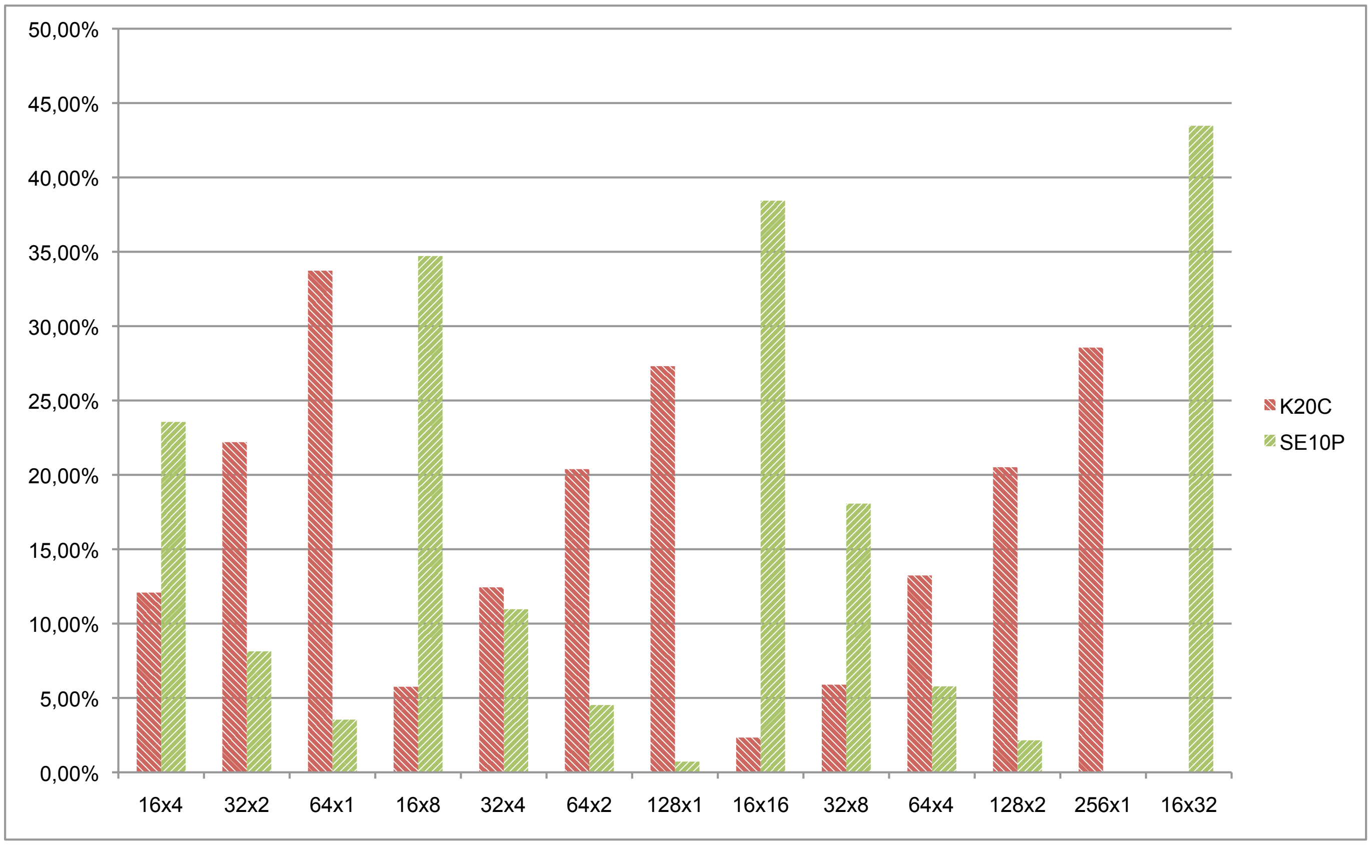

Fig. 1. Performance correlation between K20C and SE10P.

integer number of blocks can be run on a SMX, large block wastes resources by not effectively using all registers. On a K20C when using e.g. a hundred registers per thread, at most about 640 threads can run simultaneously in the 64K register file. Five blocks of 128 threads can be run on a SMX, but only two blocks of 256 threads thus wasting the last 128 available thread slots. Obviously such behaviors are hardware dependent even between two generations of GPUs from the same vendor. As expected, each hardware responded differently to his per-kernel tweaking.

The AMD7970 did not show much improvement, at 1.53% in our measure (compared to the best "local size" for this device). This was to be expected from our numbers: all kernels showed maximum speed when using 256 work-item per work-group (the maximum supported by the device), thus limiting the choice of improved kernels compared to the best case of 32×8.

The SE10P also did not show much improvement, at 1.41%. This time, the optimal number of work-item per work-group varied between kernels, but the shape was more consistent, with wide and shallow (i.e. high first number, low second number) work-group being usually the best. Only the kernel `riemann` was biased toward slightly narrower, deeper work-group, but only with a small difference. Other kernels are mostly memory-bound on the SE10P, and their preferred shape is likely more favorable to the memory subsystem.

Finally, the K20C showed an improvement of 8.39%, the only really significant improvement of the lot. The size and shape of the work-groups varied noticeably, with one kernel being fastest for 16×64 (`gatherConservativeVars`), another for 64×1 (`cmpflx`), and `riemann` being optimal at 16×8.Ten different sizes are used to attain optimal speed.

A surprising observation is that such specific optimizations do not necessarily worsen the situation for other hardware. Of course, when optimizing for the K20C or the SE10P, the code will no longer run on the AMD7970 as the work-group size used are too large. But the version optimized for the AMD7970 was not too bad, with an **E.L.** of 10.45% for the K20C, but of only 9.2% for the SE10P (the $BestExecTime$ for the metric is now the per-kernel optimized version), much better than in the first evaluation. Similarly, the version tuned for the K20C and the Phi are slightly less under-performing for the other device. Each kernel has a different behavior with regards to "local size" on each hardware, yet there is apparently some correlation. So this additional level of freedom in the choice of "local size" seems to reduce the $EfficiencyLoss$ for other hardware.

This observation allows to pick and choose "local size" to create a specific binary that will allow to minimize **E.L.** for a set of devices. For instance, if we limit ourselves to "local size" supported by the AMD7970, and then choosing for each kernel the "local size" that minimizes the sum of the execution time for all devices, then we obtain a device with an **E.L.** of 0.78%, 10.41% and 4.76% (against, $BestExecTime$ is the per-kernel optimized version) for the AMD7970, K20C and SE10P respectively.

IV. Conclusion

OpenCL provides a portable API to benefit from accelerator technology. Nevertheless, code tuning remains an issue when deploying a single code base on different platforms. If only one version of the code is to be used for all hardware, we show that it is important to look for a tradeoff between average efficiency and best performance. In the case exposed in this paper we demonstrate that performance portability is achievable if an efficiency-loss of 12% is considered as acceptable. Versions of the code tuned for maximal efficiency for a given hardware tend to strongly degrade performance on the other systems (up to 43% loss of efficiency) and may not even run.

This paper shows that, if ultimate performance is not the goal, performance portability can be achieved to an extent. As also shown in previous studies, when looking for tradeoffs, auto-tuning technology is a must. This work demonstrates that it is not always necessary to embed self-adapting codes that leads to runtime overheads and extra complexity. This study will be extended using more applications in order to better determine when self-adapting code is strictly necessary.

References

[1] CAPS entreprise, "CAPS compilers," http://www.caps-entreprise.com/products/caps-compilers/, 2012.

[2] Khronos OpenCL Working Group, "The opencl specification version 1.2," http://www.khronos.org/opencl/, 2008.

[3] OpenACC Consortium, "The openacc application programming interface," http://www.openacc-standard.org, 2011.

[4] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, "Evaluating performance and portability of opencl programs," in The Fifth International Workshop on Automatic Performance Tuning (iWAPT2010), 2010.

[5] S. Rul, H. Vandierendonck, J. D'Haene, and K. De Bosschere, "An experimental study on performance portability of opencl kernels," 2010.

[6] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in Inpar 2012, 2012.

[7] T. Lutz, C. Fensch, and M. Cole, "Partans: An autotuning framework for stencil computation on multi-gpu systems," ACM Trans. Archit. Code Optim., vol. 9, no. 4, pp. 59:1–59:24, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400718

[8] R. Miceli, G. Civario, A. Sikora, E. Cesar, M. Gerndt, H. Haitof, C. Navarette, S. Benkner, M. Sandrieser, L. Morin, and F. Bodin, "Autotune: A plugin-driven approach to the automatic tuning of parallel applications. proceedings para 2012: Workshop on state-of-the-art in scientific and parallel computing, helsinki, finland, june 10-13, 2012, springer verlag," in PARA 2012: Workshop on State-of-the-Art in Scientific and Parallel Computing, Helsinki, Finland, June 10-13, 2012, Helsinki, Finland, June 2012. [Online]. Available: http://eprints.cs.univie.ac.at/3595/

[9] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A multi-objective auto-tuning framework for parallel codes," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 10:1–10:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389010

[10] S. Zhou, J. Qiu, and K. A. Hawick, "Guest editor's introduction: Special section on challenges and solutions in multicore and many-core computing," Concurrency and Computation: Practice and Experience, vol. 24, no. 1, pp. 1–2, 2012.

[11] J. Shen, J. Fang, H. Sips, and A. Varbanescu, "Performance gaps between openmp and opencl for multi-core cpus," in Parallel Processing Workshops (ICPPW), 2012 41st International Conference on, 2012, pp. 116–125.

[12] 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 Comput., vol. 38, no. 8, pp. 391–407, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2011.10.002

[13] R. Weber, A. Gothandaraman, R. Hinde, and G. Peterson, "Comparing hardware accelerators in scientific applications: A case study," Parallel and Distributed Systems, IEEE Transactions on, vol. 22, no. 1, pp. 58–68, 2011.

[14] S. Seo, G. Jo, and J. Lee, "Performance characterization of the nas parallel benchmarks in opencl," in Workload Characterization (IISWC), 2011 IEEE International Symposium on, 2011, pp. 137–148.

[15] P.-F. Lavallée, G. Colin de Verdière, P. Wautelet, D. Lecas, and J.-M. Dupays, "Hydro," https://github.com/HydroBench/Hydro/tree/master/HydroC, 2012.

[16] R. Teyssier, "Cosmological hydrodynamics with adaptive mesh refinement: a new high resolution code called RAMSES," Astron. Astrophys., vol. 385, pp. 337–364, Nov. 2001. [Online]. Available: http://dx.doi.org/10.1051/0004-6361:20011817

[17] R. Teyssier, S. Fromang, and E. Dormy, "Kinematic dynamos using constrained transport with high order godunov schemes and adaptive mesh refinement," J. Comput. Phys., vol. 218, no. 1, pp. 44–67, Oct. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.jcp.2006.01.042

[18] S. Fromang, P. Hennebelle, and R. Teyssier, "A high order godunov scheme with constrained transport and adaptive mesh refinement for astrophysical magnetohydrodynamics," Astronomy and Astrophysics, vol. 457, no. 2, pp. 371–384, 2006.