An Experimental Study on Performance Portability of OpenCL Kernels

Sean Rul, Hans Vandierendonck, Joris D'Haene and Koen De Bosschere
Ghent University,
Dept. of Electronics and Information Systems,
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
{srul,hvdieren,kdb}@elis.ugent.be

Abstract

Accelerator processors allow energy-efficient computation at high performance, especially for computation-intensive applications. There exists a plethora of different accelerator architectures, such as GPUs and the Cell Broadband Engine. Each accelerator has its own programming language, but the recently introduced OpenCL language unifies accelerator programming languages. Hereby, OpenCL achieves functional protability, allowing to reduce the development time of kernels. Functional portability however has limited value without performance portability: the possibility to re-use optimized kernels with good performance. This paper investigates the specificity of code optimizations to accelerator architecture and the severity of lack of performance portability.

1. Introduction

Accelerator processors allow to execute computation-intensive programs with higher performance and with better energy-efficiency than general-purpose processors. As such, software developers are starting to exploit accelerators by off-loading computation-intensive tasks.

The best known and most widely available accelerators are graphical processing units (GPUs), which are available in every computer. But other accelerators such as the Cell processor [3], ClearSpeed [2] and Nallatech's Slipstream [1] are also commercially available. As maximum performance is a key reason for using these devices, programming them occurs at a very low abstraction level where architecture-specific features are exposed to the programmer. As such, each architecture is programmed using a different programming language, which leads to a real "Tower of Babel" experience: software developed for one accelerator architecture cannot be ported to a different architecture.

The OpenCL language[1] was introduced about 1 year ago to solve this situation. OpenCL is a C-based programming language that contains features of many accelerator architectures. As such, OpenCL allows to write software with functional portability: software written for one accelerator architecture should execute correctly on a different accelerator architecture. As optimizing performance is the key reason for using accelerators, functional portability has limited value without performance portability: software optimized for one accelerator should not be entirely re-optimized for a different accelerator. Without performance portability, software developers must still develop accelerator-specific versions of their code. OpenCL does not support performance portability.

The goals of this paper are (i) to study the specificity of code optimizations to the accelerator architecture and (ii) to evaluate the severity of lack of performance portability.

2. Method

We use the OpenCL language as a means for obtaining functional portability. The OpenCL language divides the computational workload into thread-blocks which can have up to three dimensions, similar to the CUDA programming model. These thread-blocks are then dispatched onto the execution units of the accelerator. Due to the similarity of program structure between the CUDA and OpenCL languages, the translation process is relatively straightforward.

This work uses Parboil[2] benchmarks to study performance portability. The Parboil benchmark suite consists of 7 kernels expressed in the CUDA language, specific to NVIDIA GPUs [5]. These benchmarks are hand-translated into OpenCL and 3 optimization parameters are exposed in order to auto-tune the kernels. These parameters correspond to the degree of loop unrolling, the use of vectorization or not (4-element float vectors) and the number of threads in a thread block.

We perform measurements on four architectures: an Intel Core i7 720-QM (a 1.60 GHz quad-core), an NVIDIA Tesla

[1]http://www.khronos.org/opencl/

[2]http://impact.crhc.illinois.edu/parboil.php

Table 1. Properties of the accelerator architectures in the experiments. SIMD widths shown assume floating-point values are held.

Accelerator	Core Freq.	Processors	DLP	Memory BW	Max. Dim XxYxZ / Max. Threads
CPU Intel i7 720-QM	1.6 GHz	4 (+4 SMT)	4-way SIMD	21 GB/s	1024x1024x1024 / 1024
Tesla c1060	1.3 GHz	240	8-way SIMT	102.4 GB/s	512x512x64 / 512
ATI FirePro V8700	750 MHz	160	5-way VLIW	108.8 GB/s	256x256x256 / 256
Cell PS3	3.2 GHz	6	4-way SIMD	25.6 GB/s	256x256x256 / 256[1]

[1] While the maximum number of thread blocks is reported as 256, the current Cell implementation requires that the threads per block is at most 1.

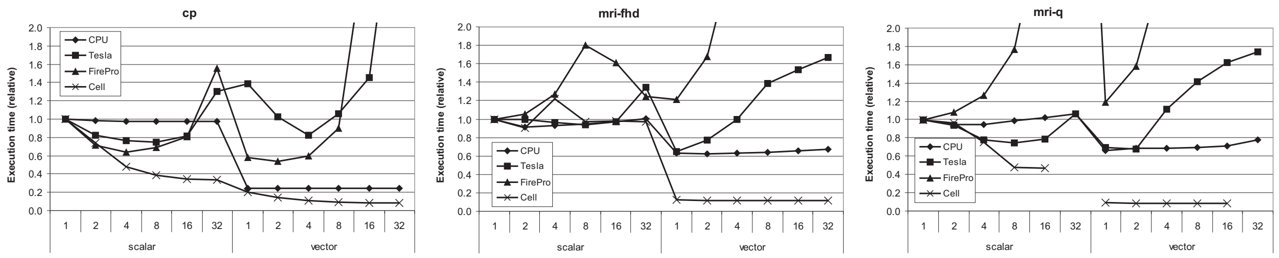

Figure 1. Impact of loop unrolling and vectorization on the performance of kernels on different architectures. Execution time is normalized per architecture with respect to the baseline kernel (no vectorization, loop unrolling factor 1). Missing data points correspond to compilation errors.

c1060, an ATI FirePro V8700 and a Sony/Toshiba/IBM Cell Broadband Engine (Playstation 3). These architectures have quite diverse properties regarding the trade-off between single-thread performance, number of processor cores and micro-architectural pipeline organization (Table 1). Furthermore, while some architectures use SIMD (single instruction multiple data) to enable power-efficient data-parallelism, NVIDIA GPUs use SIMT (single instruction multiple thread) to limit the instruction fetch overhead in data-parallel computations.

The architectural differences are so large that they inevitably percolate up to the programming language. The last column shows the maximum dimensions of thread blocks, the building block of OpenCL kernels. It is clear that thread block sizes must be tuned to the architecture.

3. Evaluation

This section presents our measurement results on the cp, mri-fhd and mri-q kernels. Results for the tpacf and rpes benchmarks are pending. The sad and pns benchmarks were excluded from this study as the first one is too small to perform meaningful optimization and the latter one did not work correctly.

Figure 1 shows the performance of the kernels when varying the loop unrolling factor and when applying vectorization or not. The thread block size is 128 for the Tesla and CPU measurements while it is necessarily 1 on Cell. Figure 2 shows the impact of the thread block size on performance. Only a single data point is possible for the Cell processor.

The impact of the optimizations varies hugely between architectures, confirming the need for architecture-specific optimization. Optimizations interact with each other: e.g.,

Table 2. Execution time in seconds of kernels on each accelerator when optimized for a particular accelerator.

		Optimized for ...			
Kernel	Accelerator	CPU	Tesla	FirePro	Cell
cp	CPU	12.62	50.50	50.54	12.62
	Tesla	2.49	0.48	0.49	2.49
	FirePro	164.60	1.83	1.71	164.60
	Cell	8.80	39.75	49.49	8.80
mri-fhd	CPU	2.90	2.93	4.66	3.11
	Tesla	0.38	0.31	0.49	0.81
	FirePro	1.84	1.3	1.10	n/a
	Cell	1.20	1.26	10.22	1.14
mri-q	CPU	5.78	5.99	8.74	6.22
	Tesla	0.80	0.77	1.15	1.87
	FirePro	1.49	1.98	1.25	n/a
	Cell	2.28	2.08	24.00	1.91

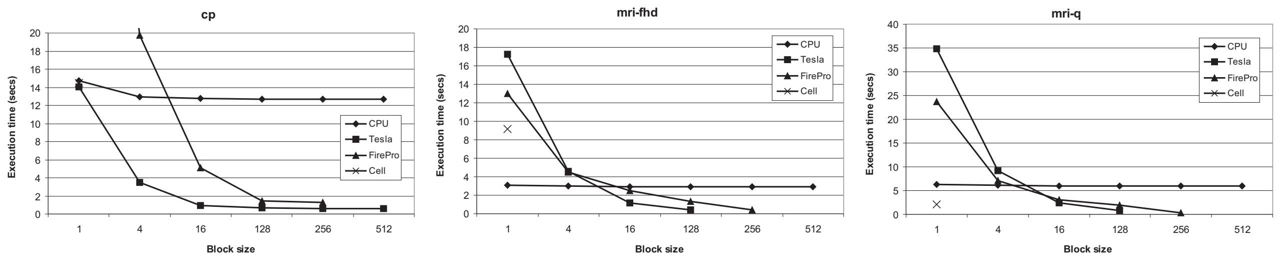

Figure 2. Impact of thread block size on kernel performance for vectorization and loop unrolling 2.

too much loop unrolling degrades performance on Tesla. When adding vectorization, this degradation becomes even larger and the optimal unrolling factor becomes smaller. Also, while the Tesla and FirePro respond largely similarly to the optimizations, the FirePro is much more sensitive to parameter values for the optimizations and it has different optimal parameter values than the Tesla. Thus, accurate exploration of the parameter space is essential to maximize performance.

Optimizations may be exclusively applicable to some architectures. E.g., when optimizing for CPUs, loop unrolling is not an important optimization for these kernels, but loop unrolling is crucial for good performance on Cell. For Tesla, the thread block size is the most important parameter. Failing to expose a particular optimization through an auto-tunable parameter may thus lead to sub-optimal performance on some architectures. In order to maximize performance portability, the developer must be careful to consider optimizations relevant to *all* target accelerators.

Table 2 shows the performance obtained on each accelerator when selecting the optimal optimization sequence for a particular accelerator. It is striking how sensitive performance is to the optimization sequence. Especially the cp kernel sees up to a factor of 4.5 performance difference on the Cell if optimizing for the wrong architecture. We conclude that performance is by no means portable across architectures.

4. Related Work

This work is concerned with the ease of programmability of accelerators, in particular with the aspect of obtaining portability of code across accelerators. Auto-tuning is an important technique to optimize the performance of software to architectural properties [6] that may be used to obtain performance portability. Hereto, code generators are constructed to create many different versions of an algorithm. As such, auto-tuning requires significant efforts and is kernel-specific [4, 7].

5. Conclusion

The recently introduced OpenCL language for accelerator processors promises functional portability, but cannot deliver performance portability.

This paper studies the sensitivity of performance on accelerators to the best optimization sequence. We find that optimizations have a strongly different impact on different architectures. In particular, each architecture is extremely sensitive to one of the optimizations, while these optimizations have much lesser impact on the other architectures.

We conclude that special measures must be taken to obtain performance portability. Auto-tuning is a set of techniques that may help reach this goal. Our results indicate however a vulnerability of auto-tuning: as optimizations are architecture-specific, they may be easily overseen or considered "not relevant" when developing on different architectures. As such, performance of a kernel must be verified on all potential target architectures. We conclude that, while being technically possible, obtaining performance portability remains time-consuming, regardless of the functional portability obtained by a language such as OpenCL.

References

[1] A. Cantle and R. Bruce. An Introduction to the Nallatech Slipstream FSB-FPGA Accelerator Module for Intel Platforms. White paper, http://www.nallatech.com, Sept. 2007.

[2] T. R. Halfhill. Floating point buoys ClearSpeed. *Microprocessor Report*, page 7, Nov. 2003.

[3] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, 2005.

[4] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, 2004.

[5] nVidia CUDA programming guide, version 3.0, Feb. 2010.

[6] C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27:2001, 2000.

[7] S. W. Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Departement, University of California, Berkeley, Dec. 2008.