Improving Performance Portability in OpenCL Programs

Yao Zhang, Mark Sinclair II, and Andrew A. Chien
Department of Computer Science
University of Chicago
{yaozhang, msinclair94, achien}@cs.uchicago.edu

Abstract. We study the performance portability of OpenCL across diverse architectures including NVIDIA GPU, Intel Ivy Bridge CPU, and AMD Fusion APU. We present detailed performance analysis at assembly level on three exemplar OpenCL benchmarks: SGEMM, SpMV, and FFT. We also identify a number of tuning knobs that are critical to performance portability, including threads-data mapping, data layout, tiling size, data caching, and operation-specific factors. We further demonstrate that proper tuning could improve the OpenCL portable performance from the current 15% to a potential 67% of the state-of-the-art performance on the Ivy Bridge CPU. Finally, we evaluate the current OpenCL programming model, and propose a list of extensions that improve performance portability.

1 Introduction

The recent development of OpenCL [2] provide an open, portable C-based programming model for highly parallel processors. In contrast to NVIDIA's proprietary programming API CUDA [17], a primary goal of OpenCL is portability across a diverse set of computing devices including CPUs, GPUs, and other accelerators [6]. Although the initial focus of OpenCL is to offer functional portability, performance portability is a critical feature for it to be widely adopted. However, it still remains unclear and lacks a systematic study on how performance-portable OpenCL is across multicores and GPUs, given that it is heavily influenced by a GPU-centric programming model, CUDA.

In this work, we study the performance portability of OpenCL programs (SGEMM, SpMV, and FFT) across diverse architectures including NVIDIA GPU, Intel Ivy Bridge CPU, and AMD Fusion CPU. The central questions we would like to answer are: (1) what is the gap between the portable performance of single-source OpenCL programs and the optimized performance of architecture-specific programs? (2) How much of this gap could be closed by certain tuning knobs that adapt OpenCL programs to diverse architectures? And what are those tuning knobs? (3) How should the OpenCL programming interface be extended to better incorporate these tuning knobs?

With these questions in mind, we make the following contributions in this work. First, our study found that the portable performance of three OpenCL programs is poor, generally achieving a low percentage of peak performance (7.5%–40% of peak GFLOPS and 1.4%-40.8% of peak bandwidth). Second, we identify a list of tuning knobs including thread-data mapping, parallelism granularity, data layout transformation, data caching, and we demonstrate that they could improve the OpenCL portable performance from the current 15% to a potential 67% of the state-of-the-art performance. Third, we evaluate current Intel and AMD OpenCL CPU compilers, particular on their features of vectorization, multithreading, and thread aggregation. Fourth, we evaluate the OpenCL programming interface and propose potential extensions on parallelism and data abstractions for performance portability.

The rest of the paper is organized as follows. Section 2 describes our test platform and selected benchmarks. Section 3 presents our experiment results on the portable performance, as well as an evaluation of compiler quality. Section 4 identifies performance critical tuning knobs and demonstrates their performance impacts. Section 5 evaluates the OpenCL programming interface, and proposes a performance portable programming framework. Section 6 discusses related work. Section 7 summarizes and describes future work.

2 Experiment Setup

2.1 Test platform

Tab. 1: Processor specifications

Processor	Cores	Vector width (32 bits)	Freq (GHz)	Peak GFLOPS	LLC size	Bandwidth (GB/s)
Fermi GPU	14	32 (ALUs)	1.15	1030.4	768 KB	144
APU CPU	4	4	2.9	92.8	4 MB	29.9
APU GPU	5	16×5 (VLIW)	0.6	480	256 KB	29.9
Ivy Bridge CPU	4	8	3.4	217.6	6 MB	25.6
Ivy Bridge GPU	16	8	1.15	166.4	6 MB	25.6

Our test processors include NVIDIA Tesla C2050 (Fermi), Intel Core i5 3570K CPU (Ivy Bridge), and AMD A8-3850 APU. Table 1 summarizes the specifications for these processors including the integrated GPUs. Our software platforms use NVIVIA CUDA 4.2 for Ubuntu 12.04, AMD Catalyst 12.4 driver for OpenCL 1.2 and Ubuntu 12.04, and Intel SDK for OpenCL 1.1 and Windows 7.

2.2 Benchmarks

We select three programs from the SHOC OpenCL benchmark suite [8] as our case studies: Single Precision General Matrix Multiply (SGEMM), Sparse Matrix Vector multiply (SpMV), and Fast Fourier Transform (FFT), which represent a range of easy to difficult, but computationally important benchmarks. Table 2 summarizes their computation characteristics and performance bottlenecks.

SGEMM is a sub-routine in the Basic Linear Algebra Subprograms (BLAS) library, and its performance behaviors are representative of other level-3 matrix-matrix operations [12]. The SHOC program is based on the implementation

Tab. 2: Benchmark characteristics.

Benchmarks	Compute complexity	Compute-to-memory ratio	Bottleneck
SGEMM	$O(N^3)$	$O(N)$	Compute-limited
SpMV	$O(N)$	$O(1)$	Bandwidth-limited
FFT	$O(NlogN)$	$O(logN)$	Compute-limited

developed by Volkov and Demmel [24]. The major improvement introduced by them is to use a block-based algorithm and an improved loop order so that only one input sub-matrix needs to be cached instead of two. Choosing an appropriate block size is critical to the SGEMM performance.

The SHOC SpMV routine is adapted from the version developed by Bell and Garland [4]. It is well known that SpMV is memory-bound, and a compact storage format is critical to its performance. Bell and Garland experimented with a number of compact formants and discovered the ELLPACK format [18] generally performs best on GPUs. This format could guarantee continuous memory access to matrix entries by adjacent threads and thus maximize the bandwidth utilization. We will use the ELLPACK format and its column-major and row-major variants for performance experiments.

The SHOC FFT routine is based on the version developed by Volkov and Kazian[25]. The program is hard-coded for processing many 512-point FFTs. The major optimization exploits the Cooley-Tukey algorithm [6] to decompose a 512-point FFT to many 8-point FFTs and process them by individual threads in registers, instead of processing a large 512-point FFT by many threads collectively in the slower on-chip scratchpad memory. A vectorization-friendly data layout and efficient twiddle factor calculation are critical performance factors.

3 OpenCL Portable Performance

In this section, we study the portable performance of the three OpenCL benchmarks for a diverse set of processors. We will also investigate the causes of the gap between the portable performance and optimized performance.

3.1 SGEMM

Figure 1a shows the nomalized SGEMM performance. The benchmark is not tailored to any of our test processors, as it was originally written in CUDA for NVIDIA G80/GT200 GPUs [24], and later ported to OpenCL in SHOC. Still, it reaches 40% of the peak Tesla C2050 (Fermi) performance, which is much higher than that of other processors, but lower than the reported 60% for the previous generation GT200 GPU [24]. Since the Fermi GPU significantly increases the hardware resources of ALUs, registers and scratchpad memory, it may need a larger sub-matrix size to achieve a higher GPU utilization.

The major inefficiency of the APU GPU (integrated) comes from the low utilization of its VLIW ALUs. Our examination of the assembly code reveals that only an average of 2.4 slots of the total 5 VLIW slots are used, and only 38% of the total dynamic instructions are compute instructions, which together bound the performance to $\frac{2.4}{5} \times 38\% = 18.4\%$, close to our measured 14.6%. We also test a SGEMM program in the AMD APP SDK, which achieves a performance of 329 GFLOPS, 68.5% of the peak, thanks to its customized vector operations. The low performance of the APU CPU is mainly due to two factors: (1) the AMD CPU compiler does not support vectorization, and (2) the expensive context switching between threads at synchronization points.

The Intel CPU compiler does a better job on supporting vectorization and thread aggregation (serialize threads to avoid unnecessary thread synchronizations), and thus achieves a higher percentage of the peak performance (13.5% vs. 7.5% for the AMD CPU compiler). The OpenCL program uses a column-major data layout, which favors GPUs by preserving the inter-thread locality, instead of the intra-thread locality favored by the CPUs. This is why the Ivy Bridge CPU performance decreases for larger matrices, which demand better locality to reduce the bandwidth requirement.

3.2 SpMV

We generate our random test sparse matrices of various sizes, with 1% non-zeros. Figure 1b shows the SpMV performance and bandwidth utilization, which we define as the ratio of the effective bandwidth to the peak bandwidth. The effective bandwidth is calculated as

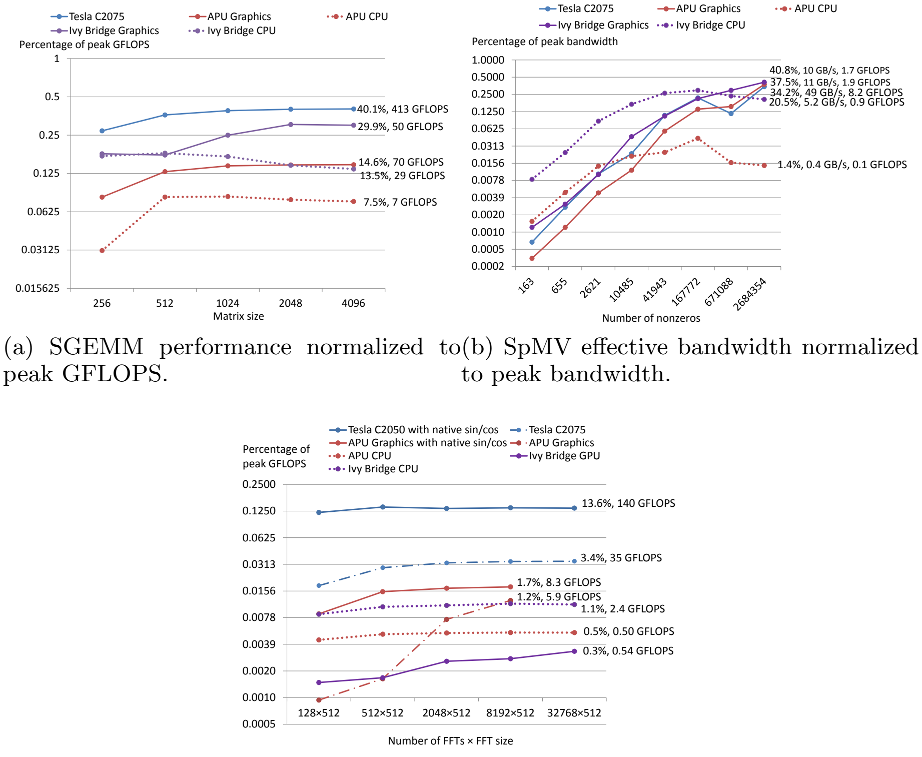
$$\frac{3 \times NumberNonzeros \times 4Bytes}{ProgramRuntime}$$

where 3 is the number of reads for a matrix entry, a column index, and a vector entry, and 4 bytes is the size for a 32-bit floating point or integer value. The performance is memory-latency-bound for small matrices, and gradually becomes bandwidth-bound as the matrix size increases. Due to the random access to vector entries, the bandwidth utilization is low on all processors. The Ivy Bridge CPU performance is higher than the integrated GPU performance for smaller matrices, mainly thanks to the L1–L2 cache. However, because of the poor locality of the column-major data layout on the CPU, the CPU performance drops as input matrix becomes too big to fit into the cache; the Ivy Bridge integrated GPU performance is not affected, because the program uses a GPU-friendly column-major data layout. The APU processor shows a similar performance trend.

3.3 FFT

We use $5Nlog_2N$ (radix-2 FFT arithmetic complexity) as the number of flops for a N-point FFT, as a convenient and normalized way to compare the performance of different FFT algorithms, as suggested in the paper by Volkov and Kazian [25]. Figure 1c shows the normalized FFT performance. Both Tesla C2050 and APU integrated GPU support native sin/cos instructions, which enables a speedup of $2 - 8\times$ over the program using software-emulated sin/cos calculations. The program defines a data type for complex numbers and uses a array-of-structure

(AoS) layout with real and imaginary parts of complex numbers stored in an interleaved way. In this SHOC FFT program, this interleaved layout is fine for GPUs, but unfriendly for CPUs, because the CPU vectorization is at the butterfly level, instead of the 8-point-FFT level in the GPU case. Both AMD CPU and Ivy Bridge CPU show low performance. The Intel OpenCL CPU compiler chooses not to vectorize the code, because of the interleaved data layout, while the AMD compiler does not support vectorization yet.



(a) SGEMM performance normalized to peak GFLOPS.

(b) SpMV effective bandwidth normalized to peak bandwidth.

(c) FFT performance normalized to peak GFLOPS.

Fig. 1: SGEMM, SpMV, and FFT performance normalized to the peak performance of each processor.

3.4 Summary

For the three OpenCL benchmarks across multicores and GPUs, the portable performance is generally low (7.5%–40% of peak GFLOPS and 1.4%-40.8% of peak bandwidth). In addition to compiler support, we identified the major causes of low performance, related to submatrix size, thread-data mapping, data layout, and native sin/cos support. Next, we show how portable performance can be significantly improved by tuning.

4 Performance Tuning

In this section, we will discuss the motivation and methodology of performance tuning, and show it could significantly improve the portable performance.

4.1 Problem Statement

Multicores and GPUs have different architecture features and configurations, and thus demand different program optimizations. However, current single-source OpenCL programs lack the ability to adapt. A number of architecture features impact programmers' tuning decisions on thread-data mapping, data layout, tiling size, and so on. These features include core types (complex ILP cores vs. simple highly-threaded cores), vectorization style (explicit SIMD vs. implicit SIMT [17]), core count, vector width, cache types (hardware vs. programmer-managed), cache size (kilobytes vs. megabytes), and bandwidth. The goal of performance tuning is to make the best choices to map a program to architecture features.

4.2 Methodology

Tab. 3: Tuning knobs and their settings for three benchmarks.

Benchmark	Tuning knob	Setting
SGEMM	Tile size	2×2
		......
		128×128
	Data layout	Row-major
		Col-major
	Prefetching/caching	Enabled
		Disabled
SpMV	Thread-data mapping	Interleaved
		Blocked
	Data layout	Row-major
		Col-major
FFT	Share sin/cos calculations	Enabled
		Disabled
	Data layout	SoA
		AoS

We adopt a systematic approach for performance tuning by summarizing all potential program optimization aspects as tuning knobs, which form a high dimensional optimization space. We explore this optimization space by experimenting with the settings of these tuning knobs. Table 3 summarizes all the tuning knobs and their settings for three benchmarks. In the following subsections, we will explain why these tuning knobs might be critical performance factors.

Tiling size Tiling is an effective technique used by block-based algorithms (e.g. SGEMM) to increase the cache reuse and thus the compute-to-memory ratio for bandwidth-limited programs. An optimal tiling size depends on multiple architecture features including bandwidth, cache size, core count, vector width, and processor frequency. A perfect size will balance between the effective cache reuse (large tiles preferred) and sufficient parallelism (small tiles preferred). GPUs usually prefer smaller tiling sizes because of limited on-chip scrachpad memory and massively parallel hardware, while CPUs often prefer bigger tiling sizes because of large cache and fewer hardware parallelism.

Data layout Data layout plays an important role in program performance optimization. GPU threads are lightweight and work in a tightly coupled, synchronized fashion in thread groups. It is highly desirable for adjacent threads in a group to access adjacent data in memory to maximize the bandwidth utilization. As a result, GPUs usually favor column-major data layout for inter-thread data locality. On the other hand, CPU threads are more independent and have a larger working set. They usually favor row-major data layout for intra-thread locality, and a column-major layout will result in inefficient strided memory access. Both CPUs and GPUs favor the SoA layout for vectorized operations. There are tricky exceptions where the AoS layout could be efficient on GPUs, and the SHOC FFT benchmark is an example, as discussed in Section 3.3.

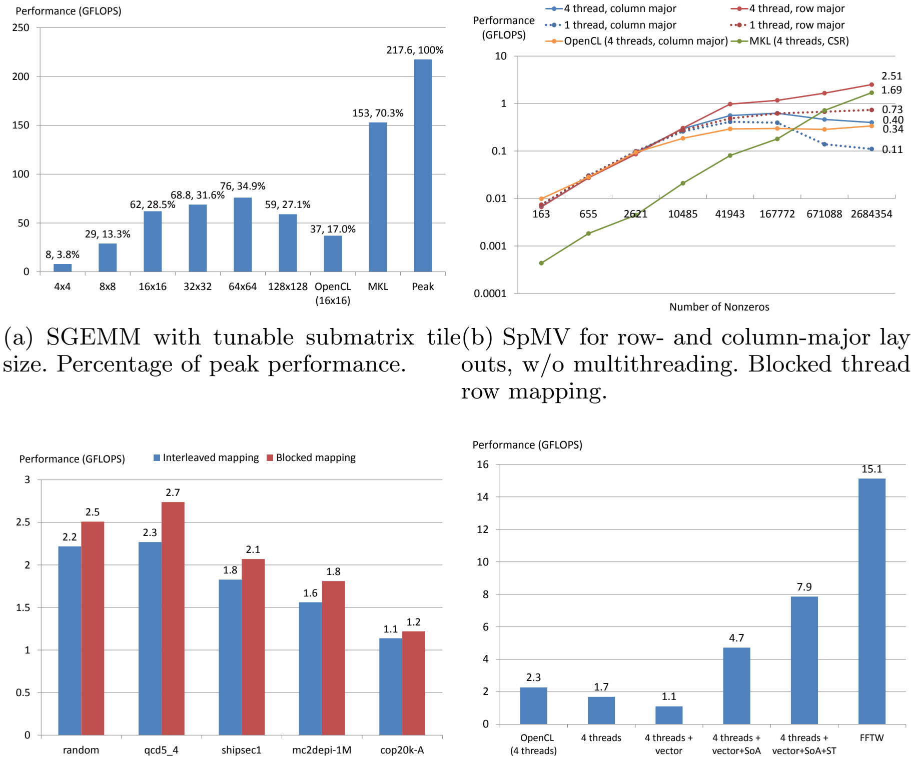
Caching and prefetching GPUs use programmer-managed scratchpad memory (as well as hardware L1 and L2 caches for post-Fermi GPUs) for data caching, while CPUs use hardware-managed cache. For CPUs, OpenCL currently simply treats arrays in scratchpad memory as the ones in external memory, in order to ensure the program correctness. The extra loads and stores to such emulated scrachpad memory for CPUs may have a performance penalty. However, the prefetching effect of such loads can help the performance, as shown in Section 4.3.

Thread data mapping CPUs and GPUs have a two-level parallelism structure with cores and vector units. However, many applications show multiple levels of parallelism (e.g. SGEMM has three levels of parallelism: sub-matrix, row, and element). It is desirable to have the ability to flexibly map the cross-level parallelism to the two-level architectures to maximize the hardware utilization. Another option is to choose between blocked and interleaved thread-data mapping. GPUs prefer interleaved mapping (adjacent data mapped to adjacent threads) for intra-thread locality, while CPUs prefer blocked mapping (adjacent data are mapped to a single thread) for intra-thread locality, because CPU threads are more independent and often have their own L1/L2 cache.

Operation-specific tuning Different generations of CPUs and GPUs may support a different set of hardware intrinsic instructions such as trigonometric, logarithmic, and thread coordination (atomic, sync, fence, etc.) operations. To avoid expensive software emulation, programs should try minimizing the use of intrinsic functions for the architectures that do not support them.

4.3 Performance Tuning Results

In this section, we will present our performance tuning results. Although we use the Intel CPU for experiments, we expect similar tuning benefits on other processors. To experiment with different optimizations, we port the baseline OpenCL programs to OpenMP, so that we can control the mapping of parallelism to the hardware, and if vectorization or multithreading is applied. For all experiments, we use Intel Core i5 3570K CPU (Ivy Bridge) and Intel C++ Compiler XE 13.0, which supports both AVX vectorization and OpenMP multithreading. All experiments are performed with single precision floating point arithmetic.



(a) SGEMM with tunable submatrix tile size. Percentage of peak performance.

(b) SpMV for row- and column-major layouts, w/o multithreading. Blocked thread-row mapping.

(c) SpMV with interleaved and blocked thread-row mapping for various matrices.

(d) FFT with various optimizations. SoA: structure of arrays. ST: shared twiddle factors.

Fig. 2: Performance of SGEMM, SpMV, and FFT, with tuning knobs incorporated on the Ivy Bridge CPU.

SGEMM *Tiling size* Figure 2a compares the performance of the original OpenCL program, the ported OpenMP program with tunable sub-matrix size, and the Intel MKL 10.2 SGEMM routine. Our ported OpenMP program is auto-vectorized by the Intel compiler using 8-wide AVX instructions on 32-bit floating point numbers. An optimal sub-matrix size of 64×64 doubles the performance of the original OpenCL program which has a hard-coded sub-matrix size of 16×16.

Caching and prefetching On the CPU, the use of scratchpad memory (which caches one input sub-matrix) is emulated by external memory. We first thought the extra copies to and from such emulated scrachpad memory would have a performance penalty. However, it turns out it even provides a slight performance improvement, most likely due to the prefetching effect of such extra load.

Data layout We experiment with both the column-major and row-major layout. Although the column-major format introduces strided memory access and is bad for cache performance, the two-dimensional block-based algorithm minimizes its performance penalty by caching the sub-matrices and making better use of the cacheline data brought in by each strided access. With a sub-matrix size of 16×16, a row of 16 32-bit values could use the entire cacheline of 64 KB brought in by a strided access. As a result, the row-major layout only offers slight performance advantage over the column-major layout.

Comparing with the state-of-the-art performance Our tuned performance is is still low compared with the MKL routine (Figure 2a). By comparing their assembly code, we find that the vectorized section of our ported OpenMP program is not as efficient as that of the MLK routine, in that it requires two extra data shuffling and replacement instructions per multiply-add.

SpMV *Data layout* We experiment with both the row-major and column-major ELLPACK format. Using four threads and the row-major layout, we achieve the best performance, which is 7.4× higher than that of the SHOC benchmark as shown in Figure 2b. Although SpMV is a memory-bound program, we find that the performance scales well with the number of threads for large inputs. This is most likely because more threads are able to use the L1 and L2 caches in more cores and thus reduce the bandwidth requirement. For small matrix sizes, data layout and multithreading do not help with the performance, because the program is memory-latency-bound and all data could fit into cache. Data layout or multithreading starts to make a performance difference at the point where the total matrix data and index size$(2 \times 41943 Nonzeros \times 4 Bytes/1024 = 327KB)$ exceeds the 256 KB of L2 cache.

Thread data mapping We experiment with interleaved and blocked thread-row mapping schemes for one random matrix plus four matrices from a collection of sparse matrices from various scientific and engineering applications, which is also used in prior work by others [7] [4]. The blocked mapping is always faster than the interleaved mapping by 7% to 21% (Figure 2c).

Comparing with the state-of-the-art performance The Intel MKL library does not support ELLPACK. We test its SpMV routine in compressed sparse row (CSR) format. It has a high startup cost for small matrices and is 33% slower than our row-major ELLPACK program for the largest matrix (Figure 2b).

FFT *Data layout* We experiment with both the SoA and AoS layout. As shown in Figure 2d, the ported program with the same AoS data layout achieves comparable performance to that of the OpenCL FFT benchmark. The auto-vectorized program by the Intel compiler even hurts the performance. However, the SoA layout is suitable for vectorization and makes the program 2× faster than the OpenCL benchmark.

Operation-specific tuning Calculating twiddle factors consumes a considerable amount of compute time, as the Intel CPU does not have hardware arithmetic for sin/cos functions. The Cooley-Tukey algorithm decomposes a 512-point FFT to 8 64-point FFTs, which share the same set of twiddle factors. Based on this observation, we further improve the program efficiency by cutting 8 times of redundant twiddle factor calculations to one time per 8 64-point FFTs. This speeds up the program by another 68% in addition to the data layout optimization.

Comparing with the state-of-the-art performance Our tuned program is still 50% slower than the state-of-art FFTW library. There are two improvement oppurtunities. First, we could speedup the twiddle factor calcuation by using lookup tables and the symmetric property of sin/cos functions. Second, currently only half of the 8-wide AVX vector unit are utilized, limited by the four concurrent butterflies in the radix-8 algorithm. A radix-16 algorithm will fully utilize the vector unit. Another option is to vectorize over 8 FFTs rather than over butterflies in a single FFT. This opition is used on the GPU, thanks to convinient data shuffling provided by the crossbar interconnect of scrachpad memory. On CPUs, however, this will involve major data shuffling overhead.

4.4 Summary

Tab. 4: Summary of the optimal settings of tuning knobs for the Ivy Bridge CPU. The improvement is compared with the baseline OpenCL programs.

Programs	Optimal knob settings	Improvement
SGEMM	64 × 64 tile size	2×
	Row-major layout	Insignificant
	Prefetching/caching	Insignificant
	Total	2×
SpMV	Row-major (small input)	Insignificant
	Row-major (larger input)	6.2×
	Blocked thread mapping	1.2×
	Total	7.4×
FFT	SoA layout	2.0×
	Minimize expensive sin/cos	1.7×
	Total	3.4×
Average	Total	4.3×

We have explored the performance optimization space formed by various tuning knobs, and demonstrated a large room of performance tuning for three benchmarks. Table 4 summarizes all the optimial setting of these tuning knobs for the Ivy Bridge CPU. On average, the optimized programs perform 4.3× faster. Another major observation is that the same optimizations may have drastically different performance impacts for different programs and input sizes. For example, the SpMV performance is much more sensitive to the row-major layout optimization than the SGEMM performance.

One primary goal of this paper is to investigate the gap between the "current" portable performance of single-source OpenCL programs and the performance of state-of-the-art programs with architecture-specific optimizations. We also want to quantify how much of this gap could be closed by "potential" portable performance, which is the performance achieved with our tuning knobs incorporated (summarized in Table 4). Figure 3 shows the current and potential portable performance of SGEMM, SpMV, and FFT on the Ivy Bridge CPU, normalized against the performance of today's state-of-the-art programs (the MKL SGEMM

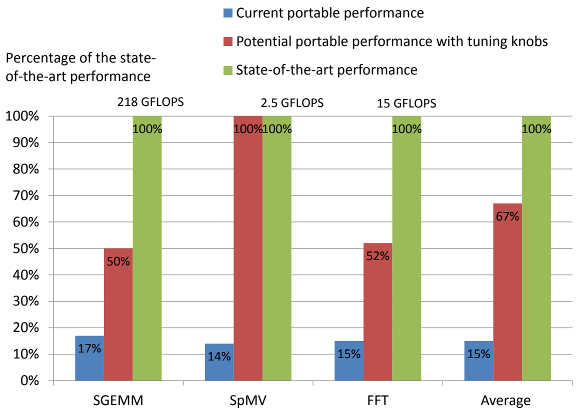


Fig. 3: Current and potential portable performance on the Ivy Bridge CPU. Performance is normalized against the state-of-the-art performance results.

routine, FFTW, and our ELLPACK SpMV routine, which outperforms the MKL CSR SpMV routine). On average, by incorporating tuning knobs to program, the OpenCL portable performance could be improved by more than 4×, from 15% to a 67% of the state-of-the-art performance.

5 Programming Model Implications

In this section, we evaluate current OpenCL programming interface, and propose extensions towards a performance-portable programming framework.

Although OpenCL provides functional portability across multicores and GPUs, its performance portability is poor (Figure 3). We have demonstrated a set of tuning knobs could significantly improve the portable performance. To incorporate these tuning knobs, however, the OpenCL programming interface needs to be raised to a more abstract level. In particular, we propose the following extensions.

First, the mapping between threads and data is currently specified by programmers in a two-level parallelism hierarchy with work-items (threads) and work-groups (groups of threads), where work-groups are mapped to cores and work-items to vector lanes. This two-level parallelism model limits the possibility to tune thread-data mapping across multiple levels of parallelism. The current model also requires programmers to specify a fixed parallelism granularity (the amount of work per work-item is fixed by the specified total number of work-items, although the OpenCL runtime could select a work-group size if not provided) and implies interleaved thread-data mapping, which is not favorable to CPUs. To support tunable thread-data mapping, we propose the notion of logical threads with more levels of parallelism hierarchy and logical dimensions, so that they could be re-mapped and re-sized to match today's two-level (core and vector) processors in an adaptive way.

Second, OpenCL currently does not support any data layout abstraction and requires programmers to specify a fixed data layout. However, multicores and GPUs favor different data layout between row major and column major, and between structure-of-arrays and array-of-structures. To solve this problem, OpenCL needs to introduce some form of data abstraction, which decouples data structure and content from the data layout, and allows programmers to write generic code without specifying an architecture-specific layout. Examples of such decoupling include layout specifiers in UPC [5] and data distributive directives in HPF [15].

Third, OpenCL currently does not have an abstract way to use or not use scrachpad memory. Although OpenCL programs could explicitly manage scratchpad memory on GPUs, such programs do not naturally fit into CPUs with hardware cache. As a result, the OpenCL compiler uses external memory as an emulated scratchpad memory for CPUs, which may cause a performance penalty. OpenCL needs to introduce a simple switch for using either programmer-managed or hardware-managed cache, such as the cache directive in OpenACC [1].

A higher-level programming interface allows a larger tuning space, but is still not sufficient. To support architecture-specific performance tuning, we need to build a more intelligent compiler and runtime framework. As we have noted previously, there is not a one-size-fit-all recipe on how to turn those tuning knobs, and the same optimizations may show totally different performance impacts for different programs and inputs. Therefore, such tuning should be analyzed on a case-by-case basis and could potentially be guided by recently proposed performance models [3, 13, 26]. Application-specific GPU performance autotuning has also been recently explored with success [7, 9, 16]. These studies, as well as ours in this paper, still require considerable manual work of programmers, and we expect a higher-level programming interface with model-guided tuning will be an important direction for future research.

6 Related Work

There have been quite a few studies on the portability of GPU programming models [10, 11, 14, 19–23]. However, the previous work focus mainly on architecture-specific optimizations for OpenCL programs. The contribution of this work is a methodically designed performance portability study that covers both compute-limited and bandwidth-limited benchmarks, systematically summarizes common tuning knobs, and discusses their programming model implications.

MCUDA is one of the pioneering work to compile CUDA programs to a CPU architecture [22]. The loop fission technique is used to convert explicitly synchronized fine-grained parallel GPU programs to implicitly synchronized coarse-grained multi-threaded CPU programs. Potential optimizations including data layout transformation, optional use of shared memory, and flexible thread-data mapping are not explored in the paper.

Du et al. [10] did an interesting study on portable performance of vendor-specific SGEMM kernels across NVIDIA and ATI GPUs. Only compute-bound dense matrix kernels and one tuning parameter (tiling size) are investigated in their proposed autotuning infrastructure. Similar studies [19, 23] explored other tuning parameters including caching, vectorization, and thread block size. Seo et al [20] also noticed the limited performance portability of OpenCL, but did not further investigate its causes.

Shen et al. [21] showed properly tuned OpenCL programs could achieve comparable performance to OpenMP versions. Various tuning aspects including data layouts and parallelism granularity are explored. Fang et al. [11] and Komatsu et al. [14] respectively compare the performance of OpenCL and CUDA, and reach a similar conclusion that the performance of OpenCL programs is comparable to those of CUDA if optimized appropriately. Both studies call for future autotuning research to adapt OpenCL programs to various processors.

7 Conclusions and Future Work

We have identified major tuning knobs for performance portable programming, and demonstrated that they could improve the OpenCL portable performance from the current 15% to a potential 67% of the state-of-the-art performance. We also evaluated the current OpenCL compilers and programming model, and made a list of proposals towards a performance portable programming framework. We believe these results will inform and inspire more thinkings on the design and evolution of OpenCL and other emergining higher-level programming models.

Future directions of this work include: (1) study the performance portability of irregular programs with data-dependent control flows and memory access patterns, (2) investigate the feasibility of incorporating a performance model to compiler or runtime for model-based tuning, and (3) extend this study to other architectures such as Intel's Xeon Phi and other emerging programming APIs such as OpenACC.

References

1. The OpenACC application programming interface 1.0, November 2011. `http://www.openacc-standard.org/`.
2. The OpenCL specification 1.2, November 2011. `http://www.khronos.org/registry/cl/`.
3. S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. W. Hwu. An adaptive performance modeling tool for GPU architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*, pages 105–114, Jan. 2010.
4. N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the 2009 ACM/IEEE Conference on Supercomputing (SC '09)*, pages 18:1–18:11, Nov. 2009.
5. W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. *Introduction to UPC and language specification.* Center for Computing Sciences, Institute for Defense Analyses, 1999.
6. A. A. Chien, A. Snavely, and M. Gahagan. 10x10: A general-purpose architectural approach to heterogeneity and energy efficiency. *Procedia CS*, 4:1987–1996, 2011.
7. J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*, pages 115–126, Jan. 2010.
8. A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 63–74, New York, NY, USA, 2010. ACM.

9. A. Davidson, Y. Zhang, and J. D. Owens. An auto-tuned method for solving large tridiagonal systems on the GPU. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, pages 956–965, May 2011.
10. 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.*, 38(8):391–407, Aug. 2012.
11. J. Fang, A. L. Varbanescu, and H. Sips. A comprehensive performance comparison of CUDA and OpenCL. In *Proceedings of the 2011 International Conference on Parallel Processing (ICPP '11)*.
12. K. Goto and R. Van De Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw.*, 35(1):4:1–4:14, July 2008.
13. S. Hong and H. Kim. An integrated GPU power and performance model. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA '10)*, pages 280–289, 2010.
14. 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*, June 2010.
15. D. Loveman. High performance Fortran. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 1(1):25–42, 1993.
16. J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*, pages 256–265, June 2009.
17. NVIDIA Corporation. NVIDIA CUDA compute unified device architecture, programming guide 5.0, October 2012. `http://developer.nvidia.com/`.
18. J. R. Rice and R. F. Boisvert. *Solving Elliptic Problems using ELLPACK*. Springer-Verlag New York, Inc., 1984.
19. S. Rul, H. Vandierendonck, J. D'Haene, and K. De Bosschere. An experimental study on performance portability of OpenCL kernels. In *2010 Symposium on Application Accelerators in High Performance Computing*, page 3, 2010.
20. S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS parallel benchmarks in OpenCL. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 137–148, Nov.
21. J. Shen, J. Fang, H. Sips, and A. Varbanescu. Performance gaps between OpenMP and OpenCL for multi-core CPUs. In *2012 41st International Conference on Parallel Processing Workshops (ICPPW)*, pages 116–125, Sept.
22. J. A. Stratton, S. S. Stone, and W.-M. W. Hwu. In J. N. Amaral, editor, *Languages and Compilers for Parallel Computing*, chapter MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs, pages 16–30. 2008.
23. P. Thoman, K. Kofler, H. Studt, J. Thomson, and T. Fahringer. Automatic OpenCL device characterization: guiding optimized kernel design. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, pages 438–452, Berlin, Heidelberg, 2011. Springer-Verlag.
24. V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*, pages 31:1–31:11, Nov. 2008.
25. V. Volkov and B. Kazian. Fitting FFT onto the G80 architecture, May 2008. `http://www.cs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6_report.pdf/`.
26. Y. Zhang and J. D. Owens. A quantitative performance analysis model for GPU architectures. In *Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA 17)*, pages 382–393, Feb. 2011.