

# Compiling SIMT Programs on Multi- and Many-core Processors with Wide Vector Units: A Case Study with CUDA

Hancheng Wu

North Carolina State University  
hwu16@ncsu.edu

John Ravi

North Carolina State University  
jjravi@ncsu.edu

Michela Becchi

North Carolina State University  
mbecchi@ncsu.edu

## ABSTRACT

In this work, we study the effective implementation of a SIMT programming model on Intel platforms with 512-bit vector extensions (hybrid MIMD/SIMD architectures). We first propose a set of compiler techniques that enable a SIMT programming model on hybrid architectures. We then evaluate the proposed techniques on various hybrid systems using microbenchmarks and real-world applications, and we point out the main challenges in supporting the SIMT model on hybrid systems.

## KEYWORDS

Xeon Phi, hybrid MIMD/SIMD systems, CUDA, SIMT, vectorization

## 1 INTRODUCTION

Manycore devices with wide vector extensions (MIMD/SIMD hybrid architecture) have played a significant role in high performance computing due to their high computational power and power efficiency [1]. Generating high performance code on such hybrid systems (e.g., Intel Xeon Phi devices) requires the use of both their x86 cores and vector units. Given that the Intel compiler only performs effective auto-vectorization on simple code patterns (mainly loops) [2] and that manual vectorization often yields low programmability and is error-prone, there has been an increasing interest in supporting SIMT programming models on hybrid architectures allowing for the simultaneous use of x86 cores and vector units, better programmability and code portability [3-5]. However, the effective implementation of the SIMT model on these hybrid architectures is not well understood. In this work, we propose a set of compiler techniques that enable a SIMT programming model on hybrid architectures and study their effectiveness on Xeon Phi and Skylake (co)processors using microbenchmarks and real-world applications. By comparing the resulting performance with that achieved on GPUs, we point out the main challenges in supporting the SIMT model on hybrid systems.

## 2 KEY COMPILER TECHNIQUES

We propose compiler techniques to transform generic programs written in SIMT model into code that leverages both the x86 cores and the vector units (VPUs) of a hybrid MIMD/SIMD

architecture. Here, we consider a subset of CUDA-C as the source SIMT programming language and three Intel (co)processors with 512-bit vector extensions as target platforms. The generated code uses the Pthreads API to implement hardware threads running on the x86-compatible cores and vector intrinsics to offload work to the VPUs.

**Threads Mapping Scheme** - The first design question is the mapping of the threads of a CUDA kernel onto the x86 cores and VPUs available on the target platforms. Our transformations map a CUDA thread onto a VPU lane and a CUDA thread-block onto one or more x86 hardware threads (*HTs*). Since the platforms considered have 512-bit VPUs and we support 32-bit variables (int, unsigned int and float), our implementation assumes VPUs with 16 vector lanes. Therefore, each x86 HT can issue instructions to 16 vector lanes simultaneously. Since a CUDA warp consists of 32 threads, it is mapped onto two HTs. Thus, each CUDA thread-block (which consists of one or multiple warps) is mapped onto multiple HTs. We refer to the set of HTs executing the same CUDA thread-block as *hthread-block*. When there are not enough concurrent hthread-blocks to map all CUDA thread-blocks, each hthread-block will execute multiple CUDA thread-blocks in an iterative fashion.

**Thread and Block Identification** - To map CUDA identifiers (*blockIdx*, *threadIdx*, etc.) on hybrid systems, our transformations generate pre-defined vector variables with the same name as their CUDA counterparts and initialize them based on the hthread-block and the vector lane each CUDA thread is mapped to.

**Thread-block Synchronization and Shared Memory** - To implement the CUDA `__syncthreads` synchronization primitive and the CUDA shared memory abstraction on hybrid architectures, we associate to each hthread-block a shared memory region and a barrier synchronization primitive implemented using the standard Pthreads library.

**Arithmetic and Compare Operations** - As scalar data types are transformed into vector data types, CUDA-C arithmetic and *compare* operations over scalar data types are replaced with the corresponding vector intrinsics. Specifically, a compare vector instruction applied to two vector variables returns a mask vector variable whose bits are set to 1 for vector lanes with true result. Compare vector instructions are used to implement control flow statements.

**Assignments** - Assignments between vector variables are naturally supported. Moving data from memory to a vector

variable requires the load or gather instructions; moving data from a vector register to memory requires the store or scatter instructions. The load and store are faster than gather and scatter, but they only work when the involved addresses are contiguous and 64-byte aligned. Therefore, the gather and scatter primitives will be used in general cases. The load and store primitives will be used only if the `#pragma aligned` compiler directive is specified to guarantee the necessary requirements.

**Control Flow Statements** - We support three control-flow statements: if-else, while-loop and for-loop statements. In CUDA-C, these statements are executed by each CUDA thread, and the control flow is maintained at the CUDA thread level. On hybrid systems the control flow is maintained by the threads running on x86 cores. Supporting control flow statements requires handling the case where different VPU lanes take different branches. This is complicated by the nesting of control flow statements. To support this functionality, we associate a mask variable to each block scope and issue all the vector instructions in the block scope using that mask. When the execution moves to a different block scope, we derive a new mask variable for the new scope.

**Function Calls** - On hybrid systems, functions are invoked on x86 cores. To transform a function written in SIMT style into a function that executes on VPUs, we address two issues. First, our transformation adds to each function an extra mask parameter. When the function is invoked, the mask variable at the caller scope is passed as a parameter and associated with the function body. Thus, vector lanes that are inactive at the caller scope will remain inactive in the callee scope. Second, our transformation makes sure that the function will not return until all vector lanes have terminated the execution of the function.

### 3 MICROBENCHMARK RESULTS

We perform our experiments on three hybrid architectures: a Xeon Phi coprocessor (*Hco*), a Xeon Phi processor (*Hpro*), a Xeon Skylake processor (*Hsky*), and we compare the resulting performance with that on three Nvidia GPUs with the following architectures: Fermi, Maxwell and Pascal (*Gfer*, *Gmax* and *Gpas*, respectively). We design microbenchmarks to evaluate the performance limiting factors of the proposed techniques on hybrid architectures. We have observed the following results.

**Kernel Launch Overhead** is incurred at kernel launch time since our framework must spawn enough pthreads to utilize the available x86 cores and VPUs. This overhead turns out to be more substantial on hybrid systems than on GPUs. We find that it increases almost linearly with the number of pthreads spawned. **Irregular Memory Accesses** are more problematic on hybrid systems than on GPUs, especially when all computing cores are utilized. In general, hybrid systems perform worse than GPUs with high access irregularity. We find **Control Divergence Overhead** on hybrid architectures to be comparable to that on GPUs. **Thread-block Barrier Synchronization Overhead** is significantly high on hybrid systems (software-based synchronization triggers heavy memory traffic across the private L2 caches). Even the Skylake processor, the fastest among the hybrid systems, has a barrier synchronization cost 500-1000x

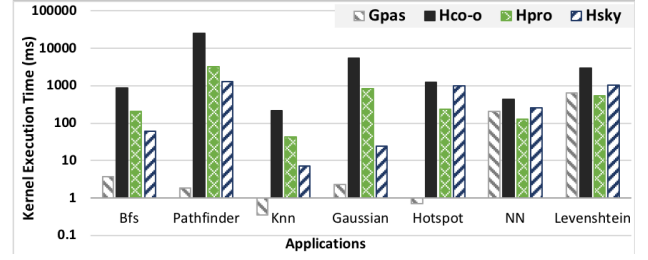


Figure 1. Results of real world applications

higher than GPU. **Shared Memory** allocated to a hthread-block may be cached in different private L2 caches. This can lead to invalidations of L2 cache entries due to coherence actions, and, as a result, to additional memory traffic. We see that, except for fully regular memory accesses, GPUs outperform Intel systems.

### 4 REAL-WORLD APPLICATION RESULTS

We evaluate our compiler transformations on the following benchmarks: BFS, Pathfinder, Knn, Gaussian Elimination, Hotspot, NN and Levenshtein Distance (*LD*) [6-8].

We find that applications with iterative kernel invocations (BFS, Hotspot, Pathfinder and Gaussian Elimination) can hardly benefit from running on hybrid architectures due to the kernel launch overhead. Hotspot performs only 4 kernel invocations, but since it uses shared memory along with thread-block synchronization, it reports significantly better performance on GPUs than on Intel hybrid platforms. For applications with a single kernel invocation (Knn, NN and LD), Knn reports better performance on GPU than on hybrid platforms due to its irregular memory accesses. NN, on the other hand, reports the best performance on the Knights Landing processor due to its use of recursive calls which is less efficient on GPU [8, 9]. LD reports the best performance on the Knights Landing processor due to several reasons. First, it has a long running kernel that amortizes the kernel launch overhead. Second, its kernel is launched with only 16 CUDA threads per thread-block, leading to 1 hardware thread per hthread-block in the transformed code. As a result, the shared memory associated with each hthread-block is always cached in a single L2 cache, avoiding constant cache entry invalidations (reduced memory traffic) and the thread-block synchronization. Lastly, the memory access patterns of the LD kernel, most of the times, meet the alignment requirement, allowing the use of fast load and store instructions.

### 4 CONCLUSIONS

We tested our code transformations with microbenchmarks and real-world applications with different computation and memory access patterns, and we compared the resulting performance with that of the original CUDA codes on GPUs. We found several performance-limiting factors that result in poor performance of existing CUDA applications on hybrid systems, such as iterative kernel launch, thread-block synchronization, and use of shared memory on large thread-block configurations. Even so, Intel Phi platforms outperform Pascal GPUs on certain CUDA codes written for GPUs.

## REFERENCES

- [1] (2017). *TOP500 Supercomputer Sites*. Available: <https://www.top500.org/lists/2017/11/>
- [2] S. Maleki, Y. Gao, M. J. Garzar, #225, T. Wong, and D. A. Padua, "An Evaluation of Vectorizing Compilers," presented at the Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques, 2011.
- [3] B. Ren, T. Poutanen, T. Mytkowicz, W. Schulte, G. Agrawal, and J. R. Larus, "SIMD parallelization of applications that traverse irregular data structures," presented at the Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2013.
- [4] X. Huo, B. Ren, and G. Agrawal, "A programming system for xeon phis with runtime SIMD parallelization," presented at the Proceedings of the 28th ACM international conference on Supercomputing, Munich, Germany, 2014.
- [5] L. Chen, P. Jiang, and G. Agrawal, "Exploiting recent SIMD architectural advances for irregular applications," presented at the Proceedings of the 2016 International Symposium on Code Generation and Optimization, Barcelona, Spain, 2016.
- [6] S. Che, M. Boyer, and J. Meng, "Rodinia: A benchmark suite for heterogeneous computing," presented at the IEEE International Symposium on Workload Characterization, Austin, TX, USA, 2009.
- [7] A. Todd, M. Nourian, and M. Becchi, "A Memory-Efficient GPU Method for Hamming and Levenshtein Distance Similarity," presented at the 2017 IEEE 24th International Conference on High Performance Computing Jaipur, India, 2017.
- [8] H. Wu and M. Becchi, "An Analytical Study of Recursive Tree Traversal Patterns on Multi- and Many-Core Platforms," presented at the 2017 IEEE 23rd International Conference on Parallel and Distributed Systems, Shenzhen, China, 2017.
- [9] M. Goldfarb, Y. Jo, and M. Kulkarni, "General transformations for GPU execution of tree traversals," presented at the Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Denver, Colorado, 2013.