Performance Evaluation of OpenMP's Target Construct on GPUs - Exploring Compiler Optimizations -

Akihiro Hayashi, Jun Shirako

Department of Computer Science,
Rice University,
Houston, TX, USA
E-mail: {ahayashi,shirako}@rice.edu

Etorre Tiotto, Robert Ho

IBM Canada Laboratory,
Markham, ON, CA
E-mail: {etiotto,robertho}@ca.ibm.com

Vivek Sarkar

Department of Computer Science,
Rice University,
Houston, TX, USA
E-mail: vsarkar@rice.edu

Abstract:

OpenMP is a directive-based shared memory parallel programming model and has been widely used for many years. From OpenMP 4.0 onwards, GPU platforms are supported by extending OpenMP's high-level parallel abstractions with accelerator programming. This extension allows programmers to write GPU programs in standard C/C++ or Fortran languages, without exposing too many details of GPU architectures.

However, such high-level programming models generally impose additional program optimizations on compilers and runtime systems. Otherwise, OpenMP programs could be slower than fully hand-tuned and even naive implementations with low-level programming models like CUDA. To study potential performance improvements by compiling and optimizing high-level programs for GPU execution, in this paper, we 1) evaluate a set of OpenMP benchmarks on two NVIDIA Tesla GPUs (K80 and P100) and 2) conduct a comparable performance analysis among hand-written CUDA and automatically-generated GPU programs by the IBM XL and clang/LLVM compilers.

Keywords: OpenMP; GPUs; Performance Evaluation; Compilers; Target Constructs; LLVM; XL Compiler; Kepler; Pascal;

Reference to this paper should made as follows: Hayashi, A., Shirako, J., Tiotto, E., Ho, R. and Sarkar, V. (201X) 'Performance Evaluation of OpenMP's Target

Construct on GPUs', *International Journal of High Performance Computing and Networking*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Akihiro Hayashi received his Ph.D. in Engineering from Waseda University, Japan, in 2012. He is a Research Scientist at the Department of Computer Science, Rice University, USA. His research interests include automatic parallelization, programming languages, and compiler optimizations for parallel computer systems.

Jun Shirako received his Ph.D. degree in Computer Science from Waseda University in 2007. He is currently a Research Scientist in Computer Science Department at Rice University. His research interests include parallel programming languages, optimizing compilers, and inter-task synchronization runtime.

Ettore Tiotto is a Senior Compiler Developer at the IBM Toronto Laboratory. He graduated "Summa Cum Laude" in Physics from the University of Torino, Italy, in 1996. Since joining the IBM XL compiler team he has worked on numerous releases of the industry leading XL C/C++ and Fortran compilers for POWER. He currently leads the static XL compiler team focusing on the development of GPU code generation and optimization strategies for OpenMP 4.5.

Robert Ho received his Ph.D. in Computer Engineering from the University of Toronto. He is a software developer working on the XL Compiler at the IBM Canada lab in Markham, Ontario. His research interests include OpenMP, compiler optimizations for NUMA multiprocessor architectures, and heterogenous parallel architectures.

Vivek Sarkar is the E.D. Butcher Chair of Engineering and Professor of Computer Science at Rice University. He currently leads the Habanero Extreme Scale Software Research Laboratory at Rice, and is PI of the DARPA-funded Pliny project on "big code" analytics. Prior to joining Rice in July 2007, Vivek was Senior Manager of Programming Technologies at IBM Research. His research projects at IBM included the X10 programming language, the Jikes Research Virtual Machine for the Java language, the ASTI optimizer used in IBM's XL Fortran product compilers, and the PTRAN automatic parallelization system. Vivek became a member of the IBM Academy of Technology in 1995, the E.D. Butcher Chair in Engineering at Rice University in 2007, and was inducted as an ACM Fellow in 2008. Vivek has been serving as a member of the US Department of Department of Energy's Advanced Scientific Computing Advisory Committee (ASCAC) since 2009, and o CRA's Board of Directors since 2015.

1 Introduction

Graphics processing units (GPUs) can achieve significant performance and energy efficiency for certain classes of applications, assuming sufficient tuning efforts by expert programmers. A key challenge in GPU computing is the improvement of programmability: reducing the programmers' burden in writing low-level GPU programming languages such as CUDA (NVIDIA 2017*a*) and OpenCL (KHRONOS GROUP 2015) without sacrificing performance. This burden is mainly because programmers have to not only 1) develop efficient compute kernels using the single instruction multiple thread (SIMT) model but also 2) manage memory allocation/deallocation on GPUs and data transfers

between CPUs and GPUs by orchestrating low-level API calls. Additionally, performance tuning with such low-level programming models is often device-specific, thereby reducing performance portability. To improve software productivity and portability, a more efficient approach would be to provide high-level abstractions of GPUs that hide GPUs' architectural details while retaining sufficient information for optimizations and code generation.

The OpenMP API (OpenMP 2015) is a de facto standard parallel programming model for shared memory CPUs, supported on a wide range of SMP systems for many years. The OpenMP model offers directive-based parallel programming for C/C++ or Fortran, which successfully integrated bulk-synchronous SPMD parallelism including barriers/parallel loops and asynchronous dynamic task parallelism. The newly introduced OpenMP accelerator model is an extension to the standard OpenMP parallel programming model and aims at not exposing too many details of underlying accelerator architectures by providing a set of high-level `device` constructs. As for GPUs, the OpenMP `target` constructs create a GPU environment and the `distribute parallel for` and `parallel for` constructs are used for expressing the block-level and thread-level parallelism on GPUs respectively. Additionally, the `map` clause enables data transfers between CPUs and GPUs. We believe that these high-level abstractions by the OpenMP accelerator model enable improved programmability and performance portability in current and future GPU programming. As of this writing, development/beta versions of IBM XL C/C++/Fortran and clang+LLVM compilers support the accelerator model on GPUs. Note that clang+ LLVM also supports Intel Xeon Phi.

However, aside from the improved programmability and performance portability, mapping the high-level OpenMP programs to GPUs imposes technical challenges on compiler optimizations and runtime execution models: generating highly-tuned code in consideration of the GPUs' architectural details such as two distinct levels of parallelism (blocks and threads) and deep/diverse memory hierarchy. As an initial step to address these challenges, we 1) evaluate a set of OpenMP programs with the `target` construct on GPUs and then 2) analyze the results and generated code for exploring further optimization opportunities.

To study potential performance improvements by compiling and optimizing high-level GPU programs, this paper makes the following contributions:

- Performance evaluation of OpenMP benchmarks on NVIDIA Tesla K80 and P100 GPU platforms.
- Detailed performance analysis among hand-written CUDA and automatically generated GPU programs by development/beta versions of the IBM XL C and clang+ LLVM compilers to explore future performance improvement opportunities.
  Our key findings from the study are summarized as follows:
  - The OpenMP versions are in some cases faster, in some cases slower than straightforward CUDA implementations written even without complicated hand-tuning.
  - Additionally, results show that more work must be done for OpenMP-enabled compilers and runtime systems to match the performance of highly-tuned CUDA code for some cases examined. The possible compiler optimization strategies for OpenMP programs are:
    1. minimizing OpenMP runtime overheads on GPUs when possible.

2. constructing a good data placement policy for the read-only cache and the shared memory on GPUs.
3. improving code generation for each thread in GPUs (e.g., math function and memory coalescing.).
4. performing high-level loop transformation (e.g. using the polyhedral model (Shirako et al. 2017)).

Because our results and analyses can apply to both OpenMP 4.0 and 4.5 programs, we don't distinguish them. In the following, we refer to OpenMP 4.0 and 4.5 as OpenMP unless otherwise indicated.

The paper is organized as follows. Section 2 provides background information on GPUs and the OpenMP accelerator model. Section 3 shows an overview of clang+LLVM and XL C compilers that compile OpenMP programs to GPUs. Section 4 presents an extensive performance evaluation and analysis on two single-node platforms with GPUs. Section 5, Section 6, and Section 7 summarize related work, conclusions, and future work.

2 The OpenMP Accelerator Model

2.1 GPUs

NVIDIA GPU architecture consists of global memory and an array of *streaming multiprocessors* (SMXs). Each SMX comprises many single- and double- precision cores, special function units, and load/store units to execute hundreds of threads concurrently. L1 cache, read-only cache, and shared memory are shared among these cores/units to improve data locality within a single SMX. Also, global memory data requested from each SMX are cached by L2 cache.

CUDA (NVIDIA 2017*a*) is a standard parallel programming model for NVIDIA GPUs. In CUDA, **kernels** are C functions that will be executed on GPUs. A **block** is a group of **threads** executed on the same SMX and is organized in a collection of blocks called a **grid** that corresponds to a single kernel invocation. All blocks within a grid are indexed as a 1- or 2-D array. Similarly, all threads within each block are indexed as 1-, 2-, or 3-D array. While **barrier synchronizations among threads** in the same block are allowed, no support exists for inter-block (global) barrier synchronizations. Instead, global barriers can be simulated by separating the phases into separate kernel invocations. For memory optimizations, the programmer and compiler must utilize registers and shared memory for improving data locality. Also, it is important to note that global memory accesses for adjacent memory locations are coalesced into a single memory transaction if consecutive global memory locations are accessed by a number of consecutive threads (normally 32 threads, called *warp*) and the starting address is aligned. This is called **memory access coalescing** and code transformations for improved coalescing can be performed by both programmers and compilers.

2.2 OpenMP directives

The OpenMP accelerator model, which consists of a set of device constructs for heterogeneous computing, was originally introduced in the OpenMP 4.0 specification. We give a brief summary of the OpenMP device constructs used in this paper.

```
1  // Combined Construct Version
2  #pragma omp target teams distribute parallel for \
3              map(from: C) map(to: B, A) \
4              num_teams(N/1024) thread_limit(1024) \
5              dist_schedule(static, distChunk) \
6              schedule (static, 1)
7    for (int i = 0; i < N; i++) {
8      C[i] = A[i] + B[i];
9    }
10 // Non-Combined Construct Version
11 #pragma omp target map(from: C) map(to: B, A)
12 #pragma omp teams num_teams(N/1024) \
13                   thread_limit(1024)
14 #pragma omp distribute parallel for \
15             dist_schedule(static, distChunk) \
16             schedule (static, 1)
17   for (int i = 0; i < N; i++) {
18     C[i] = A[i] + B[i];
19   }
```

Listing 1: Vector Addition Example.

The `target` construct specifies the program region to be offloaded to a target device, e.g., GPU grid. The `map` clause attached to the `target` construct maps variables to/from the device data environment. The `teams` construct, which must be perfectly nested in a `target` construct, creates a league of thread teams. The number of teams and the number of threads per team are respectively specified by the `num_teams` and `thread_limit` clauses. A thread team corresponds to a thread block on a GPU, and there is a master thread in each team. The `distribute` construct is a device construct to be associated with loops, whose iteration space is distributed across master threads of a `teams` construct. On the other hand, the loops associated with the `parallel for` worksharing construct are distributed across threads within a team.

These constructs can be specified as individual constructs, or can be compounded as a single combined construct when they are immediately nested. Listing 1 shows a vector addition kernel with both the combined and non-combined constructs. The whole loop kernel is specified with the `target` construct and offloaded to a GPU. According to the `map` clauses, arrays A, B, C are mapped to/from the GPU device memory and the compiler generates required data transfers. The `teams` construct with `num_teams(N/1024)` and `thread_limit(1024)` clauses creates a league of `N`/1024 teams each of which contains 1024 threads. As with the `schedule` clause attached on `for` construct, the `dist_schedule` clause for the `distribute` construct allows users to specify chunk size when distributing iterations across teams. In this example, the whole `N` iterations are divided into chunks of `distChunk` iterations, and the iterations per chunk are distributed across threads per team according to the `schedule` clause.

3 Compiling OpenMP to GPUs

This section describes a brief overview of the OpenMP compilers and their optimizations and code generation used for performance evaluation in this paper.

3.1 Compilers

Figure 1 illustrates the compilation flow of the clang+LLVM and IBM XL C compilers.

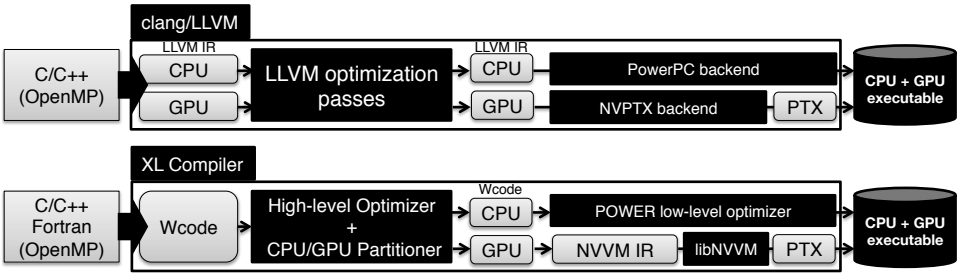

Figure 1: Compilation flow of clang+LLVM and XL compilers for C/C++ and C/C++/Fortran respectively.

3.1.1 clang+LLVM Compiler

LLVM (Lattner & Adve 2004) is a widely used compiler infrastructure and clang is a C language family front-end for LLVM. Clang first transforms OpenMP programs to LLVM's intermediate representation (LLVM IR) and then the LLVM compiler applies language- and target-independent optimization passes to LLVM IR (LLVM.org 2017*a*).

As of this writing, a development version of clang (Bertolli et al. 2014, 2015, Antao et al. 2016) for the CORAL systems (Department of Energy 2014) supports NVIDIA's GPU code generation from OpenMP's target construct. First, the clang compiler outlines GPU kernels specified by OpenMP `target` directives as separate LLVM functions and the LLVM functions are fed into standard LLVM passes followed by the NVPTX backend (LLVM.org 2017*b*) for PTX assembly (NVIDIA 2017*e*) code generation. Also, the LLVM compiler generates CPU code that invokes CUDA API calls to perform memory allocations/deallocations on GPUs, data transfers between CPUs and GPUs, and kernel launches.

3.1.2 IBM XL Compiler

Our beta XL compiler for OpenMP CPU/GPU execution is built on top of a production version of the IBM XL C/C++ and XL Fortran compilers. First, the compiler front-end transforms OpenMP programs to Wcode, which is an IR used by IBM compiler components. Then, the Toronto portable optimizer (TPO) performs high-level optimizations over the Wcode in a language- and target-independent manner.

In the case where OpenMP `target` directives are found, the GPU partitioner partitions the Wcode into CPU Wcode and GPU Wcode analogous to how the clang+LLVM outlines kernels as functions. Finally, the POWER Low-level optimizer optimizes CPU Wcode and generates a PowerPC binary including CUDA API calls for controlling GPUs. For GPU code generation, one fundamental difference between the XL and the clang+LLVM compilers is that GPU Wcode is translated into an NVVM IR (NVIDIA 2017*d*) in the XL compiler, whereas the clang+LLVM compiler generates PTX directly. The NVVM IR is eventually fed into libNVVM library to generate PTX assembly code (NVIDIA 2017*e*).

```
1  #pragma omp target teams { // GPU region
2      // sequential region 1 executed
3      // by the master thread of each team
4      if (...) {
5        // parallel region 1
6        #pragma omp parallel for
7        for () {}
8      } else {
9        ...
10     }
11 }
```

Listing 2: OpenMP code that requires multiple execution modes on GPUs.

3.2 Running OpenMP programs on GPUs

This section describes how OpenMP 's target construct is compiled and optimized for GPU execution. We mainly focus on significant optimizations affecting performance as shown in the performance results in Section 4.

3.2.1 OpenMP Threading Model on GPUs

In OpenMP specifications, `target` regions may include sequences of sequential, parallel, and potentially nested parallel regions. Consider an example of the `target` directive shown in Listing 2. First, the master thread of each team needs to execute the if-statement in Line 4. Then, if the branch is taken, the program execution switches to the parallel region (parallel for loop in Line 6-7) executed by threads within a team. In general, OpenMP programs can switch back and forth between sequential and parallel regions, and thus code generation for such program is generally challenging. As of this writing, a state machine execution scheme (Bertolli et al. 2014, 2015) and master/slave worker execution scheme (Antao et al. 2016) were proposed in prior work. A brief summary of these code generation schemes is as follows:

State Machine Execution defines logical execution states of GPU execution such as parallel and sequential regions, and state transitions occur dynamically. Listing 3 shows an example of the state machine execution scheme. In Listing 3, `SEQUENTIAL_REGION1` is a "team master only" state, where only the master of each team needs to execute it (`if (threadIdx.x != MASTER)` in Line 5). If the branch in Line 7 is taken, the execution switches to `PARALLEL_REGION1`, where the original `omp parallel for` loop is executed by all threads within a team. This can increase register pressure and incur performance penalties due to control-flow instructions. The detailed information on GPU code generation with state transitions can be found in (Bertolli et al. 2014, 2015, Hayashi et al. 2016).

Master/Worker Execution employs a similar execution scheme to the original OpenMP's fork/join execution model. In this model, the runtime distinguishes two logical types of warps within a block - i.e. master and worker warps. The master warp is dedicated for serial execution and activating worker warps when it encounters a parallel region. Worker warps wait for work from the master warp on a specific barrier number (e.g., `bar.sync 0`) and use different barrier numbers when synchronizations among parallel warps are required. Listing 4 shows an example of the execution scheme. Advantages of this code generation scheme are 1) it simplifies the code generation, 2) its register pressure is not as bad as the state machine execution, and 3) it can support orphaned parallel directives in extern functions.

```
bool finished = false;
while (!finished) {
  switch (labelNext) {
  case SEQUENTIAL_REGION1:
    if (threadIdx.x != MASTER) break;
    // code for sequential region 1
    if (...) {
      ...
      labelNext = PARALLEL_REGION1;
    }
    break;
  case PARALLEL_REGION1:
    // code for parallel region 1
    ...
    if (threadIdx.x == MASTER) {
        // update labelNext;
    }
    break;
  // other cases
  ...
  case END:
    labelNext = -1;
    finished = true;
    break;
  }
  __syncthreads();
}
```

Listing 3: An example of the state machine execution on GPUs.

```
if (masterWarp) {
   // code for sequential region 1
   if (...) {
      // code for parallel region 1
      [activate workers]
      bar.sync 0 // synchronization
      bar.sync 0 // synchronication
   }
} else  {
  // Worker Warps
  bar.sync 0 // synchronization
  // get a chunk of parallel loop
     and execute it in parallel
  executeParallelLoop();
  bar.sync 0 // synchronization
}
// outlined work for worker warps
executeParallelLoop();
```

Listing 4: An example of the master/worker execution on GPUs.

For the purpose of optimizations, these execution schemes can be simplified by an alternative code generation scheme when the body of the target region satisfies the following conditions (Bertolli et al. 2015):

- There is no "team master only" region, where only master threads need to execute it (e.g. Line 4-10 in Listing 3).
- There is no data sharing among threads in a team.
- There are no nested OpenMP pragmas through function calls.
- `schedule(static, 1)` is specified on the `#pragma omp parallel for` construct.

To activate the alternative code generation scheme, the clang+LLVM compiler additionally requires programmers to use a combined construct (OpenMP 2015), a shortcut for specifying multiple constructs in a single line (see also Section 2.2), whereas the XL C compiler can do so even with a non-combined construct.

3.2.2 Leveraging GPU's Memory Hierarchy

GPU memory optimizations such as utilizing the shared memory and the read-only data cache are essential for improving kernel performance. For OpenMP programs, it is the compiler's responsibility to perform such optimizations since OpenMP does not provide a way to place data on a particular GPU memory. However, neither the clang nor the XL C compiler performs such optimization as of this writing.

The NVPTX backend and the libNVVM library utilize the read-only cache for all data that is guaranteed to be read-only when the target architecture is `sm_35` or later. However, placing all possible data on the read-only data cache can also generate a harmful effect on performance; a more attractive approach would be to selectively optimize data placement as a part of high-level loop transformations guided by proper cost models. Further discussions can be found in Section 4.4, Section 6, and Section 7.

3.2.3 Maximizing ILP

Leveraging instruction-level parallelism (ILP) is also an important optimization strategy to increase GPU utilization. While SMXs on GPUs can take advantage of ILP interchangeably with thread-level parallelism (TLP), in some cases, it is easier to increase ILP by performing loop unrolling and other transformations. The clang+LLVM compiler, the NVPTX backend, and the libNVVM library unroll sequential loops to increase ILP. Further discussions on it can be found in Section 4.2.4 and Section 4.4.

Benchmark	Description	Data Size	Target Directives
VecAdd	Vector Addition (C=A+B)	67,108,864	1-level
Saxpy	Single-Precision scalar multiplication and vector addition (Z=A×X+Y)	67,108,864	1-level
MM	Matrix Multiplication (C=A×B)	2,048 × 2,048	1-level
BlackScholes	Theoretical estimation of the European style options	4,194,304	1-level
OMRIQ	3-D MRI reconstruction from SPEC ACCEL™ (SPEC 2015)	32,768	1-level
SP-xsolve3	Scalar Penta-diagonal solver from SPEC ACCEL™ (SPEC 2015)	5 × 255 × 256 × 256	2-level

Table 2 Benchmarks from PolyBench and SPEC ACCEL used in our evaluation

4 Performance Evaluation

This section presents the results of an experimental evaluation of OpenMP's target construct on two single-node platforms with GPUs.

4.1 Experimental protocol

Purpose: Our goal is to study potential compiler optimizations for OpenMP programs in terms of kernel performance. We do not focus on data transfers between the host and GPU devices in this paper because 1) data transfer optimizations with OpenMP are more transparent thanks to the `map` clause than kernel optimizations, and 2) there are some prior approaches (Ishizaki et al. 2015, Kim et al. 2016) from which we can leverage some of the insights. For that purpose, we focus on the performance difference among CUDA and OpenMP variants of benchmarks. In Section 4.2, we first compare naive CUDA and OpenMP variants, each of which employs straightforward GPU parallelization strategies without complicated hand-tuning. Then, we discuss the performance difference between highly-tuned CUDA and OpenMP code in Section 4.3.

Machine: We present the results on two single-node platforms with GPUs. The first platform (S824) consists of a multicore IBM POWER8 CPU and an NVIDIA Tesla K80 with the ECC enabled. The platform has two 12-core IBM POWER8 CPUs (8286-42A), operating at up to 3.52GHz with a total 1TB of main memory. Each core is capable of running eight SMT threads, resulting in 192 CPU threads per platform. The NVIDIA K80 GPU has 13 SMXs, each with 192 CUDA cores, operating at up to 875MHz with 12GB of global memory, and is connected to the POWER8 by using PCI-Express. The second platform (S822LC) consists of a multicore IBM POWER8 CPU and an NVIDIA Tesla P100 with the ECC disabled. The platform has two 8-core IBM POWER8 CPUs (8335-GTB), operating at up to 4.02GHz with a total 128GB of main memory and capable of running 128 CPU threads per platform. The NVIDIA P100 GPU has 56 SMs, each with 64 CUDA cores, operating at up to 1.36GHz with 4GB of global memory, and is connected to the POWER8 by using PCI-Express.

Benchmarks: Table 2 lists six benchmarks that were used in the experiments. We chose typical numerical computing, medical, and financial applications for the purpose of the compiler optimization exploration. For all the benchmarks, we used the variants with the float data type. Also, we used three different datasets: small (roughly 1K- elements), medium (roughly 64K- elements), and large (roughly 4M- elements). However, since the three datasets show the same trends except for what we discuss in Section 4.2.2, we only report the results with one data set. For "Data Size", Table 2 only shows the largest array size evaluated. For "Target Directives", "1-level" shows we only parallelized the outermost loop, where the OpenMP compilers accept both "combined" and "non-combined" versions.

"2-level" means that we parallelized nested loops at different levels - i.e. block- and thread-level and there are only "non-combined" versions.

Experimental variants: Each benchmark was evaluated by comparing the following versions relative to a parallel GPU execution of a baseline CUDA version:

- **CUDA:** Reference CUDA implementations
  - CUDA (baseline): A CUDA version with the read-only data cache disabled because the read-only cache does not always contribute to performance improvements (see 4.2.5).
  - CUDA-ROC (K80 Only): all read-only arrays within a kernel are accessed through the read-only data cache. Read-only arrays are specified with `const* __restrict__`. Also, the XL C compiler utilizes the read-only data cache through `ld.global.nc` instruction. Note that the read-only data cache is no longer available in the P100 GPUs.
- **OpenMP:** Combined and non-combined constructs versions compiled by the following compilers. These compilers employ the master/worker code generation scheme except for the alternative code generation scheme for simplifying the OpenMP threading model on GPUs. For more details, see Section 2.2 and Section 3.
  - clang+LLVM compiler
  - XL C compiler

For fair and clear comparisons, we carefully 1) prepared the syntactically same CUDA and OpenMP source code and 2) we used the same block and grid size among these variants. For the VecAdd, Saxpy, MM, BlackScholes, OMRIQ cases, we used a grid size of $N/1024$ and a block size of 1024, where N is the length of an input array. Also, a grid size of 254 and a block size of 254 were used for SP. Block and grid sizes used in this experiment can be found in Table 10 in Section A.

For the CUDA variants, we used the CUDA compiler driver (`nvcc`) 8.0.44 with `-O3` and `-arch=sm_37` for the Tesla K80 GPU or `-arch=sm_60` for the Tesla P100 GPU. For OpenMP variants, a development version of clang 4.0 with `-fopenmp -fopenmp-targets=nvptx64-nvidia-cuda` and IBM XL C/C++ compiler version 13.1.5 (technology preview) with `-O3 -qhot -qoffload -qsmp=omp` were used. Note that these options internally specify appropriate compute capability (either 3.7 or 6.0) through the NVPTX backend or the libNVVM library. For single-precision floating point operations, `-ftz=false -prec-div=true -prec-sqrt=true` was used for all variants. Additionally, there is no limitation on the number of registers per thread for OpenMP variants.

Performance was measured in terms of elapsed milliseconds from the start of the first loop(s) to the completion of all loops. Since **our primary focus is on kernel performance, our measurements only include kernel execution time on the GPU (for all the variants)**, and exclude host-device data transfer times. Performance numbers were obtained with NVIDIA CUDA profiler, or `nvprof` (NVIDIA 2017*f*), whose *Summary Mode* is capable of measuring/printing the average, minimum, and maximum time of the kernel execution(s) and data transfer(s) with low overheads. We ran each variant at least three times and reported the fastest run. The performance numbers are quite stable with small variations.

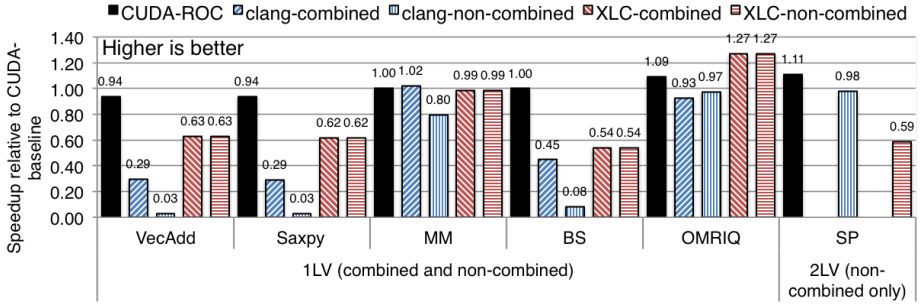

Figure 2: Relative performance over CUDA-baseline on the IBM POWER8 + NVIDIA Tesla K80 platform.

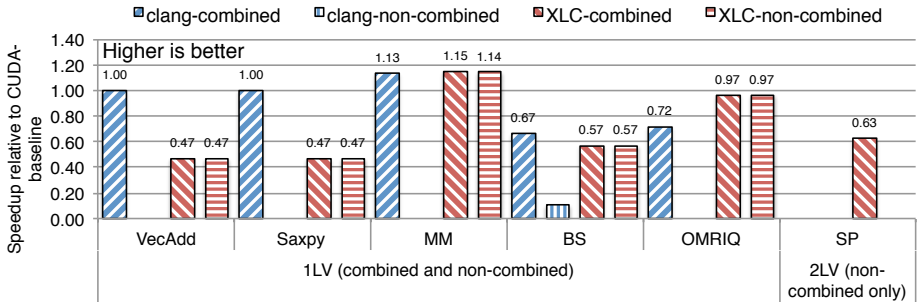

Figure 3: Relative performance over CUDA-baseline on the IBM POWER8 + NVIDIA Tesla P100 platform. Some of the clang-control variants are missing due to a runtime error.

In the following, we first compare naive CUDA and OpenMP variants, each of which employs straightforward GPU parallelization strategies without complicated hand-tuning in Section 4.2. Then, we discuss the performance difference between highly-tuned CUDA and OpenMP code in Section 4.3.

4.2 Performance Comparison with Naive Code

4.2.1 Summary of Results

This section outlines the results shown in Figure 2 and Figure 3, which show speedup factors relative to the baseline CUDA implementation (CUDA) on the NVIDIA Tesla K80 and P100 GPUs. Also, absolute performance numbers for each variant are shown in Table 10 in Section A. Note that some of the clang-control variants on the P100 platform are missing due to a runtime error, in which the kernel invocation was failed.

Overall, for both clang and XL C compilers, the OpenMP variants are in some cases faster, in some cases slower than the baseline CUDA. One of the reasons for the slowdown is that overheads of running OpenMP's threading model on GPUs are not negligible. More detailed discussions can be found in Section 4.2.2. For BlackScholes, the CUDA variants

are better than the OpenMP variants due to more efficient math function code generation (see Section 4.2.3). For MM, the OpenMP variants show performance improvements in some cases, but the CUDA variants are faster than the OpenMP variants in SP. This is mainly due to selecting proper unrolling factors (see Section 4.2.4 as well).

Also, the results in Figure 2 show that utilizing the read-only data cache does not always improve GPU kernel performance (see Section 4.2.5).

4.2.2 Overheads of OpenMP's execution model on GPUs

As we discussed in Section 3.2.1, a potential performance issue with OpenMP programs is that the OpenMP variant generally requires either the state machine or the master/worker execution to support OpenMP's thread execution model on GPUs.

Non-Combined vs. Combined Directive: An important compiler optimization is to simplify OpenMP's thread execution scheme using the alternative code generation scheme for specific programs. The XL C compiler supports such an optimization, and thereby there is no performance degradation even with non-combined constructs as shown in Figure 2 and Figure 3. The impact of the removal is obvious by comparing the combined and non-combined versions by the clang compiler shown in Figure 2 and Figure 3. The non-combined version is 3.7× slower than the combined version in geometric mean on the K80 platform. The primary cause of this is the increased number of instructions by the OpenMP execution flow. For example, our analysis with the CUDA profiler indicates that the number of integer, control flow, and load store instructions for the non-combined version of VecAdd on the Kepler platform is 1.9×, 2.2×, and 5.2× larger than that for the combined version respectively. Additionally, as discussed in (Hayashi et al. 2016), if the runtime employs the state machine execution, the non-combined version requires additional registers for state transitions. This can incur an additional performance degradation on CUDA devices due to less achieved_occupancy.

Overheads of OpenMP Runtime Library: Removing redundant OpenMP runtime library calls is another important compiler optimization even if the combined-directive is used. For example, OpenMP programs on GPUs invoke several OpenMP's offloading runtime library functions of `libomptarget` (Clang-ykt 2017) such as `__kmpc_spmd_kernel_init()`, `__kmpc_for_static_init()`, and `__kmpc_for_static_fini()` to initialize the SPMD program execution, to compute loop ranges for chunked parallel loops, and to finialize the parallel loop execution. In theory, these library calls can be eliminated if the combined directive is used. However, in practice, clang XL C failed to do so even at `-O3` level in some cases and this can add an additional runtime overhead. Table 3 summarizes overheads of the OpenMP runtime on GPUs. We measured these numbers by executing the following synthetic combined-construct program:

```
1 // a[] and b[] are float arrays
2 #pragma omp target teams distribute parallel for \
3             schedule(static, 1) \
4             num_teams(N/1024) thread_limit(1024)
5 for (int i = 0; i < N; i++) {
6   ; // do nothing
7 }
```

Listing 5: A synthetic benchmark for OpenMP's runtime overhead measurements.

Note that the synthetic benchmark is compiled with the same options shown in Section 4.1. We carefully analyzed the generated PTX files and confirmed

	Grid Size (N/1024)	1	64	1024	4096	16384	65536
K80	clang	5.5 us	20.3 us	281.3 us	1.1 ms	4.4 ms	17.6 ms
	XL C	3.6 us	9.2 us	117.9 us	464.6 us	1.8 ms	7.3 ms
P100	clang	1.1 us	1.4 us	7.3 us	26.5 us	103.5 us	411.2 us
	XL C	3.3 us	6.2 us	43.7 us	163.5 us	643.5 us	2.5 ms

Table 3 OpenMP runtime overheads measured with the synthetic combined-construct program (Listing 5) invoked with a block size of 1024.

that 1) clang on the P100 platform completely eliminated OpenMP runtime calls, meaning that it only invokes an empty kernel, 2) clang on the K80 platform failed to eliminate `__kmpc_spmd_kernel_init()`, `__kmpc_for_static_init()`, and `__kmpc_for_static_fini()`, and 3) XL C on the both platforms failed to eliminate `__kmpc_spmd_kernel_init()`. Table 3 shows that the overheads increase as "Grid Size" increases depending on how these compilers eliminate OpenMP runtime calls. That's one of the reasons why the OpenMP variants are slower than the CUDA variants in VecAdd, Saxpy, BlackScholes cases on the both platforms. For example, with VecAdd on the K80 platform, the overhead accounts for 85.0% (= 17.6/20.7) of the execution time of the clang-combined variant and 75.3% (= 7.3/9.7) of that of the xlc-combined and control variants.

These results emphasize the importance of minimizing OpenMP runtime overheads on GPUs.

4.2.3 Math function code generation

Let us consider the BlackScholes case where many math operations are heavily performed. If we subtract the corresponding OpenMP runtime overhead in Table 3 from the kernel execution time to get pure computation time only, the CUDA version is the fastest, the XL C version is the second fastest, and the clang version is the slowest on the both platforms.

This is due to the dynamic number of double- and single-precision instructions. For example, BlackScholes shows the dynamic numbers of double- and single-precision instructions executed by CUDA, clang, and XL C are 0.91×10^9, 1.18×10^9, and 1.17×10^9 respectively. To understand this, consider the following OpenMP program and suppose we have an equivalent CUDA program:

```
1 // a[] and b[] are float arrays
2 #pragma omp target teams distribute parallel for ...
3 for (int i = 0; i < N; i++) {
4     float T = exp(a[i]);        // double exp(double)
5     b[i] = (float)log(a[i])/T; // double log(double)
6 }
```

Listing 6: A synthetic Math benchmark.

Each cell in Table 4 shows absolute performance for each variant where $N = 4,194,304$ on the GPU, which shows similar trends to BlackScholes.

One key issue on the programs is the use of double-precision versions of the `exp` and the `log` functions even though their argument and the resulting value is single-precision. Our analysis shows that the clang compiler keeps the original double-precision math functions, which is why the clang version is the slowest. However, the `nvcc` and XL C compilers 1) generate the single-precision version instead when possible, which significantly eliminates redundant double-precision operations, and 2) also inline these functions in the PTX assembly code to increase opportunities for additional compiler

		CUDA	clang	XL C
K80	Original	472.5 us	734.0 us	725.4 us
	Hand Conversion	472.5 us	`ptxas` error	494.2 us
P100	Original	139.8 us	229.7 us	171.8 us
	Hand Conversion	139.8 us	`ptxas` error	153.3 us

Table 4 GPU kernel time for the synthetic Math benchmark.

optimizations. For the XL C version, the compiler only generates the `expf` and keeps the `log` function. That is why the XL C version is a slightly faster than the clang version.

However, there is a still performance gap between the CUDA and OpenMP versions even if we manually replace `exp` with `expf` and `log` with `logf` (see the second row of Table 4). This can stem from the difference between the CUDA Math API (NVIDIA 2017*c*) used by the `nvcc` compiler and the `libdevice` (NVIDIA 2017*b*) used by the clang and XL C compilers.

For the clang version with the hand conversion, the PTX assembler, or `ptxas`, was aborted due to type mismatch errors, in which the clang compiler mistakenly generates a PTX instruction invoking `exp` with float arguments while `expf` was used in the hand-converted program.

4.2.4 Loop Unrolling Factors

Loop unrolling can increase ILP and reduce control-flow instruction overheads. However, selecting a proper unrolling factor is still an open question. To study this, consider the following source code from MM:

```
#pragma unroll _UNROLLING_FACTOR_
for (int k = 0; k < N; k++) {
  // one offset access and one stride access
  sum += A[i*N+k] * B[k*N+j];
}
```

Listing 7: The inner most loop of MM

Table 5 shows the relationship between unrolling factors and MM's kernel performance numbers for the CUDA and OMP clang variants. The bold characters represent performance numbers obtained with a default unrolling factor by compilers.

	Unrolling Factor	1	2	4	8	16	32
K80	CUDA	229.5 ms	225.02 ms	228.8 ms	**232.1** ms	232.5 ms	230.6 ms
	clang	259.3 ms	**227.1** ms	231.8 ms	232.5 ms	231.2 ms	230.3 ms
P100	CUDA	44.3 ms	64.0 ms	71.4 ms	**74.7** ms	73.1 ms	73.9 ms
	clang	44.1 ms	**65.9** ms	71.0 ms	73.1 ms	73.6 ms	74.6 ms

Table 5 Relationship between unrolling factors and the kernel performance.

In this case, unrolling factors of 2 and 1 achieve the best performance on the K80 and P100 platform respectively. On closer examination with `nvprof` on the P100 platform, the achieved_occupancy of the CUDA variants with an unrolling factor of 8 (74.7 ms) and 1 (44.3 ms) are 84.5% and 96.6% respectively. On the K80 platform, the achieved_occupancy of the CUDA variants with an unrolling factor of 8 (232.1 ms) and 1 (225.0 ms) are 98.9% and 98.4% respectively. This implies that a smaller unrolling

factor can improve the performance of memory-intensive applications. Also, SP has two sequential loops and we observed that the unrolling factors affect the overall kernel performance as well. These observations emphasize the importance of selecting proper unrolling factors.

4.2.5 The read-only data cache (For K80 Only)

The read-only data cache is introduced in the Kepler architecture, but is no longer available in the Pascal architecture. In the Kepler architecture, it is programmer's and compiler's responsibility to control it using `const* __restrict__` keyword and/or `__ldg()` intrinsic. While the read-only data cache can improve memory access efficiency, it does not always contribute performance improvements since the benefit fully depends on memory access patterns during the GPU execution. Based on results of the CUDA versions shown in Figure 2, OMRIQ and SP benefit from the read-only data cache, whereas such is not the case with VecAdd, Saxpy, and BlackScholes, which has poor temporal locality. However, despite its potential spatial and temporal locality, the read-only data cache version of MM in CUDA show the same performance as the version without it. These observations emphasize the importance of data placement optimization.

4.2.6 FMA contraction

The Fused-Multiply-Add (FMA) instruction computes multiply and add operations in a single step. Saxpy is one of the benchmarks that benefit from FMA and the impact of using it can be seen when comparing the combined versions of clang and XL C because the clang compiler does not generate FMA by default. Our analysis with `nvprof` shows the dynamic number of floating point instructions made by the clang version is approximately 2x larger than by the other variants. Note that the clang shows the same number of dynamic instructions as the XL C combined version when `-mllvm -nvptx-fma-level=1 or 2` is enabled, which gives 4.3% and 0.5% performance improvements on the K80 and P100 platforms respectively excluding the OpenMP runtime overheads shown in Table 3.

4.2.7 `schedule(static, 1)` for memory access coalescing

Table 6 shows the relationship between chunk sizes in the `schedule` construct and VecAdd's kernel performance. In terms of global memory access coalescing, it is usually better to specify a chunk size of 1 so that consecutive global memory locations can be accessed by a number of consecutive threads. This is also suitable for using the alternative code generation scheme as we discussed in Section 3.2.1. In our experiments, we observed that both the clang and XL compilers used `schedule(static, 1)` by default unless specified to prevent performance degradation shown in Table 6. However, it is worth mentioning that the initial values of `schedule` and `chunk_size` are "Implementation defined" unless specified by programmers according to the OpenMP specification (OpenMP 2015, pp.36-44 and p.64) and they can be different for different target and/or the host devices.

4.3 Performance Comparison with Highly-tuned Code

This section compares highly-tuned CUDA and OpenMP code using MM, OMRIQ, and SP to study the gap between CUDA and OpenMP variants.

	Chunk Size	1	2	4	8	16	32
K80	clang	20.8 ms	37.4 ms	40.1 ms	52.1 ms	89.8 ms	228.6 ms
	XL C	9.6 ms	13.4 ms	15.3 ms	22.8 ms	42.8 ms	106.2 ms
P100	clang	2.2 ms	2.3 ms	2.5 ms	5.0 ms	16.4 ms	26.1 ms
	XL C	4.7 ms	4.8 ms	5.0 ms	5.7 ms	6.1 ms	10.2 ms

Table 6 Relationship between chunk sizes and the kernel performance.

```
1  #pragma omp target teams distribute ...
2  for (int k = 1; k <= nz2; k++) {
3    #pragma omp parallel for ...
4    for (int j = 1; j <= ny2; j++) {
5      // loop1
6      for (int i = 0; i <= gp01; i++) {
7        rhonX[k*RHONX1 + j*RHONX2 + i] = ...;
8      }
9      // loop2
10     for (int i = 1; i <= nx2; i++) {
11       lhsX[0*LHSX1 + k*LHSX2 + i*LHSX3 + j] =  0.0;
12       ...
13     }
14   }
15 }
```

Listing 8: xsolve3 kernel in SP

4.3.1 SP

Let us take the original implementation of xsolve3 kernel in SP as an example of high-level loop transformation (Listing 8).

Since `lhsX` in the `loop2` (Line 10-13) is accessed contiguously by consecutive threads, memory accesses for `lhsX` are coalesced. However, such is not the case with `rhonX`. Our measurements indicate that the original version written in CUDA achieved an average number of memory transactions per request of 31.8 for loads and 7.0 stores. Note that 32 is the worst possible value and this is caused by uncoalesced memory accesses made in the `loop1`.

For better memory coalescing accesses and additional memory optimizations, we performed loop distribution to break the original loop into two parts: the first part only contains `loop1` and the second part only contains `loop2`, each of which is individually enclosed by the `k-loop` and `j-loop`. Then, only for the first part, permute `i-loop` and `j-loop` for improving memory coalescing efficiency. This can be applied to both CUDA and OpenMP versions. Additionally, we performed loop tiling to allocate tiles on the shared memory for additional memory optimizations. This can be applied to the CUDA version only because OpenMP does not provide a way to allocate variables on the shared memory. We used 32×32 tile size, but the tile size exploration is another important research problem to be addressed in future work.

The impact of the optimizations is summarized in Table 7:

The results show that the "Transformed" version is much faster than the "Original" version. The CUDA profiler shows that the "Transformed" version achieved an average number of memory transactions per request of 1.9 for loads and 1.0 for stores on the K80 platform. This is almost ideal indicating that almost all memory accesses were coalesced (1 is the best possible value). Also, the "Transformed+SharedMemory" version achieves additional performance improvements by exploiting the shared memory.

	Variants	CUDA	clang	XL C
K80	Original	102.4 ms	104.5 ms	174.3 ms
	Transformed	27.1 ms	30.5 ms	39.3 ms
	Transformed+SharedMemory	9.1 ms	-	-
P100	Original	40.9 ms	40.9 ms	65.3 ms
	Transformed	12.6 ms	Error	11.3 ms
	Transformed+SharedMemory	3.5 ms	-	-

Table 7 The impact of hand-optimizations (SP).

4.3.2 OMRIQ

For the highly-tuned CUDA program, we evaluated an optimized CUDA code from our prior work (Shirako et al. 2017), which is comparable to the highly-tuned CUDA implemention (Stone et al. 2008). As with SP, the tuned CUDA code was optimized by performing loop tiling and shared memory allocation. For the OpenMP variants, we also performed loop tiling to see the impact of increasing the temporal locality. Table 8 shows absolute performance numbers for these variants on the Tesla K80 and P100 platforms.

	Variants	CUDA	clang	XL C
K80	Original	14.1 ms	15.7 ms	11.1 ms
	Transformed (Tiled)	12.7 ms	14.2 ms	12.8 ms
	Transformed+SharedMemory	8.8 ms	-	-
P100	Original	2.8 ms	3.9 ms	2.9 ms
	Transformed (Tiled)	2.7 ms	3.9 ms	2.8 ms
	Transformed+SharedMemory	2.3 ms	-	-

Table 8 The impact of hand-optimizations (OMRIQ).

Table 8 shows that utilizing shared memory significantly improves the performance, while loop tiling slightly improves the performance of the OpenMP variants in some cases.

4.3.3 MM

This section discusses the performance differences between 1) a hand-optimized CUDA program and 2) the CUDA and OpenMP variants. For the hand-optimized CUDA program, we evaluated a hand-tuned 2048×2048 matrix multiply CUDA code available from the CUDA SDK (Volkov & Demmel 2008). Table 9 shows absolute performance numbers for these variants on the Tesla K80 and P100 platforms.

Based on results shown in Table 9, the hand-tuned matrix multiply CUDA code is the fastest. As with SP and OMRIQ, the primary cause of the performance gap is that the hand-tuned version performs loop tiling by utilizing the shared memory.

4.4 Lessons Learned

Results show that the OpenMP versions are in some cases faster, in some cases slower than straightforward CUDA implementations written without complicated hand-tuning. Additionally, we conclude further advancements are necessary for OpenMP-enabled compilers to match the performance of highly-tuned CUDA code for some cases examined.

	Variants	CUDA	clang	XL C
K80	Original	231.7 ms	223.1 ms	234.8 ms
	Transformed (Tiling)	192.3 ms	224.9 ms	157.9 ms
	Transformed+SharedMemory	70.6 ms	-	-
P100	Original	74.7 ms	65.9 ms	65.4 ms
	Transformed (Tiling)	49.6 ms	74.6 ms	62.4 ms
	Transformed+SharedMemory	8.6 ms	-	-

Table 9 The impact of hand-optimizations (MM).

Based on our analysis, our suggestions to improve OpenMP programs' performance on GPUs are as follows:

For OpenMP programmers:

1. Using the combined construct (e.g., Listing 1) when possible (Section 3.2.1 and Section 4.2.2).
2. Using `schedule(static, 1)` for better global memory accesses (Section 4.2.7) and for simplifying OpenMP's thread execution scheme on GPUs (Section 3.2.1 and Section 4.2.2).
3. Using Math library functions very carefully (Section 4.2.3).
4. Performing high-level loop transformations and optimizing global memory accesses (e.g., memory access coalescing) like standard CUDA optimizations (Section 4.2.4 and Section 4.3).

For OpenMP compiler designers:

1. Minimizing OpenMP runtime overheads on GPUs when possible (Section 3.2.1 and Section 4.2.2).
2. Constructing a good data placement policy for the read-only cache and the shared memory on GPUs (Section 4.2.5).
3. Improving code generation for GPUs. For example, math functions, memory coalescing, and FMA generation (Section 4.2.3, Section 4.2.7, and Section 4.2.6).
4. Performing high-level loop transformation. For example, the use of the polyhedral model (Shirako et al. 2017) can improve OpenMP programs' performance on GPUs (Section 4.2.4 and Section 4.3).

We believe that those insights are helpful for OpenMP compiler designers to improve the OpenMP compilers so that non-expert programmers can easily get significant performance improvements on GPUs without complicated hand-tuning in future.

5 Related Work

5.1 OpenMP Accelerator Model

There have been several studies on the efficient support of the OpenMP accelerator model on GPUs.

Bercea et al. (Bercea et al. 2015) presented detailed performance analysis of OpenMP 4.0 implementations of LULESH, a proxy application provided by DOE as part of the CORAL benchmark suite, using clang on an NVIDIA K40 GPU. Mitra et al. (Mitra et al. 2014) explored challenges encountered while migrating the general matrix multiplication kernel using an early prototype of the OpenMP 4.0 accelerator model on the TI Keystone II Architecture. In (Martineau et al. 2016), the authors presented detailed performance analysis of OpenMP 4.5 versions of mini-apps using clang on an NVIDIA K40 GPU.

5.2 Compiling High-level/Directive-based Languages to GPUs

Many previous studies aim to facilitate GPU programming by providing high-level abstractions of GPU programming. They often introduce directives and/or language constructs expressing parallelism for semi-/fully- automated code generation and optimizations for GPUs.

OpenACC (OpenACC forum 2015) is a widely-recognized directive-based programming model for heterogeneous systems. In OpenACC, "the user specifies the regions of a host program to be targeted for offloading to an accelerator device." (OpenACC forum 2015, p.7), whereas in OpenMP "the programmer explicitly specifies the actions to be taken by the compiler and runtime system in order to execute the program in parallel." (OpenMP 2015, p.1). This implies that OpenMP is a more prescriptive parallel programming model, but we believe that compiler optimizations are still important to improve programmability and performance portability of high-level GPU programs.

OpenMPC(Lee & Eigenmann 2010) transforms extended OpenMP programs into CUDA applications.

For JVM-based languages, many approaches (Leung et al. 2009, Dubach et al. 2012, Hayashi et al. 2013, Ishizaki et al. 2015) provide high-level abstractions of GPU programming. Velociraptor (Garg & Hendren 2014) compiles MATLAB and Python to GPUs.

In terms of high-level loop transformation for GPUs. The polyhedral model is often used as a basis for GPU optimizations, including parallelization, loop transformations, and shared memory optimizations. As we discussed in Section 4.4, high-level loop transformation is one of the most important compiler optimzations. We plan to leverage some of the insights from prior work (Baskaran et al. 2010, Leung et al. 2010, Vasilache et al. 2012, Shirako et al. 2017).

6 Conclusion

To study potential performance improvements by compiler optimizations for high-level GPU programs, this paper evaluates and analyzes OpenMP benchmarks on IBM POWER8 + NVIDIA Tesla K80 and Tesla P100 platforms. For that purpose, we performed in-depth analysis of hand-written CUDA codes and automatically generated GPU codes by IBM XL and clang/LLVM compilers from the high-level OpenMP programs.

While some of the OpenMP variants show comparable performance to the original CUDA implementation, there are still several missing parts for OpenMP compilers for GPUs. As we discussed in Section 4.3, one open question is how to exploit GPU's memory hierarchy more efficiently by performing high-level loop transformations. Specifically, OpenMP compilers are required to carefully determine several factors including 1)

unrolling factors, 2) distribute chunk sizes, and 3) tile sizes for the read-only cache and the shared memory, 4) leveraging faster Math functions and FMA instructions. Note that 1)-3) are still open questions in the high-performance computing community. One possible solution would be to construct a high-level loop transformation framework with a certain cost model for proper optimization selection.

Further investigation will be required for better compiler optimizations for OpenMP programs.

7 Future Work

For future work, we plan to implement all optimizations we mentioned in this paper. However, there are several unsolved challenges to do so (e.g., tile size selection, unrolling factor exploration and so on). To tackle these challenges, our initial focus is to build a high-level loop transformation framework with a certain cost model for proper optimization selection based on our prior work (Shirako et al. 2017).

Also, selection of the preferred computing resource between CPUs and GPUs for individual kernels remains one of the most important challenges since GPU execution is not always faster than CPUs. Ideally, a preferrable device could be choosen at compile-time and/or runtime using analytical/empirical model. We first plan to add such a capability to the OpenMP runtime by extending prior approaches such as (Hayashi et al. 2015).

A Appendix

Table 10 shows absolute performance numbers for each variant on the K80 and P100 platforms. It is worth mentioning that absolute performace numbers on the P100 are always faster than those on the K80 GPU.

		grid, block	CUDA	CUDA-ROC	clang-combined	clang-control	xlc-combined	xlc-control
K80	VecAdd	65536, 1024	6.1	6.5	20.7	203.4	9.7	9.7
	Saxpy	65536, 1024	6.1	6.5	20.9	210.0	9.9	9.9
	MM	4096, 1024	231.7	230.7	227.1	291.2	234.8	234.8
	BS	4096, 1024	1.3	1.3	2.9	15.9	2.4	2.4
	OMRIQ	32, 1024	14.1	12.9	15.2	14.5	11.1	11.1
	SP	254, 254	102.3	92.2	N/A	104.5	N/A	174.3
P100	VecAdd	65536, 1024	2.2	N/A	2.2	Runtime Error	4.7	4.7
	Saxpy	65536, 1024	2.2	N/A	2.2	Runtime Error	4.7	4.7
	MM	4096, 1024	74.7	N/A	65.9	Runtime Error	65.2	65.4
	BS	4096, 1024	0.4	N/A	0.6	3.9	0.7	0.7
	OMRIQ	32, 1024	2.8	N/A	3.9	Runtime Error	2.9	2.9
	SP	254, 254	40.9	N/A	N/A	Runtime Error	N/A	65.3

Table 10 Absolute performance numbers for each variant in ms.

Acknowledgment

Part of this research is supported by the IBM Centre for Advanced Studies. We would like to thank the anonymous reviewers for their helpful comments and suggestions.

References

Antao, S. F., Bataev, A., Jacob, A. C., Bercea, G.-T., Eichenberger, A. E., Rokos, G., Martineau, M., Jin, T., Ozen, G., Sura, Z., Chen, T., Sung, H., Bertolli, C. & O'Brien, K. (2016), Offloading support for openmp in clang and llvm, *in* 'Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC', LLVM-HPC '16, IEEE Press, Piscataway, NJ, USA, pp. 1–11.

Baskaran, M. M., Ramanujam, J. & Sadayappan, P. (2010), Automatic C-to-CUDA code generation for affine programs, *in* 'Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction', CC'10/ETAPS'10, Springer-Verlag, Berlin, Heidelberg, pp. 244–263.

Bercea, G.-T., Bertolli, C., Antao, S. F., Jacob, A. C., Eichenberger, A. E., Chen, T., Sura, Z., Sung, H., Rokos, G., Appelhans, D. & O'Brien, K. (2015), Performance analysis of openmp on a gpu using a coral proxy application, *in* 'Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems', PMBS '15, ACM, New York, NY, USA, pp. 2:1–2:11.

Bertolli, C., Antao, S. F., Bercea, G.-T., Jacob, A. C., Eichenberger, A. E., Chen, T., Sura, Z., Sung, H., Rokos, G., Appelhans, D. & O'Brien, K. (2015), Integrating gpu support for openmp offloading directives into clang, *in* 'Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC', LLVM '15, ACM, New York, NY, USA, pp. 5:1–5:11.

Bertolli, C., Antao, S. F., Eichenberger, A. E., O'Brien, K., Sura, Z., Jacob, A. C., Chen, T. & Sallenave, O. (2014), Coordinating gpu threads for openmp 4.0 in llvm, *in* 'Proceedings of the 2014 LLVM Compiler Infrastructure in HPC', LLVM-HPC '14, IEEE Press, Piscataway, NJ, USA, pp. 12–21.

Clang-ykt (2017), 'LLVM OpenMP* Offloading Runtime Library (libomptarget)', [online] `https://github.com/clang-ykt/openmp/tree/master/libomptarget` (Accessed 12 February 2017).

Department of Energy (2014), 'Department of energy awards $425 million for next generation supercomputing technologies', [online] `http://energy.gov/articles/department-energy-awards-425-million-next-generation-supercomputing-technologies` (Accessed 12 February 2017).

Dubach, C., Cheng, P., Rabbah, R., Bacon, D. F. & Fink, S. J. (2012), Compiling a high-level language for gpus: (via language support for architectures and compilers), *in* 'Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation', PLDI '12, ACM, New York, NY, USA, pp. 1–12.

Garg, R. & Hendren, L. (2014), Velociraptor: An embedded compiler toolkit for numerical programs targeting cpus and gpus, *in* 'Proceedings of the 23rd International Conference on Parallel Architectures and Compilation', PACT '14, ACM, New York, NY, USA, pp. 317–330.

Hayashi, A., Grossman, M., Zhao, J., Shirako, J. & Sarkar, V. (2013), Accelerating Habanero-Java Programs with OpenCL Generation, *in* 'Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools', PPPJ '13, pp. 124–134.

Hayashi, A., Ishizaki, K., Koblents, G. & Sarkar, V. (2015), Machine-learning-based performance heuristics for runtime cpu/gpu selection, *in* 'Proceedings of the Principles and Practices of Programming on The Java Platform', PPPJ '15, ACM, New York, NY, USA, pp. 27–36.

Hayashi, A., Shirako, J., Tiotto, E., Ho, R. & Sarkar, V. (2016), Exploring compiler optimization opportunities for the openmp 4.x accelerator model on a power8+gpu platform, *in* '2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)', pp. 68–78.

Ishizaki, K., Hayashi, A., Koblents, G. & Sarkar, V. (2015), Compiling and optimizing java 8 programs for gpu execution, *in* '2015 International Conference on Parallel Architecture and Compilation (PACT)', pp. 419–431.

KHRONOS GROUP (2015), 'The OpenCL Specification'. [online] `https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf` (Accessed 12 February 2017).

Kim, J., Lee, Y. J., Park, J. & Lee, J. (2016), Translating openmp device constructs to opencl using unnecessary data transfer elimination, *in* 'SC16: International Conference for High Performance Computing, Networking, Storage and Analysis', pp. 597–608.

Lattner, C. & Adve, V. (2004), LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, CGO '04, IEEE Computer Society, Washington, DC, USA, pp. 75–.

Lee, S. & Eigenmann, R. (2010), Openmpc: Extended openmp programming and tuning for gpus, *in* 'Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis', SC '10, IEEE Computer Society, Washington, DC, USA, pp. 1–11.

Leung, A., Lhoták, O. & Lashari, G. (2009), Automatic parallelization for graphics processing units, *in* 'Proceedings of the 7th International Conference on Principles and Practice of Programming in Java', PPPJ '09, pp. 91–100.

Leung, A., Vasilache, N., Meister, B., Baskaran, M., Wohlford, D., Bastoul, C. & Lethin, R. (2010), A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction, *in* 'Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units', GPGPU-3, ACM, New York, NY, USA, pp. 51–61.

LLVM.org (2017*a*), 'OpenMP: Support for the OpenMP language', [online] `http://openmp.llvm.org/` (Accessed 12 February 2017).

LLVM.org (2017*b*), 'User Guide for NVPTX Back-end'. [online] `http://llvm.org/docs/NVPTXUsage.html` (Accessed 12 February 2017).

Martineau, M., McIntosh-Smith, S., Bertolli, C., Jacob, A. C., Antao, S. F., Eichenberger, A., Bercea, G.-T., Chen, T., Jin, T., O'Brien, K., Rokos, G., Sung, H. & Sura, Z. (2016), Performance analysis and optimization of clang's openmp 4.5 gpu support, *in* 'Proceedings of the 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems', PMBS '16, IEEE Press, Piscataway, NJ, USA, pp. 54–64.

Mitra, G., Stotzer, E., Jayaraj, A. & Rendell, A. P. (2014), Implementation and Optimization of the OpenMP Accelerator Model for the TI Keystone II Architecture, *in* 'Proceedings of the 10th International Workshop on OpenMP (IWOMP '14)'.

NVIDIA (2017*a*), 'CUDA C Programming Guide version 8.0'. [online] `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf` (Accessed 12 February 2017).

NVIDIA (2017*b*), 'Libdevice User's Guide'. [online] `http://docs.nvidia.com/cuda/pdf/libdevice-users-guide.pdf` (Accessed 12 February 2017).

NVIDIA (2017*c*), 'NVIDIA CUDA Math API'. [online] `http://docs.nvidia.com/cuda/pdf/CUDA_Math_API.pdf` (Accessed 12 February 2017).

NVIDIA (2017*d*), 'NVVM IR specification 1.3', [online] `http://docs.nvidia.com/cuda/pdf/NVVM_IR_Specification.pdf` (Accessed 12 February 2017).

NVIDIA (2017*e*), 'PARALLEL THREAD EXECUTION ISA v5.0'. [online] `http://docs.nvidia.com/cuda/pdf/ptx_isa_5.0.pdf` (Accessed 12 February 2017).

NVIDIA (2017*f*), 'PROFILER USER'S GUIDE 8.0'. [online] `http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf` (Accessed 12 February 2017).

OpenACC forum (2015), 'The OpenACC Application Programming Interface, Version 2.5'. [online] `http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf` (Accessed 12 February 2017).

OpenMP (2015), 'OpenMP Application Program Interface, version 4.5'. [online] `http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf` (Accessed 12 February 2017).

Shirako, J., Hayashi, A. & Sarkar, V. (2017), Optimized two-level parallelization for gpu accelerators using the polyhedral model, *in* 'Proceedings of the 26th International Conference on Compiler Construction', CC 2017, ACM, New York, NY, USA, pp. 22–33.

SPEC (2015), 'SPEC ACCEL benchmark'. [online] `https://www.spec.org/accel/` (Accessed 12 February 2017).

Stone, S. S., Haldar, J. P., Tsao, S. C., Hwu, W. m. W., Sutton, B. P. & Liang, Z. P. (2008), 'Accelerating advanced mri reconstructions on gpus', *J. Parallel Distrib. Comput.* **68**(10), 1307–1318.

Vasilache, N., Meister, B., Baskaran, M. & Lethin, R. (2012), Joint scheduling and layout optimization to enable multi- level vectorization, *in* 'IMPACT-2: 2nd International Workshop on Polyhedral Compilation Techniques, Paris, France, January', Paris, France.

Volkov, V. & Demmel, J. W. (2008), Benchmarking gpus to tune dense linear algebra, *in* 'Proceedings of the 2008 ACM/IEEE Conference on Supercomputing', SC '08, IEEE Press, Piscataway, NJ, USA, pp. 31:1–31:11.