Exploring Compiler Optimization Opportunities for the OpenMP 4.x Accelerator Model on a POWER8+GPU Platform

Akihiro Hayashi*‡, Jun Shirako*, Ettore Tiotto†, Robert Ho†, Vivek Sarkar*
*Rice University, †IBM Canada, ‡Email: ahayashi@rice.edu

***Abstract*—While GPUs are increasingly popular for high-performance computing, optimizing the performance of GPU programs is a time-consuming and non-trivial process in general. This complexity stems from the low abstraction level of standard GPU programming models such as CUDA and OpenCL: programmers are required to orchestrate low-level operations in order to exploit the full capability of GPUs. In terms of software productivity and portability, a more attractive approach would be to facilitate GPU programming by providing high-level abstractions for expressing parallel algorithms.**

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 parallel programming strategies generally impose additional program optimizations on compilers, which could result in lower performance than fully hand-tuned code with low-level programming models.To study potential performance improvements by compiling and optimizing high-level GPU programs, in this paper, we 1) evaluate a set of OpenMP 4.x benchmarks on an IBM POWER8 and NVIDIA Tesla GPU platform and 2) conduct a comparable performance analysis among hand-written CUDA and automatically-generated GPU programs by the IBM XL and clang/LLVM compilers.

***Index Terms*—Parallel programming**

I. 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 [19] and OpenCL [10] 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 [23] 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 4.x 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 4.x `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 4.x 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 [1].

However, aside from the improved programmability and performance portability, mapping the high-level OpenMP programs to GPUs imposes technical challenges on compiler optimizations: 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 4.x programs 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 4.x benchmarks on an IBM POWER8 and NVIDIA Tesla GPU platform.
- Detailed performance analysis among hand-written CUDA and automatically generated GPU programs by

WACCPD16; Salt Lake City, Utah, USA; November 2016
978-1-5090-6152-5/16/$31.00 ©2016 IEEE

[1] clang+LLVM also supports Intel Xeon Phi.

development/beta versions of the IBM XL C and clang+ LLVM compilers to exlore 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 without complicated hand-tuning.
- Additionally, results show that more work must be done for OpenMP-enabled compilers to match the performance of highly-tuned CUDA code for some cases examined. The possible compiler optimization strategies for OpenMP programs are:
  1) avoiding state machine execution on GPUs [4], [3] 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 math and fused multiply-add operations.
  4) performing high-level loop transformation (e.g. using the polyhedral model [24]).

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 4.x unless otherwise indicated.

The paper is organized as follows. Section II provides background information on GPUs and the OpenMP 4.x accelerator model. Section III shows an overview of clang+LLVM and XL C compilers that compile OpenMP 4.x programs to GPUs. Section IV presents an extensive performance evaluation and analysis on a single GPU paltform. Section V, Section VI, and Section VII summarize related work, conclusions, and future work.

II. The OpenMP Accelerator Model

A. 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 [19] 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) 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.

```
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.

B. OpenMP 4.x 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.

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 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

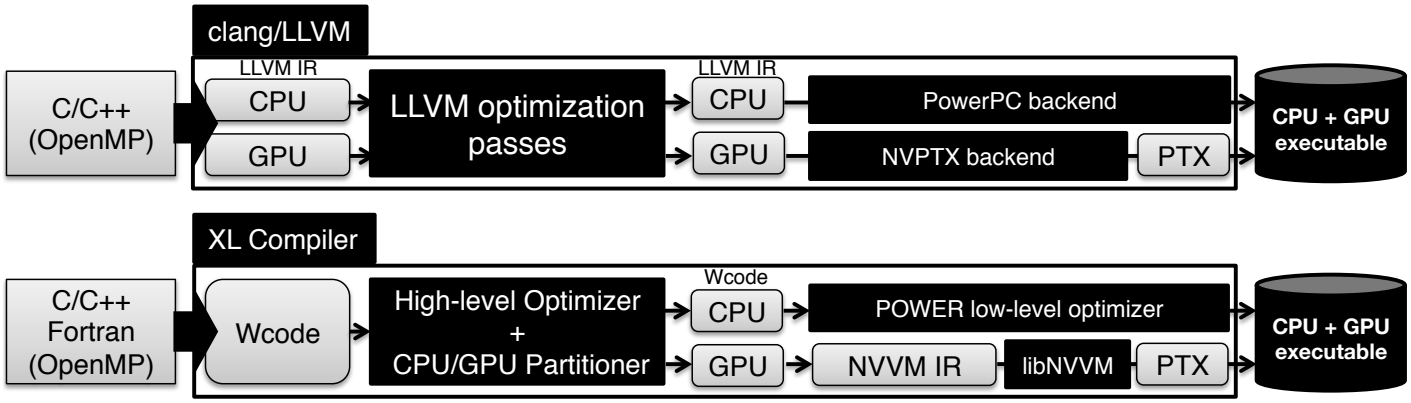

Fig. 1. Compilation flow of clang+LLVM and XL compilers for C/C++ and C/C++/Fortran respectively.

```
#pragma omp target teams { // GPU region
    // sequential region 1 executed
    // by the master thread of each team
    if (...) {
      // parallel region 1
      #pragma omp parallel for
      for () {}
    } else {
      ...
    }
}
```

Listing 2. OpenMP 4.x code that requires multiple execution modes on GPUs.

```
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.

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.

III. Compiling OpenMP 4.x to GPUs

This section describes a brief overview of the OpenMP 4.x compilers and their optimizations and code generation used for performance evaluation in this paper.

A. Compilers

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

1) clang+LLVM Compiler: LLVM [11] is a widely used compiler infrastructure and clang is a C language family front-end for LLVM. Clang first transforms OpenMP 4.x programs to LLVM's intermediate representation (LLVM IR) and then the LLVM compiler applies language- and target-independent optimization passes to LLVM IR [1].

To support GPU code generation, the clang compiler outlines GPU kernels specified by OpenMP 4.x `target` directives as separate LLVM functions and the LLVM functions are fed into standard LLVM passes followed by the NVPTX backend [14] for PTX assembly [20] 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.

2) IBM XL Compiler: Our beta XL compiler for OpenMP 4.x 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 4.x `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

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™ [25]	262,114	1-level
SP-xsolve3	Scalar Penta-diagonal solver from SPEC ACCEL™ [25]	5 × 255 × 256 × 256	2-level

TABLE I
BENCHMARKS FROM POLYBENCH AND SPEC ACCEL USED IN OUR EVALUATION

is translated into an NVVM IR [18] 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 [20].

B. Optimizing OpenMP 4.x programs for GPUs

This section describes possible optimizations for OpenMP 4.x programs. We mainly focus on significant optimizations affecting performance as shown in the performance results in Section IV.

1) Simplifying the OpenMP Execution Model on GPUs: In OpenMP 4.x 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 4.x programs can switch back and forth between sequential and parallel regions, thereby code generation for such program is challenging because it can require state machine code generation on GPUs (see Listing 3).

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 [4], [3]. In the following, "*control-loop*" is referred to as a loop that controls state transition on GPUs.

For the purpose of optimizations, the control-loop can be eliminated when the body of the target region satisfies the following conditions [3]:

- 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.

The clang+LLVM compiler additionally requires programmers to use a combined construct [23], a shortcut for specifying multiple constructs in a single line (see also Section II-B), whereas the XL C compiler can remove the control-loop even with a non-combined construct.

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 4.x programs, it is the compiler's responsibility to perform such optimizations since OpenMP 4.x 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 IV-C7.

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 finding an optimal unrolling factor can be found in Section IV-C7.

IV. PERFORMANCE EVALUATION

This section presents the results of an experimental evaluation of the XL C and the clang+LLVM compilers on an IBM POWER8 and NVIDIA Tesla K80 platform. The GPU's error-correcting code (ECC) feature was turned on in our experiment.

A. Experimental protocol

Purpose: Our goal is to study potential compiler optimizations for OpenMP 4.x programs in terms of kernel performance. We do not focus on data transfers between the host and GPU devices in this paper. Note that OpenMP 4.x has support for optimizing communications using the `map` clause. For that purpose, we focus on the performance difference among CUDA and OpenMP variants of benchmarks. The CUDA variant employs straightforward GPU parallelization strategies without complicated hand-tuning. However, we also discuss the performance difference between highly-tuned CUDA code and the OpenMP variants in Section IV-C6.

Machine: The platform consists of a multicore IBM POWER8 CPU and an NVIDIA Tesla K80. The platform has two 12-core IBM POWER8 CPUs (S824), operating at up to 3.52GHz with a total 1TB of main memory. Each core is capable of

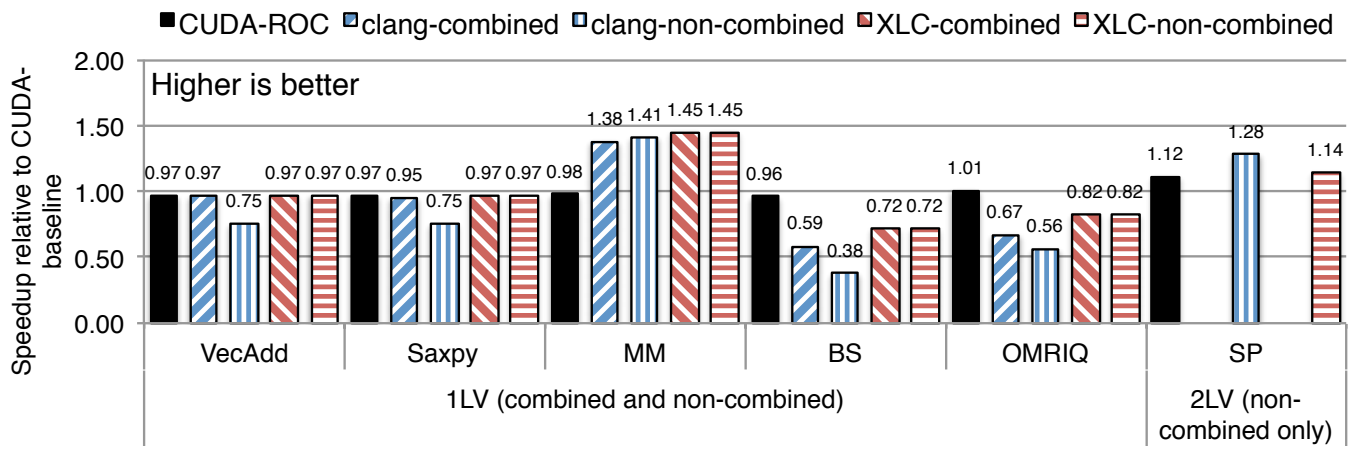

Fig. 2. Relative performance over CUDA-baseline (non-ROC version) on the IBM POWER8 + NVIDIA Tesla K80 platform.

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.

Benchmarks: Table I 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. For "Data Size", Table I 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 (control-loop)" versions. "2-level" means we parallelized nested loops at different levels, which only generates the "control-loop" version.

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 IV-C2).
  - CUDA-ROC: all read-only arrays within a kernel are accessed through the read-only data cache[2]. Since the OpenMP compilers utilize the read-only data cache by default, we focus on the performance difference between CUDA-ROC and the OpenMP variants below.
- **OpenMP:** Combined and non-combined constructs versions compiled by the following compilers. For more details, see Section II-B and Section III.
  - clang+LLVM compiler
  - XL C compiler

[2]read-only arrays are specified with `const* __restrict__`.

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 and BlackScholes cases, we used a grid size of 512 and a block size of 1024. For OMRIQ we used a grid size of 512 and a block size of 256. Also, a grid size of 254 and a block size of 254 were used for SP.

For the CUDA variants, we used the CUDA compiler driver (`nvcc`) 7.5.17 with `-O3 -arch=sm_35`. For OpenMP variants, a development version of clang 3.8 with `-O3-target powerpc64le-ibm-linux-gnu-mcpu=pwr8 -fopenmp =libomp -omptargets=nvptx64sm_35-nvidia-linux` and IBM XL C/C++ compiler version 13.1.4 with `-O3 -qsmp=omp`[3] were used. For single-precision floating point operations, `-ftz=false --prec-div=true --prec-sqrt =true` was used for all variants. Also, `-maxrregcount 64` was specified to limit 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)** which is obtained with NVIDIA CUDA profiler, or `nvprof` [21], and exclude host-device data transfer times.

B. Summary of Results

This section outlines the results shown in Figure 2, which shows speedup factors relative to the baseline CUDA implementation (CUDA) on the NVIDIA Tesla K80 GPU.

Overall, for both clang and XL C compilers, the combined version is comparable to the CUDA-ROC version in many cases. While the non-combined (control-loop) version with clang is slower than the combined version due to state machine execution on GPUs in general, the non-combined version

[3]This option internally specifies `-arch=compute_35` and `-opt=3` for the libNVVM library.

using XL C does not show any performance degradation (see Section IV-C1 for more details). Also, the results show that utilizing the read-only data cache does not always improve GPU kernel performance (see Section IV-C2). For MM, the OpenMP variants show performance improvements of up to 1.45×. This is mainly due to better read-only data cache accesses (see Section IV-C2 as well). For BlackScholes and OMRIQ, the CUDA variants are better than the OpenMP variants due to more efficient math function code generation (see Section IV-C3). For SP, the OpenMP variants show better performance than the CUDA variant due to selecting proper unrolling factors (see Section IV-C4).

Also, Section IV-C5 conducts additional experiments to show the impact of performing high-level loop transformations, and Section IV-C6 discusses the performance comparisons with hand-optimized CUDA code.

Section IV-C7 summarizes our insights on compiler optimizations for OpenMP 4.x programs.

C. Key Insights

This section discusses our insights on future compiler optimizations for OpenMP 4.x accelerator programs targeting GPUs. Also, we quantitatively analyze the impact of each factor using HW counter numbers obtained with `nvprof` shown in Table VI.

1) Remove control-loops when possible: As we discussed in Section III-B1, the non-combined version may require state machine execution on GPUs. An important compiler optimization is the removal of the state machine code if possible. The XL C compiler supports such an optimization, which resulted in no performance degradation even with non-combined constructs as shown in Figure 2.

The impact of state machine code removal is obvious by comparing the combined and non-combined versions by the clang compiler shown in Figure 2. The non-combined version is 23.6% slower than the combined version on average among VecAdd, Saxpy, BlackScholes, and OMRIQ. The primary cause of this is the increased number of instructions by the state transitions code. For example, consider VecAdd in Table VI. The number of integer, control flow, and load store instructions for the non-combined version is 2.96×, 2.38×, and 1.18× larger than that for the combined version respectively. Also, the non-combined version requires additional registers for state transitions - i.e. 56 regs for the non-combined version and 10 regs for the combined version. This can incur an additional performance degradation on CUDA devices with compute capability 3.5 or lower due to less achieved_occupancy[4].

2) Utilize the read-only data cache carefully: 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 BS, which has poor temporal locality. However, despite its potential spatial and temporal locality, the read-only data cache version of MM in CUDA is 2% slower than the version without it. These observations emphasize the importance of data placement optimization.

Let us focus on the OpenMP versions of MM that outperform the CUDA versions. In that case, some of the compiler optimizations can inhibit/improve GPU performance. On closer examinations, we observed two important things that current GPU compilers do not handle perfectly:

How to determine an optimal unrolling factor? We first analyze the performance difference between the CUDA-ROC version (0.98× over the baseline) and the clang-combined version (1.38× over the baseline). Based on our analysis, we see that a bigger loop unrolling factor can decrease L2 cache hit rate for read requests from the read-only data cache in MM. To be more specific, consider the following source code:

```
1 for (int k = 0; k < N; k++) {
2   // one offset access and one stride access
3   sum += A[i*N+k] * B[k*N+j];
4 }
```

Listing 4. The inner most loop of MM

For the CUDA-ROC version, the `nvcc` compiler unrolls the loop by a factor of 8, which results in 14.5% L2 hit rate from the read-only data cache. However, the clang compiler unrolls the loop by a factor of 2, which achieves 91.9% L2 hit rate and 1.41× performance improvement compared to the CUDA-ROC version (look for `L2 Hit (Texture Read)` for MM in Table VI). It is worth mentioning that the CUDA-ROC version shows the same performance as the clang-combined version if `#pragma unroll 2` is specified. While loop unrolling can increase instruction-level parallelism (ILP) and amortize loop overheads, in this case, multiple offset and stride access requests issued simultaneously had incurred the performance penalty due to the read-only data cache's line conflicts.

How to optimize memory access patterns under distributed chunking? Next, we investigate why the clang-non-combined version outperforms the clang-combined version. A key difference between them is shown in the `dist_schedule` clause. The current implementation of the clang compiler ignores the `dist_schedule` clause in the case of combined constructs, which could incur poor memory access patterns. To be more specific, consider pseudo code in Listing 5 that are equilavent to the CUDA variants and the clang-combined variants. `idx` is used to delinearize the 1-D global iteration space to 2-D space and `blockDim.x * gridDim.x` is the stride for the next element accessed on the same thread. If the stride is too long, this could cause poor L2 cache rate. Table II shows details of the stride size and the unrolling factor for MM. Since the stride size in the CUDA and clang-combined version is 524k elements ($= 512 \times 1024$), this is the primary cause of the lower L2 cache rate discussed above. On the other hand, in

[4] achived_occupancy in Table VI is not worse since compute capability 3.7 device allows 128 integer registers per thread.

```
// NxN Matrix
int UB = N*N;
// CUDA, CUDA-ROC, clang-combined
int gid = blockDim.x * blockIdx.x + threadIdx.x;
if (gid < UB) {
  for (int idx = gid; j < UB;
       idx += blockDim.x * gridDim.x) {
       // each thread delinearizes idx to (i, j)
       int i = idx / N;
       int j = idx % N;
       ...
```

Listing 5. Pseudo code for MM

Table II the clang-combined and the XLC version has a smaller stride, which can relatively fit the read-only data cache.

	CUDA	clang-combined	clang-control	XL C
the stride size	524,288 (`blockDim.x * gridDim.x`)		1,024	
unrolling factor@PTX	8	2	2	8
who did unroll?	nvcc	clang	clang	libNVVM

TABLE II
DETAILS OF THE STRIDE SIZE AND THE UNROLLING FACTOR FOR MM.

To summarize, selecting an optimal unrolling factor and distribute chunk size is still an open question and we believe that an analytical/empirical model would be required so as to address this important challenge.

3) Improve math function code generation: Let us consider the BlackScholes and OMRIQ cases where many math operations are performed. Figure 2 shows that the CUDA version is the fastest, the XL C version is the second fastest, and the clang version is the slowest in both benchmarks.

This is due to the dynamic number of double-precision instructions (see Table VI). For example, BlackScholes shows the dynamic numbers of double-precision instructions executed by CUDA, clang, and XL C are 2.9×10^7, 5.8×10^8, and 4.1×10^8 respectively. To understand this, consider the following OpenMP program and suppose we have an equivalent CUDA program:

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

Listing 6. A synthetic Math benchmark.

The first row in Table III shows absolute performance for each variant where $N = 4,194,304$ on the GPU, which shows similar trends to BlackScholes and OMRIQ.

	CUDA	clang-combined	xlc-combined
Compiler	515 us	933 us	922 us
Hand Conversion (expf, logf)	515 us	link error	721 us

TABLE III
GPU KERNEL TIME FOR THE SYNTHETIC MATH BENCHMARK.

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, and these functions are not inlined even at SASS assembly code level[5], 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 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 bit faster than the clang version.

However, there is a still performance gap between the CUDA and OpenMP versions even if we perform hand-conversion (see the second row of Table III). This can stem from the difference between the CUDA Math API [17] used by the `nvcc` compiler and the Libdevice [16] used by the clang and XL C compilers.

4) Other importatnt insights: This section discusses other important factors affecting GPU performance.

Perform 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 (2% performance improvement in this case). Table VI 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 performance as the XL C combined version when `-mllvm -nvptx-fma-level=1or2` is enabled.

Loop unrolling for better ALU utilization: Besides what we discussed in Section IV-C2, loop unrolling can increase ILP and reduce control-flow instruction overheads. SP has two sequential loops and the unrolling factors affect overall kernel performance. To reiterate, finding an optimal unrolling factor is very important for future compiler optimizations for OpenMP programs on GPUs.

Use `schedule(static, 1)` for memory access coalescing: 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 removing control-loops as we discussed in Section III-B1.

5) Additional Experiments: Impact of high-level loop transformations: High-level loop transformations are compiler optimizations to transform loop structures in sequential and parallel programs while keeping program semantics. Let us take the original implementation of xsolve3 kernel in SP as an example of high-level loop transformation (Listing 7). 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 shown in Table VI 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

[5]SASS is the lowest level assembly code accessible from programmers.

```
#pragma omp target teams distribute ...
for (int k = 1; k <= nz2; k++) {
  #pragma omp parallel for ...
  for (int j = 1; j <= ny2; j++) {
    // loop1
    for (int i = 0; i <= gp01; i++) {
      rhonX[k*RHONX1 + j*RHONX2 + i] = ...;
    }
    // loop2
    for (int i = 1; i <= nx2; i++) {
      lhsX[0*LHSX1 + k*LHSX2 + i*LHSX3 + j] =  0.0;
      ...
    }
  }
}
```

Listing 7. xsolve3 kernel in SP

worst possible value and this is caused by uncoalesed memory accesses made in the `loop1`.

For better memory coalescing accesses and additional memory optimizations, we performed the following optimizations manually:

Loop transformations: Perform 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.

Shared memory utilization: Perform loop tiling to allocate tiles on the shared memory for additional memory optimizations. This can be applied to the CUDA version only because OpenMP 4.x 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 IV:

	CUDA	clang	XL C
Original	91.9 ms	80.0 ms	89.9 ms
Transformed	21.6 ms	30.2 ms	28.9 ms
Transformed+SharedMemory	9.1 ms	-	-

TABLE IV
THE IMPACT OF ADDITIONAL OPTIMIZATIONS

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. 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.

6) Additional Experiments: Performance Comparison with Hand-tuned Code: 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 [26]. Table V shows absolute performance numbers for these variants on the IBM POWER8 with Tesla K80 platform in descending order.

Lang.	Variants	Absolute Performance
CUDA	Straightforward (w/ ROC)	228.4 ms
CUDA	Straightforward (w/o ROC)	223.0 ms
OpenMP	clang+LLVM (combined)	161.5 ms
OpenMP	clang+LLVM (non-combined)	158.0 ms
OpenMP	XL C	153.9 ms
CUDA	hand-tuned (w/o ROC)	70.6 ms
CUDA	hand-tuned (w/ ROC)	64.3 ms

TABLE V
ABSOLUTE PERFORMANCE COMPARISON (MM)

Based on results shown in Table V, the hand-tuned matrix multiply CUDA code is the fastest. The primary cause of the performance gap is that the hand-tuned version performs loop tiling to allocate tiles on the shared memory as we discussed in Section IV-C5.

7) Overall Summary & Discussion: While some of the OpenMP variants show comparable performance to the original CUDA implementation, there are still several missing parts for OpenMP 4.x compilers for GPUs. As we discussed in Section IV-C5 and IV-C6, one open question is how to exploit GPU's memory hierarchy more efficiently by performing high-level loop transformations. Specifically, OpenMP 4.x 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. For instance, the use of the polyhedral and AST-based transformation [24] could be an option for such an optimization framework.

V. RELATED WORK

A. OpenMP 4.x Accelerator Model

There have been several studies on the efficient support of the OpenMP accelerator model on GPUs. Bercea et al. [2] presented detailed performance analysis of OpenMP 4.0 implementations of LULESH, a proxy application provided by DOE as part of the CORAL benchmark suite. Mitra et al. [15] 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.

B. 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 [22] is a widely-recognized directive-based programming model for heterogeneous systems. OpenMPC[12] transforms extended OpenMP programs into CUDA applications. For JVM-based languages, many approaches [13], [5], [7], [9] provide high-level abstractions of GPU programming. Velociraptor [6] compiles MATLAB and Python to GPUs.

VI. Conclusion

To study potential performance improvements by compiler optimizations for high-level GPU programs, this paper evaluates and analyzes OpenMP 4.x benchmarks on an IBM POWER8 + NVIDIA Tesla K80 platform. 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 4.x programs.

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. Based on our analysis, the possible compiler optimizations to improve OpenMP programs' performance on GPUs are as follows:

1) avoiding the state machine execution on GPUs [4], [3] 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 math and fused multiply-add operations.
4) performing high-level loop transformation (e.g. using the polyhedral model [24]).

We believe that OpenMP 4.x compilers have to carefully perform these optimizations, but some of them are still open questions. Further investigation will be required for better compiler optimizations for OpenMP 4.x programs.

VII. 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. To tackle these challenges, our initial focus is to build a high-level loop transformation framework.

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 compiler/runtime using analytical/empirical model. We first plan to add such a capability to the OpenMP 4.x runtime by extending prior approaches such as [8].

Acknowledgment

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

References

[1] Github repository for exetended clang implementation supporting OpenMP 4.0. https://github.com/clang-omp/.

[2] G.-T. Bercea, C. Bertolli, S. F. Antao, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, and K. O'Brien. 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, pages 2:1–2:11, New York, NY, USA, 2015. ACM.

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

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

[5] C. D. et al. 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, pages 1–12, New York, NY, USA, 2012. ACM.

[6] R. Garg and L. Hendren. 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, pages 317–330, New York, NY, USA, 2014. ACM.

[7] A. Hayashi, M. Grossman, J. Zhao, J. Shirako, and V. Sarkar. 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, pages 124–134, 2013.

[8] A. Hayashi, K. Ishizaki, G. Koblents, and V. Sarkar. 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, pages 27–36, New York, NY, USA, 2015. ACM.

[9] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar. Compiling and optimizing java 8 programs for gpu execution. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 419–431, Oct 2015.

[10] KHRONOS. OpenCL. http://www.khronos.org/opencl/.

[11] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[12] S. Lee and R. Eigenmann. 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, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

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

[14] LLVM.org. User Guide for NVPTX Back-end. http://llvm.org/docs/NVPTXUsage.html.

[15] G. Mitra, E. Stotzer, A. Jayaraj, and A. P. Rendell. 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)*, 2014.

[16] NVIDIA. Libdevice User's Guide. http://docs.nvidia.com/cuda/libdevice-users-guide.

[17] NVIDIA. NVIDIA CUDA Math API. http://docs.nvidia.com/cuda/cuda-math-api/.

[18] NVIDIA. NVVM IR specification 1.1. http://docs.nvidia.com/cuda/nvvm-ir-spec/index.html.

[19] NVIDIA. CUDA C Programming Guide version 7.5, 2015. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[20] NVIDIA Corporation. PARALLEL THREAD EXECUTION ISA v4.1, 2014. http://docs.nvidia.com/cuda/pdf/ptx_isa_4.1.pdf.

[21] NVIDIA Corporation. PROFILER USER'S GUIDE 7.5, 2015. http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf.

[22] OpenACC forum. The OpenACC Application Programming Interface, Version 2.0, 2013. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf.

[23] OpenMP Application Program Interface, version 4.0, July 2013. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[24] J. Shirako, L. N. Pouchet, and V. Sarkar. Oil and water can mix: An integration of polyhedral and ast-based transformations. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 287–298, Nov 2014.

[25] SPEC. SPEC ACCEL benchmark. https://www.spec.org/accel/.

[26] 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, Piscataway, NJ, USA, 2008. IEEE Press.

		CUDA		OpenMP (clang)		OpenMP (XL C)
	Metrics	**w/ ROC**	**w/o ROC**	**combined**	**control**	-
VecAdd (see Section IV-C1)	Kernel Time	6.3 ms	6.1 ms	6.3 ms	8.1 ms	6.3 ms
	Registers	10	12	10	**56**	33
	Occupancy	98.1%	94.7%	98.2%	99.3%	98.3%
	Global Load Transactions per request	0.0	1.0	0.0	3.2	0.0
	Global Store Transactions per request	1.0	1.0	1.0	1.4	1.0
	Float-Point Instructions (Single)	6.7E+07	6.7E+07	6.7E+07	6.7E+07	6.8E+07
	Integer Instructions	5.4E+08	5.4E+08	5.4E+08	**1.6E+09**	1.2E+09
	Control Flow Instructions	6.7E+07	6.7E+07	6.7E+07	**1.6E+08**	6.8E+07
	Load Store Instructions	2.7E+08	2.0E+08	2.7E+08	**3.2E+08**	2.7E+08
Saxpy (see Section IV-C4)	Kernel Time	6.3 ms	6.1 ms	6.4 ms	8.1 ms	6.3 ms
	Float-Point Instructions (Single)	6.7E+07	6.7E+07	**1.3E+08**	**1.3E+08**	6.8E+07
MM (see Section IV-C2)	Kernel Time	228.4 ms	223.0 ms	161.5 ms	158.0 ms	153.9 ms
	Registers	36	29	24	61	52
	Occupancy	98.4%	98.4%	98.8%	99.6%	99.7%
	Texture Cache Hit Rate	74.6%	N/A	74.8%	74.8%	74.8%
	L2 Hit (Texture Read)	**14.5%**	N/A	**91.9%**	**93.1%**	**77.2%**
	Global Load Transactions per request	1.0	1.0	1.0	2.6	1.0
	Global Store Transactions per request	1.0	1.0	1.0	3.7	1.0
	Float-Point Instructions (Single)	8.6E+09	8.6E+09	1.7E+10	1.7E+10	8.6E+09
	Integer Instructions	3.8E+10	8.1E+09	4.73E+10	4.74E+10	4.3E+10
	Control-Flow Instructions	5.4E+08	5.4E+08	4.29E+09	4.31E+09	1.1E+09
	Load/Store Instructions	2.4E+10	1.7E+10	2.57E+10	2.58E+10	2.57E+10
BlackScholes (see Section IV-C3)	Kernel Time	1.93 ms	1.85 ms	3.16 ms	4.87 ms	2.58 ms
	Registers	29	29	36	64	56
	Occupancy	97.5%	97.6%	97.8%	99.1%	98.1%
	Global Load Transactions per request	0.0	1.0	0.0	3.2	0.0
	Global Store Transactions per request	1.0	1.0	1.0	3.0	1.0
	Float-Point Instructions (Single)	**7.0E+08**	**7.0E+08**	**4.5E+08**	**4.5E+08**	**3.4E+08**
	Float-Point Instructions (Double)	**2.9E+07**	**2.9E+07**	**5.8E+08**	**5.8E+08**	**4.1E+08**
	Integer Instructions	2.4E+08	2.4E+08	2.6E+08	3.7E+08	3.8E+08
	Control-Flow Instructions	1.9E+08	1.9E+08	2.9E+08	3.3E+08	2.0E+08
	Load/Store Instructions	1.7E+07	1.3E+07	1.7E+07	1.0E+08	2.5E+07
OMRIQ (see Section IV-C3)	Kernel Time	522.5 ms	526.5 ms	780.3 ms	933.4 ms	638.4 ms
	Registers	26	26	29	64	41
	Occupancy	98.8%	98.7%	98.2%	96.9%	97.1%
	Texture Cache Hit Rate	99.9%	N/A	99.8%	99.9%	99.9%
	L2 Hit (Texture Read)	98.6%	N/A	98.3%	97.8%	96.5%
	Global Load Transactions per request	1.0	1.0	1.0	2.1	1.0
	Global Store Transactions per request	1.0	1.0	1.0	2.5	1.0
	Float-Point Instructions (Single)	**1.8E+11**	**1.8E+11**	**2.1E+11**	**Overflow**	**1.8E+11**
	Float-Point Instructions (Double)	**6.6E+09**	**6.6E+09**	**6.6E+09**	**6.6E+09**	**6.6E+09 + 900K**
	Integer Instructions	1.4E+11	1.4E+11	1.7E+11	1.7E+11	1.4E+11
	Control-Flow Instructions	6.6E+09	6.6E+09	5.9E+10	5.9E+10	6.6E+09
	Load/Store Instructions	5.2E+10	2.6E+10	5.2E+10	5.2E+10	5.2E+10
SP (see Section IV-C5)	Kernel Time	91.9 ms	102.5 ms	-	80.0 ms	89.9 ms
	Registers	22	21	-	64	64
	Occupancy	98.4%	98.5%	-	99.2%	99.6%
	Texture Cache Hit Rate	23.1%	N/A	-	29.5%	24.1%
	L2 Hit (Texture Read)	3.8%	N/A	-	8.5%	7.6%
	Global Load Transactions per request	**31.8**	**31.8**	-	**31.3**	**31.3**
	Global Store Transactions per request	**7.0**	**7.0**	-	**6.9**	**6.9**
	Float-Point Instructions (Single)	1.8E+08	1.8E+08	-	2.8E+08	1.8E+08
	Integer Instructions	2.3E+08	1.5E+08	-	6.9E+08	4.0E+08
	Control-Flow Instructions	8.1E+06	8.1E+06	-	1.8E+07	2.6E+07
	Load/Store Instructions	2.5E+08	2.0E+08	-	2.5E+08	2.6E+08

TABLE VI

PERFORMANCE BREAKDOWN USING THE NVIDIA CUDA PROFILER. XL C ONLY HAS ONE COLUMN BECAUSE THE COMBINED AND NON-COMBINED VERSIONS SHOW THE SAME HW COUNTER NUMBERS. NUMBERS IN BOLD CHARACTER ARE PARTICULARY IMPORTANT FACTORS MENTIONED IN EACH SECTION.