Assessing One-to-One Parallelism Levels
Mapping for OpenMP Offloading to GPUs

Chen Shen
Dept. of Computer Science
University of Houston
Houston, TX
cshen6@uh.edu

Xiaonan Tian
Dept. of Computer Science
University of Houston
Houston, TX
xtian2@uh.edu

Dounia Khaldi
Institute for Advanced Computational Science
Stony Brook University
Stony Brook, NY
dounia.khaldi@stonybrook.edu

Barbara Chapman
Institute for Advanced Computational Science
Stony Brook University
Stony Brook, NY
barbara.chapman@stonybrook.edu

Abstract
The proliferation of accelerators in modern clusters makes efficient coprocessor programming a key requirement if application codes are to achieve high levels of performance with acceptable energy consumption on such platforms. This has led to considerable effort to provide suitable programming models for these accelerators, especially within the OpenMP community. While OpenMP 4.5 offers a rich set of directives, clauses and runtime calls to fully utilize accelerators, an efficient implementation of OpenMP 4.5 for GPUs remains a non-trivial task, given their multiple levels of thread parallelism.

In this paper, we describe a new implementation of the corresponding features of OpenMP 4.5 for GPUs based on a one-to-one mapping of its loop hierarchy parallelism to the GPU thread hierarchy. We assess the impact of this mapping, in particular the use of GPU warps to handle innermost loop execution, on the performance of GPU execution via a set of benchmarks that include a version of the NAS parallel benchmarks specifically developed for this research; we also used the Matrix-Matrix multiplication, Jacobi, Gauss and Laplacian kernels.

Keywords: OpenMP, OpenUH, LLVM, GPUs

1. Introduction
Most modern clusters include nodes with attached accelerators, NVIDIA GPU accelerators being the most commonly used. But recently, accelerators such as AMD GPUs, Intel Xeon Phi, DSPs, and FPGAs have been explored as well. The energy efficiency of accelerators and their resulting proliferation has made coprocessor programming essential if application codes are to efficiently exploit HPC platforms. As a result, considerable effort has been made to provide suitable programming models.

One may first distinguish between low-level and high-level programming libraries and languages; the user has to choose one programming API that suits her application code, level of programming, and timing. CUDA (cuda 2015) and OpenCL (OpenCL Standard) are low-level programming libraries. On the other hand, high-level directive-based APIs for programming accelerators, such as OpenMP (Chapman et al. 2008) and OpenACC (OpenACC), are easy to use by scientists, preserve high-level language features, and provide good performance portability features. Recently, multiple open-source compilers (e.g. GCC, LLVM, OpenUH) and commercial compilers such as Cray, IBM, Intel and PGI, have added support for these directive-based APIs (or are in the process of doing so).

OpenMP’s multithreading features are already employed in many codes to exploit multicore systems. The broadly available standard has been rapidly evolving to meet the requirements of heterogeneous nodes. In 2013, Version 4.0 of the OpenMP specification made it possible for user-selected computations to be offloaded onto accelerators; some important features were added in version 4.5 (Beyer et al. 2011; Board 2015). Implementations within vendor compilers are under way. OpenMP 4.0 added a number of device constructs: target can be used to specify a region that should be launched on a device and define a data device environment, and target data, to map variables on that device. Pragma teams can be used inside target to spawn a set of teams, each containing multiple OpenMP threads; note that teams can be mapped to a CUDA block. Finally, distribute is used to partition the iterations of an enclosed loop to each team of such a set.

Compared to OpenMP 4.0, OpenMP 4.5 changed the semantics of the mapping of scalar variables in C/C++ target regions. It also provides support for asynchronous offloading using nowait and depend clauses on the target construct. The mapping of data can also be performed asynchronously in OpenMP 4.5 using target enter data and target exit data. The clause is_device_ptr was added to target to indicate that a variable is a device pointer that is already in the device data environment, so it should be used directly. The clause use_device_ptr has been
added to target data to convert variables into device pointers to the corresponding variable in the device data environment. Finally, device memory routines were added to support explicit allocation of memory and memory transfers between hosts and offloading devices, such as *omp_target_alloc*.

A typical GPU architecture, for which some of the above-mentioned features were designed, provides three levels of parallelism, namely thread blocks, warps, and threads inside each warp. OpenMP 4.5 provides three levels of loop hierarchy parallelism as well, namely teams, parallel, and SIMD. For traditional CPU architectures and CPU-like accelerators, the mapping of these three levels of parallelism is straightforward. However, a typical GPU architecture differs substantially from a CPU, and thus the design of an efficient implementation of OpenMP 4.5 on GPUs is not a trivial task. In this work, we describe an efficient implementation of the OpenMP offloading constructs in the open source OpenUH compiler (Chapman et al. 2012) for NVIDIA GPUs. Our implementation of OpenMP 4.x offloading is based on a one-to-one mapping of OpenMP levels of parallelism to CUDA’s levels of parallelism, i.e., grid, thread block, and warp. To assess the suitability of this approach, we also implemented an OpenMP accelerator version of the NAS Parallel Benchmarks, and used it to test our implementation.

This paper makes the following contributions:

- we propose a one-to-one mapping of the loop hierarchy parallelism available in OpenMP 4.x to the GPU thread hierarchy and implement this mapping in the OpenUH compiler;
- we use a set of benchmarks to assess the impact of this mapping, especially the use of GPU warps to handle innermost loop execution, on the performance of GPU execution;
- we adapt the NAS parallel benchmarks to use OpenMP offloading, and make them available for use by the OpenMP community to test other implementations and platforms.

The organization of this paper is as follows. Section 2 provides an overview of the NVIDIA GPUs architecture and the OpenUH compiler. Section 3 describes the one-to-one loop hierarchy mapping to the GPU thread hierarchy that we have adopted. Performance results are discussed in Section 4. Section 5 highlights the related work in this area. Finally, we conclude our work in Section 6.

2. Background

In this section, we provide a brief overview of the architecture of GPUs, and of the OpenUH compiler.

2.1 GPU Architecture

Modern GPUs consist of multiple streaming multiprocessors (SMs or SMXs); each SM consists of many scalar processors (SPs, also referred to as cores). Each GPU supports the concurrent execution of hundreds to thousands of threads following the Single Instruction Multiple Threads (SIMT) paradigm, and each thread is executed by a scalar core. The smallest scheduling and execution unit is called a warp, which is typically composed of 32 threads. Warps of threads are grouped together into a thread block, and blocks are grouped into a grid. Both the thread blocks and the grid can be organized into a one-, two-, or three-dimensional topology. SIMT execution within a warp in GPUs can be compared to SIMD execution in a CPU. In each case, the same instruction is broadcasted to multiple arithmetic units. However, the CPU comes with vector instructions to process several elements of short arrays in parallel, whereas, in SIMT mode (within a warp), several elements of an array are processed in parallel.

Modern GPUs deploy a deep memory hierarchy, which includes several different memory spaces. Each of them has special properties. The global memory is the main memory in GPUs that all threads can make use of it. The so-called shared memory is shared by threads within a thread block only. The texture memory is randomly memory that allows adjacent reads in a warp.

Each kind of memory requires specific attention. For example, accesses to global memory may be coalesced or not; accesses to texture memory could come with a spatial locality penalty; accesses to shared memory might suffer from bank conflicts; and accesses to constant memory may be broadcast. Memory coalescing is a key enabler of data locality in modern GPU architectures. Under it, memory requests by individual threads in a warp are grouped together into large transactions. When consecutive threads access consecutive memory addresses, this enables the exploitation of spatial data locality within a warp.

2.2 OpenUH Compiler

The OpenUH compiler (Chapman et al. 2012) is a branch of the open-source Open64 compiler suite for C, C++, and Fortran 95/2003. It offers a complete implementation of OpenACC 1.0 (Tian et al. 2014), Fortran Coarrays (Eachempati et al. 2012), and OpenMP3.0 (Liao et al. 2007). OpenUH is an industrial-strength optimizing compiler that integrates all the components needed of a modern compiler; it serves as a parallel programming infrastructure in the compiler research community. OpenUH has been the basis for a broad range of research endeavors, such as language research (Ghosh et al. 2013; Huang et al. 2010; Eachempati et al. 2012), static analysis of parallel programs (Chapman et al. 2003), performance analysis (Pophale et al. 2014), task scheduling (Ahmad Qawasmeh 2014) and dynamic optimization (Besar Wicaksono 2011).

The major functional parts of the compiler are the front ends (the Fortran 95 front end was originally developed by Cray Research and the C/C++ front end comes from GNU GCC 4.2.0), the inter-procedural analyzer/optimizer (IPA/OPO), and the middle-end/backend, which is further subdivided into the loop nest optimizer (LNO), including an auto-parallelizer (with an OpenMP optimization module), global optimizer (WOPT), and code generator (CG). Currently, x86-64, IA-64, IA-32, MIPS, and PTX are supported by its backend. OpenUH may also be used as a source-to-source compiler for other machines using its IR (Intermediate Representation)-to-source tools. OpenUH uses a tree-based IR called WHIRL, which comprises 5 levels, from Very High (VH) to Very Low (VL), to enable a broad range of optimizations. This design allows the compiler to perform various optimizations on different levels.

3. OpenUH-Based OpenMP Offloading Support

In the OpenMP offloading model, it is assumed that the main program runs on the host while the compute-intensive regions of the program are offloaded to the attached accelerator. OpenMP constructs for device and host execution are similar: the *target* directive is used to distinguish between the two. A target region consists of two parts: (1) a device data environment, which maps host variables to the device memory and (2) a computing region that will run on the device. The device and the host may have separate memories; data movements between them may be controlled explicitly. Currently, OpenMP provides a rich set of data transfer directives, clauses and runtime routines as part of the standard to create device data environments and offload regions.

OpenMP also provides multiple levels of parallelism. First, the *teams* directive can be used to spawn a league of teams; each team contains a number of OpenMP threads. Next, *distribute* may be attached to a loop to partition its iterations among the master
threads of these teams. Then, the `parallel for` directive can be used to workshare the enclosed loop within each team. Finally, the `simd` directive gives a hint to the compiler to vectorize the loop iterations using SIMD instructions.

With the traditional CPU architecture and in CPU-like accelerators, the mapping of these three levels of parallelism is straightforward. In the latest Intel 72-Core Knights-Landing (KNL) accelerator, for example, each core has two 512-bit vector units and supports four threads. In this case, we might e.g. create 72 teams containing 4 threads each. Each thread can then perform 512-bit SIMD operations. The typical GPU architecture does not permit this straightforward approach, and thus designing an efficient implementation of OpenMP 4.x on the GPU is not a trivial task. In this work, we introduce, and subsequently assess, an approach based on a one-to-one mapping of the OpenMP levels of parallelism to the levels of parallelism found in CUDA, i.e. grid, thread block, and warp (cud 2015).

3.1 Frontend and Backend Support

We have implemented the OpenMP accelerator constructs in the OpenUH compiler. Note that OpenUH also supports OpenACC, another directive-based API for device programming. The existing OpenACC implementation generates CUDA/OpenCL kernel functions for NVIDIA and AMD GPUs, respectively. In this work, we target NVIDIA GPUs because of their wide popularity in the HPC community. We plan to address AMD GPUs in future work.

The creation of an OpenMP compiler for accelerators requires a significant implementation effort to meet the challenges of mapping high-level loop constructs to low-level threading architectures (as shown in Figure 1). This implementation is divided into the following phases.

**Frontend** We have extended the existing frontend to support the following directives and clauses of OpenMP 4.5: `target`, `target data`, and the loop parallelism directives, namely `distribute`, `teams`, and `simd`.

We parsed these constructs and generated new corresponding intermediate representations nodes.

**Backend** In this phase, we transform the generated IR nodes into runtime library calls, which further invoke lower-level CUDA library routines to allocate/deallocate memory within GPUs; this corresponds to the **LOWERING** module in Figure 1. Then, we outline each OpenMP `target region` as a CUDA kernel function that will be fed to another module of Figure 1, called **WHIRL2CUDA**, in order to generate CUDA source code. Finally, we replace the original outlined target region with a runtime function call that is used to launch the kernel. Note that, during the outlining step, we perform a one-to-one mapping of parallel loops to the threading architectures; we detail this mapping in the following subsection.

**Binary Generation** Since our translation is based on source-to-source translation to handle target regions, we use the CUDA SDK environment to translate the CUDA kernel functions into NVIDIA GPU assembly code, also called PTX code.

Our main goal is to follow the OpenMP target model specification while attaining high performance on GPUs as well as CPU-like accelerators despite the significant differences in their architectures. The individual cores in CPU-like accelerators contain SIMD units. Their ability to issue SIMD instructions makes it very easy for the cores to exploit the SIMD directives in OpenMP, which are important for performance on such platforms. In contrast, GPUs are highly suitable for massively parallel processing. Their thread hierarchy is flatter than on CPUs. The warps are basic scheduling units on GPUs and this makes it challenging to map different levels of parallelism.

3.2 One-to-One Loop Hierarchy Mapping of OpenMP 4.5 Parallelism

Performance portability, where a single source code runs well across different target platforms, is increasingly demanded. Progress toward this goal in the context of OpenMP requires that accelerator directives be implemented efficiently on GPUs as well CPU-like accelerators. In order to achieve this, it is necessary to to effectively exploit each level of architectural parallelism, between and within groups of threads, and to map all three levels of OpenMP 4.x parallelism onto the GPU. Mapping the `simd` construct of OpenMP is critical to fully exploit the corresponding computing capabilities. Table 1 shows the mapping of OpenMP parallelism concepts to CUDA parallelism levels that was used in this work. The teams construct is translated directly into thread blocks in CUDA because the thread blocks are independent of one another and consist of a group of threads that run on the same execution unit (SMX).

<table>
<thead>
<tr>
<th>OpenMP Abstraction</th>
<th>CUDA Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>teams</td>
<td>thread blocks within the grid</td>
</tr>
<tr>
<td>parallel</td>
<td>warps within a thread block</td>
</tr>
<tr>
<td>simd</td>
<td>32 threads within a warp</td>
</tr>
</tbody>
</table>

In the OpenMP `target region`, the environment of the compute kernel is initiated, which includes device data creation and CUDA kernel generation. At runtime, target data will be transferred between host and device according to the information from map clause, then the CUDA kernel will be launched.

When the program encounters the `omp teams distribute` directive, a league of thread teams will be created and the execution will be distributed across the associated loop. This mapping strategy will basically create a grid that consists of thread blocks; then the computation load will be spread across those created thread blocks. Strictly following the specification of OpenMP, the workload is shared among the master threads of thread blocks. As we are generating CUDA kernel calls, there is no way to specify synchronization among different teams; the thread synchronization calls are only effective within a thread block.

The implementation of the `omp parallel for` directive inside the `omp target region` is based on a single thread block. A thread block will get a chunk of data and distribute the associated loop across threads. In OpenUH, the distribution is actually accomplished among warps within a thread block, and the master thread in...
each warp finishes the work load and waits at a synchronize point. This kind of mapping strategy makes it possible to utilize shared memory resources in each thread team on GPU-like accelerators.

Each warp consists of 32 threads; the first thread of each warp corresponds to the thread with the smallest thread-id. A parallel region is executed by each first thread in all the warps within the same thread block. If for is combined with parallel, these first threads will workshare the parallel loop region.

The GPU architecture supports the SIMT approach by executing the same instruction on several elements concurrently, within a warp. Therefore, a single instruction that is executed by a warp can be considered to be a SIMD operation as defined by the OpenMP standard. Thus, in our implementation of OpenMP offloading for GPUs, we interpret the simd construct as the execution unit of a warp. Specifically, the iterations of a loop that is annotated by the simd directive are distributed or vectorized among threads within the same warp.

```c
#pragma omp target teams distribute
for (i=1;i<=isize-1;i++)
#pragma omp parallel for
for (j=1;j<=jsize;j++)
#pragma omp simd
for (k=1;k<=ksize;k++)
temp1 = dt * tx1;
temp2 = dt * tx2;
lhsX[i][j][k] = ...
```

Figure 4 shows an example code that contains just a single level of OpenMP offloading parallelism in a single loop. Since we do not have a teams construct, only one thread block is created and the workload is shared among the threads inside this thread block. As described above, in a naive implementation only the master thread in each warp is working and the other threads simply wait at the synchronization point. There is a large overhead for synchronizing threads, which further degrades performance. In this kind of scenario, OpenUH’s implementation will automatically spread the workload to warps which has an effect similar to that of using a combined parallel for simd construct. Note that this approach, and the previous automatic insertion of simd constructs, does not affect the semantics of the program.

4. Evaluation

In this section, we present the results of experiments that used our implementation of the OpenMP 4.5 offloading model to translate the NAS parallel benchmarks (NPBs) for a GPU. We also compare our implementation with that of LLVM.

4.1 Experimental Setup

The NVIDIA GPU used for this evaluation includes a host with 2 Intel Xeon E5-2640 CPUs (16 threads per CPU) and 32GB main memory; the attached GPU is a K20Xm with 5GB global memory. CUDA 6.5 is used for the OpenUH backend GPU code compilation with “-O3” optimization. For the comparative analysis, we used the Clang 3.8 implementation of OpenMP that is available at https://github.com/clang-omp/libomptarget. We use “-openmp -omp-targets=nvptx64-sm_35-nvidia-linux -O3” for the LLVM compiler options. To obtain reliable results, all experiments were performed five times and then the average performance was computed. We also compare the GPU execution with 2 Intel Xeon E5-2640 CPUs.

4.2 Performance Evaluation using NAS Benchmarks

In order to assess the performance of the OpenUH implementation of the OpenMP offloading model, we modified the NAS parallel benchmarks (NPB) to make use of OpenMP offloading directives, including data movement and parallel region constructs with the corresponding clauses. We then measured the execution time for data movement and for launching and running the kernels. The data
sizes used for NAS benchmarks are CLASS A (not discussed here), CLASS B, and CLASS C, the data sizes for each class are different.

Figures 5 and 6 show the performance of our offloading implementation on the NPB variant that we specifically developed for this work. Note that we did not perform comparisons with LLVM, because the backend of the currently available implementation does not support some constructs used in NPB-OpenMP4.5.

4.3 Comparison with Clang OpenMP Implementation

We compared the performance of OpenUH with the LLVM implementation that uses libomptarget as an offloading runtime. We used four kernels: 2d-jacobi, matrix-matrix multiplication, gauss and laplacian. In all kernels, our implementation outperforms the LLVM implementation (see Figure 7 and 8).

For the matrix-matrix multiplication kernel, the performance gap between OpenUH and LLVM is huge (notice the logarithmic scale on the y axis) for very large data sizes. This is because Clang maps the loop hierarchy to the GPU threads in an inefficient way. In fact, LLVM does not fully utilize GPU warps as we believe it should be handled while executing loops with a parallel for clause. Indeed, LLVM mainly addresses two levels of parallelism on the GPU: distribute and parallel. In the current implementations (Bertolli et al. 2014, 2015), the teams construct is mapped to the x dimension of the thread blocks in the grid. The parallel directive is mapped to the x dimension of threads within each thread block. This indicates that LLVM does not handle the innermost loop efficiently; yet when only master threads in warps are executing, a large overhead for thread synchronization is incurred as described in the previous section. For the laplacian kernel, the performance gap between OpenUH and LLVM is smaller than for the other three kernels. The laplacian has three levels of parallelism: distribute, parallel and simd, in this case, LLVM could make use of GPU warps for the innermost loop, which could then be spread across threads within each warp, resulting in much better performance.

In OpenUH, the fact that we implicitly use GPU warps to handle innermost loop execution leads to a much better performance than LLVM’s approach in general. Therefore, we believe the one-to-one mapping introduced in this paper should be used to fully exploit the thread hierarchy and to get better thread occupancy within GPUs.
5. Related Work

Support for OpenMP 4.5 in vendor compilers is still a work in progress. The Intel compiler 16.0, released in August 2015, fully supports OpenMP 4.0 for the Intel Xeon Phi coprocessor. The Cray Compiling Environment 8.4 was released in September 2015. It provides support for the OpenMP 4.0 specification for C, C++, and Fortran, enabling the execution of OpenMP 4.0 target regions on NVIDIA GPUs.

Regarding open-source compilers, GCC 6 was released in April 2016; it fully supports the OpenMP 4.5 specification for C and C++. GCC targets Intel Xeon Phi Knights Landing and AMD’s HSAIL (Heterogeneous System Architecture Intermediate Language).

LLVM 3.8.0 was released in March 2016. Its front-end Clang supports all of OpenMP 3.1 and some elements of OpenMP 4.0 and 4.5. LLVM uses an OpenMP offloading library called libomp-target.

The current implementation of this library can be divided into three components: target-agnostic offloading, target-specific offloading plugins, and target-specific runtime library. The libomp-target library was designed with the goal of supporting multiple devices; it currently targets the IBM Power architecture, x86_64, Nvidia GPUs, and Intel Xeon Phi. In (Bertolli et al. 2015), its OpenMP 4.0 offloading strategy is described and some optimizations are applied, such as a control loop scheme (Bertolli et al. 2014) that avoids dynamic spawning of GPU threads inside the target region; this implementation targets NVIDIA GPUs. As already discussed in the experimental section, the current implementation of LLVM does not map the #pragma OpenMP level to GPU warps, thereby hurting performance.

6. Conclusion

In this paper, we describe our design and implementation of a compilation scheme based on a one-to-one mapping of the loop hierarchy available in OpenMP 4.x to the thread hierarchy found in GPUs. The implementation was carried out in the OpenUH compiler, which is well suited for prototyping novel techniques to support parallel languages and new compiler optimizations. A set of benchmarks were employed to assess how this mapping, especially the use of GPU warps to handle innermost loop execution, impacts the performance of GPU execution. The benchmarks include new versions of the NAS parallel benchmarks that we specifically developed for this research; we also used matrix-matrix multiplication and Jacobi kernels. We show that our one-to-one mapping technique significantly outperforms the Clang implementation of OpenMP on these two last kernels. In future work, we plan to fully support the OpenMP 4.5 specification as well as target other accelerators such as AMD GPUs and Intel Xeon Phi.

References


