Performance Analysis of OpenMP on a GPU using a CORAL Proxy Application

Gheorghe-Teodor Bercea, Carlo Bertolli, Samuel F. Antao, Arpith C. Jacob, Alexandre E. Eichenberger, Tong Chen, Zehra Sura, Hyojin Sung, Georgios Rokos, David Appelhans, Kevin O'Brien
IBM T.J. Watson Research Center
1101 Kitchawan Rd. Yorktown Heights NY, U.S.A.
{gbercea,cbertol,sfantao,acjacob,alexe,chentong}@us.ibm.com
{zsura,hsung,grokos,dappelh,caohmin}@us.ibm.com
Imperial College London, Department of Computing,
180 Queen's Gate, London, SW7 2AZ, United Kingdom
gheorghe-teodor.bercea08@imperial.ac.uk

ABSTRACT

OpenMP provides high-level parallel abstractions for programing heterogeneous systems based on acceleration technology. Active areas of research are looking to characterise the performance that can be expected from even the simplest combinations of directives and how they compare to versions manually implemented and tuned to a specific hardware accelerator. In this paper we analyze the performance of our implementation of the OpenMP 4.0 constructs on an NVIDIA GPU. For performance analysis we use LULESH, a complex proxy application provided by the Department of Energy as part of the CORAL benchmark suite.

NVIDIA provides CUDA as a native programming model for GPUs. We compare the performance of an OpenMP 4.0 version of LULESH obtained from a pre-existing OpenMP implementation with a functionally equivalent CUDA implementation. Alongside our performance analysis we also present the tuning steps required to obtain good performance when porting existing applications to a new accelerator architecture. Based on the analysis of the performance characteristics of our application we present an extension to the compiler code-synthesis process for combined OpenMP 4.0 offloading directives. The results obtained using our OpenMP compilation toolchain show performance within as low as 10% of native CUDA C/C++ for application kernels with low register counts.

1. INTRODUCTION

The next generation High Performance computing architecture by IBM[1]is a heterogeneous system based on the OpenPower technology that couples Power processors with NVIDIA Graphics Processing Units (GPUs) and a network based on Mellanox technology. Two such systems are scheduled to be delivered to Oak Ridge and Lawrence Livermore National Laboratories, part of the U.S. Department of Energy (DoE) [4].

A critical feature of these systems is the ability to optimize application execution by mapping every code region to the most appropriate execution resource through an offloading-based model. In the last few years this has been coded through native programming languages, like CUDA [5].

CUDA C/C++ has the advantage of presenting the user with an abstraction that can be used to tune an application to the GPU hardware. This is useful in real-world use cases where performance considerations take over development or porting time constraints.

In a large number of situations, drawn from different scientific areas, accelerator-specific implementations are not a viable solution. Users, such as scientists at DoE, are often required to develop and maintain applications across different hardware technologies. Adopting one language per technology would result in multiple different versions of the same application, a situation that can quickly lead to prohibitive development and maintenance costs. The related issues are prominent in the scientific community at the time this paper is being written and they are characterized under the *performance portability* umbrella.

Abstract programming languages like OpenCL [16], OpenACC [15] and OpenMP [17] have been developed to tackle performance portability. OpenMP has a platform agnostic approach to parallelism and it gives user control over thread mapping on diverse systems. OpenMP version 4.0 [13] introduces new offloading constructs that expose abstract acceleration mechanisms, both in terms of data layout and placement and expression of concurrency. The application programmer retains a high-level view of parallelism avoiding any accelerator-specific thinking. OpenMP, as a standard,

PMBS2015 November 15-20, 2015, Austin, Texas USA
Copyright 2015 ACM 978-1-4503-4009-0/15/11 ...$15.00.
DOI:http://dx.doi.org/10.1145/2832087.2832089.

[1]The following symbols used throughout the paper are registered trademarks: IBM®, OpenPower®, Power®, NVIDA®, CUDA®, Mellanox®, OpenMP®, OpenCL®, OpenACC®, Kepler®, Linux®, Ubuntu®, AMD®, Intel®, Xeon Phi®

is implemented on a wide range of hardware platforms and is an instance of the "write once, run everywhere" goal.

While the high-level approach of OpenMP 4.0 is appealing from an abstraction and application portability viewpoint, it is still an open research issue whether efficient code synthesis and support can be achieved on specific accelerators. When not employing CUDA C/C++ or any native libraries like Thrust [19], NVIDIA GPUs are a particularly challenging platform to achieve high performance on.

In this paper we explore the challenges involved in achieving high performance when using OpenMP (as supported by our LLVM-based toolchain) as a viable alternative to using CUDA in conjunction with the native NVCC compiler stack. This builds on the previous paper [2] where we introduced our design and implementation of the OpenMP 4.0 offloading constructs on NVIDIA GPUs based on the LLVM compiler infrastructure [9].

Although we only run our experiments on NVIDIA GPUs, the approach we take could be extended to other hardware platforms such as AMD GPUs.

The performance evaluation of our toolchain is focused on LULESH [11], a DoE proxy application that was ported to several languages and platforms. LULESH was developed to expose performance challenges representative of a class of static unstructured mesh applications of interest to DoE scientists. LULESH-type simulations account for almost 30% of the entire DoE application runs [12]. The large number of offloading regions provides a varied context for identifying the generic performance characteristics and kernel-specific issues of our implementation.

As reference of best performance we consider the GPU-native CUDA C/C++ implementation of LULESH [11] based on the CUDA Thrust STL library and compiled using NVCC. We then consider the performance of several versions of our own OpenMP 4.0 implementation of LULESH coupled with increasingly-optimized versions of our LLVM toolchain.

The initial version has been obtained by replacing the parallel for regions of the original OpenMP implementation of LULESH [7] with OpenMP 4.0 device offloading constructs. Since our compiler abstracts away any device-specific details, this version involves minimal performance parameter tuning. Successive versions incorporate changes to the OpenMP 4.0 LULESH implementation including loop interchange, loop fusion, inlining as well as compiler improvements that tune shared memory usage and register allocation.

Additionally, we present an alternative code synthesis to the one described in [2]. Unlike the default compilation synthesis that supports all possible pragmas in an offloading region, the alternative applies to a combined construct scheme that is widely used. Due to this specialization, we were able to improve performance metrics by an order of magnitude over the baseline OpenMP 4.0 performance. For certain kernels this delivers performance that is similar to the CUDA-based implementation.

The contributions of this paper are the following:

- We report on the end-to-end porting of the LULESH CORAL proxy application onto NVIDIA GPUs using OpenMP 4.0 directives.
- We describe the main implementation mechanisms that affect performance of OpenMP for LULESH on GPUs, and we relate them to performance metrics obtained from application profiling.
- We introduce optimizations to the basic mechanisms and we report improved performance. We track performance improvements back to relevant metrics and features of generated code.

The paper is organized as follows: in Section 3 we give background information on OpenMP 4.0 constructs, briefly describe main features of our implementation, and show two optimization techniques. Section 4 describes relevant aspects of LULESH. Section 5 includes a full performance analysis of LULESH for various problem sizes and optimization levels. Section 6 summarizes the findings reported in this paper.

2. RELATED WORK

In our previous paper [2] we reported on efforts to port parallel directive languages onto GPUs. Particularly relevant to this paper is [20]. The authors describe their solution to multiple implementation challenges, including data sharing within a block, thread selection for nested pragmas and thread coordination between sequential and parallel regions.

Several other attempts were made towards a high performance implementation of OpenMP 4.0 on GPUs. An exploratory work is presented in [18], where the OpenSs compiler is extended to include OpenMP offloading directives. The paper shows a prototype implementation and suggests the adoption of new clauses for data sharing into OpenMP. This constitutes a basis for a further proposal to support OpenMP offloading directives in the gcc compiler as discussed in [6]. The gcc proposal relies on the use of dynamic parallelism to straightforwardly re-organize scheduling of tasks to threads. This will permit improved data access coalescing, a critical performance feature of GPU applications.

Full implementation details of the OpenMP 4.0 accelerator offloading directives in our CLANG compiler can be found in [1]. Other relevant work is represented by the implementation of the OpenMP 4.0 standard for x86 platforms including the Intel Xeon Phi acceleration architecture.

Previous work on benchmarking LULESH [8] shows comparisons between NVIDIA GPU run times and OpenMP run times on CPUs. We choose the problem sizes in the performance analysis section such that they match the range used in [8]. Although previous work does not report individual kernel run times (only overall application time), to comprehensively present the impact on performance of our CLANG/LLVM compiler extensions (detailed in Section 3) the performance is evaluated on a per-kernel basis.

3. IMPLEMENTATION OF OPENMP 4 ON NVIDIA GPUS

In this section we explain in more detail the aspects of the implementation of OpenMP target regions that are relevant to the performance analysis of LULESH. We give a brief description of the OpenMP constructs considered and present the baseline compiler code synthesis. As highlighted later in performance analysis (see Section 5), we note that, on average, OpenMP offloading regions are characterized by low GPU occupancy compared to the baseline version. We thus present two general optimizations that aim to: (i) customize the amount of shared memory required to compile an OpenMP offloading region; (ii) minimize the amount of registers needed per thread. Reducing these two metrics results in higher occupancy and therefore improved performance.

3.1 OpenMP Offloading Constructs

OpenMP 4 offloading is based on a host-centric viewpoint. An application starts executing on an initial (host) device, possibly including OpenMP regions. Offloading to a secondary device is performed when a *target* region is encountered. A target region defines: a device data environment which maps variables allocated on the host memory to the device memory; an executable region of code that is to be run on the device. The implementation of target regions includes possibly allocating and transferring data between host and device and launching a CUDA C/C++ kernel that corresponds to the executable region of the pragma.

Target regions can be programmed with any OpenMP construct with some minor restrictions. There are OpenMP constructs that are only available when placed inside a target region. Pragma *teams* can be used to spawn a league of teams, each containing multiple OpenMP threads. The main feature is that two threads in different teams are not allowed to communicate in any native way. For instance, it is not possible to express a barrier between threads in different teams. The implementation of GPUs naturally maps each team to an independent CUDA block, as these feature the same execution independence requirements as teams i.e. CUDA C/C++ does not expose a native synchronization operation between two blocks. It is important to note that when a *teams* region is encountered only one thread (team master) actually starts executing. The remainder of the threads in each team are only activated when a pragma *parallel* is encountered.

A pragma *distribute* is associated with a loop and is used to partition it into chunks which are scheduled to teams. This is done by the team master of each team.

Function calls and global variable references are allowed in target regions. The user needs to mark every declaration and definition of a function or a global variable that is called/used within a target region using a special *declare target* pragma.

3.2 Control Loop

The programming model of NVIDIA GPUs involves restrictions that impact the OpenMP implementation. Relevant ones for OpenMP implementations are: threads within the same warp execute in lock-step and expose limited means of expressing data- or control-flow dependencies; block synchronizations need to be encountered by all threads in the block. In [2] we introduced a Control Loop scheme that enables implementation of OpenMP target regions in a correct and efficient way based on the two restrictions above.

The coordination scheme supports an implementation choice where all threads that will be needed for a target region are started when launching the related kernel. In our previous paper, we reported that the alternative of using the CUDA C/C++ dynamic parallelism support induces large overheads compared to our coordination scheme. Our solution is based on a state-machine approach that simplifies integration of the code generation scheme in the compiler, while at the same time satisfies the restrictions listed above.

A synthetic and simplified version of the scheme for a simple example is shown in Figure 1. The team master (thread 0 in each CUDA block) executes the sequential regions and leads the other threads across the different regions. Each sequential and parallel region is mapped to one or more switch cases and we prevent unnecessary threads from executing anything depending on the region kind and OpenMP clauses. For instance, sequential regions are only executed by the team master, while all other threads are waiting at the barrier. Parallel regions are executed by all threads, in this simplified version. Note that there is a single barrier (line 24) in the whole kernel and that idle threads wait on the same barrier that will be used by the active threads before making any progress.

```
1  #pragma omp target teams
2  {
3    int pi = 3.14;
4    #pragma omp distribute parallel for
5    for (int i = 0 ; i < N ; i++)
6      a[i] = b[i] + c[i] * pi;
7  }
```

```
1  bool finished = false;
2  __shared__ int nextLabel = Seq1;
3  while (!finished) {
4   switch(nextLabel) {
5   case Seq1:
6     if (threadIdx.x != 0) break;
7     // code gen for sequential region 1
8     // ...
9     nextLabel = Par1;
10    break;
11  case Par1:
12    // code gen for parallel region
13    // ...
14    if (threadIdx.x == 0) nextLabel = Seq2;
15    break;
16  case Seq2:
17    if (threadIdx.x != 0) break;
18    // code gen for sequential region 2
19    // ...
20    nextLabel = -1;
21    finished = true;
22    break;
23  }
24  __syncthreads();
25 }
```

Figure 1: Device-offloaded target region (top) and corresponding thread coordination code generated by the Control Loop scheme for GPUs (bottom).

While this scheme is easy to implement in a modular way in the compiler, it complicates the produced code with a multi-target switch and a while loop over a condition variable. This negatively impacts register allocation, as explored in the performance analysis section 5.

3.3 Local Data Sharing

According to OpenMP, a given thread may need to share local data with other threads executing parallel regions in the same data declaration scope. Figure 2 presents an example where local data is being shared. In this paper we show a design for the data sharing mechanism that is general enough to support two levels of parallelism and nested SIMD regions. In this model we partition CUDA threads available in each team/block into multiple levels of parallelism. For instance, in the example of Figure 2 a thread may only be activated inside the SIMD region. We refer to CUDA threads that execute in a SIMD region as *lanes.*

In this example, the iteration variable `i` of loop `L1` is a local variable of the master thread of the OpenMP team,

```
#define N 256
int a[2*N][N][N];
...
#pragma omp target teams num_teams(8) thread_limit
    (64)
{
  #pragma omp distribute
L1:for (int i = 0 ; i < N ; i++) {
    int tmp = 0;
    #pragma omp parallel for firstprivate(i)
L2: for (int j = 0 ; j < N ; j++)
    tmp += j;
    #pragma omp simd
L3: for (int k = 0 ; k < N ; k++){
        a[i][j][k] *= omp_get_thread_num();
    }
  }
}
```

Figure 2: Example of OpenMP code that requires thread local data sharing.

given that only the team master thread gets to execute the code closely nested inside the *distribute* directive. Therefore, `i` has to be shared among the team master thread and the OpenMP parallel threads (lane master threads) in order to initialize their private instances of `i`. The code closely nested within the *parallel for* directive is executed only by the lane master thread, thus they will have a local instance of the iteration variable `j` of loop `L2` that has to be shared among all threads in the same SIMD group, as `j` is used in loop `L3`.

The Clang implementation of OpenMP for CPUs outlines functions for the parallel and SIMD regions forwarded to the OpenMP runtime library along with a set of arguments that contain a reference to the data being shared. The runtime library will then spawn threads to execute that region (outlined function) and each of the spawned threads can access that data by using the shared data reference. However, this approach cannot be used in the OpenMP implementation for NVIDIA GPUs for two reasons:

- When the target region starts to be executed on an NVIDIA GPU, all threads that can ever participate in the computation are already launched. Therefore, there is no point in creating outlined functions meant to be executed by newly spawned threads. Instead, the entire code of the different regions is kept inlined in the same function;

- Data can be communicated between threads only by using a shared address space. This address space can either be CUDA shared memory or CUDA global memory. Also, a reference to data that is local (automatic variables, register spill slots) to a thread (CUDA local memory) can only be used by that thread. In other words, the same address in CUDA local memory points to different physical storage if dereferenced in different threads. This is due to automatic address translation performed by the device hardware and meant to improve data access coalescing and therefore use more bandwidth while accessing local data; different threads in the same CUDA warp may have a local instance of the same declaration that is accessed by all threads in the same cycle. As a result, there is no point in sharing references if the data being referred to is in CUDA local memory.

A consequence of the two observations above is that all local variables that need to be shared among threads have to be promoted to CUDA shared or CUDA global memory. This forces the compiler to manage the different instances of the same declaration by reserving appropriate storage and then generating the code that correctly accesses it. Currently, all storage reserved for data sharing is allocated statically with the following (per team) memory layout: *i)* one slot for team-master/SIMD-master data sharing and *ii)* N_{SG} slots for SIMD-master/SIMD-lane data sharing, N_{SG} being the number of SIMD groups. For the example in Figure 2, the slot *i)* will store `i` and each of the slots *ii)* will store a private instance of `i` and the instance of `j`. The compiler is able to compute the size of each slot by looking at the type of the declarations being shared (variable-length array types are not currently supported), but it still needs to know the number of threads and teams. If CUDA shared memory is used for data sharing, this is not required because this is allocated per team/CUDA block. These numbers can be retrieved from the *thread_limit* and *num_teams* clauses, respectively, if constant expressions are being used. Otherwise, the compiler uses a default number that can be customized by the user with proper compiler options.

By default, the compiler uses CUDA shared memory to share data unless the amount of data being shared exceeds its capacity. In that case CUDA global memory is used. An option to force the use of CUDA global memory is also provided that can be useful for cases where the CUDA shared memory limits the occupancy of the GPU resources.

3.4 Optimized Scheme for Combined Constructs

The complexity of code patterns that can be found in a target region requires a complex thread coordination scheme and data sharing support. In particular, we note that the following features are critical to the complexity of the control loop implementation:

- A team region contains sequential (team master only) and parallel regions. A sequential region can contain state modifications and, more generally, side effects that cannot be executed by all threads in a team in a redundant way.

- Data modified in a team master only region and used in a parallel region requires sharing, as highlighted above.

- Function calls can be found in a target region and these can contain further OpenMP pragmas. The compiler does not necessarily have knowledge of what is contained in a function called from within a target region, as it could be defined in a separate compilation unit.

These features are not shared by all OpenMP pragma configurations and all programs. Our analysis of LULESH, as reported in Section 4, shows that one OpenMP pragma *parallel for* pattern is ubiquitous in the program. Transforming these loops into the OpenMP 4.0 offloading model is essentially based on combining *parallel for* with pragma *target teams distribute*. From a control flow perspective the parallel computation is performed in the loop body, and support pragma actions can be executed in a redundant way. Consider the simple program at the top of Figure 3 where a combined target construct is used. The annotated loop is chunked and and each chunk distributed to a team. The chunks are then executed in parallel by all threads in each team. The main characteristics of this program are: (i)

```
#pragma omp target teams distribute
     parallel for schedule(static,1)
for (int i = 0 i < n ; i++)
  a[i] = b[i] + c[i];
```

```
for(int idx = threadIdx.x + blockId.x*
     blockDim.x;
       idx < n ; idx += blockDim.x *
            gridDim.x)
  a[idx] = b[idx] + c[idx];
```

Figure 3: OpenMP program that satisfies constraints for applying the combined directive optimizations and CUDA-like code synthesis.

there is no team-master only region, except the scheduling of loop chunks to team masters (*distribute* implementation); (ii) there is no data sharing, except possibly the lower and upper bounds of the iteration variable of the chunk assigned to a team; (iii) there are no function calls and therefore no possible nested OpenMP pragmas. The latter point does not limit the extensibility of the combined construct to support the SIMD and other pragmas.

For (i) we note that the distribution can be calculated by all threads in each team in a redundant way. This impacts (ii) because a thread can directly compute the lower and upper bounds of the iteration block assigned to the team it belongs to. No explicit sharing of the bounds is required. The static schedule with chunk size equal to one requires that each thread execute the team-assigned block one iteration at a time.

This program can be implemented using a CUDA-like notation as shown at the bottom of Figure 3. The iteration space is divided into partitions of size equal to the number of CUDA blocks multiplied by the number of threads in each block. The computation of a partition is fully parallel with one iteration assigned to every thread. This achieves full coalescing for global memory accesses.

To correctly apply this code synthesis, the compiler is required to perform conservative code analysis. For instance, it checks that no OpenMP pragmas are present within the combined construct region, and that no functions are called. In the implementation, there are no calls to an OpenMP runtime, as it is instead required in the more general case, and no need to coordinate threads across OpenMP regions.

4. PORTING OF LULESH TO OPENMP 4

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH [12]) is an implementation of one of the five challenge problems in the DARPA Ubiquitous High Performance Computing (UHPC) program. LULESH has been widely studied and ported to different programming models such as OpenMP, CUDA C/C++, OpenACC, Charm++, Liszt[8]. LULESH is also part of the set of benchmarks of the CORAL collaborative exascale effort.

4.1 Features

4.1.1 Parallelism

LULESH uses a hexahedral mesh which exposes parallelism at both vertex and element levels. Elements store thermodynamic variables (such as pressure) and vertices store the kinematic values (such as velocity or position). Each element can usually be processed independently of the others. The number of elements in the mesh is determined at runtime and it has a default value of 30^3.

LULESH is implemented using a timestepping Lagrange leapfrog algorithm consisting of two main phases for advancing the vertex and element quantities followed by the calculation of the time constraints. In this paper we focus on the first two phases because they expose a high degree of parallelism, they account for over 95% of the runtime and their instruction mix (integer, floating point and control flow instructions) represents a challenging use case for the exploration of compiler level optimizations.

4.1.2 OpenMP and CUDA C/C++ implementations

We convert the parallel loops (marked with *pragma omp parallel for*) in the LULESH 2.0 OpenMP implementation to use OpenMP 4.0 target regions marked with the *target teams distribute parallel for* directive combination. To compare the performance achieved by OpenMP 4.0 on the device with that achieved by the CUDA C/C++ implementation we need to first take into consideration the number of kernels and parallel regions and identify the correspondence between them. The OpenMP implementation consists of about 45 statically scheduled OpenMP parallel loops over elements and vertices while the CUDA C/C++ implementation uses ten such regions (kernels). For example, on the OpenMP-side, the computation of nodal forces is split into several parallel regions while in CUDA C/C++ there are only two.

The number of calls per timestep to each kernel or target region is not guaranteed to be the same across implementations. For example, OpenMP has separate calls to process each multi-material region while in CUDA only one kernel call is used.

The only one-to-one correspondence which can be established between the two codes is the acceleration computation for the interior and along the boundary of the domain.

To make the OpenMP 4.0 target regions match the CUDA C/C++ workloads, we fuse some of the OpenMP *parallel for* loops. Since this requires manual function inlining and loop interchanges, we have only done this for a subset of *target* regions. The number of OpenMP 4.0 target regions is initially reduced to 18. In Table 1 we show the different CUDA kernels and their corresponding OpenMP 4.0 GPU offloaded regions named after the functions they are contained in.

In order to be able to include more kernels in our performance evaluation we further reduce the number of OpenMP 4.0 target regions to 16 by fusing the last three regions under the *ApplyMaterialPropertiesForElems* kernel (K_{16}). This will allow us to directly compare the performance of this target region with that of the corresponding CUDA kernel.

Another discrepancy between the two codes is that the OpenMP parallel loops allow any problem size (limited only by the range of the *int* type) while the CUDA C/C++ implementation is limited by the number of thread blocks that the device can support. This means that the CUDA implementation cannot support large iterations spaces, like those that can be expressed with the 64-bit integer data type. This difference may translate at compiler level into a distinct set of low-level optimisations being applied (e.g. different register allocation heuristics).

The OpenMP implementation of LULESH is capable of switching between SoA (structure of arrays) and AoS (ar-

CUDA kernels	**OpenMP 4.0 target regions**	**Kernel ID**
CalcVolumeForceForElems_kernel and *AddNodeForcesFromElems_kernel*	*CalcForceForNodes*	K_1
	InitStressTermsForElems	K_2
	IntegrateStressForElems_1	K_3
	IntegrateStressForElems_2	K_4
	CalcHourglassControlForElems	K_5
	CalcFBHourglassForceForElems_1	K_6
	CalcFBHourglassForceForElems_2	K_7
CalcAccelerationForNodes_kernel	*CalcAccelerationForNodes*	K_8
ApplyAccelerationBoundaryConditionsForNodes_kernel	*ApplyAccelerationBoundaryConditionsForNodes*	K_9
CalcPositionAndVelocityForNodes_kernel	*CalcVelocityForNodes*	K_{10}
	CalcPositionForNodes	K_{11}
CalcKinematicsAndMonotonicQGradient_kernel	*CalcLagrangeElements*	K_{12}
	CalcKinematicsForElems	K_{13}
	CalcMonotonicQGradientsForElems	K_{14}
CalcMonotonicQRegionForElems_kernel	*CalcMonotonicQRegionForElems*	K_{15}
ApplyMaterialPropertiesAndUpdateVolume_kernel	*ApplyMaterialPropertiesForElems*	K_{16}
	EvalEOSForElems	
	UpdateVolumeForElems	

Table 1: Correspondence between the kernels of the CUDA C/C++ implementation of LULESH and the offload regions (regions marked with the *target* directive) in the OpenMP 4.0 implementation.

ray of structures) data structure representations. We have only considered the SoA representation since this is a better match for the memory model of the device.

4.2 Porting challenges

Every OpenMP 4.0 converted parallel region requires a *pragma omp target data* directive to explicitly map host data to device data. The implicit data access descriptor for every map is *tofrom* which triggers the copying of data to and from the device. To reduce the number of transfers between host and device we inspect the parallel regions and derive the appropriate OpenMP clause (*to*, *from*, *alloc*).

Scientific applications are often architected using object oriented principles which lead to deep object hierarchies. Encapsulating the representation of data may result in data structures that cannot be trivially mapped using current OpenMP syntax. Even in the case of LULESH's flat hierarchy, obtaining references to the underlying arrays required breaking the encapsulation principle.

Both porting challenges will be made simpler in the OpenMP 4.1 release. A more robust application-wide data offloading mechanism will be introduced (*target data enter/exit*) as well as support for limited deep copying of object hierarchies.

5. PERFORMANCE ANALYSIS

In this section we evaluate the performance of the *target* pragma offloading by comparing the performance of our LULESH OpenMP 4.0 implementation with that achieved by the reference CUDA C/C++ implementation [11]. We track performance differences down to the low-level code by analyzing generated PTX and SASS code and inspecting performance counters.

The OpenMP 4.0 implementation of LULESH has been tested for correctness against the original OpenMP version by comparing the values of the arrays at every timestep. The number of timesteps for each problem size is the same as in the original version.

5.1 Experimental Setup

The tests are run on an OpenPower node using two Power 8 sockets (model PowerNV 8247-42L) and two NVIDIA Kepler GPUs K40m. The operating system run by the host processor is a *bare-metal* Linux distribution (Ubuntu version 14.04.1). The OpenMP tests are run using our CLANG/LLVM compiler which extends the current trunk repository with additional OpenMP constructs. Our CLANG/LLVM compiler which implements the control loop scheme is derived from an open source version available on Github [3]. The implementation of the combined construct scheme is not currently available as open source.

For the OpenMP runtime library we chose the following configuration:

- For the host OpenMP library and the offloading logic that communicates with the NVIDIA CUDA device driver we employ the IBM Lightweight OpenMP implementation (LOMP).
- For the GPU OpenMP library we use the open source implementation available on Github [14].

For CUDA C/C++ tests we use NVCC for CUDA release 7.0, version 7.0.27. The compilation flags for CLANG are: `-fopenmp=libomp -O3 -omptargets=nvptx64sm_35-nvidia-linux`. The compilation flags for NVCC are `-O3 -arch sm_35`.

Both compilation toolchains generate PTX code which is processed by the same low-level tools, namely PTXAS and NVLINK. NVCC and CLANG/LLVM use two different tools for low level optimization and code generation of PTX. NVCC uses LIBNVVM [5], which is only available in binary format and is based on LLVM IR version 3.5. Our OpenMP compiler relies on LLVM version 3.8 backend for NVPTX, which is instead open source [10]. This may result in different generated PTX code.

In all our experiments we focus on the runtimes of the kernels and offloaded regions. We do not take into consideration the time spent moving data between the host and the GPU. We assume that the time taken to perform the data transfers is the same for OpenMP and CUDA. For each offloaded region we measure the runtime using nvprof. For the cases analyzed in this paper, each offloaded region or CUDA C/C++ kernel is called once per LULESH timestep.

The problem size is given by the number of elements in the mesh. We run experiments with the following mesh sizes: 12^3, 30^3, and 100^3. As shown in Table 1, there is a limited number of direct correspondences between the OpenMP 4.0 offloaded regions and the CUDA C/C++ kernels. To improve the quality of comparison we fuse at source level the OpenMP 4.0 target regions that are responsible for applying the material properties to obtain K_{16}.

Even though we analyzed the performance of the OpenMP 4.0 version of LULESH for every offloaded region, in this paper we focus on two of the four regions which have one-to-one correspondences, namely K_8 and K_{16}. The two kernels are interesting from a compiler perspective as they have a significantly different instruction mix, number of operations and control flow instructions. K_8 is similar to a benchmarking kernel as it contains no control flow instructions and performs three double precision acceleration computations (floating point multiplications), one for each component of the 3D space. K_{16} is characteristic of production-grade kernels as it contains a considerable number of floating point operations (some of which are divisions and square root operations), data-dependent control flow instructions and a large number of intermediate values. It also contains an inner loop with a trip count chosen at runtime from a small set of values. This affects load balance and prevents both compilers from performing loop unrolling optimizations. Overall, the two kernels are good representatives of the range of kernels found in the application.

For each kernel and mesh size we run two distinct LULESH implementations: the original CUDA C/C++ code and our OpenMP 4.0 version. For the latter we use two CLANG implementations: the first is based on the Control Loop scheme and the second is based on the Combined Construct optimization scheme.

The CUDA C/C++ implementation of LULESH assigns a thread to every element- or vertex-centered computation. This matches the parallelism exposed at the level of the OpenMP implementation. It allows us to use the CUDA C/C++ runtime and performance metric values as a benchmark for the OpenMP 4.0 implementation.

For kernels with high register usage, the CUDA C/C++ version has been optimized in terms of data layout and computation granularity [8]. Similarly, code transformations have been performed to the corresponding K_{16} target region in OpenMP 4.0. The iteration over regions has been replaced with the iteration over elements which matches the iteration scheme present in the corresponding CUDA C/C++ code.

The OpenMP runtime scheduling of threads is different from the one chosen in the CUDA C/C++ program. The default behavior of OpenMP static scheduling assigns contiguous chunks to threads, which may result in poor global memory coalescing when using a GPU. To improve this, we select static schedule, with chunk size equal to one, for the for loop that assigns iterations to threads. This requires the runtime to assign chunks of one iteration each to threads in a round-robin fashion [13]. This ensures coalesced accesses in the same places as in the CUDA C/C++ implementation.

For each region we identify the optimal number of teams and the thread limit. The number of *teams* is equivalent to the number of CUDA blocks while the *thread limit* represents the largest number of threads that can be allocated to a team (provided enough registers and shared memory are available). For each analyzed offloaded region we consider the number of teams and threads that minimizes the runtime.

This is obtained by varying the number of teams and threads per team in powers of two. For completeness, we also considered the case where the number of blocks is obtained by dividing the size of the iteration space over the mesh by the number of threads in each block. The latter is the same scheme used by the CUDA C/C++ implementation to determine the number of blocks and threads per block.

As shown in this section, a direct consequence of using the Control Loop is the large number of registers allocated to each thread. The large number of registers used by every kernel can reduce the number of concurrent blocks which were able to run on a given Streaming Multiprocessor (SM) of the GPU at a time. The Control Loop also uses shared memory to allow communication between the master thread of each team and the rest of the threads in the team. Shared memory has similar effects on the number of concurrent blocks and occupancy.

The version of the compiler based on the Combined Construct Optimization was designed to support OpenMP 4.0 loops which do not require any fine-grained synchronization within the kernel. This is characteristic of all LULESH offload regions as the parallelism is over the main elements of the mesh and the kernel applications are independent. No shared memory is used in this version.

Preliminary PTX code analysis suggested that a large number of registers was exclusively used by the Control Loop itself i.e. outside the body of the main kernel. We gradually lower the number of registers per thread in powers of two starting from the maximum allowed of 256, by using the *maxregcount* option for ptxas. For the particular kernels we analyze, the best runtime is obtained when capping the number of registers to 64. This is now the default register cap for the CLANG/LLVM toolchain.

Using fewer registers results in load and store spills to local memory. We will show results in which register spilling is a performance bottleneck, when the overall local memory size used by all active threads is larger than the L1 cache. In such cases registers are spilled further down the memory hierarchy to L2 or even global memory. Limiting the number of registers leads to a trade-off between the number of threads (degree of parallelism) and the amount of local memory used.

5.2 Parameter Tuning and Overall Performance

The achieved running times of the CUDA C/C++ and OpenMP 4.0 versions of LULESH (with the two OpenMP 4.0 runtime environments for the *Control* Loop and *Combined* Construct Optimization) for the kernels K_8 and K_{16} and varying problem sizes are presented in Tables 2 and 3.

The minimum runtime is achieved for different numbers of teams and threads: across the same kernel, the resources allocated per thread differ (see Table 4) due to different im-

Problem Size	K_8								
	CUDA			**OpenMP 4.0 (Control)**			**OpenMP 4.0 (Combined)**		
	Blocks	**Threads**	**Time(μs)**	**Teams**	**Threads**	**Time(μs)**	**Teams**	**Threads**	**Time(μs)**
12^3	(*)	128	3.264	32	64	32.512	64	256	4.352
30^3	(*)	128	8.224	64	128	49.087	128	128	8.32
100^3	(*)	128	304.45	128	64	567.29	1024	128	318.4

Table 2: Run times for the K_8 kernels for different problem sizes. The table includes the number of teams and threads which produce the best runtimes. The number of CUDA blocks is computed dynamically by dividing the number of iterations by the number of threads (i.e. 128).

Problem Size	K_{16}								
	CUDA			**OpenMP 4.0 (Control)**			**OpenMP 4.0 (Combined)**		
	Blocks	**Threads**	**Time(μs)**	**Teams**	**Threads**	**Time(μs)**	**Teams**	**Threads**	**Time(μs)**
12^3	(*)	128	92.928	32	64	295.65	128	32	102.83
30^3	(*)	128	113.47	64	256	525.34	1024	64	222.72
100^3	(*)	128	2015.8	512	128	5494.6	1024	128	4634.5

Table 3: Run times for the K_{16} kernels for different problem sizes. The table includes the number of teams and threads which produce the best runtimes. The number of CUDA blocks is computed dynamically by dividing the number of iterations by the number of threads (i.e. 128).

plementations and compilation processes.

The Control Loop-based OpenMP 4.0 implementation is significantly slower than CUDA C/C++. Increasing the problem size lowers the difference between Control Loop and CUDA C/C++ execution times, as size-independent overheads are amortized. For K_8 the difference in performance is reduced significantly from ten times to about 1.8 times. For complex kernels like K_{16} the difference is reduced only slightly from 3.2 times to 2.7 times.

The run time for the OpenMP 4.0 implementation based on the Combined Construct Optimization is consistently lower than the version using the Control Loop. For K_8 performance is similar to CUDA C/C++ and approaches the CUDA performance further, as the problem size increases. For a complex kernel like K_{16} the run time is similar to CUDA for small problem sizes. As the problem size increases the difference in performance increases up to about 2.3 times the CUDA C/C++ run time.

5.3 Detailed Analysis of Results

To understand the performance trends in Tables 2 and 3 we use nvprof to report the values of the GPU hardware counters (Table 4). We also use the report generated by PTXAS and NVLINK regarding the number of registers used, shared memory, stack frame and load/store spill sizes. We further inspect the PTX and disassembled SASS codes to track the performance characteristics to the backend generated code.

The Combined Construct Optimized version produces code which is similar to the one generated by NVCC in terms of frequency of each instruction type. By comparing the PTX and SASS codes we observe that, for the payload body, the instruction mix is comparable but the overall register count is not. The code outside the main payload body is different due to reasons described in the next section. The performance is affected by GPU specific resource management such as registers, shared memory and local memory accesses which limit latency hiding and occupancy. The high usage of these resources results in higher recorded instructions counts (using nvprof) at runtime.

5.3.1 *Register usage*

We identified register usage as the main source of overhead for the Combined Construct version. Both OpenMP 4.0 implementations use a larger number of registers than CUDA C/C++.

High register counts are due to several factors:

- **Arguments to target regions are passed as reference to pointer:** Arrays are required to be dereferenced twice which uses twice as many registers. The register allocator does not consider the registers holding the initial pointer references free and hence does not re-use them over the course of the kernel. Ongoing work on the CLANG/LLVM toolchain is currently removing this requirement in order to comply with the newest OpenMP 4.1 standards.

- **Different looping strategies** On the CUDA C/C++ side, the iteration over threads is done by computing the thread identifier based on the position of the thread in the CUDA grid and then comparing it to the maximum number of iterations. This excludes unnecessary threads, but it limits the iteration space size of loops to the maximum number of CUDA blocks multiplied by the maximum number of threads per block. This is enough to support a 32-bit integer variable, but it is not for a 64-bit one. Code synthesis with the Combined Construct Optimization generates a loop over threads in order to deal with arbitrarily large data. The Control Loop uses a while loop and a switch statement which chooses the next thread state. Extra registers are required to maintain the thread states and to handle the various function calls in the generated code.

- **Potentially different register allocation optimizations:** The CLANG/LLVM toolchain uses the NVPTX while the NVCC compiler uses LIBNVVM. This may lead to different register usage analysis being applied. We have manually checked that the number of register between the PTX and disassembled SASS is consistent

		K_8			K_{16}		
		CUDA	**OpenMP (Control)**	**OpenMP (Combined)**	**CUDA**	**OpenMP (Control)**	**OpenMP (Combined)**
	Registers	16	64	38	58	64	64
	Stack Frame Size (Bytes)	0	184	0	0	376	232
	Spill Stores Size (Bytes)	0	100	0	0	364	308
	Spill Loads Size (Bytes)	0	132	0	0	416	404
12^3	Control-Flow Instructions	2304	58037	16384	147802	240895	182047
	Load/Store Instructions	15379	244486	78905	67014	578018	404642
	Achieved Occupancy	7%	6%	27%	6%	6%	3%
	Dev Mem Read BW (GB/s)	0.22	7.81	11.79	0.03	2.84	1.38
	Dev Mem Write BW (GB/s)	11.64	11.09	8.70	1.03	7.45	3.53
	L2 BW (L1 Reads) (GB/s)	12.44	14.23	9.31	0.66	7.19	2.41
	L2 BW (L1 Writes) (GB/s)	9.32	19.08	6.98	0.83	7.97	2.86
	Loc Mem Store BW (GB/s)	0	4.60	0	0	3.98	2.52
	Loc Mem Load BW (GB/s)	0	10.28	0	0	10.63	10.17
	L1 Local Hit Rate	N/A	49.42%	N/A	N/A	48.26%	63.94%
30^3	Control-Flow Instructions	29824	273375	29791	5362030	6778635	6287259
	Load/Store Instructions	208537	1266418	564530	937758	10003592	9520272
	Achieved Occupancy	79%	26%	45%	37%	40%	35%
	Dev Mem Read BW (GB/s)	89.02	45.06	74.76	22.15	37.28	27.96
	Dev Mem Write BW (GB/s)	68.95	66.22	56.98	13.58	44.94	53.16
	L2 BW (L1 Reads) (GB/s)	76.90	59.15	62.99	8.54	78.27	97.20
	L2 BW (L1 Writes) (GB/s)	58.76	67.73	47.24	10.98	40.03	43.65
	Loc Mem Store BW (GB/s)	0	17.81	0	0	20.06	22.71
	Loc Mem Load BW (GB/s)	0	27.25s	0	0	86.42	111.76
	L1 Local Hit Rate	N/A	22.60%	N/A	N/A	19.46%	15.44%
100^3	Control-Flow Instructions	1030400	2275229	1030301	125543531	153960083	150184019
	Load/Store Instructions	7212107	15279126	15014038	35758744	217453536	220972064
	Achieved Occupancy	87%	26%	69%	48%	45%	41%
	Dev Mem Read BW (GB/s)	125.20	84.31	120.23	58.96	72.26	64.98
	Dev Mem Write BW (GB/s)	94.334	71.91	90.99	28.88	52.77	56.98
	L2 BW (L1 Reads) (GB/s)	107.28	83.16	101.79	18.45	122.93	144.04
	L2 BW (L1 Writes) (GB/s)	80.47	59.27	76.34	23.06	44.82	48.77
	Loc Mem Store BW (GB/s)	0	7.55	0	0	16.62	17.72
	Loc Mem Load BW (GB/s)	0	37.32	0	0	126.58	154.30
	L1 Local Hit Rate	N/A	44.96%	N/A	N/A	12.67%	13.24%

Table 4: Performance metrics for different implementations of LULESH K_8 and K_{16} kernels (CUDA C/C++, OpenMP 4.0 using the control loop and OpenMP 4.0 using the thread loop version). The metrics are provided for each problem size: 12^3, 30^3 and 100^3 mesh elements.

for the kernels analyzed in this paper. This supports our conclusion. Solutions to this would be to switch to using either ptxas or LIBNVVM for both codes or generate GPU machine code directly.

Using a large number of registers may lead to spilling. When the extra virtual registers cannot get mapped to physical registers directly, they end up being pushed to the thread stack frame. The stack frame is part of the local memory of the thread which typically uses the L1 cache but can get pushed down to L2 or even global memory for large data sizes.

5.3.2 *Register spilling*

Register spilling increases the memory traffic generating a higher number of load and store instructions. This is also confirmed by *nvprof* which shows significantly larger numbers of load/store instructions being used for both OpenMP 4.0 versions where spilling occurs.

As the problem size is increased, for large kernels, register spills can generate enough load/store memory traffic to become the performance bottleneck. This can be seen in Table 4 on the rows showing the local load/store memory bandwidth and a small L1 Local Hit Rate.

Register spilling can be used to explain some of the observed performance trends for LULESH kernel K_{16}. By lifting the register cap we can get ptxas to report on the actual number of registers required by this kernel. The register numbers reported by ptxas which would have been used by the Control Loop and Combined Construct compiler variants are 238 and 176 respectively. Since we cap the number of registers to 64, the excess registers are spilled to local memory. The quantitative impact of spills is given by the size in local memory of the spilled loads and stores reported by ptxas.

For small problem sizes the Combined Construct OpenMP 4.0 version of K_{16} is as fast as CUDA C/C++ (only 1.1 times slower). The penalties of register spills are avoided simply due to threads being able to make efficient use of the L1 cache (up to 64% hit rate). This makes the performance for small sizes bound by the kernel body execution.

For larger problem sizes the L1 hit rate becomes significantly worse (between 13% and 15%) and hence the performance degrades in comparison to CUDA C/C++. The bandwidth generated by the extra load/store traffic is high enough to become the bottleneck. For the current GPU architecture, the peak bandwidth to L1 is 187GB/s. K_{16} achieves a large percentages of that: 66% and 91% for the 30^3 and 100^3 problem sizes respectively.

The Combined Construct optimization relieves register pressure but for K_{16} this is not enough to avoid register spills completely. For LULESH kernel K_8 considerations similar to the case of the Control Loop version can be applied. For the Combined Construct Optimization of K_8, register usage is low enough to avoid register spills. This makes the memory traffic profile similar to the K_8's CUDA C/C++ implementation: no local memory traffic is detected during profiling. The lack of register spills for K_8 also allows better latency hiding at larger problem sizes (occupancy increases from 27% to 69%) which leads to running times which approach those achieved by the CUDA C/C++ implementation at large problem sizes.

The performance of K_8 is bound by the bandwidth to global memory. For the largest problem size, out of the peak bandwidth figure of 288GB/s the K_8 Combined Construct implementation achieves 73% of combined load/store bandwidth.

5.3.3 *Volume of Instructions*

The number of double precision floating point operations (not included in the tables) is constant across the three implementations.

The generated PTX and SASS codes suggest that the control loop is also responsible for a high number of additional control flow instructions (see Table 4) needed by the state switches. The absence of the control loop lowers the number of control flow instructions used. The number of control flow instructions has a larger impact for smaller problem sizes and smaller kernels like K_8. For complex kernels like K_{16}, which contain a large number of control flow instructions in their payload, the impact is negligible.

6. CONCLUSION

In this paper we show the performance evaluation for a representative set of kernels drawn from the LULESH proxy application. The results were obtained using our CLANG/LLVM compiler stack which implements the OpenMP 4.0 accelerator offloading constructs. The compiler contains two code generation schemes with different levels of generality. At compile time, one of the two schemes is chosen based on the combination of OpenMP 4.0 directives provided. The performance shows that for simple benchmark-style kernels the code generation changes are effective and make the performance similar to a native CUDA C/C++ implementation. For complex kernels where the register usage is high, we have outlined the set of optimizations required to bridge the performance gap between OpenMP 4.0 and a native CUDA C/C++ implementation.

Native device programming is usually more constrained and does not have to handle the multitude of cases which typically occur on CPUs. The Control Loop handles the generality required by OpenMP 4.0 at the cost of more instructions being executed, more registers being allocated and more shared memory being used.

Having a more specialized code generation scheme capable of producing PTX/SASS codes for OpenMP 4.0 target regions similar to the codes generated by NVCC/LIBNVVM for equivalent CUDA C/C++ implementations enables us to reduce the instruction overhead for small kernels and improve register allocation.

Ongoing work is exploring the impact on the number of allocated registers of different optimizations (such as using single indirections for the kernel arguments) while maintaining the loop over threads. This work is currently being applied to small benchmark kernels. Following full integration in the compiler, similar considerations for complex kernels will be explored. Further work also involves the extension of the applicability of the specilaized code generation scheme to the OpenMP 4.0 vectorization *simd* pragma, global reductions and nested parallelism.

7. ACKNOWLEDGMENTS

The authors would like to thank scientists at U.S. DoE laboratories and NVIDIA for their valuable input to the optimization process of the OpenMP compiler and runtime libraries for the OpenPower platform. This paper is partially supported by the CORAL project LLNS Subcontract No. B604142.

8. REFERENCES

[1] 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 *Submitted to the 2015 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '15, 2015.

[2] 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.

[3] Github repository for extended Clang implementation supporting OpenMP 4.0. https://github.com/clang-omp/clang_trunk.

[4] CORAL award announcement. http://energy.gov/articles/department-energy-awards-425-million-next-generation-supercomputing-technologies.

[5] CUDA toolkit webpage. http://docs.nvidia.com/cuda/index.html.

[6] A proposal for OpenMP offloading on GPUs in gcc. https://gcc.gnu.org/ml/gcc/2015-03/msg00331.html.

[7] I. Karlin. LULESH Programming Model and Performance Ports Overview. Technical report, LLNL, 2012. https://codesign.llnl.gov/pdfs/lulesh_Ports.pdf.

[8] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still. Exploring traditional and emerging parallel programming models using a proxy application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, may 2013.

[9] The LLVM Compiler Infrastructure webpage. http://llvm.org/.

[10] LLVM Backend component for NVPTX archietecture (Nvidia GPUs). http://llvm.org/docs/NVPTXUsage.html.

[11] LULESH webpage. https://codesign.llnl.gov/lulesh.php.

[12] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

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

[14] Github repository for libomptarget offloading and GPU OpenMP runtime. https://github.com/clang-omp/libomptarget.

[15] OpenACC webpage. http://openacc.org.

[16] OpenCL standard webpage. https://www.khronos.org/opencl.

[17] OpenMP standard webpage. http://openmp.org/.

[18] G. Ozen, E. Ayguade, and J. Labarta. On the roles of the programmer, the compiler and the runtime system when programming accelerators in OpenMP. In L. DeRose, B. de Supinski, S. Olivier, B. Chapman, and M. MÃijller, editors, *Using and Improving OpenMP for Devices, Tasks, and More*, volume 8766 of *Lecture Notes in Computer Science*, pages 215–229. Springer International Publishing, 2014.

[19] *CUDA-enable STL Thrust Library*, March 2015. http://docs.nvidia.com/cuda/thrust.

[20] Y. Yang and H. Zhou. CUDA-NP: Realizing nested thread-level parallelism in GPGPU applications. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 93–106, New York, NY, USA, 2014. ACM.