Coordinating GPU Threads for OpenMP 4.0 in LLVM

Carlo Bertolli, Samuel F. Antao, Alexandre E. Eichenberger, Kevin O'Brien
Zehra Sura, Arpith C. Jacob, Tong Chen, Olivier Sallenave
IBM T.J. Watson Research Center
1101 Kitchawan Rd., Yorktown Heights, NY, USA
{cbertol,sfantao,alexe,caohmin}@us.ibm.com
{zsura,acjacob,chentong,ohsallen}@us.ibm.com

ABSTRACT

GPUs devices are becoming critical building blocks of High-Performance platforms for performance and energy efficiency reasons. As a consequence, parallel programming environment such as OpenMP were extended to support offloading code to such devices. OpenMP compilers are faced with offering an efficient implementation of device-targeting constructs.

One main issue in implementing OpenMP on a GPU is related to efficiently supporting sequential and parallel regions, as GPUs are only optimized to execute highly parallel workloads. Multiple solutions to this issue were proposed in previous research. In this paper, we propose a method to coordinate threads in an NVIDIA GPU that is both efficient and easily integrated as part of a compiler. To support our claims, we developed CUDA programs that mimic multiple coordination schemes and we compare their performances. We show that a scheme based on dynamic parallelism performs poorly compared to inspector-executor schemes that we introduce in this paper. We also discuss how to integrate these schemes to the LLVM compiler infrastructure.

Categories and Subject Descriptors

D.3 [**Processors**]: Compilers

General Terms

Algorithms

1. INTRODUCTION

In recent years, High-Performance Computing (HPC) applications have increasingly leveraged GPUs to increase performance, either via specialized numerical libraries or carefully tuned algorithms. To obtain significant accelerations on current generations of GPUs, it is essential to expose a high degree of parallelism, often much higher than currently found in most HPC applications. Thus GPU acceleration does not come for free: HPC programmers need to expose more parallelism, usually by hand-tuning their code [4], performing architecture-specific transformations, or using compilers for domain-specific libraries and languages [7, 1, 6].

OpenMP® and other High-Performance parallel languages were mainly designed for CPU execution, but many have recently added support for accelerators. For example, the latest OpenMP 4.0 specification [10] introduced significant support for *offloading* directives that allows an application programmer to annotate regions of code whose execution is to be transferred to an accelerator device.

When designing OpenMP directives for offloading to accelerator devices, OpenMP maintained its overall programming philosophy of enabling users to express software optimizations in a highly flexible manner. Offloaded regions of codes are referred as *target regions* and may include sequences of sequential, parallel regions, and possibly nested parallel regions. This flexible design forces high performance compilers to efficiently support a wide range of OpenMP constructs on accelerator devices. Some of these patterns are especially challenging on GPUs; for example, while highly parallel regions are efficiently supported on GPUs, sequential regions typically underperform.

To handle sequential code in an offloaded region, a seemingly simple approach is to map the sequential regions to the host CPU while executing parallel regions on the GPU. This approach requires multiple invocations to the GPU to execute offloaded regions of code with multiple parallel regions separated by sequential code. It may also require complex data movement between host and device, depending on the memory model between the host and the GPUs. In addition, this solution cannot be applied for sequential regions within nested parallel regions. This approach also violate the implicit assumption that offloaded regions of code are expected to fully execute on the offload device.

Another solution to handle parallel and sequential regions of code on a GPU is to rely on GPU-initiated parallel regions. This technique, referred to as dynamic parallelism by NVIDIA®, is supported by the CUDA® language [5]. Using this mechanism, we map each sequential and parallel region to an independent CUDA kernel and nested regions are translated into multiple nested kernel calls. The main drawback of this approach is the non-negligible overheads of kernel launches in current GPU technology. Our analysis and related work [15] show that solutions based on multiple kernel invocations deliver lower performance compared to single kernel solutions.

A more advanced solution [15] is based on launching multiple blocks and threads in a single kernel call, and by guarding every statement in the sequential region with an if-statement that limits execution to a single thread only. Every if-statement is followed by a synchronization between participating threads to guarantee consistency. This solution may incur in an unnecessary number of thread synchronizations, and it is hard to implement in a modular way in a compiler. Our analysis of the LLVM compiler shows that implementing this scheme would require code generation support for every C/C++ statement, rather than only affecting the OpenMP implementation.

In this paper, we introduce a novel thread coordination scheme that enables the launch of a single fully-parallel GPU kernel to offload regions that include arbitrary code sequences of sequential, and parallel regions. The scheme can be integrated in a modular way in the LLVM compiler as it only affects OpenMP code generation components. With this scheme, we program CUDA blocks and threads to efficiently coordinate their execution across sequential and parallel regions while never returning execution to the host or resorting to dynamic parallelism.

We compare the performance of multiple thread coordination schemes, and we show how the best ones can be integrated in the LLVM compiler. Support for OpenMP 4.0 in LLVM is currently in progress, and the goal of this paper is to develop an approach that is compatible with its design and implementation characteristics.

The contributions of this paper are:

- We describe existing thread coordination schemes that can be used to implement interspersed sequential and parallel regions in OpenMP 4.0 target regions, and we introduce a novel scheme.
- We report performance associated with the coordination schemes and discuss results.
- We show that our scheme is easy to integrate in the LLVM compiler in a modular way.

This paper is organized as follows. Section 2 describes related work for OpenMP and other accelerator directive languages. Section 3 contains background concepts including OpenMP offloading constructs, CUDA and Clang. Section 4 describes thread coordination schemes for sequential and parallel regions on a GPU. Section 5 reports about the performance analysis of the coordination schemes. Section 6 briefly highlights the implications for implementing the proposed thread coordination scheme in Clang. Section 7 concludes the paper.

2. RELATED WORK

Several notable experiences have been recently made to implement OpenMP 4.0 constructs on acceleration architectures (e.g. see [2, 9, 13]). Caballero et al. [2] describe an implementation of a barrier synchronization on the Intel® MIC architecture, and [13] describes an initial implementation of the OpenMP run-time on a ARM® plus TI® DSP processor combination. A work that more closely matches our contributions can be found in [9]. The authors describe the implementation of target regions and discuss solutions to implement sequences of parallel and sequential regions. The preferred solution is based on dynamic parallelism, when available. When dynamic parallelism is not available, sought solutions include: (a) moving sequential parts to the host, while leaving parallel regions on the GPU, resulting in multiple kernel calls per parallel region; (b) starting a CUDA kernel with multiple blocks and threads. The former solution is the authors' choice, whereas the latter solution is not further explored in the paper.

Yang et al. [15] present a solution to support sequential and parallel regions that is similar to one of the schemes we describe (*if-master* in Section 4). Threads are distinguished between master and slaves using their identifiers, possibly using the three available dimensions for these when large parallelism is required. Unlike this, our schemes are studied in the context of LLVM and we design them as extensions of the LLVM IR code generation performed by Clang.

An alternative to the OpenMP 4.0 acceleration model is offered by the OpenACC® 2.0 [11] pragma language. The design of OpenACC specifically targets acceleration devices. For instance, it includes native constructs to configure execution for alternative acceleration architectures. Similarly to OpenMP, OpenACC parallelism model is based on a three level hierarchy, including *gangs*, *workers* in a gang, and *vector* parallelism in each worker. However, synchronizations at any level (gangs, workers, or vector) are not explicitly required, and may fail in some implementation. This is opposed to OpenMP semantics, as synchronization between teams is not allowed, while it is allowed between threads in the same team. If OpenMP threads in a same team are mapped to multiple CUDA blocks, the implementation is required to permit inter-block synchronization. This is not a native operation in CUDA and its implementation requires careful resource scheduling.

3. BACKGROUND

This section reviews background concepts required for the next sections, including offloading extensions to OpenMP, CUDA, and LLVM.

3.1 OpenMP 4.0 Offloading Constructs

Offloading directives are used to take advantage of accelerators (or devices, in OpenMP terminology) like GPUs present on the system (see [10] for details). These directives apply to the code block that they enclose. The following concepts are used in OpenMP. **Target** regions are structured code blocks whose execution should be offloaded onto a device. This is conditional on the run-time availability of a device, the ability of the compiler to generate device code, and other factors. **Device data environments** are used to allocate and transfer data to and from the device.

The **target** directive expresses the creation of both a device data environment and an offloading target region. It may have associated clauses to specify further details, like the exact device to use if more than one is present in the system (**device** clause) or whether the data should be moved to/from the device or allocated in the device memory (**map** clause).

As an example, the **target** directive of the example in Figure 1 creates a data device environment including the arrays *a* and *b*, as specified by the map clauses. The **to** and **from** clauses specify the directions of copy between host and device data environments for the two variables.

The **teams** directive groups the threads of a device into sets (called teams), creating a parallelism abstraction where the threads in each team collaborate more closely.

The programmer can control the number of teams and the maximum number of threads in each team by specifying the **num_teams** and **thread_limit** clauses along with the **teams** directive, respectively. One of the threads in each team is designated **team master** and the structured block following the directive is executed by all team masters across the different teams. We denote those regions where only the team masters execute as team-sequential. For instance, the statements in Figure 1 at lines 7 and 16 are only executed by team masters. Only when a **parallel** directive is encountered, all the threads in each team start execution, to collaborate in the execution of the enclosed structured block. We denote those regions where all threads in each team execute as team-parallel. As an example, the for-loop at line 10 of Figure 1 is scheduled to all threads in the team. The **distribute** directive is associated to a for-loop, whose iterations will be distributed across teams. Only static scheduling is allowed for this directive.

Finally, the use of a *teams* directive is not mandatory in a target region. This can contain any OpenMP construct without the intervention of a teams construct. In such a case, there is a single team in the target region, and threads in the team can arbitrarily synchronize amongst themselves.

3.2 CUDA

This section is a synthetic description of the main features of the CUDA programming language for NVIDIA GPUs that are needed for the understanding of the following sections. We assume basic knowledge of NVIDIA GPU terminology. The reader is encouraged to refer to [5] for full information.

CUDA is built as an extension of other languages, namely C/C++ and Fortran. A CUDA program is composed of: host functions, which are common C/C++ or Fortran functions/subroutines; global functions (or kernels) which represent entry points from the host to GPU execution; device functions, which execute on the GPU and can only be called from another device function, or a kernel. Both global and device functions are programmed in terms of the *block* and *thread* abstractions. A CUDA thread is the control- and data-flow unit. A block includes a set of threads that are mapped onto the same physical resource. When launching a kernels, the programmer specifies the number of blocks and threads in each block.

In kernels and device functions, to distinguish between blocks and threads in each block, each thread can access private identification variables. For instance, *threadIdx.x* is the identifier of a thread in its block on the x-axis of indices. Similarly, *blockIdx.x* is the identifier of the block to which a thread is assigned on the x-axis of block indices. In this paper we will only use a single dimension for threads and blocks.

Synchronization between threads is only natively defined for threads in the same block. This is obtained by calling the primitive *__syncthreads* function. No semantics is specified for synchronizations across threads in different blocks. The placement of synchronization points is critical in programming terms, as the compiler needs to ultimately guarantee that the same syntactic synchronization instruction is executed by all threads in the same block [5].

3.3 LLVM

The LLVM [8] compiler infrastructure is composed of the following passes: the Clang C/C++ frontend, which generates code in the LLVM Intermediate Representation (IR); the LLVM high-level optimizer; the LLVM back-end. Here, we focus on Clang, as it currently implements all OpenMP transformations.

Clang has been developed side-by-side with the LLVM library and therefore follows most of its implementation guidelines. One of these is the effort to make each (new) feature to be structured into logical modules in order to ease the integration and reduce disruption caused by interdependence. Following this modularity design, each action accomplished by the consumers of each basic element (token, AST node) tends to be self-contained, either by extending a default action applied on top of a class of elements or by creating a new class.

An effort to add OpenMP support to Clang has been recently carried out whose implementation is being maintained in [3]. This implementation currently fully supports the full OpenMP 3.1 specification and part of OpenMP 4.0 [10] has been gradually merged into the Clang main project. Similar to other features in Clang, the OpenMP support is implemented into self-contained consumers of tokens and AST nodes of the directives and clauses.

An important aspect is that code generation for an OpenMP directive consists of the emission of the appropriate Runtime Library (RTL) calls. All sub-statements enclosed in a directive are emitted in the exact same way they would if no OpenMP implementation was in place. This enables software scalability in the face of new directives added to the OpenMP specifications, as well as a clean modular development.

The LLVM official OpenMP Runtime Library is available at [3] and it is the default interface used by Clang during code generation. Some features added by the OpenMP 4.0 specification, in particular offloading, require an extended RTL, including device-specific functionalities. It also requires customization of the code generation, which is currently under discussion by the LLVM community at the time this paper was produced.

```
# pragma omp target map(to:b), map(from:a) {
   # pragma omp teams num_teams(N), \
           thread_limit(M) {
      // sequential part
      a[omp_get_team_num()] = b*2;

      # pragma omp parallel for
      for (int i = 0; i < NUM_IT; i++) {
        // use of a.. (omitted)
      }

     // back to sequential
     a[omp_get_team_num()] *= 2;
  }
}
```

Figure 1: OpenMP program using target and teams construct with team-sequential and team-parallel regions.

```
 1 __global__ void parallelfor_kernel (..) {
 2  <codegen for parallel region>
 3 }
 4
 5 __global__ void teams_kernel (..) {
 6   a[omp_get_team_num()] = b*2;
 7
 8   parallelfor_kernel <<<1, M>>> (..);
 9   cudaDeviceSynchronize ();
10
11   a[omp_get_team_num()] *= 2;
12 }
13
14 __global__ void target_kernel (..) {
15   teams_kernel <<<N, 1>>> (..);
16   cudaDeviceSynchronize ();
17 }
18
19 void hostFunction (..) {
20   target_kernel <<<1, 1>>> (..);
21 }
```

Figure 2: Sketch of CUDA program implementing the example in Figure 1 using dynamic parallelism.

```
 1 #define MASTER 0
 2 __global__ void kernel (..) {
 3   // only team master
 4   if (threadIdx.x == MASTER) {
 5     a[omp_get_team_num()] = b*2;
 6   }
 7
 8   // all threads in block wait for master
 9   __syncthreads ();
10
11   <codegen for parallel for>
12   __syncthreads (); //required by #for
13
14   // only team master
15   if (threadIdx.x == MASTER) {
16     a[omp_get_team_num()] *= 2;
17   }
18
19   // all threads in block wait for master
20   __syncthreads ();
21 }
```

Figure 3: Sketch of CUDA program implementing the example in Figure 1 using coordination of threads.

4. COORDINATION SCHEMES FOR TARGET REGIONS

A main feature exposed by target constructs is related to the presence of sequential and parallel regions. This represents an issue when targeting GPUs, both when these regions are themselves nested inside a teams region, or outside of it. In this section we discuss the challenges associated to the case when the teams construct is used. We show examples with increasing complexity and we present four different implementation schemes and their associated performances using CUDA. We also discuss briefly the implications for Clang code generation.

4.1 Dynamic Parallelism and If-Master Coordination Scheme

Figure 1 shows a target region with a teams construct. This contains interspersed sequential and parallel regions. As discussed above, the teams construct creates a league of teams and it starts execution of each team master. All other threads in each team will only be activated upon encountering a parallel region.

The coordination of threads for sequential and parallel regions can be implemented using dynamic parallelism (see Figure 2). The host starts a kernel (*target_kernel* in the example code) with one block and one thread (line 20). When the thread encounters the teams region, it launches a further kernel (*teams_kernel*) choosing the requested team number as number of blocks (parameter *N* in the <<<>>> notation). Each block contains a single thread (parameter *1*). Notice that we assume to implement teams using blocks of threads. The thread in each block will execute the code as team master. Team masters will execute the sequential code sections independently of each other. Upon encountering a parallel region, each team master launches a further kernel (one for each team/block) with the requested number of threads for each team (*parallelfor_kernel* with 1 block and M threads in each block). These will execute the parallel region.

The described implementation maps each OpenMP pragma into a a kernel call. A relaxed implementation removes the first blocking level, corresponding to the target pragma, and it directly starts N blocks with the team masters in them.

Even in the relaxed version, this scheme makes extensive use of dynamic parallelism: every parallel region is mapped to a kernel call. If we follow this in a strict manner, nested parallel regions are also mapped to further nested kernel launches. This can result in a large number of possibly nested kernel calls, which will incur in large overheads [15].

An alternative to this solution removes the need to use dynamic parallelism, but it comes at the cost of a more difficult coordination of threads within a block. In this solution, we start N blocks each with M threads in them upon encountering the target region. The kernel generated by the compiler is required to coordinate threads in such a way that only the team masters execute team-sequential regions. This is shown in Figure 3.

All team-sequential regions are guarded by an if-statement that enables execution of only team master threads. Intra-block synchronizations are added at lines 9 and 20, to prevent non-master threads from executing the following instructions before the sequential region has been completed by the master. We call *if-master* this thread coordination

```
 1 # pragma omp target teams num_teams(N),
       thread_limit(M), map(to:b), map(from:a[0:N])) {
 2   int me = omp_get_team_num();
 3   if (++a[me] > 0 && ++b[me] > 0) {
 4     # pragma omp parallel for
 5     for (int i = 0; i < NUM_IT/2; i++) {
 6       // parallel region 1
 7     }
 8   } else {
 9     # pragma omp parallel for
10     for (int i = NUM_IT/2; i < NUM_IT; i++) {
11       // parallel region 2
12     }
13   }
14 }
```

Figure 4: OpenMP example showing inter-mixed data and control statements in a team-sequential region.

```
  int me = omp_get_team_num ();
  if (threadIdx.x == MASTER)
    ++a[me];

  __syncthreads ();

  // decision for all team threads
  if (a[me] > 0) {
    if (threadIdx.x == MASTER)
      ++b[me];

    __syncthreads ();

    // decision for all team threads
    if (b[me] > 0) {
      // codegen parallel region 1
    } else {
      // other regions ...
    }
```

Figure 5: Sketch of CUDA program implementing the example of Figure 4 using the *if-master* scheme.

scheme.

As we will report in the next section, performance results encourage us to follow the thread coordination path. However, the integration of this scheme in Clang is an issue. It may also provoke an unnecessary number of intra-block synchronizations. In particular, we notice that all sequential sections must be followed by an intra-block synchronization, which may represent a significant overhead when there are many such sections.

4.2 Motivation for Inspector/Executor

To understand the limitations of the if-master scheme, we introduce a further example (see Figure 4).

A team-sequential region takes decisions on what control-flow path is to be followed - ultimately, which parallel region is to be executed. The if-statement condition (line 3) is composed of two increments consumed by compare instructions. Only team master execute the Increments. However, the decision on which control-path to follow has to be taken by all threads in the team.

Figure 5 shows the implementation of the if-condition executed by team master of the example in Figure 4 using the if-master scheme. In this version, we need to carefully split those statements to be performed by the team masters only,

```
#define P1 1
#define P2 2

__global__ void kernel (..) {
  __shared__ int par_code = NULL;

  int me;

  // inspection phase (masters only)
  if (threadIdx.x == MASTER) {
    me = omp_get_team_num ();

    if (++a[me] > 0 && ++b[me] > 0)
      par_code = P1;
    else
      par_code = P2;
  }

  __syncthreads ();

  // execution phase: all threads
  if (par_code == P1) {
    <codegen for parallel region 1>
  } else if (par_code == P2){
    <codegen for parallel region 2>
  }
}
```

Figure 6: Sketch of CUDA program implementing the example in Figure 4 using the inspector-executor scheme.

from those that need to be executed by all threads. This results in two block synchronizations.

In general the requirement of having a conditional and synchronization for each basic statement outside a parallel region is a significant overhead. It is also easy to notice that the code generation for the increments and if-statements in the original program now strictly depends on the fact that we are compiling code inserted inside a “target teams” region. *This means that the OpenMP implementation affects code generation also for those statements that are not immediately related to its pragmas. This makes impractical the integration of this scheme in Clang.* In Clang terms, this means that we have to add conditionals during the code generation of all C statements based on the OpenMP context in which they are located. It would make code generation not scalable in terms of number of supported directive languages. This is highly undesirable, as we should aim at using a classical OpenMP code generation approach, which only requires direct intervention of the compiler on constructs which are directly related to the directives.

A solution to this problem can be found by adding complexity to the code generation scheme. We show this by modifying the if-master solution borrowing the idea of inspector-executor scheme [14]. In this scheme, the compiler generates code from a single code section that is split in two parts: *i)* the *inspection* phase, which typically analyzes the state of the computation; and *ii)* the *execution* phase, which effectively implements the code section. In our case, the inspection phase corresponds to team-sequential sections, while the executor phase corresponds to team-parallel regions. This is shown in Figure 6. The first part of the figure shows the inspection phase. A variable shared by thread in the same team is allocated into shared memory and used by team masters to guide the execution of all threads in the execution phase. The inspection phase is executed only by master

```
 1 #target teams num_teams(N), thread_limit(M) {
 2   if (++a[omp_get_team_num()] > 0 &&
 3      ++b[omp_get_team_num()] > 0) {
 4     int cond_gbl = 0;
 5
 6     #parallel for
 7     for (k = ..) { // par reg 1.1
 8       cond[k] = ..;
 9     }
10
11     // back to master only
12     for (k = 0 ; k < N ; k++)
13       cond_gbl = cond_gbl && cond[k];
14
15     if (cond_gbl) {
16       #parallel for
17       for (l = ..) {} //par reg. 1.2
18     }
19     else {
20       #parallel for
21       for (l = ..) {} //par reg. 1.3
22     }
23   } else {
24     #parallel for
25     for (..) {}  //par reg. 2
26 }
```

Figure 7: OpenMP example where the innermost parallel for directives are controlled by the results of an outermost parallel for.

threads (line 10), and it contains all sequential code contained in the input program. The if-then-else statement at lines 13 and 15 maps the conditionals to control-taking decisions, where the masters set the shared memory variables to the right values. After the inspection phase, a synchronization guarantees that non-master threads wait for the masters to take decisions on their behalf. When masters finish the inspection phase, all threads can proceed to the execution phase. Here, based on the decision taken in the inspection phase, the threads execute one of the parallel regions.

In comparison with the if-master scheme, we removed an intra-block synchronization. More importantly, from a code generation viewpoint, the compiler generates two (super) blocks: an *inspection block*, in which it generates all code that is inside the teams construct and outside of parallel regions. An *execution block*, in which it will generate all code that is inside parallel regions, appropriately guarded. The creation of the two super-blocks, possibly including multiple basic blocks, is triggered by the "target teams" construct. Switching between the super-blocks is performed when a parallel construct is encountered, and when it is closed. This means that no additional code generation effort is needed for constructs which are not related to OpenMP - all code generation logic is limited to OpenMP constructs.

This solution combines better performance results, due to the use of less synchronizations, and scalable code generation.

4.3 Control-Loop with Inspector/Executor

Figure 7 shows an example that requires a more complex inspector/executor scheme, because it introduces a further level of decision from data calculated in a parallel region.

The *cond_gbl* variable is calculated at line 13 as the *logical and* of an array of values that is produced by a parallel region (lines 6-9).

In terms of the inspector/executor, we cannot cleanly divide the program into two sections. The executor part that contains the codegen for parallel region 1.1 will be continued in a further nested level of inspector/execution scheme. This is needed because it is only after having executed parallel region 1.1. that we are able to take further inspection decisions. In addition, this solution would require more syntactic intra-block synchronizations, which should be carefully generated, as discussed in the background Section 3.

To support this case in a clean way, we designed an extended inspector/executor solution. This solution only includes a single inspection and execution section. Synthetically, we add a "control-loop" around the inspection/execution scheme. At the end of each sequential or parallel block a decision is taken about the next section to be executed and the participants to that section.

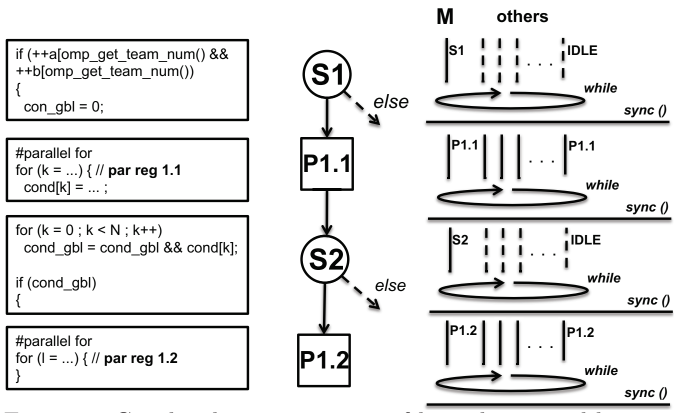
We describe a compact scheme that is followed by the code generator (see Figure 8). A label is defined for each sequential and parallel region (lines 2-4). The core of the scheme is a switch construct from line 24 to 40. Each case in the switch construct corresponds to a sequential or parallel region. For instance, the **SEQ_REG1** case contains the code generation for a sequential region, and **PAR_REG1** the one for a parallel region. The team master coordinates all threads by using two variables **labMaster** and **labOthers**. These variables contain the value of the next switch case to be executed and they can be used by the team master to distinguish between execution of itself, the master, and of all other threads in the team. For instance, when a parallel region is executed, the master selects the same label for its variable **labMaster** and for all other thread variable **labOthers**. This is done both at initialization (lines 8-10) and at the end of each parallel and sequential region (lines 30 and 36). The switch has a special case called **IDLE** that is taken by non-master threads when the master executes a sequential region case. For instance, the initialization at lines 8-10 shows that the master will execute **SEQ_REG1** while all the other threads will go into **IDLE**.

The switch case is inserted in a control-loop starting at line 14. All threads execute an iteration of the control-loop for each parallel and sequential region. Notice that the correctness of execution is guaranteed by a team synchronization at line 17. This guarantees that the master has selected the next label to execute for all threads at the end of each region case, even when the non-master threads take the **IDLE** case.

Figure 9 is a graphical representation of how the scheme of Figure 8 is applied to the example of Figure 7. The figure only shows the control-flow when all "then" cases are taken. In the left part of the figure, we show the code of the example in boxes - each box corresponds to a case in the switch. At the center, the control-flow graph is shown, using labels of the switch cases as they would be generated by the compiler. In the right part we represent the execution of threads in a team, where **M** indicates the team master and **others** the non-master threads. Vertical lines represent execution, and they are associated with labels of

```
// labels for regions
#define SEQ_REG1 0
#define PAR_REG1 1
__global__ void kernel () {
  __shared__ int labMaster;
  __shared__ int labOthers;

  if (threadIdx.x == MASTER) {
    labelMaster = SEQ_REG1;
    labelOthers = IDLE;
  }

  bool finished = false;
  while (!finished) {
    int nextLabel;

    __syncthreads ();

    if (threadIdx.x == MASTER) {
      nextLabel = labMaster;
    else
      nextLabel = labOthers;

    switch (labelNext) {
    case IDLE:
      break;

    case SEQ_REG1:
      <code gen for seq region 1>
      <assign labMaster and labOthers>
      break;

    case PAR_REG1:
      <code gen for par region 1>
      if (threadIdx.x == MASTER)
        <assign labMaster and labOthers>
      break;

    <..other cases.>
    }
}
```

Figure 8: Scheme for the solution based on a control-loop enclosing the inspection/execution scheme.



```
if (++a[omp_get_team_num() &&
++b[omp_get_team_num())
{
  con_gbl = 0;

#parallel for
for (k = ...) { // par reg 1.1
  cond[k] = ... ;

for (k = 0 ; k < N ; k++)
  cond_gbl = cond_gbl && cond[k];

if (cond_gbl)
{

#parallel for
for (l = ...) { // par reg 1.2
}
```

Figure 9: Graphical representation of how the control-loop scheme with inspector/executor is applied to the example of Figure 7.

the switch cases. For instance, in the first vertical box the team master is executing **S1** while all other threads in the team are in the **IDLE** case. Vertical lines are full when the corresponding thread is actually executing a region of code, dotted when it is waiting for the team master to execute a sequential region. At the end of each vertical box there is a loop labelled with *while* denoting that, after finishing a code region, all threads return back to the while construct

	Host	Device
Model	IBM Power8LE (S824)	NVIDIA Tesla K40m
#Cores/#SMs (#Cores)	16	15 (2880)
Core Freq.	4GHz	0.88GHz
Memory	L1=64KB, L2=512KB, L3=8192KB Main=512GB	L1=16KB, L2=1.5MB, Global=12GB
Compiler	gcc 4.8.2	nvcc 5.5
Options		-arch sm_35

Table 1: Specifications of the experimental platform configuration.

```
#pragma omp target map(to:a,b), map(c)
{
  #pragma omp teams num_teams(N), thread_limit(M)
  {
    #pragma omp distribute
    for (int i0 = 0 ; i0 < VECTOR_SIZE ;
      i0 += blockSize)
    {
      #pragma omp parallel for
      for (int i = i0 ;
           i < min (i0+blockSize, VECTOR_SIZE) ;
           i++)
        c[i] += a[i] + b[i];
    }
  }
}
```

Figure 10: Vector-add example in OpenMP using teams in a target region.

for new work assignment. Following this, a horizontal line marked with *sync()* denotes a team synchronization. Notice that non-master threads are only executing the parallel regions, while waiting on the synchronization in sequential regions.

5. PERFORMANCE ANALYSIS

This section reports on the performance of the thread coordination strategies described in the previous section. Experiments were performed on a CPU+GPU system with an IBM$^{®}$ Power$^{®}$ 8 (S824) CPU and an Nvidia Tesla$^{®}$ K40m GPU. Table 1 lists the features of the platform plus the compilers used. Apart from those specified in the table, standard options were used in both host and device compilation. Our experiments are conducted by implementing a CUDA program that mimics what the LLVM compiler will generate for a certain input OpenMP program. We either use OpenMP or CUDA to describe the program subject of an experiment, depending on the aspects that we want to highlight. There is no use of shared memory for application data. Iteration variables (e.g. loop bounds) are mapped onto shared memory when a team/block master exclusively calculates data to be shared in the team.

The first study that we report is related to analyzing the difference in performance between two implementations of a vector-add benchmark, respectively using dynamic parallelism and the *if-master* scheme. The OpenMP program that describes the benchmark is shown in Figure 10. This benchmark uses teams to distribute the vector-add operations in blocks of *blockSize* elements each. It is inspired by example 54.2c of the OpenMP example document [12]. The iteration space is distributed onto the available blocks and each team executes in parallel the iteration block. As discussed in Section 3.1, the distribute directive only enables static scheduling of iterations to teams. We also implemented the innermost parallel for using static scheduling.

The implementation based on dynamic parallelism is based on three kernels. The first kernel starts the parallel region and it contains one block and one thread in it. The thread launches a further kernel with N blocks each with a single thread in them. This implements the spawning of the league of teams where we only activate the master thread in each team (team master-only kernel). The distribute directive implementation is executed by the team masters in their assigned blocks. These successively launch a further kernel with 1 block and M threads in each block (innermost kernel). This last level of threading executes the parallel for directive and its body. All threads in a block cooperate for the execution of the innermost loop. At the end of the innermost kernel invocation, we return to the team master-only kernel, where team masters are waiting for the innermost kernel to finish execution. This implements both the for directive semantics (implicit barrier at the end of directive), but it also enables switching back control to team masters.

The thread coordination implementation is based on launching N blocks and M threads in a single kernel call. The only sequential part of the kernel is represented by the implementation of the distribute directive: the scheduling of the outermost loop iterations across blocks/teams is only executed by team masters. We guard the scheduling procedure as shown in the previous section (if-master strategy) and we successively synchronize the threads in the same block using a *_syncthreads()* call. At the end of each parallel for directive, we also add a block synchronization call, following the semantics of the for directive.

Figure 11 shows the performance obtained for the two implementations by varying the parameters N, and M. We show the case for vector size equal to 1024, where similar results were obtained for other sizes. Each test is executed 10^4 times to mask the overhead associated with the initial kernel call. Reported performance numbers are average of the total execution time over the number of tests.

Figure 11 contains two charts. The left chart shows the execution time obtained for the benchmark when using dynamic parallelism. The right one reports execution time when using the if-master strategy. The x-axis shows the number of blocks used, while the z-axis shows the number of threads in each block. The y-axis follows the execution time in milliseconds. Different shades of grey are used for different ranges of execution time. The legend shows the values associated with each iso-performance curve.

Performance numbers show that the best time obtained with dynamic parallelism is above 100 milliseconds, while the worst performance obtained with the if-master coordination scheme is below 20 milliseconds. On average, there is an order of magnitude of difference between the two schemes.

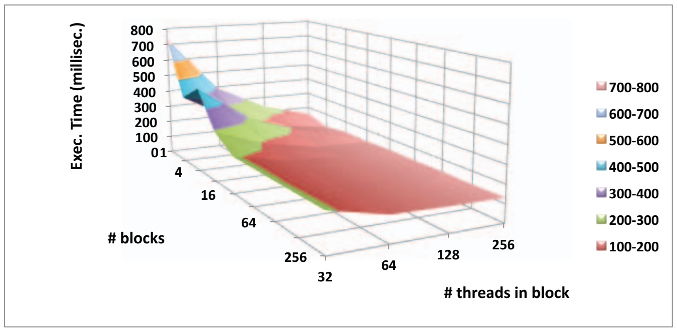
```
__global__ void microbench (..)
{
  for (int it = 0 ; it < num_tests ; it++)
  {
    // only team masters
    for (int i = 0 ; i < L ; i++)
      for (int j = 0 ; j < K ; j++)
        <sequential beta pre-calculation>
    __syncthreads ();

    // distribution across blocks and threads
    int lb, ub;
    get_my_bounds (0, N, lb, ub);
    for (int i = lb ; i < ub ; i++)
      c[i] += beta*a[i] + b[i];

    // only team masters
    for (int i = 0 ; i < L ; i++)
      for (int j = 0 ; j < K ; j++)
        <sequential beta post-calculation>
    __syncthreads ();
  }
}
```

Figure 12: Skeleton of micro-benchmark used for performance analysis.

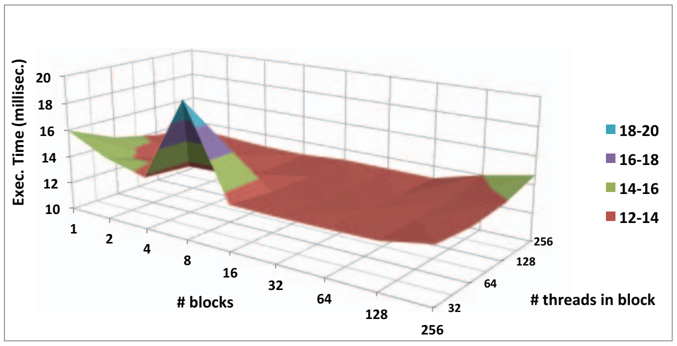
As discussed in the previous section, the if-master coordination scheme performs better than using dynamic parallelism. While this was expected, we further discussed that the implementation of the if-master scheme requires complex compiler re-structuring. Our alternative solution, based on the control-loop with inspection/execution, is easier to integrate in the compiler. In this section we prove that its performance is similar to the one provided by the if-master strategy.

For the vector-add example, the difference in performance between the if-master scheme and the control-loop with inspector/executor is not relevant, because of the small size of the team-sequential part. To better study the difference in performance between the two versions, we developed a micro-benchmark that exposes the characteristics of sequential and parallel regions in a configurable manner. The skeleton of the microbenchmark is shown in Figure 12. The calculation is a weighted vector-vector addition, and it includes: an initial team-sequential region; a parallel region that performs the vector addition; and a final sequential region that reduces the results of the parallel region. All threads synchronize their execution while switching regions. The L and K variables can be set by the testing environment to tune the amount of work performed in the sequential regions, while N is used to tune the parallel region. In the experiments, each test is run 10^6 times (*num_tests* variable).

Results are shown in Table 2. The table shows the percentage difference in performance between the if-master scheme and the control-loop one for a representative choice of configuration parameters ((N = 16384, K = 100, L = 1). For all configurations, the if-master scheme offers better performance, where the maximum percentage difference between the two schemes is lower than 5%. For the configuration parameters used in the tables, the largest difference between the two schemes was recorded for the case of 256 blocks each with 256 threads. The actual execution time of the if-master scheme is 122.021 milliseconds, while the control-loop scheme executed in 126.947 milliseconds. Similar differences



(a)



(b)

Figure 11: Execution time of thread coordination schemes for the first benchmark (see Figure 10). Results are for the the dynamic parallelism scheme (left) and the if-master one (right).

were obtained for different vector sizes (parameter N), with the maximum percentage increase in run-time of 4.79%.

To understand the difference in performance between the two strategies, we analyzed the output produced by the nvcc compiler. For the switch construct in the control-loop strategy, the compiler generates a sequence of cascaded compare instructions with the label variable and one of the values of the switch cases as input. Each compare instruction is followed by a conditional branch: if the label variable is equal to the case value, the control-flow branches to the basic block implementing the switch case block. For the studied benchmark the compiler generated five compare plus branch instructions.

For sequential regions, all warps that do not contain the master will evaluate the first compare to true and jump into the *IDLE* label. The warp with the master will scan, in the worst case, the entire set of compares for the master thread to reach the one evaluating to true. This means that we pay an entire round of compares in the worse case, and half of it in the average case, because all warps will be waiting for the master to execute the sequential region. Similarly, when executing a parallel region, all warps will have to execute the same number of compares, which is equal to the total number of labels in the worst case scenario.

An alternative implementation of this is based on setting up a table of labels and by indexing it using switch labels. The implementation of the switch would consist in a load from the table and a branch to the read value. This would avoid the execution of the if-statements discussed above. This scheme can be implemented in the LLVM compiler when targeting PTX. We will explore this scheme as future work.

6. IMPLICATIONS FOR CLANG/LLVM

The code generation proposals presented in this paper aim at minimal changes to the design of Clang and its current OpenMP implementations. The required modifications in Clang are solely contained in the OpenMP implementation and do not require changes in any other Clang code generation component.

The OpenMP implementation is shared by any target supported by LLVM. However, some specialization of the OpenMP code generation features is required for a given target. In the previous sections we discussed that a team region may spawn multiple blocks each containing multiple threads, but only team masters are active in the team region, until a parallel region is encountered. As described in the proposal, forking all threads, even the ones that will not do any immediate computation (non-masters), is remarkably more efficient for GPUs. However, this is a GPU-specific optimization: making several threads active without any real computation on a different target platform may be excessive. Therefore, the code generation of the threads forking, as well as the corresponding RTL calls, need to be overloaded if the target is a GPU. The default implementation should only be used for those targets that can efficiently fork several threads from the master thread of a team.

For this purpose, we implemented a facility that permits overloading calls to the OpenMP RTL, where we can specialize the implementation of the calls depending on the target platform. This is used by the implementation of the teams directive, as described above, when targeting the NVPTX backend (NVIDIA GPU assembly language). Other specializations include code inlining of critical OpenMP functionalities.

Another modification relates with the code insertion points for basic blocks that are going to be executed either by the master or all threads in a team. This is useful for implementing the switch construct in the control-loop solution. New basic blocks have to be created each time the code generation switches from a single thread (master) to multiple threads and vice-versa, where each basic block implements a case of the switch construct. This requires extending the initial and final code generation of parallel regions so that the appropriate control flow variables (jump table labels) are updated when the parallel region is contained in a target region. As we have seen in the performance analysis section, this would prevent the creation of a sequence of if-statements to implement the switch construct.

To implement this, we extend the Clang code generation with "hook" RTL functions. These are properly overloaded for NVIDIA GPU targets, while they do not produce any action by default, to support other targets. This means that for any target other than NVIDIA GPUs there will be no corresponding statement in the produced code.

					# blocks					
		1	2	4	8	16	32	64	128	256
# threads	32	1.1456%	0.4656%	0.6406%	0.8377%	1.0458%	2.3288%	2.4617%	2.2883%	2.4351%
	64	0.9367%	1.4086%	1.5228%	1.5741%	1.5936%	2.1161%	2.2796%	2.3831%	2.1455%
	128	1.3590%	1.2749%	1.3364%	1.5621%	1.6310%	2.8337%	2.5220%	2.4574%	3.6567%
	256	1.3873%	1.1753%	1.3644%	1.2606%	1.6260%	3.0446%	2.3633%	3.6675%	**4.0370%**

Table 2: Performance results for the second benchmark (see Figure 12). This table shows the difference in percentage of execution time of the if-master strategy and the control loop one (N = 16384, K = 100, L = 1). The largest difference is highlighted.

7. CONCLUSION

This paper describes thread coordination schemes for the efficient implementation of OpenMP target regions on GPUs. The studied issue is to efficiently support target regions that contain interspersed and nested sequential and parallel regions. An initial scheme based on dynamic parallelism is shown to deliver one order of magnitude worse performance than a scheme based on launching a fully parallel kernel and by guarding sequential regions with if-statements. We discuss how this if-master scheme is hard to implement in the target compilation toolchain, i.e. in Clang/LLVM. A main contribution of this paper is to show that the if-master scheme is disruptive to potentially all C/C++ constructs and it does not follow Clang modularity.

A further scheme is described based on the concept of control-loop and inspection/execution strategy, which is easy to integrate in a modular fashion in Clang and that delivers performance results similar to the if-master scheme. We report a maximum performance difference smaller than 5% between the if-master and the control-loop scheme. We also give hints on how the compiler can be modified to optimize the control-loop scheme implementation on GPUs.

8. REFERENCES

[1] C. Bertolli, A. Betts, N. Loriant, G. Mudalige, D. Radford, D. Ham, M. Giles, and P. Kelly. Compiler optimizations for industrial unstructured mesh cfd applications on gpus. In H. Kasahara and K. Kimura, editors, *Languages and Compilers for Parallel Computing*, volume 7760 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2013.

[2] D. Caballero, A. Duran, and X. Martorell. An OpenMP barrier using SIMD instructions for Intel Xeon Phi^{TM} coprocessor. In A. P. Rendell, B. Chapman, and M. S. Müller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 99–113. Springer, 2013.

[3] Llvm support for the openmp language. http://openmp.llvm.org/.

[4] cuBLAS. http://docs.nvidia.com/cuda/cublas.

[5] CUDA C Programming Guide version 6.0. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[6] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[7] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 311–320, New York, NY, USA, 2012. ACM.

[8] C. Lattner. Llvm and clang: Advancing compiler technology. Keynote Talk, FOSDEM 2011: Free and Open Source Developers European Meeting. http://llvm.org/pubs/2011-02-FOSDEM-LLVMAndClang.pdf.

[9] C. Liao, Y. Yan, B. R. de Supinski, D. Quinlan, and B. Chapman. Early experiences with the openmp accelerator model. In A. P. Rendell, B. Chapman, and M. S. Müller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 84–98. Springer Berlin Heidelberg, 2013.

[10] OpenMP Application Program Interface. http://www.openmp.org/.

[11] OpenACC - Directives for Accelerators. http://www.openacc-standard.org/.

[12] OpenMP Application Program Interface Examples version 4.0. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[13] E. Stotzer, A. Jayaraj, M. Ali, A. Friedmann, G. Mitra, A. Rendell, and I. Lintault. OpenMP on the low-power TI keystone II ARM/DSP system-on-chip. In A. Rendell, B. Chapman, and M. S. Müller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 114–127. Springer, 2013.

[14] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck. Sparse tiling for stationary iterative methods. *Int. J. High Perform. Comput. Appl.*, 18(1):95–113, Feb. 2004.

[15] 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)*, 2014.