Integrating GPU Support for OpenMP Offloading Directives into Clang

Carlo Bertolli, Samuel F. Antao, Gheorghe-Teodor Bercea, 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.
{cbertol,sfantao,gbercea,acjacob,alexe,chentong}@us.ibm.com
{zsura,hsung,grokos,dappelh,caohmin}@us.ibm.com
Department of Computing, Imperial College London,
180 Queen's Gate, London, SW7 2AZ, United Kingdom
gheorghe-teodor.bercea08@imperial.ac.uk

ABSTRACT

The LLVM community is currently developing OpenMP 4.1 support, consisting of software improvements for Clang and new runtime libraries. OpenMP 4.1 includes offloading constructs that permit execution of user selected regions on generic devices, external to the main host processor. This paper describes our ongoing work towards delivering support for OpenMP offloading constructs for the OpenPower system into the LLVM compiler infrastructure. We previously introduced a design for a control loop scheme necessary to implement the OpenMP generic offloading model on NVIDIA GPUs. In this paper we show how we integrated the complexity of the control loop into Clang by limiting its support to OpenMP-related functionality. We also synthetically report the results of performance analysis on benchmarks and a complex application kernel. We show an optimization in the Clang code generation scheme for specific code patterns, alternative to the control loop, which delivers improved performance.

1. INTRODUCTION

The LLVM[1]compiler infrastructure is the leading solution as open source compilation toolchain for next generation High Performance systems. There are several recent contributions that show improvements in both the quality of the generated code [4] and its support for accelerators [1] [14] and Enterprise systems [20]. LLVM is the selected framework for a new system that IBM is building under the CORAL Department of Energy framework program [9], in collaboration with NVIDIA and Mellanox. The new CORAL system will include nodes based on OpenPower by coupling Power processing cores with NVIDIA Graphics Processing Units (GPUs).

LLVM-HPC2015 November 15-20, 2015, Austin, Texas USA

DOI: http://dx.doi.org/10.1145/2833157.2833161.

These heterogeneous systems are especially attractive from a performance viewpoint as they permit users to partition their applications and execute each subcomponent using the computing resources that better fit its requirements. At the same time they pose a critical program portability issue: to obtain the best performance possible users are typically required to specialize their applications using a native programming language, like CUDA C/C++ for NVIDIA GPUs. This is an issue in terms of performance portability as an application implemented in CUDA C/C++ is, by definition, not generally portable to different systems. The users would be required to provide many different versions of their applications, one for each target architecture. Furthermore, there is also the productivity limiting factor of forcing the user to master not only the algorithmic details of applications, but also several different programming languages.

This issue is tackled by the OpenMP [17] directive-based language, starting with version 4.0, with the introduction of new offloading constructs. Offloading constructs specify regions of code and data environments that are to be mapped to a generic external device. From a programming viewpoint, there is no distinction between the programming abstractions that can be used inside or outside of offloading regions. It is responsibility of the OpenMP software toolchain to optimize this general programming paradigm for each specific device type. This is a challenge for NVIDIA GPUs: compiler and runtime are required to provide optimal code synthesis and efficient support for constructs that are not derived from GPU-specific hardware characteristics. Critical features for an OpenMP implementation, including dynamically spawning threads or recruiting them, have a known optimized implementation on a CPU. On a GPU and under certain conditions, support for these features can only be implemented with large overheads that cannot be easily masked.

Our work is focused on delivering an efficient implementation of the OpenMP offloading model for OpenPower systems into the LLVM framework. In a previous paper [6] we

[1]The following symbols are registered trademark: LLVM®, IBM®, CORAL®, NVIDIA®, Mellanox®, OpenPower®, CUDA®, OpenMP®, Clang®, Intel®, Xeon Phi®, Texas

described the design of a thread coordination scheme, called the *control loop*, for the Clang/LLVM compilation toolchain. It has the following characteristics:

- It avoids the use of a potentially expensive implementation mechanism available in CUDA C/C++, i.e. dynamic parallelism, which enables the dynamic spawning of threads while executing on the device. Under the control loop scheme, a large amount of threads is spawned at the beginning of an offloading construct implementation and no further dynamic spawning is allowed. As all threads are immediately and actively executing an offloading region, we need to coordinate their execution: a master thread per CUDA block is used to guide all other threads through OpenMP regions during offloading; the compiler generates code to automatically exclude unnecessary additional threads from executing certain regions (e.g. sequential); the scheme is designed to follow the strict programming requirements for GPUs, related to the ability of proving thread convergence across OpenMP regions.

- It is designed to be easily and modularly integrated into the Clang code generation module. The control loop scheme is designed in such a way that its implementation can be limited to emission routines of Clang that are specifically related to OpenMP constructs. That is, all other OpenMP and C/C++ constructs are generated as we would normally do when targeting a CPU.

In this paper we focus on two significant steps towards the full integration of the control loop scheme into Clang:

- We show a design for integration of OpenMP 4.0 offloading constructs into Clang. We describe an implementation for the NVIDIA GPU target and based on the control loop. Most constructs are implemented and available in an extended Clang repository [8].

- We conducted extensive performance analysis on several benchmarks and a full-fledged application. We describe overheads incurred by the control loop scheme, characterizing the inherent trade-off between: modular integration in Clang; constraints given by the target GPU execution model; overheads incurred by applications.

- We describe enhancements in code generation derived from performance analysis, to support relevant OpenMP constructs and their combinations. We show cases in which our implementation can match the performance of CUDA-based implementations and we describe issues that are still preventing a full performance matching in remaining cases.

Through the description of a complex code generation scheme for GPUs we aim at disseminating concepts of our design to the LLVM community with the goal of improving them through collaborations.

This paper is organized as following: Section 2 describes related work in Clang/LLVM for high-performance computing. Section 3 introduces the control loop scheme and shows how this is modularly integrated into Clang. We give specific details on where code generation routines are extended. Section 4 describes the alternative code generation for patterns based on combined constructs and its integration into Clang. Section 5 shows results of experiments on three kernels using the control loop scheme and the alternative scheme. We discuss the main performance metrics that characterize the results we obtained. Finally, Section 6 gives the conclusion of the paper.

Instruments®, OpenACC®, Github®, NVIDIA Kepler®, Linux®, Ubuntu®.

2. RELATED WORK

There are several contributions to support High Performance scientific computing in the Clang/LLVM toolchain. A homogeneous programming layer called PACXX based on C++14, which shares the same goal as our work, is described in [11]. The authors describe a compiler for PACXX and show preliminary results for a benchmark, comparing performance of the same program implemented in different languages (OpenCL, CUDA C/C++) and executed on different host acceleration platforms. PACXX programs are based on C++14 abstractions including STL containers and concurrency constructs. OpenMP allows more control over parallelism and how this is executed and mapped to resources. For instance, memory management can be completely controlled by the user in OpenMP, while it is implicitly managed by the compiler and runtime in PACXX. Another abstraction with similar goals to PACXX is SYCL [7].

OpenMP 4.0 offloading constructs are implemented for several kinds of devices. Experiments for OpenMP on Intel Xeon Phi are presented in [2]. Further examples of devices supporting OpenMP are a DSP accelerator by Texas Instruments [18], and mobile devices [19].

As for OpenMP, OpenACC targets performance portability across diverse acceleration architectures as main goal. A similar coordination scheme than the one we implemented for OpenMP in Clang is necessary to support OpenACC: execution starts in *gang-redundant* mode, where only one lane in one worker in each gang is executing. This corresponds to execution within a teams construct in OpenMP, and outside of parallel or simd regions. Execution transitions into *gang-partitioned* mode when a loop region, marked for worker- or vector-level parallelism, is encountered. This corresponds to execution in a parallel or a simd region in OpenMP. We will analyze in future work the solutions to this coordination issue on NVIDIA GPUs in commercial and open source compilers.

An alternative approach to programming GPUs is described in [3]. CUDA code is automatically generated from C/C++ programs by a compiler based on the polyhedral model. Experimental results show that, by applying loop unrolling and taking advantage of shared and constant GPU memory, near hand-tuned CUDA performance can be obtained and in some cases overcome. The limitation of this approach is that it only applies to loop nests that belong to the category that can be modeled through the polyhedral abstractions. This is not true, for instance, in the case of loop nests using arrays as indirection pointers, as it is the case of the LULESH kernel studied in the experimental section (see 5).

3. INTEGRATION OF CONTROL LOOP

In this section we give a brief description of OpenMP of-

floading directives and we motivate the need for the control loop scheme. We sketch the control loop scheme to coordinate GPU threads when executing an OpenMP target region. We thus show how this scheme can be integrated into the OpenMP implementation modules of Clang.

3.1 OpenMP Offloading Constructs

OpenMP 4.0 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 data device 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 may include allocating and transferring data between host and device and launching a GPU 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 on GPUs naturally maps each team to an independent CUDA block, as these feature the same execution independence requirements as teams: CUDA C/C++ does not expose a native synchronization operation between threads in different blocks. It is important to note that when a teams region is encountered, only one thread (team master) actually starts execution. The other threads in each team are only activated when a pragma *parallel* is encountered.

A pragma *distribute* is associated to a loop and can be used to partition it into chunks which are assigned 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 the definition of a function or a global variable that is called/used within a target region using a special *declare target* pragma.

3.2 Sketch of Control Loop Scheme

The programming model of NVIDIA GPUs includes restrictions that impact the OpenMP implementation. More specifically: threads within the same warp execute in lockstep and expose limited means of expressing data- or control-flow dependencies; block synchronizations need to be syntactically unique for all threads in the block. In [6] 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 the related kernel is launched. In the 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.

```
1  #pragma omp target teams
2  {
3    double 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
2  void tgt_function(..) {
3    int ControlIdx = 0;
4    __shared__ int ControlState[2];
5    int nextState = Seq1;
6
7    while (!finished) {
8      switch(nextState) {
9      case Seq1:
10       if (threadIdx.x != 0) break;
11       kmpc_kernel_init();
12       //<alloca and assign pi variable,
             share within team>
13       ControlState[ControlIdx] = Par1;
14       break;
15     case Par1:
16       int CUDAThreadsInParallel =
             kmpc_parallel_start();
17       if(threadIdx.x >=
             CUDAThreadsInParallel) break;
18      // <implement loop>
19       if (threadIdx.x == 0)
20         ControlState[ControlIdx] = Seq2;
21       kmpc_parallel_end();
22       break;
23     case Seq2:
24       if (threadIdx.x != 0) break;
25       //<close region>
26       finished = true;
27       break;
28     }
29     // all threads re-converge here
30     __syncthreads();
31     nextState = ControlState[ControlIdx];
32     ControlIdx = ControlIdx^0x1;
33   }
34   return;
35 }
```

Figure 1: Input OpenMP program with target region and corresponding implementation using the control loop scheme.

In this section we revisit the control loop and we go further to introduce a full-fledged scheme that enables the implementation and optimization of most OpenMP constructs in offloading regions.

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. At the beginning of each case code is generated to exclude unnecessary threads depending on the region kind and OpenMP clauses associated to it. For instance, sequential regions (lines 9 and 23) are only executed by the team master, while all other threads are waiting at the barrier (__syncthreads() at line 30). Parallel regions (e.g. line 15) are executed by multiple threads. The exact number of threads executing the parallel region depends on many factors, including the value of the num_threads clause of the parallel construct, if used. The OpenMP runtime library for the GPU target provides a function (*kmpc_parallel_start*, line 16) that returns a calculated number of CUDA threads that

are required for the parallel region. All threads with identifier larger than the required number jump forward to the synchronization point (line 17). A similar *kmpc_parallel_end* call is provided by the runtime and invoked at the end of the parallel region (line 21). These two functions, start and end parallel, are used by the runtime to create and delete a new OpenMP context while executing the parallel region. This is necessary to implement basic OpenMP functionalities. For instance, a call to *omp_get_num_threads()* depends on the context: if called outside the parallel case, it returns one; if inside the parallel case, it returns the number of active OpenMP threads.

Note that there is a single barrier in the whole kernel and that idle threads wait on the same barrier that will be used by the active threads before progressing work. The control state variable is an array of length two. This is used by the master thread to store the identifier of the next OpenMP region to execute and, as such, is allocated into shared memory, as this is accessible by and private to all threads in the same CUDA block. The threads alternate reading and writing between the two positions to prevent data races between a thread that is late to read a certain current state and the master that is attempting to write the next one.

This scheme is easy to implement in a modular way in Clang, as we show below. However, it complicates the produced code with a multi-target switch and a while loop using a variable for the condition expression. This impacts register allocation negatively, as we study in the performance analysis (see Section 5).

3.3 Integration into Clang

Support for OpenMP in Clang has been developed in two separate code bases: in [8](Github repository) and [13] (trunk repository). The initial prototype of the different OpenMP offloading features have been developed in [8] and are now being gradually ported to [13]. At the time this paper is being written, GPU offloading support is only available in [8]. Notwithstanding the two code bases being different (rewriting of the implementation is necessary as part of the reviewing process), the way the code is organized is similar and considerations about the integration of OpenMP support into Clang are valid in both cases. The compilation toolchain is shown in Figure 2. Input C/C++ code using OpenMP is transformed into LLVM IR by Clang. The backend of Clang, which generates LLVM IR code from its internal Abstract Syntax Tree representation, is responsible for implementing OpenMP constructs. It partially transforms code regions that are annotated with OpenMP pragmas and inserts calls to the OpenMP runtime library. LLVM transforms the output of Clang into the target assembly language. For NVIDIA GPU targets, LLVM generates PTX language, which is successively passed to ptxas and nvlink as shown in the figure. These are tools that are provided by NVIDIA and transform PTX programs into low-level native GPU assembly, called SASS, which is packed into CUDA binary ELF sections (also called cubin files).

The current implementation of OpenMP in Clang is mainly confined to two compilation units: CGStmtOpenMP.cpp and CGOpenMPRuntime.cpp. These are part of the code generation phase of Clang:

- CGStmtOpenMP.cpp exposes an Emit* function for almost every OpenMP construct/clause. These functions are grouped together when appropriate, e.g. when dealing with combined directives. These functions can access the OpenMP library runtime calls through an OpenMPRuntime object contained in the CodeGenModule object associated to the current function.

- CGOpenMPRuntime.cpp implements the OpenMPRuntime class used to generate calls to the OpenMP runtime functions. In some more complex cases, it also generates specific code to support the runtime calls, e.g. to implement the task construct. The OpenMPRuntime class implementation is currently the same for all target architectures.

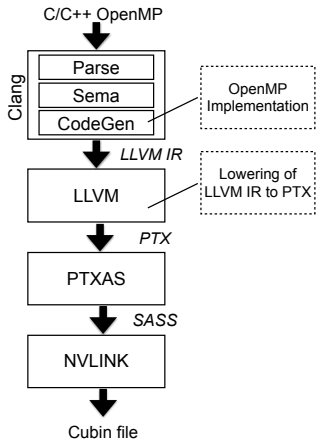

Figure 2: Graphical representation of Clang/LLVM compilation toolchain for OpenMP target regions on NVIDIA GPUs.

As we described above, generating the control loop scheme requires support at the code generation level, apart from target-dependent code specialization of the runtime library. Our goal is to minimize the amount of target-dependent changes to the OpenMP support in Clang and completely avoid code specialization for generic C/C++ constructs, thus implementing a modular extension of Clang. In other words, code generation logic for C/C++ constructs should not be dependent on the fact that we are generating code for an OpenMP target region for the NVPTX backend.

As motivated in our previous paper [6], modular integration cannot be achieved when adopting simpler thread control schemes like the so-called *if-master*. This would require alternative code generation for every C/C++ statement based on the target. Unlike this, the control loop implementation only requires a code-generation interception mechanism when entering and exiting certain OpenMP constructs, including target and parallel.

To further make GPU code specialization less intrusive, we make it possible to extend the OpenMPRuntime class. For NVIDIA GPU targets we created a new OpenMPRuntimeNVPTX class that extends OpenMPRuntime. We added new virtual methods in the base class, as described below, that are called in target mode from the CodeGenFunction methods at appropriate sites. The OpenMPRuntimeNVPTX class implements specialized versions of those methods which enable target-specific complex code generation. The same methods are left empty in the base class (generic

```
1  void CodeGenFunction::EmitOMPDirectiveWithTarget
       (..) {
2    // first create the target function and codegen
         for it
3    <create the function declaration with type, plus
         handle clauses>
4    ..
5    {
6      CGF.StartFunction(FD, getContext().VoidTy, Fn,
           FI, FnArgs, SourceLocation());
7
8      OMPRegion.StartRegion();
9
10     if (isTargetMode)
11       CGM.getOpenMPRuntime().EnterTarget(
12         CS->getLocStart(), CGF, FD->getName(),
13         DKind, SKind, S);
14     switch (SKind) {
15     default:
16       llvm_unreachable("Unexpected target
             directive subkind");
17       break;
18     case OMPD_teams:
19       CGF.EmitOMPDirectiveWithTeams(
20         OMPD_target_teams, OMPD_unknown, S);
21       break;
22     case OMPD_teams_distribute:
23       CGF.EmitOMPDirectiveWithTeams(
24         OMPD_target_teams_distribute,
25         OMPD_distribute, S);
26       break;
27      // etc..
28       }
29       if (isTargetMode)
30         CGM.getOpenMPRuntime().ExitTarget(
31           CS->getLocStart(), CGF, false,
32           FD->getName(), SKind);
33       CGF.FinishFunction();
34     }
35   }
36   ..
37   // second, invoke the runtime for offloading
         using
38   // the target function
39   <set up parameters to tgt_target_teams call>
40   llvm::Value *Offload = Builder.CreateCall(
         TgtTargetFn, TgtArgs,
41         "offloadret");
42   <if offload fails, execute on host by calling
         host target function>
43 }
```

Figure 3: Scheme of implementation of OpenMP pragma *target* where we highlight the new function calls to the OpenMP runtime object. These are specialized for the NVIDIA GPU target.

OpenMP target). This mechanism can be used to support specialization of the OpenMPRuntime class for different targets than the one considered in this paper. Figure 3 contains a scheme of the implementation of target regions in the Github version of the compiler, where we highlight new calls to the OpenMPRuntimeNVPTX class. For the target construct, two new calls *EnterTarget* and *ExitTarget* are the only extension needed to the pragma *target* code generation to implement the control loop. Further calls are required for relevant pragmas (e.g. *parallel*).

The new specialized OpenMPRuntimeNVPTX class offers the following methods to implement the control loop scheme:

EnterTarget This function is called at the beginning of a target function code generation phase (line 11), after the function type and prologue are generated. It generates the control loop scheme and prepares the function body for emission of the target region body. With reference to Figure 1: it creates new data structures to allow the team master thread to guide the other threads through the OpenMP regions (ControlState, etc..); it sets up a while-loop with a local condition variable that is set by all threads in the last OpenMP region case. The alternative is to directly use a return statement inside a final switch case. Our original solution (using a condition variable over the while-loop) is more robust when lower compilation passes (ptxas) require proving thread convergence. It creates a switch construct with a final case and an initial sequential case. The initial sequential case contains an if-then-else statement that provokes all threads with a thread identifier different than the team master (0) to break out of the case. As this is a sequential case, only the team master thread needs to execute it. The initial sequential case is set as the first one executed by all threads when entering the CUDA kernel. The last statements inside the while-loop is where all threads converge when executing any break-statement inside switch cases. We generate a call to the team/block synchronization primitive (__syncthreads() call in CUDA C/C++ terminology). We also generate code to select the next switch case to execute, after the synchronization. Before returning control to the caller, we set the insert point for subsequent code generation to the first sequential basic block that is only executed by the team master.

ExitTarget This function is called at the end of code generation for the body of a target region, as shown in Figure 3. It fills up the finished case by having all threads set the local thread condition variable to true. At this point the entire target region code has been generated and Clang can perform post-generation code analysis and parameter configuration. For instance, Clang could select a different thread scheduling strategy based on the OpenMP directives and C/C++ statements that were encountered in a target region. This is relevant to improve support for nested parallelism and pragma *simd*, as well as OpenMP tasks.

EnterParallel As described in Figure 1, parallel regions require Clang to generate new switch cases. Assume that we are generating code for a team-master sequential region. When Clang encounters a pragma *parallel* in the AST, it invokes a function in CGStmtOpenMP.cpp to perform the appropriate code generation. The OpenMP implementation for standard targets invokes the *kmpc_fork_call* routine to recruit threads for executing the region. In our scheme and unlike more common OpenMP implementations, all threads are already actively executing in the control loop with the team master. They are being excluded by the thread control mechanism. EnterParallel creates a new initial switch case for the parallel region. Before closing the current sequential switch case, it generates code for the team master to set the next label to the new parallel case and issues a break for the team master that will provoke it to join all other CUDA threads at the barrier point. Clang moves the insert point to the new parallel case and it first excludes all unnecessary threads. In a relevant example, the OpenMP offloading runtime may have started 1024 CUDA threads, but a *num_threads* clause on *par-*

allel only requires 32 threads. Clang generates code to check the thread identifier against the calculated number of OpenMP threads. For all excluded threads, it uses a out-of-case break statement which will lead to the synchronization point. All other threads that are required to execute will remain in the current parallel switch case. Before returning control to the pragma *parallel* code emission, Clang sets the insert point to the new basic block for non-excluded threads in the parallel region.

ExitParallel At the end of a parallel region Clang will issue instructions that provoke returning to a sequential (team master only) execution. At runtime some threads will be waiting at the barrier point because they were excluded from executing the parallel region. Some others are instead executing the instructions generated so far in the parallel basic blocks that implement the parallel switch case. Clang generates a new sequential switch case and it provokes the team master to set the next switch label to this new case, while all other threads that were not excluded will break to the barrier point. Clang sets the insert point to the first basic block of the new sequential switch case and generates code that excludes all threads except the team master from executing the region, using an if-statement with a break.

EmitOMPBarrier When a pragma *barrier* is encountered in the standard OpenMP host implementation, we emit a call to the runtime barrier routine. In the control loop we can to map it to the single synchronization point. Clang closes the current parallel region which provokes all threads to join at the barrier point. The master sets as next switch case a new parallel case that will contain the code generated for the remaining code after a barrier, with a similar thread exclusion mechanism as discussed above. This specialization is expressed as an alternative implementation of EmitOMPBarrier.

The complexity of the behavior described here can be more easily synthesized in a single function that permits switching between control loop cases based on the following information: current region type (sequential or parallel), next region type (sequential or parallel). Nested parallelism can be easily intercepted in the Enter/ExitParallel regions by keeping track of the nested parallel regions encountered during code generation. If a nested parallel region is detected, the current compiler implementation provokes the serialization of the region through appropriate calls to the runtime library. Explicit barriers can be generated by using the same single function and instructing it that both the current and next regions are of type parallel.

A limitation of this scheme is that we are currently required to serialize all parallel regions inside function body calls invoked from within a target region. This is necessary in the general case, when the function definition is in a different compilation unit from the target region. The compiler is not able to infer whether the parallel region is a nested one, as it would need accessing both caller and callee regions. The cons of this approach is that non-nested parallel regions in function bodies, which could be in principle treated as fully parallel, are instead sequentialized by default.

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

Figure 4: OpenMP program using combined target directives.

4. AN ALTERNATIVE TO CONTROL LOOP FOR A COMBINED CONSTRUCT

We have seen that the control loop scheme introduces a skeleton around the target function body and scatters it among different cases in the switch statement. In general, the generated code is harder to process from the lower compilation passes: more control-flow instructions are used; a global state must be kept across the entire target body. As we highlight in the performance section 5, this results in a worse result for register allocation, which in turn limits the amount of GPU occupancy that can be achieved by kernel executions, thus lowering performance.

This section describes an alternative code generation scheme that can be used for a combined construct directive with strict constraints on the target region body. This scheme is, however, a common pattern in OpenMP programs that employ target regions. It was originally introduced in [5].

4.1 Code Synthesis for Combined Construct

The improved scheme is based on the following observations on OpenMP target regions:

- A team region contains sequential 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.

- Function calls are allowed in target regions and they can contain further OpenMP pragmas. The compiler may not be able to access a function definition that is called 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. It is possible to characterize a subset of OpenMP programs that are free of these properties. Consider the simple program of Figure 4. The for-loop is first blocked and scheduled to different teams to implement pragma *distribute*. The blocks are then executed in parallel by all threads in each team to implement pragma *for*. The main characteristics of this program are: (i) there is no team-master only region, except the distribution of loop blocks to team masters; (ii) there is no data sharing within a team, possibly except the lower and upper bounds of the iteration variable of the block assigned to a team; (iii) there are no function calls and, as such, no possible nested OpenMP pragmas.

For (i) we note that distribution could be calculated by all threads in each team in a redundant way. This impacts (ii) because every threads in each team calculates directly the lower and upper bounds of the block of iterations assigned to the team it belongs to. No data sharing of the bounds is thus needed. The static schedule with chunk size equal to one requires each thread in each team to execute the team-assigned block in chunks of one iteration each.

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

Figure 5: Optimized implementation of the program in Fig. 4.

This program can be implemented as shown in Figure 5, where we use a CUDA-like notation. The iteration space is divided into chunks the size of which is equal to the number of CUDA blocks multiplied by the number of threads per block. One sweep of the loop executes one chunk and full parallelism is used where each thread in each block executes one iteration of the chunk. In the implementation there are no calls to the OpenMP runtime, contrary to what is required in the more general case, and no need to coordinate threads across OpenMP regions.

4.2 Integration into Clang

The integration of this alternative code synthesis scheme into Clang is expressed as an extension of the EnterTarget function of the OpenMPRuntimeNVPTX object. There are two phases. Initially, Clang detects if the target construct that is being subject to code generation is a combined one. It checks that all required constructs are used, including the static scheduling with chunk size equal to one. If one or more constructs or clauses are missing from the combined directive, or other clauses are used, the normal control loop implementation is invoked.

When this condition is true we still need to analyze the content of the target region to exclude cases where an external function is invoked or further pragmas are present (e.g. pragma *parallel*). This is done through a pre-processing analysis of the AST representing the target region body. If this check fails, the normal control loop implementation is invoked; otherwise, we enter the code generation phase.

Code generation is trivial. The loop described in Figure 5 is generated and its iteration variable is linked to the original loop body iteration variable. We still need to exclude code generation of further independent constructs of the combined directive, as these would normally be emitted independently in a sequentialized fashion by Clang (lines 14-34 of Figure 3). This is done by emitting directly the original loop body and dropping all further pragmas in the combined directive.

5. PERFORMANCE ANALYSIS

The goal of the OpenMP compiler toolchain for the CORAL system is to match as closely as possible corresponding CUDA-based implementations.

In this section we consider two benchmarks and two kernels drawn from the LULESH [15] proxy application. We show the performance results for two OpenMP 4.0 implementations which we compare against equivalent CUDA C/C++[2] versions. Full performance analysis for LULESH is described in a related work [5]. For all kernels, we use the combination of pragmas *target teams distribute parallel for*, and we add the clause *schedule(static,1)* to pragma *for*.

The first OpenMP implementation is based on the control loop scheme. Scheduling at team level, as required by pragma *distribute*, is performed by team masters. Each team master must then communicate the lower and upper bounds of the portion of global data assigned to the team when encountering a *parallel* directive. This is implemented using shared memory.

The second OpenMP implementation is based on the optimized code synthesis for combined constructs, which we described in Section 4. This version does not make explicit calls to the OpenMP runtime library on the GPU and all threads in all teams redundantly perform the scheduling phase corresponding to the *distribute* pragma. There is no need to share data between threads in the same team in this version.

We run tests on an OpenPower node which includes two Power 8 sockets (model PowerNV 8247-42L) and two NVIDIA Kepler GPUs K40m. The host processor runs Linux Ubuntu distribution, version 14.04.1. For the OpenMP tests we use our compiler based on Clang/LLVM which extends the current trunk repository with new OpenMP constructs. The code is open source for the control loop scheme and available on Github [8]. 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 [16].

For CUDA tests we use nvcc release 7.0, version 7.0.27. The compilation flags are: for Clang *-fopenmp=libomp -O3 -omptargets=nvptx64sm_35-nvidia-linux*; for nvcc *-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 [12], which is only available in binary format and takes as input NVVM IR, an independent merger between LLVM IR version 2 and 3. Our OpenMP compiler relies on LLVM version 3.8 backend for NVPTX, which is instead open source [14]. This may result in different generated PTX code.

5.1 Control Loop Overhead

The first benchmark that we consider performs a vector addition. The input OpenMP and CUDA programs are show in Figure 6. There are two double precision additions in the loop body, plus integer instructions for the implementation of the loop.

The purpose of this simple example is to describe performance metrics when the control loop is the main and only source of overhead. We run tests with varying team sizes and number of teams, which correspond to CUDA block size and number of blocks. We report results for the best independent configuration for the three implementations. We repeat each test one hundred times and we show the best result as reported by the nvprof tool [10].

Table 1 shows results for array size roughly equal to 8 million elements (512*512*32).

The table shows values for the following metrics:

[2]In the remainder of this section all uses of the term "CUDA" refer to CUDA C/C++.

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

```
__global__ void vectorAdd(int n, double *a
     , double *b, double *c)
{
   for (int i = threadIdx.x + blockIdx.x *
        blockDim.x ;
   i < n; i+= blockDim.x * gridDim.x)
      a[i] += b[i] + c[i]
}
```

Figure 6: Vector add implementation in OpenMP 4 and CUDA.

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

```
__global__ void vectorAddUnroll(int n,
     double *a, double *b, double *c)
{
   for (int i = threadIdx.x + blockIdx.x *
        blockDim.x ;
   i < n; i+= blockDim.x * gridDim.x)
    for(int j=0; j<n_loop; j++)
       a[i] += b[i] + c[n_loop * i + j];
}
```

Figure 7: Vector add implementation with unrollable nested loop in OpenMP 4 and CUDA.

	CUDA	Control Loop	Combined
(#blocks, #threads)	(512,1024)	(64,128)	(1024,1024)
#registers	16	64	21
Shared memory (bytes)	0	280	0
Occupancy	95.9%	26.6%	96.0%
Execution time (μsec.)	1523.5	1988.5	1523.1
Valuable Memory Bandwidth (GB/sec.)	176.20	134.99	176.24

Table 1: Performance metrics for vector add implemented in CUDA, OpenMP using the control loop, and with the combined construct optimization.

	CUDA	Control Loop	Combined
(#blocks, #threads)	(1024,256)	(512,256)	(1024,512)
#registers	18	64	30
Shared memory (bytes)	0	280	0
Occupancy	97.3%	49.5%	97.7%
Execution time (μsec.)	70832.0	78333.0	70456.0
Valuable Memory Bandwidth (GB/sec.)	97.59	88.24	98.11

Table 2: Performance metrics for vector add with large payload implemented in CUDA, OpenMP using the control loop, and with the combined construct optimization.

- The optimal number of: blocks in the grid and threads in each block (using CUDA terminology) for the three different implementations of the test.
- The number of registers per thread obtained from verbose mode of ptxas.
- The amount of shared memory used, as reported by ptxas and nvlink.
- The occupancy factor, obtained by executing nvprof.
- The execution time reported by nvprof for the kernel.
- The calculated valuable global memory bandwidth as the ratio between the size of the arrays being loaded and stored during the test and the execution time.

The first three metrics influence the occupancy factor: the device scheduler for blocks is able to allocate more blocks and warps, on average, onto GPU Streaming Multiprocessors (SMs) when they require fewer registers per thread and shared memory per block. It is easy to notice that the combined construct implementation and the CUDA version can be compiled from PTX into a GPU executable module (cubin file) by using fewer registers per thread (16 and 21), while the control loop requires the maximum amount available (64). This is due to the presence of the control state variables that are live across the entire kernel, plus a complex control-flow pattern.

In principle, ptxas allows the use of more registers per thread than available on a SM. This results in fewer blocks running concurrently on an SM and, for some block size configuration, the scheduler is unable to allocate SMs for the kernel. This results in a kernel launch failure. In these tests we limit the maximum amount to 64 to be able to run correctly all configurations for the block size.

In the table we can notice that the occupancy for the control loop version is half of that obtained by the combined optimization version and by CUDA. Execution times reflect the same behavior, with the OpenMP version using the combined construct perfectly matching CUDA.

5.2 Loops with Large Payloads

To gain a better insight on the overheads induced by the control loop, we run a slightly modified vector add test in which the payload of the loop includes many more (by orders of magnitude) double precision operations. Running these experiments shows whether the control loop overhead is constant in nature, i.e. if it always induces a constant additional time compared to CUDA execution or if it depends on the problem or program size.

Figure 7 shows the OpenMP and corresponding CUDA programs for the benchmark.

We run tests with *n_loop* variable equal to one hundred,

		ApplyAccelBCForNode			CalcMonotonicQRegionForElems		
		CUDA	**OpenMP (Control)**	**OpenMP (Combined)**	**CUDA**	**OpenMP (Control)**	**OpenMP (Combined)**
	Registers	6	64	22	32	64	64
	Shared Memory (Bytes)	0	280	0	0	280	0
12^3	(#blocks, #threads)	(*,128)	(32,64)	(64,256)	(*,128)	(32,64)	(64,32)
	Occupancy	5.5%	6.7%	43.8%	5.9%	6.6%	5.6%
	Execution Time (μsec)	5.184	20.928	5.024	11.169	62.751	15.775
30^3	(#blocks, #threads)	(*,128)	(32,32)	(256,64)	(*,128)	(32,256)	(128,128)
	Occupancy	6.1%	3.3%	14.7%	70.6%	26.5%	43.3%
	Execution Time (μsec)	5.568	22.912	4.96	29.184	178.18	67.296
100^3	(#blocks, #threads)	(*,128)	(32,128)	(512,64)	(*,128)	(128,64)	(1024,256)
	Occupancy	27.8%	13.2%	33.8%	93.0%	26.4%	47.3%
	Execution Time (μsec)	6.72	50.944	14.976	1287.4	4833.2	2563.4

Table 3: Performance metrics for two kernels of LULESH, ApplyAccelBoundaryConditionsForNodes and CalcMonotonicQRegionForElems, in three different implementations: CUDA, OpenMP with Control Loop, OpenMP with combined directive optimization. The results are shown for three mesh sizes: 12^3, 30^3 and 100^3.

and n roughly equal to 8 million elements (512*512*32). Like the previous benchmark, we show results for the best configuration of team size and number of teams for each case individually.

Table 2 shows the results.

As expected, the number of registers and shared memory size used by the three versions is on par with the previous tests. The large number of registers used by the control loop version results in half the occupancy of the other two versions. In this test we notice about 10% overhead of the control loop version compared to the CUDA one, whereas in the previous test the same metric was about 25%. This points to the fact that the overhead induced by the control loop is amortized by the amount of useful operations performed per iteration. Occupancy is still very low and this can represent a major limitation for kernels that rely on the global memory latency hiding mechanism, i.e. when data streaming from global memory into SMs is large compared to the number of instructions per iteration. A solution is to employ the combined construct which results in performance figures similar to the ones obtained with CUDA.

The difference in the number of registers used by the CUDA and OpenMP versions is larger than for the previous test. We identified several sources for this difference. We discuss this issue at the end of this section.

Additionally, OpenMP compilation uses LLVM's NVPTX backend, while CUDA uses libnvvm, as previously noted. In complex cases we may expect a difference in code lowering. For this test we analyzed the resulting PTX files and noticed that both backends successfully unroll partially or completely the innermost loop, depending on its size. We leave full PTX analysis as future work.

5.3 Results for LULESH Kernels

The last test that we consider are two kernels in LULESH (available online [15]), respectively located in functions *ApplyAccelBoundaryConditionsForNodes* and *CalcMonotonicQRegionForElems* of the main file lulesh.cc. The first kernel is a simple one similar to the vector add benchmark with additional control-flow instructions (three if-then statements). The second one is more compute- and data-intensive than the previous tests and exposes more control-flow logic. The goal is to study further compilation issues for Clang/LLVM compared to CUDA. These two kernels are representative of the range of the computational complexity exposed by proxy applications.

Both kernels use OpenMP parallel loop in the original LULESH version, and they respectively iterate over boundary mesh nodes and elements. We modified the implementation by adding OpenMP 4.0 directives that specify data mappings (*pragma target data*) and added *pragma target team distribute* as combined with the existing *pragma parallel for*. Results for CUDA experiments are from runs of an unmodified LULESH CUDA implementation available online [15].

ApplyAccelBoundaryConditionsForNodes is implemented differently for CUDA and OpenMP. In OpenMP there is a single loop over nodes, while CUDA partitions it further into three kernels, each iterating over mesh dimensions X, Y, and Z. This results in three CUDA kernel calls when executing the enclosing function.

For kernel CalcMonotonicQRegionForElems, Table 4 shows profiling metrics of the PTX code resulting from compilation with Clang/LLVM. It shows the number of PTX instructions issued for each iteration of the loop, limited to the payload. It can be noticed that the loop contains a relevant number of control flow instructions compared to the previous tests. We run tests for three problem sizes (12^3, 30^3, and 100^3), corresponding to the number of elements in the unstructured mesh used by LULESH.

Floating Point Instructions	104
Integer Instructions	35
Control Flow Instructions	42
4-byte Loads and Stores	7
8-byte Loads and Stores	18

Table 4: Lulesh kernel profile as number of instructions, by type, per loop iteration.

Table 3 shows experiment results. The table is organized in two blocks of columns, corresponding to the two kernels. Each block is split into multiple sub-columns, each corresponding to different implementations: CUDA, OpenMP using the control loop (*Control Loop*), and OpenMP using the combined construct implementation (*Combined*). The num-

ber of registers used to compile the kernels and the shared memory size are independent of the size of the problem and shown in the first two rows. These values are returned by ptxas and nvlink. The remaining rows are divided into three main blocks, each corresponding to a different problem size, and they list various performance metrics.

The main difference between OpenMP and CUDA compilations is the amount of registers used. The CUDA implementation of both kernels requires the fewest amount of registers. For OpenMP, the control loop always requires the maximum available per thread (64). Using larger values is allowed by the low level compilation tools (ptxas and nvlink) but our experiments on LULESH showed that the best performance is always achieved by limiting the maximum value to 64. The combined construct implementation shows fewer registers than the control loop but still larger than CUDA for both kernels

For ApplyAccelBoundaryConditionsForNodes, the CUDA and combined construct versions perform similarly, where execution time for the combined version is slightly lower than CUDA for problem sizes 12^3 and 30^3. This is also reflected by results for occupancy. There are two reasons for this result. First, the CUDA version uses three kernels call, as we noted above, over the three X, Y, and Z dimensions. OpenMP uses a single kernel call for the three dimensions. This reduces the overhead depending on the problem size. Second, both versions use fewer registers than the ones available in the hardware (respectively 6 and 22), and as such the OpenMP combined version is not constrained by low occupancy levels. The improved performance results from a better choice of number of blocks and number of threads in each block: the OpenMP results are shown for a careful selection of the values, by running multiple experiments; the CUDA implementation uses a fixed amount of threads per block (128) and it calculates the number of blocks by dividing the iteration space size (number of elements in the mesh) by the block size. This results in oversubscription which, under certain circumstances, may induce overheads.

For CalcMonotonicQRegionForElems, CUDA shows the best results, with about 30% difference between CUDA and the best OpenMP run. This is due to the larger number of registers used per thread by both OpenMP implementations and it is reflected in the occupancy values reported by nvprof.

As noted for the previous test, there are two main areas of improvement for register usage that apply to the compilation toolchain. In the current implementation, pointers to array sections used in a target region are passed to the related GPU kernel as parameter of type *reference to pointer*. This gives the kernel the ability to access the pointer value itself, and possibly modify it. When lowered at PTX level, this results in an additional load instruction for each kernel argument to a register that can be used as base address of the array. When passed to PTXAS, this code results in an additional hardware register for each thread and for each kernel input parameter that is a pointer to array. In OpenMP 4.1 this behavior is not required and we need to pass the pointer to the array section directly to the kernel, e.g. as a parameter. This represents a first improvement to current code generation. It removes the need of transforming the input parameter to the actual array address.

Second, we need to analyze potential improvements in code synthesis when using libnvvm through CUDA. As part of our future work, we will continue analyzing the differences in code produced by LLVM for the nvptx backend and by libnvvm. We will suggest improvements to LLVM following this analysis. Directly using libnvvm as compiler backend to the OpenMP implementation in Clang is harder, due to the requirement of transforming LLVM IR version 3, as generated by Clang, into NVVM IR.

6. CONCLUSION

In this paper we showed that modular integration into Clang of the control loop scheme is possible and we described software modifications that are in line with Clang/LLVM's general programming strategy. We showed performance results for two simple kernels that highlight the limitations of the control loop scheme. We described an optimization to overcome these limitations for certain recurrent programming patterns based on combined OpenMP pragma constructs. We also showed that such limitations induce larger overheads to the OpenMP program when considering more complex kernels derived from the LULESH application.

The paper characterizes occupancy as the limiting factor. There is a large difference in the amount of registers per thread required to execute the OpenMP versions and the CUDA C/C++ one, which is exacerbated by the complexity of the kernel. To improve register usage we suggested two ways forward: improving the LLVM NVPTX backend also analyzing libnvvm; and improving the kernel parameter passing mechanism.

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] A. Baker. Custom hardware state-machines and datapaths: Using llvm to generate fpga accelerators, October 2014. http://llvm.org/devmtg/2014-10/Slides/Baker-CustomHardwareStateMachines.pdf.

[2] J. Barker and J. Bowden. Manycore parallelism through openmp. In A. P. Rendell, B. M. Chapman, and M. S. Muller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 45–57. Springer Berlin Heidelberg, 2013.

[3] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] A. Bataev. Openmp support in clang/llvm: Status update and future directions, October 2014. http://llvm.org/devmtg/2014-10/Slides/Bataev-OpenMP.pdf.

[5] G.-T. Bercea, C. Bertolli, S. F. Antao, A. C. Jacob, A. E. Eichenberger, L. Duran, T. Chen, Z. Sura,

H. Sung, G. Rokos, D. Appelhans, and K. O'Brien. Performance analysis of openmp on a gpu using a coral proxy application. In *Submitted to 6th International Workshop in Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS15)*, 2015.

[6] 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.

[7] G. Brown. Implementing the sycl for opencl shared source c++ programming model using clang/llvm. https://www.codeplay.com/public/uploaded/publications/SC2014_LLVM_HPC.pdf.

[8] Github repository for extended clang implementation supporting openmp 4.0. https://github.com/clang-omp/clang_trunk.

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

[10] Cuda toolkit webpage. http://docs.nvidia.com/cuda/index.html.

[11] M. Haidl and S. Gorlatch. Pacxx: Towards a unified programming model for programming accelerators using c++14. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, pages 1–11, Piscataway, NJ, USA, 2014. IEEE Press.

[12] Nvidia libnvvm library manual. http://docs.nvidia.com/cuda/libnvvm-api/modules.htm.

[13] The llvm compiler infrastructure webpage. http://llvm.org/.

[14] Llvm backend component for nvptx archietecture (nvidia gpus). http://llvm.org/docs/NVPTXUsage.html.

[15] Lulesh webpage. https://codesign.llnl.gov/lulesh.php.

[16] Github repository for libomptarget offloading and gpu openmp runtime. https://github.com/clang-omp/libomptarget.

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

[18] 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. P. Rendell, B. M. Chapman, and M. S. Muller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 114–127. Springer Berlin Heidelberg, 2013.

[19] Vikas, T. Scott, N. Giacaman, and O. Sinnen. Using openmp under android. In A. P. Rendell, B. M. Chapman, and M. S. Muller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 15–29. Springer Berlin Heidelberg, 2013.

[20] U. Weigand. Supporting the new ibm z13 mainframe and its simd vector unit, April 2015. http://llvm.org/devmtg/2015-04/slides/Euro-LLVM-2015-Weigand.pdf.