Efficient Execution of OpenMP on GPUs

Joseph Huber  
Oak Ridge National Laboratory  
Oak Ridge, USA  
huberjn@ornl.gov

Melanie Cornelius  
Illinois Institute of Technology  
Chicago, USA  
mdooley1@hawk.iit.edu

Giorgis Georgakoudis  
Lawrence Livermore National Laboratory  
Livermore, USA  
georgakoudis1@llnl.gov

Shilei Tian  
Stony Brook University  
Stony Brook, USA  
shilei.tian@stonybrook.edu

Jose M Monsalve Diaz  
Argonne National Laboratory  
Lemont, USA  
jmonsalvediaz@anl.gov

Kuter Dinel  
Düzcé University  
Düzcé, Turkey  
kuterdinel@gmail.com

Barbara Chapman  
Stony Brook University  
Stony Brook, USA  
barbara.chapman@stonybrook.edu

Johannes Doerfert  
Argonne National Laboratory  
Lemont, USA  
jdoerfert@anl.gov

Abstract—OpenMP is the preferred choice for CPU parallelism in High-Performance-Computing (HPC) applications written in C, C++, or Fortran. As HPC systems became heterogeneous, OpenMP introduced support for accelerator offloading via the target directive. This allowed porting existing (CPU) code onto GPUs, including well established CPU parallelism paradigms. However, there are architectural differences between CPU and GPU execution which make common patterns, like forking and joining threads, single threaded execution, or sharing of local (stack) variables, in general costly on the latter. So far it was left to the user to identify and avoid non-efficient code patterns, most commonly by writing their OpenMP offloading codes in a kernel-language style which resembles CUDA more than it does traditional OpenMP.

In this work we present OpenMP-aware program analyses and optimizations that allow efficient execution of the generic, CPU-centric parallelism model provided by OpenMP on GPUs. Our implementation in LLVM/Clang maps various common OpenMP patterns found in real world applications efficiently to the GPU. As static analysis is inherently limited we provide actionable and informative feedback to the user about the performed and missed optimizations, together with ways for the user to annotate the program for better results. Our extensive evaluation using several HPC proxy applications shows significantly improved GPU kernel times and reduction in resources requirements, such as GPU registers.

Index Terms—OpenMP, Offloading, Optimization, LLVM, GPU

I. INTRODUCTION

The triumph of general-purpose computing on GPUs in HPC was driven in large parts by the development of explicit “kernel languages”, especially CUDA. In contrast to traditional HPC programming languages, like C, C++, or Fortran, which were designed with a CPU execution model in mind, kernel languages are tailored towards the execution model of a GPU. While on first glance those might look similar, there are conceptual differences that inherently limit the ability to efficiently execute an arbitrary CPU-centric program on a GPU. Nevertheless, common CPU languages have been extended to target GPUs in an effort to natively integrate portable GPU programming into legacy HPC software.

One promising candidate to bring offloading capabilities to existing codes is OpenMP. While OpenMP offloading provides a native way to program GPUs of various vendors, it inherently follows the CPU-centric model that is fundamental to the base languages. Further, OpenMP offloading was designed as an extension to the existing CPU parallelism capabilities which means that not only the base language semantics but also the parallelism paradigms do often not match the native execution model of the GPU.

The artificial code in Figure 1 showcases how the OpenMP programming model and the GPU execution model diverge. First, OpenMP requires the right hand side of the team_val assignment to be executed only by the main thread of each team. Given that each team runs on a streaming multiprocessor (SM) which executes 32 (or more) threads simultaneously, the compiler has to ensure the side-effects of compute for all but one thread do not become visible. Once a parallel directive is reached by the main thread, which could also happen inside compute, the user expects some, if not all, of the threads in the SM to execute the parallel region. Thus, the compiler needs to guard arbitrary code across translation units, e.g., if compute is defined elsewhere, while allowing parallel execution of encapsulated parallel directives. Further, code like compute can be executed from sequential and parallel regions alike, both should execute correctly and provide the expected execution behavior with regards to nested parallelism.

In addition to thread management, the compiler has to implement local variable sharing in an execution environment

978-1-6654-0584-3/22/$31.00 © 2022 IEEE
that does not support it. In the example, `team_val` as well as
`thread_val` are local (stack or automatic) values of which the
address is passed into the `combine` function. Without further
knowledge of their use the compiler needs to assume these
variables are potentially read or written by another thread.
While this is not a problem on CPUs which allow cross thread
accesses to the “stack”, it is on a GPU where these variables
would by default reside in thread-local memory.

```cpp
#pragma omp target teams distribute
for (int block_id = 0; block_id < NBlocks; ++
    block_id){
    auto team_val = compute(); // once per team
#pragma omp parallel for
for (int thread_id = 0; thread_id < NThreads; ++
    thread_id){
    auto thread_val = compute(); // once per thread
    combine(team_val, &thread_val);
}
}
```

Fig. 1: OpenMP target region to show how the CPU-centric model diverges
from the native GPU execution model.

In this work we introduce novel, inter-procedural compiler
optimizations to bridge the gap between the CPU-centric
OpenMP programming model and the GPU execution model.
First we discuss background and related work in Section II
and III respectively. In Section IV we detail our approach,
including implementation notes, optimization descriptions, and
user interaction strategies. A thorough evaluation of four
HPC proxy applications is presented in Section V. We show
that these codes are affected by the same implementation
complexities described for Figure 1 and that our approach
effectively eliminates the associated overheads. Section VI
concludes the paper and provides thoughts on future efforts.

The main contributions of this work are:
- novel OpenMP-aware inter-procedural analysis and opti-
mizations that operate on the LLVM-IR level,
- effective translation of HPC kernels from the CPU-centric
OpenMP model to the GPU execution model,
- automatic optimizations for common cases combined with
actionable user feedback and assumption handling when
static information is insufficient, and
- integration into the community version of the LLVM
compiler including documentation for the user feedback
and compiler-usable program assumptions.

While our techniques are applicable for any GPU, we
explicitly do not address the technical challenges of:
- shortcomings in the LLVM/OpenMP GPU runtime [1],
unrelated to compilation, which hamper performance
compared to kernel languages,
- secondary effects of our optimizations, e.g., source sim-
plifications that lead to changed heuristics (unroll, inline,
etc.) which decrease program performance, and
- the effect on AMD and Intel GPUs as the LLVM
implementation does not target the latter and the former
is not mature enough to run our benchmark set.

II. BACKGROUND

OpenMP offloading features a host-centric execution mode.
The host (CPU) coordinates scheduling and synchronization
of target tasks (i.e. kernels), as well as memory allocation and
movement between host and devices (e.g. GPUs and FPGAs).
On the device, the main thread of a team begins execution of
the target task. The teams directive controls the creation of a
league of teams. Each team begins with a single main thread
and it can spawn other worker threads by using the parallel
directive. The worksharing directives for and distribute
allow scheduling loop iterations across threads in a team and
teams in a league, respectively.

GPU execution models have a similar hierarchical organiza-
tion. GPUs are composed of multiple streaming multiprocessors
(SM), each capable of executing several threads grouped into
scheduling units that execute at the same time (e.g. a warp in
NVIDIA GPUs and wavefront for AMD GPUs).

When lowering OpenMP onto GPUs it is common to map a
team to an SM and the threads of a team to hardware threads
within the SM. Figure 2 shows the corresponding mapping
between the GPU execution model and the OpenMP execution
model. The figure also illustrates what memory kinds are
accessible by the thread(s) on each level.

![Mapping of the OpenMP programming model to the GPU](image)

Fig. 2: Simplified mapping of the OpenMP programming model to the GPU.
Top row: Outermost parallelism across streaming multiprocessors (SMs) onto
which OpenMP teams are mapped. All threads on this level share the global
memory. Middle row: A single SM corresponding to a team of threads in
OpenMP. Both shared and global memory are accessible by these threads.
Bottom row: A single GPU/OpenMP thread which has exclusive access to
local memory and registers.

III. RELATED WORK

OpenMP target offloading was introduced in version 4.0,
firstly adopted to the Intel Xeon Phi co-processor, and an ARM
based DSM by Texas Instrument [2], [3]. In LLVM/Clang,
OpenMP offloading support for GPUs was first presented by
[4], [5], proposing a control loop to coordinate GPU
threads for executing sequential and parallel regions within an
offloaded region, by implementing a state machine to track
execution semantics. \cite{6} introduce in the IBM XL C/C++ compiler a lowering of OpenMP that executes without the control loop state machine in a mode where all threads execute in parallel, deemed SPMD mode of execution, when the target offloaded region encloses a single parallel construct. Since IBM XL C/C++ compiler is derived from Clang/LLVM, those changes have been upstreamed Clang/LLVM too. \cite{7} introduced the first front-end based optimizations for NVIDIA GPUs in LLVM/Clang, related to choosing the number of teams and threads for parallel loops to avoid idle threads and reduce register usage. The (PGI) Fortran front-end, known as Flang, supports OpenMP offloading via the LLVM/OpenMP runtimes. The comparative analysis of multiple OpenMP compilers by \cite{9, 10} demonstrated severe performance issues in the LLVM/OpenMP implementation, among other compilers. This work builds upon designs and implementations from this initial LLVM/Clang support of GPU offloading. It differs in that it proposes simplifying code generation in the front-end to enable semantic analysis and optimization in LLVM (middle-end), which results in significant performance improvements, as we show later in the paper.

Regarding compiler-based optimizations on OpenMP, \cite{11} presented the TRegion interface which delayed the discovery of SPMD regions into LLVM, by contrast to the Clang-based approach, which enabled more kernels to execute in SPMD mode. For host OpenMP, \cite{12} demonstrated that LLVM’s outlining approach hinders the application of existing compiler optimizations. By analyzing the semantics of OpenMP runtime functions, authors re-enabled such optimizations in the presence of OpenMP parallelism. In addition to some ideas shared with \cite{11}, such as the device function de-virtualization, they also mention SPMD code generation for the distributed parallel loop constructs in a target team region. \cite{13, 14} proposed extensions to the LLVM IR, in the form of intrinsics, to represent parallelism present in OpenMP directives. However, those extensions are hard to integrate in the existing LLVM/Clang infrastructure and provide little benefit in semantic analysis in comparison to other approaches \cite{12, 11} that directly analyze OpenMP runtime functions. Further, tools that translate between OpenACC and OpenMP \cite{15}, \cite{16} include optimizations designed to resolve execution execution model mapping mismatches, both on CPUs and GPUs, such as using nested parallelism and the SIMD clause. Our work concurs to previous approaches on the necessity of semantic analysis for parallelism in the IR. OpenMP in this case, but significantly expands both the analysis scope and applicability, extends existing optimizations, and proposes new ones.

While we are concerned with efficient GPU execution of CPU-centric codes, there is a body of work to efficiently translate codes in the opposite direction \cite{17, 18, 19}. Similar to our work, reverse transformations require explicit optimizations to bridge the conceptual execution mode differences effectively. Differently to our work, those efforts aim for best effort performance translating from the, typically faster, GPU execution to CPU, while we strive to achieve parity in performance with native programming models on GPUs.

IV. OPENMP-AWARE LLVM-IR ANALYSES AND OPTIMIZATIONS

We implemented the analysis and optimization of OpenMP programs as a pass in the default LLVM optimization pipeline. It is run early on the entire module and again late on each strongly connected component of call graph. Dealing with OpenMP on the LLVM intermediate language (LLVM-IR) level has benefits and drawbacks. \cite{20}. While we loose syntactic structure of the pragmas embedded in high-level source code, we gain access to existing analysis passes and the ability to write data-flow analysis in a familiar manner. Further, our work is applicable to C/C++ input from LLVM/Clang and Fortran input from LLVM/Flang. To embed OpenMP domain knowledge into the LLVM-IR we look for uses of known LLVM/OpenMP runtime functions that have been emitted by the front-end in response to user pragmas.

To deal with abstractions introduced during the OpenMP lowering and with the complexity of real world applications we designed our optimizations to be inherently inter-procedural. As such, the analysis performs best if it has full visibility of the kernel, called functions, and the callers of all functions. The latter is important as otherwise an external caller could call a routine from a context where a desired transformation might be illegal.

We explicitly avoided relying on aggressive inlining to bypass external callers as it comes with various side-effects, including an often significant compile time impact. Said differently, we believe the inliner heuristic for a given architecture should be in charge of inlining decisions rather than any non-inlining related optimization. To nevertheless avoid precision loss of our analysis in the presence of externally visible functions we performed aggressive internalization. In essence, we duplicate functions with external linkage\footnote{LLVM/Flang is technically not mature enough to emit LLVM-IR but the OpenMP code generation is shared between Clang and Flang which means the input is indistinguishable for our optimization.} to create an internal only copy, used when invoked from a kernel within the translation unit, and an external only copy, which is used otherwise. This way we preserve semantics for all unknown external callers while the current translation unit has full visibility of each usage and can act accordingly.

A. Optimizing Variable Placement

The example in Figure \ref{fig:example} features two variables with stack (or automatic) storage, namely team_val and thread_val. Such local variables, as well as arguments of device functions, are commonly allocated on the stack if their address is taken. Most CPU architectures allow different threads to freely access any variables allocated on the stack. When this example is compiled for the host, the declarations of these variables are therefore not treated differently by the compiler. However, if the architecture disallows accesses to local (stack) variables across threads the situation changes drastically.

\footnote{Not all linkage types allow this and we take this into account. An optimization remark (ref. Section \ref{sec:optimizations}) is emitted if internalization fails.}
On the GPU, as illustrated in Figure 2, only the thread itself has access to local memory, or registers. This means that local variables cannot be placed there if the programming model allows the user to access them from different threads.

Prior to this work, thus LLVM 12 and earlier, Clang would “globalize” all local variables whose address may be taken during the generation of the LLVM-IR. This means, the local variables would instead be allocated (and deallocated) from global or shared memory through an OpenMP GPU device runtime call. Clang used syntactic information to combine all globalized locals in a structure type and allocate them all at once, preferably as a struct-of-arrays (SoA) to allow coalesced accesses by the threads in a warp. Further, if the globalization was not nested syntactically in a target region, e.g., it happens in a device function, Clang generated runtime checks to determine how many threads reached the program location. Depending on the result, either local memory (aka. registers), a single copy of the globalized structure, or the SoA version for an entire warp would be used. As a performance improvement, Clang would not globalize any variables for code executed in SPMD mode (ref. Section IV-B) or when the command line option -fopenmp-cuda-mode was used. However, both are in general unsound and the former, using local memory in SPMD mode, fails for the code illustrated in Figure 3.

Fig. 3: OpenMP target region executed in SPMD mode with cross thread accesses to the local variable Lcl which consequently requires globalization.

In this work, we removed the incorrect SPMD mode optimization performed by Clang and instead globalize all variables that are potentially shared. Furthermore, we instead globalize each variable individually rather than combining them into a single allocation. Globalizing all eligible variables prevents miscompilations while simplifying the code generation, and allows for each variable to be analyzed individually. These changes significantly decrease the performance of OpenMP device code in favor of correctness. To recover performance, we developed a novel inter-procedural optimization in the middle-end to effectively undo globalization or utilize a specialized and more efficient implementation.

An example showing the difference in globalization using LLVM/Clang 12 and our new method is shown in Figure 4. Before our work, the generated code would not be optimized further, causing globalization to remain on the device and significantly impacting performance. When generating the OpenMP code, the front-end can only perform simple intra-procedural analysis. Because the front-end has a limited view of the code, it will introduce globalization whenever it is possible that a variable could be shared between threads in order to maintain OpenMP semantics. If, for example, the combine function used in Figure 4a is later known to not share the pointer arguments with other threads, we can safely undo the globalization and use local memory instead.

In this work, we removed the incorrect SPMD mode optimization performed by Clang and instead globalize all variables that are potentially shared. Furthermore, we instead globalize each variable individually rather than combining them into a single allocation. Globalizing all eligible variables prevents miscompilations while simplifying the code generation, and allows for each variable to be analyzed individually. These changes significantly decrease the performance of OpenMP device code in favor of correctness. To recover performance, we developed a novel inter-procedural optimization in the middle-end to effectively undo globalization or utilize a specialized and more efficient implementation.

An example showing the difference in globalization using LLVM/Clang 12 and our new method is shown in Figure 4. Before our work, the generated code would not be optimized further, causing globalization to remain on the device and significantly impacting performance. When generating the OpenMP code, the front-end can only perform simple intra-procedural analysis. Because the front-end has a limited view of the code, it will introduce globalization whenever it is possible that a variable could be shared between threads in order to maintain OpenMP semantics. If, for example, the combine function used in Figure 4a is later known to not share the pointer arguments with other threads, we can safely undo the globalization and use local memory instead.

In this work, we removed the incorrect SPMD mode optimization performed by Clang and instead globalize all variables that are potentially shared. Furthermore, we instead globalize each variable individually rather than combining them into a single allocation. Globalizing all eligible variables prevents miscompilations while simplifying the code generation, and allows for each variable to be analyzed individually. These changes significantly decrease the performance of OpenMP device code in favor of correctness. To recover performance, we developed a novel inter-procedural optimization in the middle-end to effectively undo globalization or utilize a specialized and more efficient implementation.

An example showing the difference in globalization using LLVM/Clang 12 and our new method is shown in Figure 4. Before our work, the generated code would not be optimized further, causing globalization to remain on the device and significantly impacting performance. When generating the OpenMP code, the front-end can only perform simple intra-procedural analysis. Because the front-end has a limited view of the code, it will introduce globalization whenever it is possible that a variable could be shared between threads in order to maintain OpenMP semantics. If, for example, the combine function used in Figure 4a is later known to not share the pointer arguments with other threads, we can safely undo the globalization and use local memory instead.

In this work, we removed the incorrect SPMD mode optimization performed by Clang and instead globalize all variables that are potentially shared. Furthermore, we instead globalize each variable individually rather than combining them into a single allocation. Globalizing all eligible variables prevents miscompilations while simplifying the code generation, and allows for each variable to be analyzed individually. These changes significantly decrease the performance of OpenMP device code in favor of correctness. To recover performance, we developed a novel inter-procedural optimization in the middle-end to effectively undo globalization or utilize a specialized and more efficient implementation.

An example showing the difference in globalization using LLVM/Clang 12 and our new method is shown in Figure 4. Before our work, the generated code would not be optimized further, causing globalization to remain on the device and significantly impacting performance. When generating the OpenMP code, the front-end can only perform simple intra-procedural analysis. Because the front-end has a limited view of the code, it will introduce globalization whenever it is possible that a variable could be shared between threads in order to maintain OpenMP semantics. If, for example, the combine function used in Figure 4a is later known to not share the pointer arguments with other threads, we can safely undo the globalization and use local memory instead.

In this work, we removed the incorrect SPMD mode optimization performed by Clang and instead globalize all variables that are potentially shared. Furthermore, we instead globalize each variable individually rather than combining them into a single allocation. Globalizing all eligible variables prevents miscompilations while simplifying the code generation, and allows for each variable to be analyzed individually. These changes significantly decrease the performance of OpenMP device code in favor of correctness. To recover performance, we developed a novel inter-procedural optimization in the middle-end to effectively undo globalization or utilize a specialized and more efficient implementation.

An example showing the difference in globalization using LLVM/Clang 12 and our new method is shown in Figure 4. Before our work, the generated code would not be optimized further, causing globalization to remain on the device and significantly impacting performance. When generating the OpenMP code, the front-end can only perform simple intra-procedural analysis. Because the front-end has a limited view of the code, it will introduce globalization whenever it is possible that a variable could be shared between threads in order to maintain OpenMP semantics. If, for example, the combine function used in Figure 4a is later known to not share the pointer arguments with other threads, we can safely undo the globalization and use local memory instead.

In this work, we removed the incorrect SPMD mode optimization performed by Clang and instead globalize all variables that are potentially shared. Furthermore, we instead globalize each variable individually rather than combining them into a single allocation. Globalizing all eligible variables prevents miscompilations while simplifying the code generation, and allows for each variable to be analyzed individually. These changes significantly decrease the performance of OpenMP device code in favor of correctness. To recover performance, we developed a novel inter-procedural optimization in the middle-end to effectively undo globalization or utilize a specialized and more efficient implementation.

An example showing the difference in globalization using LLVM/Clang 12 and our new method is shown in Figure 4. Before our work, the generated code would not be optimized further, causing globalization to remain on the device and significantly impacting performance. When generating the OpenMP code, the front-end can only perform simple intra-procedural analysis. Because the front-end has a limited view of the code, it will introduce globalization whenever it is possible that a variable could be shared between threads in order to maintain OpenMP semantics. If, for example, the combine function used in Figure 4a is later known to not share the pointer arguments with other threads, we can safely undo the globalization and use local memory instead.
transformation performs two separate checks of which at least one must succeed. The first will follow all uses of the heap pointer inter-procedurally and report if any of the uses might expose the pointer to another thread. The second will determine if the associated deallocation call has to be reached. In both cases the analysis has to consider potential synchronization between threads.

If heap-to-stack is not able to modify the storage location of a variable, we employ a second inter-procedural transformation that aims to replace the runtime calls with statically allocated shared memory. In contrast to the runtime allocation, which also uses a shared memory buffer organized as a stack, the newly introduced static allocation eliminates all instructions associated with the globalization and allows further memory optimizations, e.g., store-to-load-forwarding, as the lifetime and exact location are known to the compiler. The transformation inter-procedurally determines if the runtime allocation is only executed by the main thread of the OpenMP team. If so, the runtime allocation and deallocation are replaced by a (global) static allocation in shared memory. While this causes the allocation to be present during the entire lifetime of the kernel, we have not encountered a problematic case that would have benefited from an allocation on a software managed shared memory stack instead.

Figure 5 illustrates how Figure 4a could be connected to other program parts. The combine function in Figure 5a, together with either call site of the device_function, thus Figure 5b or 5c respectively, exhibit the three potential outcomes for the globalized variables, Arg and Lcl, featured in Figure 4a.

```c
double combine(float* ArgPtr, double* LclPtr) {
    unknown(ArgPtr);
    return LclPtr + *ArgPtr;
}
```

(a) A potential implementation of the combine function used in Figure 4a.

```c
void one_thread_only() {
    #pragma omp target teams
    device_function(get_arg());
}
```

(b) A potential call site of the device_function shown in Figure 4a that enters it with a single thread per team.

```c
void many_threads() {
    #pragma omp target teams
    #pragma omp parallel
    device_function(get_arg());
}
```

(c) A potential call site of the device_function shown in Figure 4a that enters it with multiple threads per team.

**Fig. 5:** Functions to provide specific context to the generic code shown in Figure 4a.

If all call sites of device_function are known to be executed only with the main thread of an OpenMP team, e.g., as shown in one_thread_only, the optimization outcome is as sketched in Figure 6a. Both transformations, heap-to-stack and the static shared memory allocation, triggered for one variable respectively. Lcl was only read in the combine function and is therefore simply replaced with local (stack) memory. Arg escaped into the unknown function and could there be shared among multiple threads. However, as long as the runtime call allocating memory for Arg is known to be executed only by the main thread in an OpenMP team we can use statically allocated shared memory. This is effectively equivalent to using an OpenMP allocator or CUDA __shared__ annotation. If the runtime call allocations could be reached by a thread other than the main thread, e.g., if Figure 5c is a call site of device_function, we do not create a static allocation but instead emit an optimization remark to the user (ref. Section IV-D). As noted in Figure 5b, it would be possible to create a static allocation in shared memory even for this case. However, it would require us to scale the allocation that is live for the entire target region by the maximal number of threads in a team. These optimizations allow users to disable globalization and get the performance of using -fopenmp-cuda-mode without sacrificing correctness. So far, we have not found a case where globalization for many threads was actually required and not only a side-effect of insufficient static analysis information. While we allow users to opt-in to large static allocations, we believe it is often more useful to indicate the problem via a remark (ref. Section IV-D) together with actionable advice on how to provide the required domain knowledge for the heap-to-stack transformation to become valid.

3It is worth noting that this will automatically move static sized arrays allocated in an OpenMP target region and used subsequently in an OpenMP parallel region into shared memory, a transformation that CUDA programmers are applying explicitly with the __shared__ annotation on buffers.
B. Optimizing Thread Execution

The OpenMP programming model follows the multi-threaded CPU execution model where there is a single active thread, deemed the main thread, at the beginning of each (implicit) teams region that spawns or activates worker threads for parallel computation. However, this model deviates from the SPMD execution model of GPUs, exposed by the kernel-language model, in which all threads start execution at kernel launch. Conceptually, when offloading a target region, the compiler needs to emulate a sequential region on the GPU by guarding the effects of all but the main thread of a team until a parallel region is reached and those threads become active.

In generic (execution) mode, introduced by [4], the threads of a team are separated at the beginning of a target region into the main thread and worker threads. Workers enter a front-end generated state machine while the main thread executes the user code. When a parallel directive is encountered the outlined function containing the parallel region code is passed from the main thread to the workers for parallel execution before they go back into their waiting state. However, to avoid costly indirect calls through function pointers, LLVM/Clang performed simple static analysis in the front-end. Effectively, an if-cascade was embedded in the state machine which checked the communicated function pointer against known parallel regions in order to call them directly [4]. Since parallel regions may be unknown, e.g., hidden under functions defined in other translation units, it is not always possible to list them all statically. An indirect fallback call after the if-cascade was emitted as a catch-all.

The generic mode incurs overhead through the diverging control flow and synchronization required by the state machine. To avoid those overheads, another mode of execution, the so-called SPMD mode [5], was introduced. If syntactic analysis in the front-end determines that it is safe to execute a target region with all threads, e.g., the OpenMP target directive is syntactically combined with the Parallel directive, SPMD mode is used. In this mode all threads are active at the kernel beginning, effectively replicating the kernel-language model. However, syntactic analysis in the front-end for determining the applicability of SPMD mode is inherently limited. If there is non-trivial code executed by the OpenMP main thread alone generic mode was required.

[11] introduces the first SPMDzation optimization that transforms a generic mode kernel into an SPMD mode kernel. Through guarding of side-effects in sequential regions, and broadcasts of values where necessary, a consistent program state was achieved while keeping all threads active. Further, for generic mode execution, the state machine was customized via inter-procedural reachability information to completely avoid the indirect fallback call.

1) Contributions: In this work we extend those ideas, make them applicable to realistic applications, and perform novel optimizations to increase their efficiency.

2) Eliminating Function Pointers in the If-Cascade.: The use of function pointers to communicate what parallel region to execute next is necessary for the general case as the parallel region might be compiled separately from the target region. However, taking the address of a function can lead to spurious call edges assumed by the GPU vendor toolchains that increase the required register count [4]. Our optimization eliminates the use of function pointers if it can prove that all target regions which may reach a particular parallel region are known statically. Instead of the parallel region address we introduce a new unique identifier that is used as a stand-in replacement when the main thread informs the workers that the particular parallel region has to be executed next. If unknown target regions may reach a parallel region, we cannot make them aware of the relationship between the parallel region and the generated identifier. Consequently, we need to keep the function pointer communication. An optimization remark (ref. Section [IV-D]) is issued in this case.

3) Expanding the Scope and Efficiency of SPMDzation.: Compared to previous work [6], [11] we perform SPMDzation based on inter-procedural analysis at the LLVM-IR level. Starting from generic mode kernels, all sequentially executed code is analyzed inter-procedurally and all encountered side-effects are check for compliance with our local guarding scheme. Only if the side-effects are always reached by the main thread alone we employ SPMDzation. This limitation is self-imposed as our experiments show that a more complex guarding alternative, needed to handle single and multi-threaded execution, often hampers performance. If a guarded code generates values used outside the guarded region we employ the broadcast logic described by [11].

Our SPMDzation analysis explicitly interacts with the data placement optimization (ref. Section [IV-A]). This allows us to avoid guards and broadcasts for OpenMP specific allocation related code which effectively does not require it.

To minimize the number of guarded regions, and the associated synchronization overhead, we perform local instruction reordering prior to guard generation. Figure [7] illustrates this optimization for two side-effects that are separated by some SPMD amenable code, e.g., side-effect free code or annotated function calls (ref. Section [IV-D]). While by default each side-effect requires a guarded region and one barrier (two if we need to broadcast values), we can amortize the cost if multiple side-effects share the same guarded region. Our reordering optimization groups side-effects at the basic block level without violating data-flow dependencies among them to minimize the number of required guarded regions.

C. Optimizing OpenMP Runtime Calls

The device runtime, roughly the GPU equivalent of the host OpenMP runtime, provides OpenMP utility functions, e.g., to split a work sharing loop at runtime based on a dynamically chosen schedule policy. The flexibility of the OpenMP programming model requires the device runtime to work in generic and SPMD mode and across all possible GPU kernel launch parameters.

[The problem is described in more detail here: http://llvm.org/PR46450]
#pragma omp target teams
{
A[0] = ...; // Access needs guarding in SPMD mode.
< SPMD amenable code >
B[0] = ...; // Access needs guarding in SPMD mode.
#pragma omp parallel
{ ... }
}
(a) Generic mode region with side-effects in the sequential part interleaved with SPMD amenable code, e.g. side-effect free code.

#pragma omp target teams
{
if (omp_get_thread_num() == 0)
A[0] = ...;
#pragma omp barrier
< SPMD amenable code >
if (omp_get_thread_num() == 0)
B[0] = ...;
#pragma omp barrier
#pragma omp parallel
{ ... }
}
(b) Local guarding as described by [11] which requires up to two barriers per guarded instruction to ensure consistency.

#pragma omp target teams
{
< SPMD amenable code >
if (omp_get_thread_num() == 0)
A[0] = ...;
B[0] = ...;
}
#pragma omp barrier
#pragma omp parallel
{ ... }

(c) Optimized code with a single guarded region generated through grouping of side-effect instructions prior to guarding.

Fig. 7: Generic mode region (top) that is amenable to SPMDization together with the naive guarding scheme [11] (middle) and our optimized result (bottom).

In this work we optimize runtime calls statically if the queried execution characteristics are known through OpenMP-aware inter-procedural program analysis. So far, our optimization tries to replace the following four categories of runtime calls with constant values.

Execution Mode The runtime often handles generic and SPMD mode differently and therefore checks the status in various places. If the execution mode of all target regions reaching a check is known and the same, execution mode checks can be replaced by constants.

Parallel Level The parallel level is required to version ICV values in OpenMP and to identify nested parallelism. Tracking the parallel level and replacing dynamic checks with constant values, allows to remove the sequential fallback code path that would be required for nested parallelism.

Thread Execution Runtime calls executed in generic mode need to determine if they are reached by the main thread in a sequential code region or by many threads in a parallel region. We already track this information to create static shared memory allocations and can consequently use it to simplify runtime checks for which the result is statically known.

Launch Parameters If the user provided a constant team or thread (limit) as part of the target directive we can statically replace queries of the GPU grid with such values. This is especially important to improve work sharing loops and reductions.

D. Compiler Remarks and User Assumptions

All optimizations described in this work come with optimization remarks that inform and guide the user. The remarks are identified by unique numbers and briefly describe the performed transformation or missed opportunity. Each remark identifier has a webpage\footnote{All remark pages and information on how to extract the remarks is available via https://openmp.llvm.org/remarks/OptimizationRemarks.html} with further explanation, an example code, and actionable suggestions on how to improve the code in case the remark represents a missed optimization opportunity. Figure \ref{fig:example_remarks} shows how remarks could look for the example configuration discussed in Figure \ref{fig:example_mode}.

The webpages for remarks identifying missed opportunities describe the problem in more detail and, if available, suggest ways to introduce domain knowledge into the compilation for better static optimization results. Such information is usually provided in form of C/C++ attributes (e.g., __attributes__ (noescape)), or the OpenMP 5.1 assumption directive we integrated into LLVM/Clang for this work. As an example, the webpage suggests enclosing problematic functions, e.g., ones defined in a separate translation unit, that are known to be amenable to SPMDization, thus that can be safely executed by all threads in a team and not only the main thread, with

```
#pragma omp target teams
{
A[0] = ...;
B[0] = ...;
}
```

\begin{verbatim}
double device_function(float Arg) {
  double
  ...
}
```

double Lc1;
```

Fig. 8: Example remarks issues for the scenario illustrated in Figure \ref{fig:example_mode}. The remark identifier is shown in brackets.

V. EVALUATION

For our experiments we used a single GPU of a SuperMicro SuperServers (1029GQ-TVRT) containing 4x NVIDIA Tesla V100 GPUs (SXM2 w/32GB HBMs) and 2x Intel Xeon Gold 6152 CPUs (2.10GHz) bundled with 192GB DDR4 RAM. We used CUDA 10.1 for all experiments and collected kernel times with nvprof. Benchmarks were compiled with nvcc 10.1 (+ gcc 7.5.0), LLVM/Clang 12.0.1, and a development version based on LLVM/Clang git: 29a3e3dd7bed. All benchmarks used in this work are available in the accompanying artifact \footnote{https://openmp.llvm.org/remarks/OptimizationRemarks.html}.

A. Benchmarks

For our performance study we looked at four scientific proxy applications, part of the ECP proxy applications suite, and compared the performance of their main GPU kernel...
in different configurations. The selected proxy applications model real scientific workloads, are developed by domain scientists, and are used across the HPC community to validate compilers, runtimes, and systems in production. Each is executed as presented by the original developers, with no modification to their source code. On the other hand, we avoid Rodinia benchmarks [22] since the small subset of OpenMP target codes does not proxy real GPU applications. Additionally, the SPEC Accel v1.3 benchmarks contain, by design, exclusively SPMD regions without globalized variables to optimize; this is because all compilers performed poorly otherwise. The SPEChpc benchmark suite was not released into the applications is given in the following:

### XS Bench and RS Bench

are two proxy applications for the Open Monte Carlo (OpenMC) project. OpenMC [23] simulates the transport of neutrons and photons using the Monte Carlo methodology. Both proxy applications compute the continuous energy macroscopic neutron cross-section lookup when studying neutron transport and both are available in multiple programming languages and frameworks. While XS Bench [24] extracts one of the main kernels in OpenMC, which is in our setup memory bound, RS Bench [25] provides a compute bound alternative implementation.

**SU3 Bench** is a micro benchmark intended to explore the maturity and strategies of different programming models, HPC architectures and systems (e.g., understand the maturity and evolution of compilers). This application implements a sparse matrix-matrix multiply routine for the Special Unitary group of order 3 (hence SU(3)). This kernel is extracted from MLIC-Lattice QCD [26], an application to study the quantum chromodynamics (QCD) theory that studies the strong interaction between quarks and gluons. For the OpenMP evaluation we used version 0 of the kernel which is a “native” (or CPU-style) OpenMP implementation. The other OpenMP versions (1-4), that are in style closer to the CUDA implementation, have not been investigated.

**MiniQMC** is a scientific proxy application that implements state-of-the-art Quantum Monte Carlo (QMC) algorithms solving the Schrodinger equation. It is mainly used in the study of electronics molecular structures and 2D/3D solid state systems. miniQMC is a simplified version of QMCPACK [27] distributed independently. MiniQMC is meant to test a subset of the original scientific computation in different programming models, algorithms and optimization methodologies. The mini-application simulates electrons as they flow in a volume of immobile atoms that for a bulk nickel oxide (NiO) arranged in a 3D checkboard pattern. We evaluated the performance of the batched spline evaluation as exposed by the most compute intensive kernel of the check_spo_batched executable.

### B. Optimization Opportunities

Figure 9 shows how often our optimizations triggered for the selected kernels. There were no missed optimization opportunities. All globalized variables were successfully replaced by stack or shared memory. We expect that some of the 18 shared memory variables for miniQMC will be placed on the stack as our analysis matures. Both generic mode kernels (SU3 Bench and miniQMC) were successfully converted to SPMD mode, thus eliminating the need for the custom state machines we could generate that did not require function pointers for comparisons or indirect calls. Remarks are emitted for all successful optimizations except for runtime call optimizations because runtime calls might not originate in user code.

---

<table>
<thead>
<tr>
<th>Build</th>
<th>Time (s)</th>
<th>SMem (KB)</th>
<th># Regs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>XS Bench:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA (NVCC)</td>
<td>1.95</td>
<td>0.043</td>
<td>30</td>
</tr>
<tr>
<td>CUDA (Clang Dev $P$)</td>
<td>2.09</td>
<td>0.043</td>
<td>32</td>
</tr>
<tr>
<td>LLVM 12</td>
<td>26.59</td>
<td>1.0</td>
<td>154</td>
</tr>
<tr>
<td>LLVM Dev $P$</td>
<td>1.99</td>
<td>2.4</td>
<td>255</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Build</th>
<th>Time (s)</th>
<th>SMem (KB)</th>
<th># Regs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SU3 Bench:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA (NVCC)</td>
<td>0.35</td>
<td>0.047</td>
<td>32</td>
</tr>
<tr>
<td>CUDA (Clang Dev $P$)</td>
<td>0.35</td>
<td>0.047</td>
<td>32</td>
</tr>
<tr>
<td>LLVM 12</td>
<td>0.75</td>
<td>1.0</td>
<td>144</td>
</tr>
<tr>
<td>LLVM Dev $P$</td>
<td>0.49</td>
<td>2.4</td>
<td>170</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Build</th>
<th>Time (s)</th>
<th>SMem (KB)</th>
<th># Regs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MiniQMC:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA (NVCC)</td>
<td>0.081</td>
<td>0.1</td>
<td>26</td>
</tr>
<tr>
<td>CUDA (Clang Dev $P$)</td>
<td>0.082</td>
<td>0.1</td>
<td>26</td>
</tr>
<tr>
<td>LLVM 12</td>
<td>2.6</td>
<td>1.1</td>
<td>70</td>
</tr>
<tr>
<td>LLVM Dev $P$</td>
<td>0.29</td>
<td>0.035</td>
<td>40</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Build</th>
<th>Time (s)</th>
<th>SMem (KB)</th>
<th># Regs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MiniQMC:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA (NVCC)</td>
<td>0.24</td>
<td>1.1</td>
<td>254</td>
</tr>
<tr>
<td>CUDA (Clang Dev $P$)</td>
<td>0.47</td>
<td>0.047</td>
<td>196</td>
</tr>
</tbody>
</table>

---

---

---

---
Fig. 11: Performance evaluation for the four proxy applications (ref. Section V-A) on an NVIDIA V100 GPU normalized to performance of LLVM 12. Values over 1 indicate improvements and values below denote slowdowns. Our LLVM Dev branch (/code_fork) performance is marked explicitly as we restricted each plot to the configurations that impact performance.

C. Performance Analysis

To evaluate the effect of the different optimizations we measured the kernel time with various configurations that enable only a subset at a time. The results are shown in Figure 11 relative to the LLVM 12 baseline. CUDA results, compiled with the same LLVM/Clang we used for OpenMP experiments as well as with NVIDIA's nvcc, are given as a watermark, except for the OpenMP-only miniQMC kernel. The numbers on top of the bars represent speedup over the LLVM 12 (base) performance. The performance of the LLVM Development branch are indicated in each plot (LLVM Dev P). The globalization optimizations (heap-2-stack and heap-2-shared, ref. Section IV-A) are together referred to as h2s\(^2\) for brevity. Custom state machine generation (ref. Section IV-B) is denoted as CSM and runtime call optimization (ref. Section IV-C) is abbreviated by RTOpt. To keep the plots concise we only show configurations of optimizations that triggered. In the following results for each benchmark are discussed in detail.

**XSBench** compiled for OpenMP offloading with LLVM 12 took roughly twice as long as the CUDA alternative. By disabling our explicit OpenMP optimizations in the LLVM Development branch this gap considerable diminished and OpenMP reaches almost 80% of the CUDA performance. We traced roughly 20% of the improvement to our changes in LLVM/Clang implementing the simplified globalization scheme, shown in Figure 4c, which replaced the complicated alternative (Figure 4b). While we are investigating the remaining difference we expect at least some to be caused by generic LLVM advances. With our explicit OpenMP optimizations the performance drops to 1.53× of the LLVM 12 base. This is a secondary effect caused by the replacement of runtime calls with a constant value which shows the fragility of compiler heuristics.

**RSBench** reached only 7.3% of CUDA performance when compiled with LLVM 12. As described in Section IV-A we eliminated the erroneous and unconditional use of stack memory in SPMD mode kernels as part of this work. Without our OpenMP specific optimizations this caused increased memory usage resulting in an out-of-memory (OOM) error, or, with an increased heap-size (i.e. LIOMPRTARGET_HEAP_SIZE), tremendous slowdowns. However, as heap-2-stack is applied...
all 7 globalized variables are converted to stack locations and performance soars to 97% of our watermark. Runtime call optimization improves the result further causing the LLVM Development branch with OpenMP offloading to reach 98% of the CUDA performance.

**SU3Bench** started out at 3% of the performance demonstrated by the CUDA version. Disabling OpenMP optimizations showed a slowdown in parts caused by the simplified globalization scheme. Successful application of heap-2-stack, together with runtime call optimization and a custom state machine, recovered most of that regression. As the kernel has a very lightweight parallel region the overhead of any generic mode execution would cause performance problems. However, once SPMDzation was applied we observed a 10.8× speedup over our baseline which brings the LLVM Development branch performance to roughly a third of the CUDA version.

**miniQMC** exhibits a significant slowdown without our optimizations caused by the missing coalescing in our simplified globalization scheme. As heap-2-stack only triggers for a fraction of the variables for now, it could only improve performance slightly. Once heap-2-shared is enabled as well the performance recovers to LLVM 12 levels. Using a runtime call optimization and especially a custom state machine without function pointer involvement will provide a 1.6× speedup. Replacing the latter with SPMDzation increases this improvement to 2.26×.

### VI. Conclusion

In this work we presented a novel set of methods and techniques for efficient execution of OpenMP on GPUs by enhancing compiler-based optimization in LLVM. Specifically, our approach proposes novel inter-procedural semantic analysis of OpenMP at the LLVM-IR level that enables key transformations to optimize performance while preserving correct execution, including memory placement optimizations, SPMD mode execution, and constant folding of OpenMP runtime function calls. Moreover, we expose the results of analyses and optimizations to the developer, through compiler optimizations remarks that provide actionable advice to improve performance via source code changes. Results on a set of scientific proxy application kernels show that our novel analyses and optimizations significantly improve kernel execution time compared to previous LLVM versions. With speedups up to a factor of 13.35× we can often reduce the performance gap compared to non-portable CUDA implementations considerably while using generic GPU reasoning that applies to GPUs of other vendors as well.

For future work, we plan to extend both semantic analysis and data-flow analysis for OpenMP in LLVM to design and implement more optimizations with the goal of achieving parity with native kernel programming models. Specifically, we intend to expand SPMD mode optimizations further using kernel fusion techniques and optimize memory accesses through coalescing techniques.

### ACKNOWLEDGMENTS

Part of this research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation’s exascale computing imperative.

Part of this research was supported by the Lawrence Livermore National Security, LLC (“LLNS”) via MPO No. B642066. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

This work was partially supported by LLNL under Contract DE-AC52-07NA27344 (LLNL-CONF-826728) through the LLNL-LDRD Program Project No. 21-ERD-018.

### APPENDIX

#### A. Abstract

Our artifact provides the benchmarks used to evaluate the inter-procedural OpenMP optimizations implemented for this work. These benchmarks were evaluated using LLVM 12.0.1 as the baseline against a development branch of LLVM containing our changes with CUDA version 11.0. All but one of these patches have landed upstream, so any build of LLVM containing the commit hash 29a3e3dd7bed should be sufficient for general testing. Evaluation was done on a single Nvidia V100 GPU node, only kernel time was considered for benchmarking to measure the impact of our optimizations on the GPU.

#### B. Description

This artifact contains the benchmarks and some scripts to build an OpenMP offloading compatible LLVM compiler. The benchmarks are taken directly from their repositories and only the build systems have been modified to build for V100 GPUs with LLVM OpenMP offloading.

1) **Artifact check-list:**

- **Algorithm:** Inter-procedural optimization using OpenMP runtime knowledge.
- **Program:** miniQMC, XSSBench, RSBench, and SU3Bench compiled with Clang.
- **Compilation:** Clang with OpenMP Nvidia offloading post commit 29a3e3dd7bed, Approximately August 5th 2021.
- **Transformations:** OpenMP runtime call and general code transformation.
- **Hardware:** Tests were run using an Nvidia V100 GPU, compute capability sm_70 on a Linux system.
- **Software:** Tests were run using CUDA 11.0 for the and LLVM release 12.0.1 with OpenMP offloading as the baseline compiler.
- **Metrics:** Results were measured as the total time spent in GPU kernels via nvprof.
- **How much time is needed to prepare workflow (approximately)?**: A clean LLVM build should take under one hour.
- **Publicly available?**: Yes.
2) How to Access: Our benchmarks and associated helper scripts for this artifact are available at the following link (https://doi.org/10.5281/zenodo.5791919).

3) Hardware Dependencies: Our benchmarks were run on an Nvidia V100 GPU, whose compute capability is sm_70. Running these tests will require a GPU accelerated system with functional offloading via the CUDA RTL.

4) Software Dependencies: Building and running all the benchmarks requires an up-to-date CUDA installation (We used version 11.0), libelf, at least CMake version 3.17, and a BLAS/LAPACK library in addition to the standard dependencies for building LLVM listed at (https://llvm.org/docs/GettingStarted.html#software).

C. Installation

We have provided a script to assist in building an OpenMP offloading compatible version of the LLVM compiler that contains our contributions. Running the script will attempt to build a functioning compiler that supports OpenMP offloading. We have included a script to build the baseline LLVM 12.0.1 release, and the version tested.

```bash
$ ./build.llvm
$ ./build.llvm12.sh
```

If the build completed without errors, add the newly installed compiler to your environment.

The scripts build.sh and run.sh are provided to build and run the OpenMP offloading and CUDA versions for each benchmark. The workflow can be modified to perform individual tests.

To build the benchmarks, run:

```bash
$ ./build.sh
```

For a simple run of all benchmarks using nvprof run:

```bash
$ ./run.sh
```

D. Evaluation and Expected Results

The expected results should show improvements in execution time compared to the LLVM 12.0.1 release for all applications. Each build should also show remarks indicating which optimizations were triggered. The optimizations triggered should match those described in the paper except for SU3Bench, due to the missing pull request that is in progress of upstreaming.

E. Experiment Customization

The experiments can be customized as we did using special LLVM flags to disable certain features, these flags are:

- `openmp-opt-disable-spmdization`
- `openmp-opt-disable-deglobalization`
- `openmp-opt-disable-state-machine-rewrite`
- `openmp-opt-disable-folding`

Flags can be added to the Makefile for each benchmark, or to the CMake configuration for miniQMC.

F. Notes

The SU3Bench evaluation done in the paper uses a patch that has not landed yet, (https://reviews.llvm.org/D102107). This means that the local variables will not be put in stack memory, and will be placed in shared memory with HeapToShared. Some results will vary because of the moving nature of LLVM.

REFERENCES


