"The final publication is available at ACM via https://doi.org/10.1145/3378678.3391881"

OpenMP to CUDA graphs: a compiler-based transformation to enhance the programmability of NVIDIA devices

Chenle Yu
chenle.yu@bsc.es
Barcelona Supercomputing Center

Sara Royuela
sara.royuela@bsc.es
Barcelona Supercomputing Center

Eduardo Quiñones
eduardo.quinones@bsc.es
Barcelona Supercomputing Center

ABSTRACT

Heterogeneous computing is increasingly being used in a diversity of computing systems, ranging from HPC to the real-time embedded domain, to cope with the performance requirements. Due to the variety of accelerators, e.g., FPGAs, GPUs, the use of high-level parallel programming models is desirable to exploit the performance capabilities of them, while maintaining an adequate productivity level. In that regard, OpenMP is a well-known high-level programming model that incorporates powerful task and accelerator models capable of efficiently exploiting structured and unstructured parallelism in heterogeneous computing. This paper presents a novel compiler transformation technique that automatically transforms OpenMP code into CUDA graphs, combining the benefits of programmability of a high-level programming model such as OpenMP, with the performance benefits of a low-level programming model such as CUDA. Evaluations have been performed on two NVIDIA GPUs from the HPC and embedded domains, i.e., the V100 and the Jetson AGX respectively.

KEYWORDS

CUDA Graphs, OpenMP, Programmability, Compiler Optimization

1 INTRODUCTION

Heterogeneous computing is being deployed in a diverse range of computing systems, varying from HPC to embedded computing. This is especially relevant in application domains such as automotive, manufacturing equipment or aerospace among others, to cope with the high-performance requirements of the most advanced system functionalities (e.g., autonomous driving). As a result, current embedded processor architectures combine general purpose multi-core systems with accelerators, such as Graphics Processing Units (GPU) and Field Processing Gate Arrays (FPGA), the so-called Multi-Processor Systems-on-Chip, MPSoC.

However, these architectures are typically programmed using diverse low-level models (e.g., CUDA, OpenCL), which reduces the programmability and portability of the system, forcing programmers to (1) tune the applications to the target architectures, and so (2) master different languages and understand the peculiarities of each system. Consequently, the adoption of new embedded architectures becomes difficult. The use of high-level parallel programming models is therefore desirable to leverage the computational capabilities of the different processor architectures.

OpenMP is a high-level parallel programming model that has become the de-facto standard for shared memory systems in HPC by virtue of its productivity, including programmability, portability and performance. Moreover, in recent years, there is a growing interest in the embedded domain as well. The reason is that OpenMP includes a very powerful *tasking model* capable of supporting highly-dynamic and unstructured parallelism, augmented with an *accelerator model* for heterogeneous systems. These two models allow the representation of OpenMP programs in the form of a *Task Dependency Graph* (TDG) in which nodes are tasks (to be executed either in the host or in the target device), and edges are data dependencies between tasks.

Recent works have proposed the extraction of the TDG by means of compiler analysis techniques to bound the amount of memory used by the OpenMP run-time [29], and characterize the timing behavior of both dynamic and static scheduling approaches [25, 26]. This paper proposes to further exploit the expressiveness of the TDG representation to enhance the programmability of NVIDIA GPUs through OpenMP. CUDA 10 introduced support for CUDA graph, i.e., an unstructured sequence of CUDA kernels or memory operations connected by dependencies. Graphs enable a *define-once-run-repeatedly* execution flow that reduces the overhead of kernel launching. They also allow a number of optimizations because the whole workflow is visible, including execution, data movement, and synchronizations. They do so by separating the definition of the graph from its execution.

Interestingly, the TDG representation of the OpenMP tasking and accelerator models and CUDA Graphs have several similarities: (1) the creation, instantiation, and execution of a task/kernel is uncoupled, (2) tasks/kernels are synchronized at specific points in the code (e.g., the `taskwait` construct in OpenMP, or the *cudaStreamSynchronize* call in CUDA), and (3) both models allow defining a data-flow execution model of tasks (in OpenMP) or kernels (in CUDA) that can be mapped into a TDG.

Concretely, this paper presents a novel compiler transformation that translates OpenMP code into CUDA graphs. The contributions are the following: (1) an intermediate representation in the form of static TDG that captures the data-flow behavior of a parallel application implemented with the OpenMP tasking/accelerator model; (2) a compiler analysis and optimization phase able to transform OpenMP target tasks into a CUDA graph under certain limitations (further explained in Section 4); (3) an evaluation that shows the potential of the static TDG to exploit parallelism from two NVIDIA platforms from the HPC and embedded domain.

2 CONVERGING PERFORMANCE AND PROGRAMMABILITY

NVIDIA introduced General-Purpose GPUs (GPGPU) in 2007 accompanied by the CUDA development environment [19], designed for GPUs. Two years later, the Khronos Group introduced its open

standard OpenCL [28], for CPU and GPU programming. While CUDA disregards portability in favor of performance, OpenCL clearly boosts portability, and performance has been shown to be similar for the two models [7]. Unfortunately, both require a steep learning curve to achieve the best performance, becoming an obstacle to deliver programmability.

Later, in 2011, NVIDIA and other companies released OpenACC [30], a user-driven, directive-based, performance-portable parallel programming model. It was designed to enhance the programmability and portability of heterogeneous systems. However, the performance capabilities of the model where still far from reaching those of CUDA [8, 21].

At that time, OpenMP [20] was already a well-known programming model for shared-memory machines by virtue of its performance [9, 27], portability and programmability [15]. These features boosted OpenMP as a gateway to explore automatic transformations of OpenMP into CUDA [1, 11, 17]. Although limited regarding the input sources accepted, the tools proved that automatic transformations lead to performance levels comparable to hand-written CUDA codes for a number of benchmarks.

In 2013, OpenMP released its accelerator model, a host-centric model in which a host device drives the execution and offloads kernels to an accelerator device. The thread model mimics that of CUDA: OpenMP threads belong to OpenMP teams, which belong to OpenMP leagues and CUDA threads belong to CUDA blocks, which belong to grids. This accelerator model, although promising [2, 10], is still far from getting performance levels similar to CUDA [14]. Overcoming this limitation, the OmpSs model [6] introduced a way to easily exploit CUDA and OpenCL kernels in C/C++ applications by augmenting them with pragmas for kernel offloading [24], while offering a convenient fast-prototyping platform.

Although centered in the productivity of HPC systems, the OpenMP tasking and accelerator models have caught the attention of the embedded systems community [4, 13]. The data-flow execution these models define allows the characterization of the program as a Task Dependency Graph, where nodes are tasks, and edges are dependencies between tasks.

The TDG of an OpenMP program is typically extracted dynamically at run-time, when the values of the tasks are instantiated and the addresses of the dependencies are solved. This mechanism can be used at runtime for dynamic scheduling based, for example, on the critical path [5]. However, in critical real-time systems, the TDG can typically be derived at compile-time [29]. This ahead of time characterization of the program allows optimizations such as bounding the amount of memory required and avoiding the use of dynamic memory by the parallel runtime [16], and enables deriving the timing properties of the system, and hence ensuring its schedulability [25]. In the same line, the OpenMP specification has been analized and augmented regarding time criticality [26] and functional safety [22].

This work takes advantage of the recently introduced CUDA graphs, in 2018, and its mapping into a Directed Acyclic Graph (DAG) equivalent to the OpenMP TDG, to enhance the productivity of current MPSoCs, considering programmability, portability and performance. Specifically, it proposes the use of the OpenMP accelerator model as a facilitator for leveraging the benefits of CUDA graphs while managing tedious tasks such as data movements, dependencies, and synchronizations.

3 THE NEED FOR HIGH-LEVEL PARALLEL PROGRAMMING MODELS

High-level parallel programming models are a *defensive programming* mechanism against low-level APIs, such as CUDA and OpenCL, because of several reasons: (1) they usually hide most of the complexities exposed to the programmer in the latter (e.g., data transfers, load balancing, etc.); (2) they offer a condensed form of expressing parallelism (generally through compiler directives), that is less error-prone than low-level APIs, which typically require rewriting many parts (if not all) of the code; and (3) they can easily be deactivated at compile time, enhancing debugging. What's more, they not only favor programmability, but also portability, still considering performance as a key aspect. There are two mainstream high-level user-directed models for offloading kernels to GPU devices: OpenMP and OpenACC. Additionally, OmpSs is also a high-level paradigm for fast prototyping that has been a forerunner of the OpenMP tasking model. In this paper, we focus on OpenMP for several reasons: (1) it targets both SMPs and heterogeneous systems, (2) it is supported by most chip and compiler vendors (including NVIDIA), and (3) it provides great expressiveness by means of an exhaustive interface. Overall, OpenMP is a well candidate to leverage the power of GPUs by virtue of its programmability.

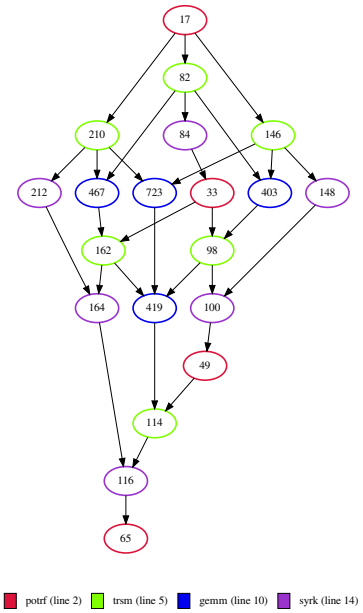
3.1 OpenMP

OpenMP describes a fork-join model where parallelism is spawned through the `parallel` construct, and distributed based on two paradigms, the *thread model* providing work-sharing constructs, e.g., `for`, and the *tasking model* providing tasking constructs, e.g., `task`. OpenMP allows synchronizing threads through synchronization constructs, e.g., `barrier` and `taskwait`, and data dependency clauses for tasks, i.e., `depend([in,out, inout])`. The tasking model defines a data-flow execution that can be characterized as a TDG. The *accelerator model*, built on top of the tasking model, allows offloading tasks to an accelerator device for further exploiting parallelism by means of device directives, e.g., `target`. These tasks, called *target tasks*, can be executed synchronously or asynchronously (i.e., using the `nowait` clause). As regular tasks, they can be represented in a TDG as well.

As an illustration, Figure 1a shows a snippet of an OpenMP Cholesky decomposition using the OpenMP tasking and accelerator models. The most computationally intensive kernel (`potrf`, at line 3) is sent to the accelerator using the `target` construct, while the others (`trsm`, `gemm` and `syrk` at lines 6, 12 and 15 respectively) are executed on the host using the `task` directive. The unstructured nature of Cholesky is expressed by means of the `depend` synchronization clauses in the `target` and `task` directives. The corresponding TDG, when considering `NB=4`, is shown in Figure 1b (it has been statically computed with the mechanisms presented in Section 4.1). Red nodes correspond to the `potrf` function executed in the device; green, blue and purple nodes correspond to the `trsm`, `gemm` and `syrk` functions, respectively, executed on the host. Each node is identified by an unique id. The edges correspond to the data dependencies among nodes as expressed by the `depend` clauses.

```
1  for (k = 0; k < NB; k++) {
2    #pragma omp target depend(inout:Ah[k][k])
3      potrf(Ah[k][k],ts,ts);
4    for (i = k + 1; i < NB; i++) {
5      #pragma omp task depend(in:Ah[k][k]) depend(inout:Ah[k][i])
6        trsm(Ah[k][k],Ah[k][i],ts,ts);
7    }
8    for (l = k + 1; l < NB; l++) {
9      for (j = k + 1; j < l; j++) {
10       #pragma omp task depend(in:Ah[k][l]) depend(in:Ah[k][j]) \
11                        depend(inout:Ah[j][l])
12         gemm(Ah[k][l],Ah[k][j],Ah[j][l],ts,ts);
13     }
14     #pragma omp task depend(in:Ah[k][l]) depend(inout:Ah[l][l])
15       syrk(Ah[k][l],Ah[l][l],ts,ts);
16   }
17 }
```

(a) OpenMP tasking source code snippet.



(b) TDG composed of 20 OpenMP tasks.

Figure 1: Cholesky decomposition using OpenMP tasking/acceleration and its corresponding TDG for `NB=4`.

3.2 CUDA Graphs

CUDA graphs, first introduced in CUDA 10, is a new model that allows defining work as a DAG rather than as single kernel operations. A CUDA graph is a set of nodes representing operations, i.e., memory operations and kernel launches, connected by edges representing run-after dependencies. Graphs enable a *define-once-run-repeatedly* execution flow by separating the definition and the execution of the graph, and allowing graphs to be launched several times. CUDA 10 includes explicit APIs for creating graphs, e.g., *cudaGraphCreate*, to create a graph; *cudaGraphAddKernelNode*/*cudaGraphAddHostNode*, to add a new node to the graph with the corresponding run-after dependencies with previous nodes, to be executed on the host/GPU; *cudaGraphInstantiate*, to create an executable graph; and *cudaGraphLaunch*, to launch an executable graph in a stream. The synchronization of graph launches is done by calling *cudastreamSynchronize* of the corresponding stream. Additionally, graphs can also be captured from an existing stream-based API (this feature is further discussed in Section 6).

Interestingly, the TDG representation of the OpenMP tasking and acceleration models and CUDA Graphs have several similarities. Figure 2 shows a snippet of the API to generate the CUDA graph of the two first nodes of the Cholesky TDG shown in Figure 1b, i.e., nodes with the unique ids 17 and 82, corresponding to the first execution of the `potrf` and `trsm` functions respectively. The former is executed on the GPU and so created with the `cudaGraphAddKernelNode` API at line 9. The API includes the function pointer (defined at line 5) and arguments (defined at line 6). Since this is the first node of the graph, no dependencies are defined (the third parameter at line 9 is set to `NULL`). The latter is executed on the host and so created with the `cudaGraphAddHostNode` API at line 18. In this case, a dependency with the previously created node 17 is set (defined by the third parameter of the `cudaGraphAddHostNode` API).

```
1  ...
2  // Creation of the kernel node 17
3  cudaGraphNode_t node_17;
4  cudaKernelNodeParams nodeArgs_17 = {0};
5  nodeArgs_17.func = (void *) potrf;
6  void *kernelArgs_17[3] = {&Ah[1][1], &ts, &ts};
7  nodeArgs_17.kernelParams=(void**)kernelArgs_17;
8  cudaGraphAddKernelNode(
9    &node_17, graph[0], NULL, 0, &nodeArgs_17);
10 // Creation of the host node 82
11 cudaGraphNode_t node_82;
12 cudaHostNodeParams nodeArgs_82 = {0};
13 nodeArgs_82.func = (void *) trsm;
14 void *hostArgs_82[4] =
15   {&Ah[1][1], &Ah[1][1], &ts, &ts};
16 nodeArgs_82.kernelParams=(void **)hostArgs_82;
17 cudaGraphAddHostNode(
18   &node_82, graph[0], &node_17, 1, &nodeArgs_82);
19 ... // Creation of the rest of the nodes
```

Figure 2: Snippet of the CUDA graph corresponding with the TDG shown in Figure 1b.

Manually generating the CUDA graph of a parallel application as the Cholesky decomposition is a very tedious and error-prone process due to the unstructured parallel nature of the application. When the number of nodes in the TDG increases (e.g., in Figure 1a, the parallel execution for `NB = 20` generates a TDG of 1540 nodes), this process becomes impossible.

4 OPENMP TO CUDA GRAPH TRANSFORMATION

This section presents a new compiler transformation that, based on the generation of a TDG at compile-time, is able to transform code using the offloading mechanism of OpenMP (i.e., the `target` directive) into CUDA graphs. This new technique has two benefits: (1) enhance the OpenMP `target` performance by avoiding offloading overheads and exploiting the *define-once-run-repeatedly* model allowed by CUDA graphs, and (2) improve the programmability and testability of NVIDIA CUDA graphs. The compiler transformation is based on first statically generating a TDG that holds all the information needed to execute the user code, and then transforming the OpenMP directives into calls to the CUDA graphs API. The following subsections describe these steps.

4.1 Augmenting the static TDG

The static generation of a TDG has already proven its applicability in embedded systems, as introduced in Section 2. The originally presented TDG [29] is constructed based on the information extracted from classic control-flow and data-flow analyses extended to represent OpenMP semantics, including (a) Parallel Control-Flow Graph (PCFG) [23] (for representing the flow of the parallel application), (b) constant propagation and common sub-expression elimination (to simplify the computation of values), (c) induction variables analysis and scalar evolution (to decide the values of variables within loops), and (d) loop unrolling (to recognize task instances).

Drowning from that mechanism, this section presents an augmented version of the TDG that enables its use to generate CUDA graphs. The modifications are listed as follows:

- A new directive with the syntax
  `#pragma omp taskgraph dims(`*integer-expr*`,` *integer-expr*`)`
  encloses the region of code that is to generate the TDG (or a subpart of the TDG) that will be transformed to a CUDA graph. This enlightens the analysis needed by the compiler to understand what part of a TDG, or complete TDG, is to be transformed into a CUDA graph, and what part is to be lowered to the SMP. However, computing the proper dimensions for a CUDA kernel is not straightforward. Hence, the directive includes the `dims` clause, which allows users to provide this information to the compiler.
- The TDG now includes regular tasks and target tasks, and also includes a new type of node that represents the `taskgraph` region.
- During the expansion process, not only dependencies are resolved, but also the data consumed by each task. This is possible when there is no aliasing, and the addresses (i.e., offsets) of each access can be computed at compile time.

Next section explains the proposed compiler transformation based on the augmented TDG.

4.2 Compiler Transformation

When the TDG of an application can be derived at compile-time, i.e., the values involved in the execution of the tasks are statically defined (which is common in critical real-time systems), and it contains all the user code that needs to be executed, i.e., all code is *taskified* within a given region, then, the TDG of that region represents an execution flow that can be perfectly mapped to a CUDA graph. Taking advantage of this, we have implemented a new stage in the Mercurium source-to-source compiler that, based on the static TDG previously described, lowers OpenMP `target` constructs into calls to the CUDA graph API. The transformation occurs, for each `taskgraph` node in the TDG, as follows:

(1) Allocate the memory structures in the device (e.g. through *cudaMalloc*).
(2) Declaration of generic variables, needed by CUDA Graph API functions.
(3) For each node within the corresponding `taskgraph`:
   (a) Set up a list of parameters, of type *cudaKernelNodeParams* or *cudaHostNodeParams*, including (i) the pointer to the kernel to be executed, (ii) the grid dimension (number of blocks), and the block dimension (threads per block) for device functions, and (iii) the required data, among others.
   (b) Associate the set of parameters with a *cudaGraphNode_t* node.
   (c) Add the node to the graph, including its dependencies, i.e., the list of previous nodes with which the new node has *run-after* dependencies.
(4) Instantiate the graph, through *cudaGraphInstantiate*.
(5) Execute the graph, via *cudaGraphLaunch*.
(6) Synchronize the graph, by calling *cudastreamSynchronize*.

In order to exploit the *define-once-run-repeatedly* execution flow allowed by the CUDA graphs model, the compiler generates extra code: it wraps the code creating the graph in a condition statement that is true only the first time this code is traversed at run-time. In this case, the graph executed is stored for use in subsequent executions. When the condition is false, the corresponding stored graph is launched and synchronized.

5 EVALUATION

This section evaluates the benefits of our proposal in terms of performance and programmability. The former shows the gain in terms of execution time of using CUDA graphs instead of OpenMP offloading for each CUDA kernel separately. The latter shows the gain, in terms of code lines, of using OpenMP instead of CUDA graphs to parallelize task-based parallel applications.

5.1 Experimental Setup

Applications. Evaluations are performed on two different applications: (1) Saxpy, Single Precision AX Plus Y, a structured benchmark using a vector of 1024x1024 elements, and blocks of 1024 elements, resulting in 1024 tasks, and (2) Cholesky decomposition, an unstructured benchmark with a matrix of 4000x4000 elements, and blocks of 20x20 elements, resulting in 1540 tasks. We have implemented the applications using the OpenMP `target` construct to offload CUDA kernels to the GPU, and we have used our automatic tool to generate the CUDA graph version of both benchmarks.

Tools. Three types of tools were used: (1) compilers, particularly Mercurium v2 [3] as source-to-source compiler for static TDG and code transformation, IBM XL 16.1.1 as back-end compiler for the OpenMP version, and NVCC v10 for the CUDA and CUDA graphs

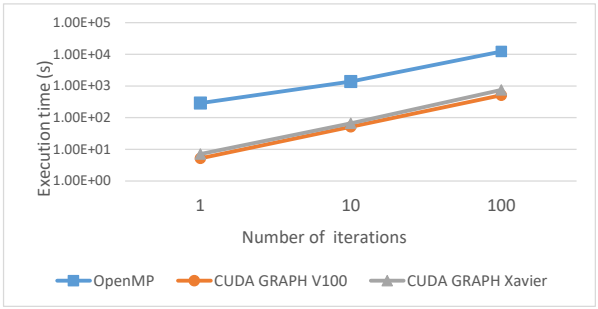
version; (2) runtimes, particularly Nanos++ implementing OpenMP plus extensions for offloading work to the GPU based on the target directive; and (3) analysis tools, particularly nvproof [18].

Platform. Experiments have been conducted in two different heterogeneous systems: (1) a node of the CTE-Power cluster, running a Linux OS, and containing 2 IBM Power9 8335-GTH processors @2.4GHz, 512GB of memory, and 4 GPU NVIDIA V100 (Volta) with 16GB HBM2, and running CUDA version 10.1, and (2) a Jetson AGX Xavier, running a Linux OS, and containing an 8-core ARM v8.2 64-bit CPU, 8MB L2 + 4MB L3, and a 512-core Volta GPU. Although it is inconceivable to use the V100 in an embedded system because of the amount of energy it consumes, we use the CTE-Power cluster because it supports OpenMP, while the Jetson does not. The CUDA graphs versions are evaluated in both architectures.

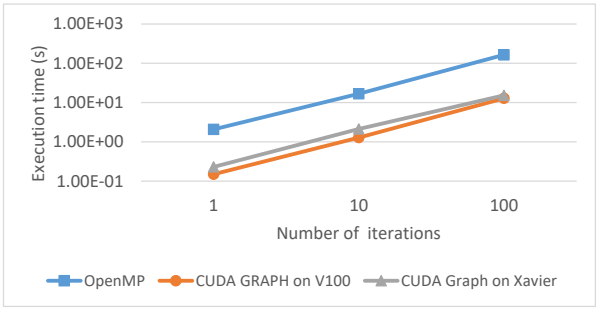
5.2 Performance evaluation

This section shows the performance evaluation of the different applications and configurations, to demonstrate the benefits, considering execution time, of exploiting CUDA graphs instead of the OpenMP native mechanism for offloading.

Figures 3a and 3b show the evolution of the execution time of the two benchmarks (Saxpy and Cholesky), when increasing the number of times the TDG executes (form 1 to 100). In both cases, the OpenMP offloading version is much slower than the automatically generated CUDA graph version. In this sense, the *nvproof* tool reveals that synchronizations take most of the time in the OpenMP version, while in the generated CUDA graphs applications, time spent in synchronizations is minimal. The increase of time, nonetheless, is exponential in both cases, indicating that, in OpenMP, the overhead of dynamically creating the TDG each time it is launched is not too high.



(a) Saxpy



(b) Cholesky

Figure 3: Execution time evolution based on the number of TDG executions for different benchmarks.

	OpenMP	*CUDA Graphs*
Saxpy	1	25
Cholesky	4	15500

Table 1: Number of lines needed to parallelize Cholesky and Saxpy depending on the parallel programming model.

5.3 Programmability evaluation

To evaluate the usefulness of our tool, we compare the effort needed to implement the presented benchmarks by transforming OpenMP to CUDA graphs, and using only CUDA graph API functions. The comparison is in terms of complexity, including the amount of code lines needed, and also the difficulty of characterizing the benchmark with the two models.

Table 1 approximates the number of lines needed to, based on a sequential version of a benchmark, produce its parallel version using OpenMP and CUDA graphs. Numbers are revealing: on one hand, Saxpy, a structured algorithm with independent tasks, allows programmers creating the CUDA graph within the original loop traversing the vector; even so, the number of lines needed to generate this version is one order of magnitude larger than using OpenMP; on the other hand, Cholesky, an unstructured algorithm where the dependencies of the different tasks/kernels change across iterations, forcing the explicit definition, not only of each node (1540 in our case), but also of their dependencies (which need additional computations to determine), reaching 4 orders of magnitude more lines than OpenMP. It is therefore unrealistic to use the CUDA graphs API to express unstructured parallelism such as that exploited in the Cholesky decomposition.

6 FURTHER DISCUSSION

This paper presents the OpenMP tasking model as an enabler to exploit unstructured kernels with CUDA graphs by means of the static generation of the Task Dependency Graph that represents the flow of the application. In this work, we also present a prototype implementation that uses the proposed `taskgraph` directive to wrap the regions of code that can be mapped to CUDA graphs. For the sake of simplicity, the prototype is limited to OpenMP codes using the tasking/accelerator models to call CUDA kernels. In this sense, other works have explored the possibility of transforming OpenMP code into CUDA kernels [1, 12, 17]. Such works, combined with ours, could transform native OpenMP codes into CUDA graphs. However, these works do not consider the accelerator model, because it was introduced later. There is no work, to the best of our knowledge, that explores the automatic transformation of the OpenMP accelerator model (including directives such as `teams`, `distribute`) into CUDA code, and this remains out of the scope of this paper.

As a second point, in this work we are using the CUDA API to exploit CUDA graphs. However, CUDA also includes another mechanism to create CUDA graphs based on capturing information on GPU activities that are submitted to the stream between the

cudaStreamBeginCapture and *cudaStreamEndCapture* calls. This mechanism could also be applied to SMPs and easily exploited with OpenMP by dynamically capturing regions of code in the form of a TDG, for later execution. This feature is remarkable in HPC where environments are highly dynamic, and the cost (both in time and energy) and the performance are paramount. The exploration of this possibility remains also as a future work.

The potential of the TDG representation is not only tied to OpenMP and CUDA graphs. Other models, such as OpenACC and OmpSs, also allow describing the parallelism as a series of kernels/tasks with dependencies, i.e., the `async` and `wait` clauses in OpenACC, and the `in`, `out`, and `inout` clauses in OmpSs. For this reason, the same TDG representation can be used to characterize the applications written with these models, and thus, similar transformations to the one presented in this paper could be applied.

Last but not least, a research line within the OpenMP tasking subgroup of the OpenMP Language Committee for the next OpenMP 6.0 specification, expected by November 2023, is the concept of *re-usable tasks* or *task graphs*. What's more, the concept of *recurrent task*, as a task containing a DAG (and maybe a period and a deadline) that is created once and can be executed several times, is also being studied within the OpenMP community for environments such as critical real-time systems [26]. These features strongly resemble the implementation of the presented `taskgraph` directive, only that the latter is executed in NVIDIA GPU by means of CUDA graphs, while the former is meant for SMPs (at least, initially).

All the interesting research lines just outlined support the importance of our contribution. The prototype presented shows, on one hand, the capabilities of the OpenMP tasking/acceleration models to describe unstructured parallelism, and, on the other hand, the potential of the TDG representation to map this highly programmable language into other task-based models like CUDA graphs. Furthermore, the evaluation presented shows the significance of using the adequate language for each platform. However, as revealed in Section 3, in some cases, such as highly unstructured algorithms, this is only possible by means of a high-level abstraction parallel programming model such as OpenMP.

ACKNOWLEDGMENTS

This work has been supported by the EU H2020 project AMPERE under the grant agreement no. 871669.

REFERENCES

[1] Muthu Manikandan Baskaran, Jj Ramanujam, and P Sadayappan. 2010. Automatic C-to-CUDA Code Generation for Affine Programs. In *CC*. 244–263.
[2] Carlo Bertolli, Samuel F Antao, Alexandre E Eichenberger, Kevin OBrien Zehra Sura, Arpith C Jacob, Tong Chen, and Olivier Sallenave. 2014. Coordinating GPU threads for OpenMP 4.0 in LLVM. In *LLVM Compiler Infrastructure in HPC*.
[3] BSC. 2020. Mercurium. https://pm.bsc.es/mcxx
[4] Barbara Chapman, Lei Huang, Eric Biscondi, Eric Stotzer, Ashish Shrivastava, and Alan Gatherer. 2009. Implementing OpenMP on a High Performance Embedded Multicore MPSoC. In *IPDPS*.
[5] Kallia Chronaki, Alejandro Rico, Rosa M Badia, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. Criticality-aware Dynamic Task Scheduling for Heterogeneous Architectures. In *ICS*.
[6] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. OmpSs: a Proposal for Programming Heterogeneous Multi-core Architectures. *Parallel Processing Letters* 21, 02 (2011).
[7] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. 2011. A comprehensive Performance Comparison of CUDA and OpenCL. In *ICPP*. 216–225.
[8] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. 2013. CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound CFD application. In *CCGRID*. IEEE, 136–143.
[9] Géraud Krawezik. 2003. Performance Comparison of MPI and Three OpenMP Programming Styles on Shared Memory Multiprocessors. In *SPAA*. 118–127.
[10] V Vergara Larrea, Wayne Joubert, M Graham Lopez, and Oscar Hernandez. 2016. Early Experiences Writing Performance Portable OpenMP 4 Codes. In *Cray User Group Meeting, London, England*.
[11] Seyong Lee and Rudolf Eigenmann. 2010. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *SC*. 1–11.
[12] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. 2009. OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization. *ACM Sigplan Notices* 44, 4 (2009).
[13] Andrea Marongiu, Paolo Burgio, and Luca Benini. 2011. Supporting OpenMP on a Multi-cluster Embedded MPSoC. *Microprocessors and Microsystems* 35, 8 (2011).
[14] Matt Martineau, Simon McIntosh-Smith, and Wayne Gaudin. 2016. Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model. In *IPDPSW*. 338–347.
[15] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. 2017. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption. In *Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*. 1–6.
[16] Adrián Munera Sánchez, Sara Royuela, and Eduardo Quiñones. 2020. Towards a Qualifiable OpenMP Framework for Embedded Systems. In *DATE*.
[17] Gabriel Noaje, Christophe Jaillet, and Michaël Krajecki. 2011. Source-to-source Code Translator: OpenMP C to CUDA. In *HPCC*. 512–519.
[18] NVIDIA. 2019. Profiler's user guide. https://docs.nvidia.com/cuda/profiler-users-guide/
[19] NVIDIA. 2019. Programming Guide :: CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/
[20] OpenMP ARB. 2018. OpenMP Application Program Interface, version 5.0. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
[21] Ruymán Reyes, Iván López, Juan J Fumero, and Francisco de Sande. 2013. A preliminary evaluation of OpenACC implementations. *The Journal of Supercomputing* 65, 3 (2013), 1063–1075.
[22] Sara Royuela, Alejandro Duran, Maria A Serrano, Eduardo Quiñones, and Xavier Martorell. 2017. A Functional Safety OpenMP* for Critical Real-Time Embedded Systems. In *IWOMP*.
[23] Sara Royuela, Roger Ferrer, Diego Caballero, and Xavier Martorell. 2015. Compiler analysis for OpenMP tasks correctness. In *CF*. 1–8.
[24] Florentino Sainz, Sergi Mateo, Vicenç Beltran, Jose L Bosque, Xavier Martorell, and Eduard Ayguadé. 2014. Leveraging OmpSs to Exploit Hardware Accelerators. In *SBAC-PAD*. 112–119.
[25] Maria A Serrano, Alessandra Melani, Roberto Vargas, Andrea Marongiu, Marko Bertogna, and Eduardo Quiñones. 2015. Timing Characterization of OpenMP4 Tasking Model. In *CASES*. 157–166.
[26] Maria A Serrano, Sara Royuela, and Eduardo Quiñones. 2018. Towards an OpenMP Specification for Critical Real-time Systems. In *IWOMP*. 143–159.
[27] Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. 2012. Performance Gaps between OpenMP and OpenCL for Multi-core CPUs. In *ICPP*. 116–125.
[28] John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering* 12, 3 (2010).
[29] Vargas, Roberto E and Royuela, Sara and Serrano, Maria A and Martorell, Xavi and Quiñones, Eduardo. 2016. A lightweight OpenMP4 Run-time for Embedded Systems. In *ASP-DAC*.
[30] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. 2012. OpenACC – First Experiences with Real-world Applications. In *Euro-Par*.