OpenMP static TDG runtime implementation and its usage in Heterogeneous Computing

Chenle YU*†, Sara Royuela*, Eduardo Quiñones*
*Barcelona Supercomputing Center, Barcelona, Spain
†Universitat Politècnica de Catalunya, Barcelona, Spain
E-mail: {chenle.yu, sara.royuela, eduardo.quinones}@bsc.es

Keywords— *OpenMP, CUDA, CUDA graph.*

I. Extended Abstract

OpenMP being the standard to use in shared memory parallel programming, it offers the possibility to parallelize sequential program with accelerators by using target directive. However, CUDA Graph as a new, efficient feature is not supported yet. In this work, we present an automatic transformation of OpenMP TDG to CUDA Graph, increasing the programmability of the latter.

A. Introduction

With the ever growing number of cores on modern CPUs, applications are more likely to be designed to have high scalability, i.e. to have better utilization of the computation power. This is true in Safety-Critical Embedded System (such as ADAS, Advanced Driver Assistance System), High Performance Computing domain and many other computer related areas.

However, efficiently parallelizing an application could be challenging, since there are different existing thread APIs in different programming languages (e.g. Java thread), or in different standards, as pthread for POSIX. In order to render this process easier, Parallel Programming Models have been introduced. Among them, OpenMP is considered as the standard model in shared-memory platform, it has been widely used in the past decades thanks to its performance and programmability. Besides, CUDA as the standard pragramming model to use if we aim to fully exploit Nvidia cards' performance, although it has a steep learning curve if one is not familiar with thread/block/grid concept of it.

Interestingly, Nvidia introduced a novel task execution model named CUDA Graph [1], which has numerous similarities with Task Dependency Graph (TDG) used in OpenMP, e.g., this feature allows user to define an execution graph once, and reuse it multiple times in the future, the same execution pattern is under consideration to add into the next OpenMP specification. Consequently, having CUDA Graph supported in OpenMP could be used for the next OpenMP specification implementation, and also for increasing the programmability of CUDA Graph. Thus, we hereby present our research aiming to automatically transform from OpenMP tasking program to a CUDA Graph application.

B. Transforming OpenMP to CUDA Graph

OpenMP 4.0 has defined $target$ directive, which allows users to offload the associated task to an accelerator device (e.g. a GPU). However, it does not support CUDA Graph. In order to deploy CUDA Graph in OpenMP, we proceed as the following:

- build a Task Dependency Graph (TDG) from the source OpenMP code
- use this TDG to build CUDA Graph

The first step is accomplished in a previous work by Royuela S. [2], where the author builds the TDG at compile time (referred as static TDG) with a source-to-source compiler: Mercurium[3]. This process requires the knowledge of all task related information at compile time, such as data to consume by tasks, number of loop iterations, etc. to generate the TDG. Although the framework slows down the compilation phase of the application, it parses all $task$ constructs and their corresponding $depend$ clauses to generate intermediate files, containing execution order information of the OpenMP program. Based on this work, we will see that an implementation of CUDA Graph in OpenMP is possible.

Generating CUDA graph from the intermediate files is done with CUDA Graph API. Host function nodes are created and inserted into the graph through *cudaGraphAddHostNode* function call, while *cudaGraphAddKernelNode* is used for kernel function node. Dependencies among tasks are managed via arrays of *CudaGraphNode_t*. Finally, the CUDA Graph, mapped from static TDG, is actually instantiated by calling *cudaGraphInstantiate*, and the function *cudaGraphLaunch* is used to launch the executable graph on a specified stream.

Currently, the automatic transformation of OpenMP task program to CUDA Graph has been tested over different benchmarks that have numerous iterations. This is done because CUDA Graph was initially introduced to reduce the host-device communications overhead, the performance gain is supposed to be greater when the number of iterations increases. Secondly, cases where applications run repeatedly before ended by the user are omnipresent, especially in real-time systems. Hence, such comparison is valuable and meaningful. In figure. 1, we show the evolution of *Cholesky* decomposition execution time based on the number of iterations, over a matrix of 4000 x 4000 elements (decomposed to 1540

tasks, or *cudaGraphNode_t*). The execution time of *Cholesky* decomposition using CUDA Graph is roughly 10 times shorter than the original OpenMP program, either for 1 iteration or repetitive execution, as shown in the chart.

In addition, manually writing this example with CUDA Graph is nearly impossible: every task node operates on different data block, and needs almost 10 lines to create the node, set up the correct argument, etc., resulting to a program having more than 15 000 lines. However, such example only had 25 additional lines in our OpenMP program.

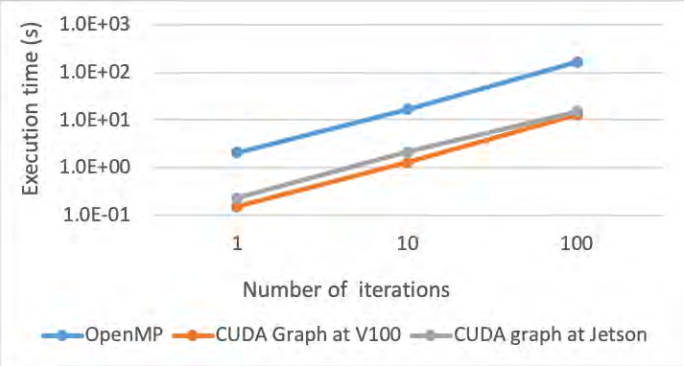


Fig. 1. Execution time evolution of Cholesky decomposition w.r.t the number of iterations

C. Further discussion

As mentioned in section I-A, next OpenMP specification is about to include reusable task graph. The transforming of OpenMP TDG to CUDA Graph fulfills such execution paradigm with the use of GPU accelerator. Regarding homogeneous programming, one implementation of such execution pattern could be: i) if all task information are known at compile time, generate the static TDG, ii) storage of the graph in memory, and execute it as many times as necessary. Based on our current work, this implementation should be straightforward, and it is in progress.

D. Conclusion

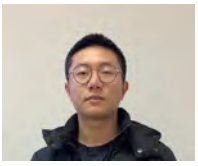
We have presented our current research work based on the concept of static TDG in OpenMP. Regarding the use of CUDA Graph, we drastically increased its programmability and the performance results satisfied our expectations. In addition, we discussed about how this implementation could be used in the reusable task graph of next OpenMP specification.

II. Acknowledgment

The realization of this work was possible thanks to the tremendous effort of my PhD advisors: Sara Royuela, Eduardo Quiñones, and Xavier Martorell. This work is part of Ampere european project.

References

[1] Nvidia, "CUDA 10 Features Revealed: Turing, CUDA Graphs, and More," 2018, https://devblogs.nvidia.com/cuda-10-features-revealed/.
[2] S. Royuela Alcazar, "High-level Compiler Analysis for OpenMP," Ph.D. dissertation, 2018.
[3] Barcelona Supercomputing Center, "Mercurium," 2019, pm.bsc.es/mcxx.



Chenle majored in HPC and Cryptography at Sorbonne University, France, during his M.S. He joined BSC in February 2019 as a Ph.D student. His research is focused on HPC, including OpenMP runtime implementation and Heterogeneous Computing.