GPU Mekong: Simplified Multi-GPU Programming using Automated Partitioning

Alexander Matz
Institute of Computer Engineering
Heidelberg University, Germany
alexander.matz@ziti.uni-heidelberg.de

Holger Fröning
Institute of Computer Engineering
Heidelberg University, Germany
holger.froening@ziti.uni-heidelberg.de

ABSTRACT

The objective of GPU Mekong is to provide a simplified path to scale out the execution of GPU programs from one GPU to almost any number, either within one host or distributed across multiple nodes. Mekong maintains the GPU's native programming model and its bulk-synchronous, thread-collective execution model. Key of Mekong is an automated partitioning, for which we use polyhedral compilation. As a result, we can maintain the simplicity and efficiency of GPU computing in the scale-out case, together with high productivity and performance. In this work, we introduce the basic concept of Mekong and provide preliminary results for a proxy application based on a 2D stencil code.

ACM Reference Format:
Alexander Matz and Holger Fröning. 2017. GPU Mekong: Simplified Multi-GPU Programming using Automated Partitioning. In *Proceedings of ACM conference, Denver, CO, 2017 (SC'17),* 3 pages.
https://doi.org/10.475/123_4

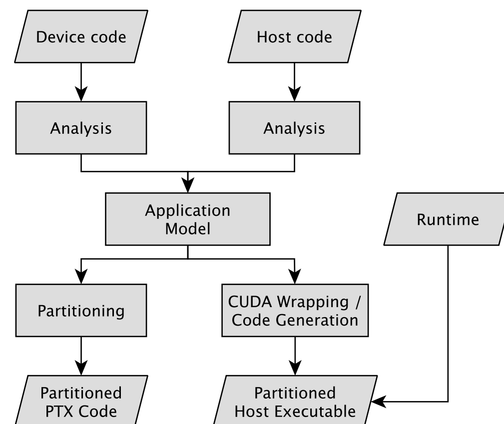

Figure 1: Compilation pipeline of Mekong

1 INTRODUCTION

GPUs have established themselves in the computing landscape, convincing users and designers by their excellent performance and energy efficiency. The inherent domain decomposition principle for these languages ensures a fine granularity when partitioning the code, typically resulting in a mapping of one single output element to one thread and reducing the need for work agglomeration. Their use has been pushed by the introduction of data-parallel languages like CUDA or OpenCL.

While these languages are rather easy to learn and use for single GPUs, programming multiple GPUs has to be done in an explicit and manual fashion that dramatically increases complexity. The user has to manually orchestrate data movements and kernel launches on the different processors. Key task in this procedure of parallelization is the problem partitioning, which today is typically done using domain decomposition. Considering that a large subset of GPU applications are regular in nature, it seems very promising to explore the use of static code analysis and transformation to implement automatic partitioning.

Such an automatic partitioning requires static analysis of host and device code. Identifying memory access patterns is the key when choosing a good partitioning. Memory access patterns of GPU kernels can be represented using a polyhedral model, respecting the inherent 3D grid of the GPU execution model. This allows to accurately identify the input and output data sets of a collection of threads (e.g. collaborative thread arrays or any consecutive partition). A tracking mechanism provides information about the location of the distributed data. This combination identifies stale data in GPU memory and allows minimizing data transfers between kernel iterations.

In this poster, we introduce the GPU Mekong project that is based on the LLVM compiler infrastructure [2] and uses polyhedral code analysis in order to model memory access patterns. Based on a single-device data-parallel program, we analyze host and device code for features to choose a partitioning strategy. The partitioning strategy itself determines how to decompose kernels into multiple sub-kernels, one for each target GPU, and how to implement communication to resolve data dependencies between GPUs. We explore the feasibility and performance of this approach using a stencil code proxy application, showing how this workload behaves in terms of strong scaling and how different components like kernel execution, communication, and dependency resolution contribute to overall execution time.

2 GPU MEKONG

Figure 1 pictures a diagram of the high level architecture of Mekong. From a high level point of view, Mekong is split into three parts: (1)

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SC'17, 2017, Denver, CO
© 2017 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

static analysis (building the application model), (2) code transformation (partitioning kernels and adapting host code) (3) runtime library (providing common routines).

The fundamental idea is to build a polyhedral model of the application, and use the results from this analysis to split one large kernel into multiple small kernels, mapping each small kernel to another GPU. As we confine a kernel to only operate on memory local to the its current device, dependencies have to be resolved on-the-fly between kernel iterations. Dependencies of a kernel execute on a given GPU can be located on either the host, other GPUs, or the local GPU. While the latter does not require transmission of any data, the former two cases require communication. Initial data distribution (`cudaMemcpyHostToDevice`) and final data collection (`cudaMemcpyDeviceToHost`) integrate seemlessly into this concept.

While we try to perform as many tasks as possible during compile time, certain tasks necessarily have to be performed at execution time. For this purpose, Mekong includes a runtime system that handles tasks like buffer management and initiating data movements.

3 POLYHEDRAL COMPILATION

The Polyhedral Model is a way to represent and reason about nested loops at a high level of abstraction [1]. It allows extensive data flow analysis regarding array usage and certain aggressive optimizations (in particular tiling) of applications.

At the heart of this model are Z-Polyhedra, which are convex sets of points in $\mathbb{Z}^n$, and relationships between these sets. In this model, the iterations of a nested loop are modeled as integer points in a multidimensional space. Each dimension represents one loop level. In addition to modeling the iteration space of a loop nest, memory accesses are modeled as a mapping from an n-dimensional iteration to an m-dimensional array index. A map can then by applied to a in iteration domain, resulting for example in a set containing the array elements that are read by a particular subset of a GPU kernel. All constants used in the description of sets and maps can be parametrized for additional flexibility.

Parametrized sets that describe the elements read and written by a particular rectangular subset of kernel threads are at the heart of Mekongs model. Mekong is limited to applications that can be represented by a polyhedral model, requiring memory accesses to be (quasi-)affine functions with constant parameters. Indirections, like they are often used in sparse linear algebra or graph computations, are currently not supported by Mekong. However, especially workloads performance numerical computations, like dense linear algebra, stencil codes or particle physics, often exhibit this regular behavior.

4 INITIAL RESULTS

For the initial performance analysis, we are using a single node system with 8 NVIDIA K80 graphics cards. As each graphics card houses two GPUs, we can perform scalability experiments with up to 16 GPUs. The GPUs are interconnected by PCIe trees to two Intel Xeon CPUs (Ivy Bridge-class). Our initial workload is based on a 2D 5-point stencil computation on a grid with a size of 32k times 32k elements. Figure 2 shows initial scaling results, including all data movements.

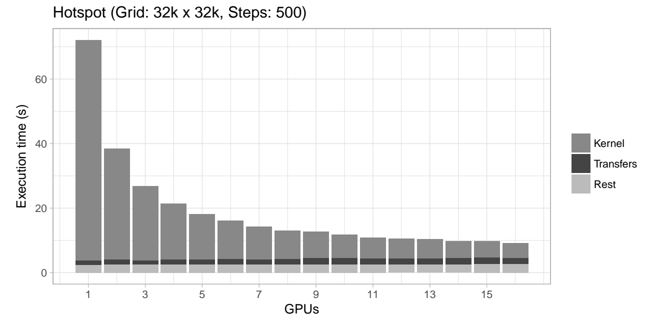

Figure 2: Initial scaling results for a 2D stencil code proxy application

As expected, the kernel execution time scales almost linearly with the number of GPUs. Overhead introduced by communication grows with the number of GPUs used and limits strong scaling for small problem sizes. However, using polyhedral code generation and a fast memory tracker for the identification and location of data dependencies helps keeping this overhead manageable.

5 RELATED WORK

The most important related work we are aware of is as follows: Lee et al. introduce an automatic system for mapping multiple kernels across multiple computing devices (MKMD) [3]. Compared to our work, they do not use polyhedral compilation and focus on scheduling optimizations instead. Pai et al. describe the use of page migration to manage distinct address spaces of general-purpose CPUs and discrete accelerators like GPUs, based on the X10 compiler and run-time [4].

6 CONCLUSION AND FUTURE WORK

We introduced the concept of GPU Mekong, a software tool that eases the programming of multi-GPU systems by using polyhedral analysis to automatically partition a single-device code for multiple GPUs. We show early results for a 2D stencil code and its scalability, demonstrating the feasibility of this approach.

In the near term, we anticipate most efforts to be spend on making the compilation stack more robust. Furthermore, array reshaping is mandatory to address scalability by optimizing buffer utilization and in turn supporting problem sizes too large for a single GPU. Also, we plan to investigate other optimizations, in particular overlap, scheduling and mapping.

7 ACKNOWLEDGMENTS

We thank Christoph Klein (Heidelberg University) for his work on previous concepts of this research. We are also thankful for the many discussions we had with Sudhakar Yalamanchili (Georgia Tech), Mark Hummel (NVIDIA), Tobias Grosser (ETHZ), and Sebastian Hack and Johannes Doerfert (Saarland University). Mekong received funding in the form of a Google Faculty Research Award and by the German Federal Ministry of Education and Research (BMBF).

REFERENCES

[1] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly - Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Vol. 2011.

[2] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. 75–86. https://doi.org/10.1109/CGO.2004.1281665

[3] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. 2015. Orchestrating Multiple Data-Parallel Kernels on Multiple Devices. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Vol. 24.

[4] Sreepathi Pai, R Govindarajan, and Matthew J Thazhuthaveetil. 2012. Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 33–42.