

Moving Heterogeneous GPU Computing into the Mainstream with Directive-Based, High-Level Programming Models (Position Paper)

DOE X-Stack Vancouver Project

<http://ft.ornl.gov/trac/vancouver>

Seyong Lee and Jeffrey S. Vetter

Oak Ridge National Laboratory, Oak Ridge, TN 37831 USA

Email: {lees2,vetter}@ornl.gov

I. INTRODUCTION

Graphics Processing Units (GPUs)-based heterogeneous systems have emerged as promising alternatives for high-performance computing. However, their programming complexity poses significant challenges for developers. Recently, several directive-based, GPU programming models have been proposed by both academia and industry ([1], [2], [3], [4], [5], [6]) to provide better productivity than existing ones, such as CUDA and OpenCL. Directive-based models provide different levels of abstraction, and programming efforts required to conform to their models and optimize the performance also vary. Understanding the differences in these new models will give valuable insights on the general applicability and performance of the directive-based GPU programming models. This position paper evaluates the existing high-level GPU programming models to identify current issues and future directions that stimulate further discussion and research to push the state of the art in productive GPU programming models.

II. DIRECTIVE-BASED, HIGH-LEVEL GPU PROGRAMMING MODELS

General directive-based programming systems consist of directives, library routines, and designated compilers. In the directive-based GPU programming models, a set of directives are used to augment information available to the designated compilers, such as guidance on mapping of loops onto GPU and data sharing rules. The most important advantage of using directive-based GPU programming models is that they provide very high-level abstraction on GPU programming, since the designated compiler hides most of the complex details specific to the underlying GPU architectures. Another benefit is that the directive approaches provide incremental parallelization of applications, like OpenMP, so that a user can specify the regions of a host program to be offloaded to a GPU device in an incremental way, and then the compiler automatically creates corresponding host+device programs.

Several directive-based, GPU programming models have been proposed, such as OpenMPC [1], hiCUDA [2], PGI Accelerator [3], HMPP [4], R-Stream [5], OpenACC [6], etc.

To understand the level of the abstraction that each directive-based model gives, Table I summarizes the type of information that GPU directives can provide. The table implies that R-Stream offers the highest abstraction, and hiCUDA provides the lowest abstraction among the compared models, since in the R-Stream model, the compiler covers most features implicitly without a programmer's involvement, while programmers using hiCUDA should control most of the features explicitly. However, lower level of abstraction is not always bad, since low level of abstraction may allow enough control over various optimizations and the features specific to the underlying GPUs to achieve optimal performance. Moreover, actual programming efforts required to use each model may not be directly related to the level of abstraction that the model offers, and high-level abstraction of a model sometimes puts limits on its application coverage.

III. CURRENT ISSUES AND FUTURE DIRECTIONS

A. Functionality

The existing models target structured blocks with parallel loops for offloading to a GPU device, but there exist several limits on their applicability; first, most of the existing implementations work only on loops, but not on general structured blocks, such as OpenMP's parallel regions. Second, most of them either do not provide reduction clauses or can handle only scalar reductions. Third, none of existing models supports critical sections or atomic operations (OpenMPC accepts OpenMP critical sections, but only if they have reduction patterns.). Fourth, all existing models support only course-grained synchronizations. Fifth, most of them do not allow function calls in mappable regions, except for simple functions that compilers can inline automatically. Sixth, all work on array-based computations but support pointer operations in very limited ways. Last, some compilers have limits on the complexity of mappable regions, such as the shape and depth of nested loops and memory allocation patterns.

Because each model has different limits on its applicability, programming efforts required to conform to their models also differ in each model, incurring portability problems. OpenACC is the first step toward a single standard on directive-based

TABLE I

FEATURE TABLE, WHICH SHOWS THE TYPE OF INFORMATION THAT GPU DIRECTIVES CAN PROVIDE. IN THE TABLE, *explicit* MEANS THAT DIRECTIVES EXIST TO CONTROL THE FEATURE EXPLICITLY, *implicit* INDICATES THAT A COMPILER IMPLICITLY HANDLES THE FEATURE, *indirect* IS THAT PROGRAMMERS CAN INDIRECTLY GUIDE THE WAY FOR THE COMPILER TO HANDLE THE FEATURE USING DIRECTIVES, AND *imp-dep* MEANS THAT THE FEATURE IS IMPLEMENTATION-DEPENDENT.

Features		OpenMPC	hiCUDA	PGI	HMPP	R-Stream	OpenACC
Code regions to be offloaded		structured blocks	structured blocks	loops	loops	loops	loops
Loop mapping		parallel	parallel	parallel vector	parallel	parallel	parallel vector
Data management	GPU memory allocation and free	explicit implicit	explicit	explicit implicit	explicit implicit	implicit	explicit implicit
	Data movement between CPU and GPU	explicit implicit	explicit	explicit implicit	explicit implicit	implicit	explicit implicit
Compiler optimizations	Loop transformations	explicit		implicit	explicit	implicit	imp-dep
	Data management optimizations	explicit implicit	implicit	explicit implicit	explicit implicit	implicit	imp-dep
GPU-specific features	Thread batching	explicit implicit	explicit	indirect implicit	explicit implicit	explicit implicit	indirect implicit
	Utilization of special memories	explicit implicit	explicit	indirect implicit	explicit	implicit	indirect imp-dep

GPU programming, but many technical details are still left unspecified. Therefore, in-depth evaluation and research on existing models will be essential to address these issues.

B. Scalability

All existing models assume *host+accelerator* systems where one or small number of GPUs are attached to the host CPU. However, these models will be applicable only to small scale. To program systems consisting of clusters of GPUs, hybrid approaches such as *MPI + X* will be needed. To enable seamless porting to large scale systems, research on unified, directive-based programming models, which integrate data distribution, parallelization, synchronization, and other additional features, will be needed.

C. Tunability

Tuning GPU programs is known to be difficult, due to the complex relationship between programs and performance. Directive-based GPU programming models may enable an easy tuning environment that assists users in generating GPU programs in many optimization variants without detailed knowledge of the complex GPU programming and memory models. However, most of the existing models do not provide enough control over various compiler-optimizations and GPU-specific features, posing a limit on their tunability. To achieve the best performance on some applications, research on alternative, but still high-level interface to express GPU-specific programming model and memory model will be necessary.

D. Debuggability

The high-level abstraction offered by directive models puts significant burdens on debugging. The existing models do not give users an idea on how the translation works, and some implementations either do not always observe directives inserted by programmers or translate them incorrectly, if they conflict with internal compiler analyses, adding more dimensions to the complex debugging space. For debugging purpose, all existing models can generate CUDA codes as outputs, but most of existing compilers generate CUDA codes

by unparsing low-level intermediate representation (IR), which contain implementation-specific code structures and thus are very difficult to understand. More research on traceability mechanisms and high-level IR-based compiler-transformation techniques will be needed for better debuggability.

IV. SUMMARY

This paper evaluates directive-based GPU programming models. The directive-based models provide very high-level abstraction on GPU programming, since the designated compilers hide most of the complex details specific to the underlying GPU architectures. The evaluation of the existing directive-based GPU programming models shows that these models provide different levels of abstraction, and programming efforts required to conform to their models and optimize the performance also vary. The evaluation also reveals that the current high-level GPU programming models may need to be further extended to support an alternative way to express GPU-specific programming model and memory model to achieve the best performance in some applications.

ACKNOWLEDGMENT

This position paper was developed with funding from the Vancouver project, X-Stack program, U.S. Department of Energy Office of Advanced Scientific Computing Research.

REFERENCES

- [1] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP programming and tuning for GPUs," in *SC'10: Proceedings of the 2010 ACM/IEEE conference on Supercomputing*. IEEE press, 2010.
- [2] T. D. Han and T. S. Abdelrahman, "hicuda: High-level gpgpu programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 78–90, 2011.
- [3] "The Portland Group, PGI Fortran and C Accelerator Programming Model [Online]. Available: <http://www.pgroup.com/resources/accel.htm>."
- [4] "HMPP Workbench, a directive-based compiler for hybrid computing [Online]. Available: <http://www.caps-entreprise.com/hmpp.html>."
- [5] "R-Stream, a High Level Compiler for embedded signal/knowledge processing and HPC algorithms [Online]. Available: <https://www.reservoir.com/rstream>."
- [6] "OpenACC: Directives for Accelerators [Online]. Available: <http://www.openacc-standard.org>."