# Exploring LLVM Infrastructure for Simplified Multi-GPU Programming

Alexander Matz[1], Mark Hummel[2] and Holger Fröning[3]

[1,3] Ruprecht-Karls University of Heidelberg, Germany,
alexander.matz@ziti.uni-heidelberg.de
holger.froening@ziti.uni-heidelberg.de

[2] NVIDIA, US
mhummel@nvidia.com

**Abstract.** GPUs have established themselves in the computing landscape, convincing users and designers by their excellent performance and energy efficiency. They differ in many aspects from general-purpose CPUs, for instance their highly parallel architecture, their thread-collective bulk-synchronous execution model, and their programming model. In particular, languages like CUDA or OpenCL require users to express parallelism very fine-grained but also highly structured in hierarchies, and to express locality very explicitly. We leverage these observations for deriving a methodology to scale out single-device programs to an execution on multiple devices, aggregating compute and memory resources. Our approach comprises three steps: 1. Collect information about data dependency and memory access patterns using static code analysis 2. Merge information in order to choose an appropriate partitioning strategy 3. Apply code transformations to implement the chosen partitioning and insert calls to a dynamic runtime library. We envision a tool that allows a user write a single-device program that utilizes an arbitrary number of GPUs, either within one machine boundary or distributed at cluster level. In this work, we introduce our concept and tool chain for regular workloads. We present results from early experiments that further motivate our work and provide a discussion on related opportunities and future directions.

## 1 Introduction

GPU Computing has gained a tremendous amount of interest in the computing landscape due to multiple reasons. GPUs as processors have a high computational power and an outstanding energy efficiency in terms of performance-per-Watt metrics. Domain-specific languages like OpenCL or CUDA, which are based on data-parallel programming, have been key to bring these properties to the masses. Without such tools, graphical programming using tools like OpenGL or similar would have been too cumbersome for most users.

We observe that data-parallel languages like OpenCL or CUDA can greatly simplify parallel programming, as no hybrid solutions like sequential code enriched with vector instructions is required. The inherent domain decomposition

principle ensures finest granularity when partitioning the problem, typically resulting in a mapping of one single output element to one thread. Work agglomeration at thread level is rendered unnecessary. The Bulk-Synchronous Parallel (BSP) programming paradigm and its associated slackness regarding the ratio of virtual to physical processors allows effective latency hiding techniques that make large caching structures obsolete. At the same time, a typical code exhibits substantial amounts of locality, as the rather flat memory hierarchy of thread-parallel processors has to rely on large amounts of data reuse to keep their vast amount of processing units busy.

However, this beauty of simplicity only is applicable to single-GPU programs. Once a program is scaled out to any number of GPUs larger than one, the programmer has to start using orthogonal orchestration techniques for data movement and kernel launch. These modification are scattered throughout host and device code. We understand the efforts behind this orchestration to be high and completely incompatible with the single-device programming model, independent if these multiple GPUs are within one or multiple machine boundaries.

With this work we introduce our efforts on GPU Mekong[1]. Its main objective is to provide a simplified path to scale-out the execution of GPU programs from one GPU to almost any number, independent if the GPUs are located within one host or distributed at cloud or cluster level. Unlike existing solutions, this work proposes to maintain the GPU's native programming model, which relies on a bulk-synchronous, thread-collective execution. No hybrid solutions like OpenCL/CUDA programs combined with message passing are required. As a result, we can maintain the simplicity and efficiency of GPU Computing in the scale-out case, together with high productivity and performance.

We base our approach on compilation techniques including static code analysis and code transformations regarding host and device code. We initially focus on multiple GPU devices within one machine boundary (single computer), allowing avoiding efforts regarding multi-device programming (cudaSetDevice, streams, events and similar). Our initial tool stack is based on OpenCL programs as input, LLVM as compilation infrastructure and a CUDA backend to orchestrate data movement and kernel launches on any number of GPUs. In this paper, we make the following contributions:

1. A detailed reasoning about the motivation and conceptual ideas of our approach, including a discussion of current design space options
2. Introduction of our compilation tool stack for analysis passes and code transformation regarding device and host code
3. Initial analysis of workload characteristics regarding suitability for this approach

[1] With Mekong we are actually referring to the Mekong Delta, a huge river delta in southwestern Vietnam that transforms one of the longest rivers of the world into an abundant number of tributaries, before this huge water stream is finally emptied in the South China Sea. Similar to this river delta, Mekong as a project gears to transform a single data stream into a large number of smaller streams that can be easily mapped to multiple GPUs.

4. Exemplary performance analysis of the execution of a single-device workload on four GPUs

The remainder of this work is structured as follows: first, we establish some background about GPUs and their programming models in section 2. We describe the overall idea and our preliminary compilation pipeline in section 3. In section 4, we describe the characteristics of BSP applications suitable for our approach. We present the partitioning schemes and their implementation in section 5. Our first experiments are described and discussed in section 6. Section 7 presents background information and section 8 discusses our current state and future direction.

## 2 Background

A GPU is a powerful high-core-count device with multiple Shared Multiprocessors (SMs) that can execute thousands of threads concurrently. Each SM is essentially composed by a large number of computing cores, and a shared scratchpad memory. Threads are organized in blocks, but the scheduler of a GPU doesn't handle each single thread or block; instead threads are organized in warps (typically 32 threads) and these warps are scheduled to the SMs during runtime. Context switching between warps comes at negligible costs, so long-latency events can easily be hidden.

GPU Computing has been dramatically pushed by the availability of programming languages like OpenCL or CUDA. They are mainly based on three concepts: (1) a thread hierarchy based on collaborative thread arrays (CTAs) to facilitate an effective mapping of the vast number of threads (often upwards of several thousands) to organizational units like the SMs. (2) shared memory that is an explicit element of the memory hierarchy, in essence enforcing users to manually specify the locality and thereby inherently optimizing locality to a large extend. (3) barrier synchronization for the enforcement of a logical order between autonomous computational instances like threads or CTAs.

Counter-intuitively, developing applications that utilize multiple GPUs is rather difficult despite the explicitly expressed high degree of parallelism. These difficulties stem from the need to orchestrate execution and manage the available memory. In contrast to multi-socket CPU systems, the memory from multiple GPUs in a single system is not shared and data has to be moved explicitly between the main memory and the memory of each GPU.

## 3 Concept and Compilation Pipeline

We base our approach on the following observations: first, the inherent thread hierarchy that forms CTAs allows an easy remapping to multiple GPUs. We leverage this to span up BSP aggregation layers that cover all SMs of multiple GPUs. While this step is rather straight-forward, the huge bandwidth disparity

between on-device and off-device memory accesses requires an effective partitioning technique to maximize locality for memory accesses. We use concepts based on block data movement (cudaMemcpy) and fine-grained remote memory accesses (UVA) to set-up a virtual global address space that embraces all device memory. We note that such an approach is not novel, it is well-explored in the area of general-purpose computing (shared virtual memory) with known advantages and drawbacks. What we observe as main difference regarding previous efforts, is that GPU programs exhibit a huge amount of locality due to the BSP-like execution model and the explicit memory hierarchy, much larger than for traditional general-purpose computing. We leverage this fact for automated data placement optimization.

The intended compilation pipeline is assembled using mostly LLVM tools as well as custom analysis and transformation passes. The framework targets OpenCL device code and C/C++ host code using the CUDA driver API, which allows us to implement the complete frontend using Clang (with the help of libCLC to provide OpenCL built-in functions, types, and macros).

With the frontend and code generation being handled by existing LLVM tools, we can focus on implementing the analysis and transformation passes that present the core of our work. These passes form a sub-pipeline consisting of three steps (see figure 1):

1. The first step is solely an analysis step that is applied to both host and device code. The goal is to extract features that can be used to characterize the workload and aid in deciding on a partitioning strategy. Examples for these features include memory access patterns, data dependencies, and kernel iterations. The results are written back into a database, which is used as the main means of communication between the different steps.
2. The second step uses the results from the first step in order to reach a decision on the partitioning strategy and its parameters. These parameters (for example the dimension along which the workload is partitioned), are written back into the database.
3. With the details on the partitioning scheme agreed on, the third step applies the corresponding code transformations in host and device code. This phase is highly dependent on the chosen partitioning scheme.

## 4 Workload Characterization

In order to efficiently and correctly partition the application, its specific nature needs to be taken into account (see table 2 for example features). Much the same way dependencies have to be taken into account when parallelizing a loop, data and control flow dependencies need to be considered when partitioning GPU kernels. Some of these dependencies can be retrieved by analyzing the device code while others can be identified in host code. Regarding what kind of transformations are allowed, the most important characteristic of a workload if whether it is regular or irregular.

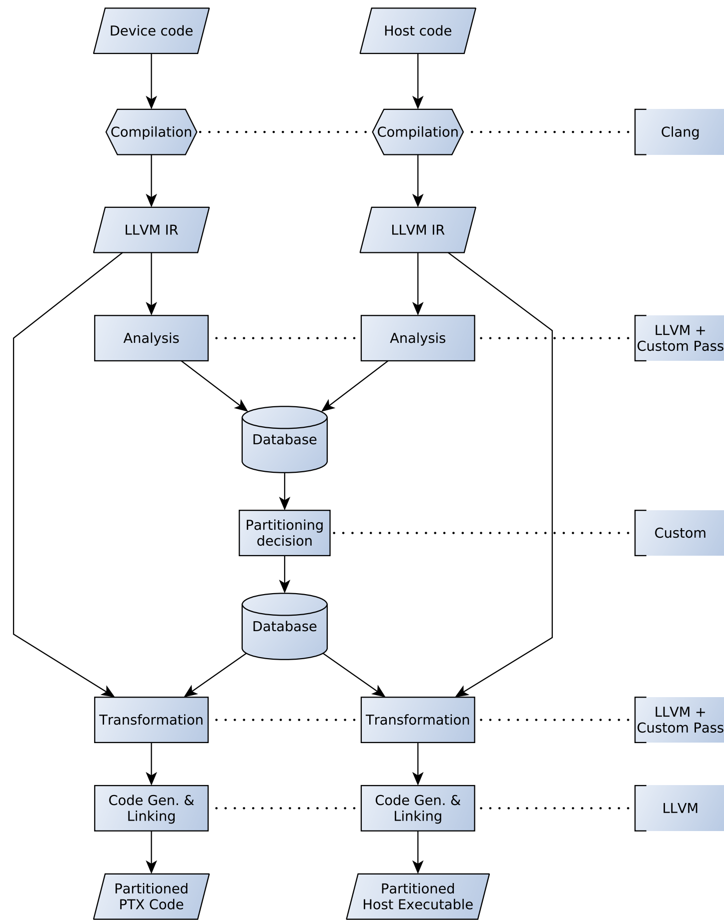


**Fig. 1.** High level overview of the compilation pipeline

Regular workloads are characterized by their well-defined memory access patterns in kernels. Well-defined in this context means that the memory locations accessed over time depend on a fixed number of parameters that are known at kernel launch time. The memory locations accessed being the result of dereferencing input data (as is the case in sparse computations for example) is a clear exclusion criterion.

They can be analyzed to a very high extend, which allows for extensive reasoning about the applied partitioning scheme and other optimization schemes. They usually can be statically partitioned according to elements in the output data. Device code can be inspected for data reuse that does not leverage shared memory and can be optimized accordingly. On the host code side, data movement and kernel synchronization are the main optimization targets.

| Workload | Classification | Data reuse | Indirections | Iterations |
|---:|---|---|---|---|
| Dense Matrix Multiply | Regular | High | 1 | 1 |
| Himeno (19 point stencil) | Regular | High | 1 | many |
| Prefix sum | Regular | Low | 1 | log2(N) |
| SpMV/Graph traversal | Irregular | Low | 2 | many |

**Fig. 2.** Characterization of selected workloads

## 5 Analysis and Transformations

This section details some of the analysis and transformations that build the core of our project. The first sub section focuses on the analysis phase, where the applicable partitioning schemes are identified and one of them is selected, while the second sub section goes into how and which transformations are applied.

As of now we focus on implementing a reasonably efficient 1D partitioning. Depending on how data movement is handled, it can be divided into further sub-schemes:

**UVA** This approach leverages NVIDIA Unified Virtual Addressing (UVA), which allows GPUs to directly access memory on different GPUs via peer-to-peer communication. It is the easiest algorithm to implement and both host and device code only need small modifications. For all but one device, all data accesses are non-local, resulting in peer-to-peer communication between GPUs. With this scheme, the data set has to fully fit into a single GPU.

**Input replication** Input replication is similar to the UVA approach in that it does not require any data reshaping and the device code transformations are exactly the same. But instead of utilizing direct memory access between GPUs, input data gets fully replicated among device. Results are written into a local buffer on each device and later collected and merged by the host code.

**Streaming** This is the approach that we suspect solves both the performance and problem size issues of the first two approaches. Both input and output data are divided into partitions and device buffers are reshaped to only hold a single partition at a time. This strategy requires extensive modifications on both host and device code but also presents more opportunities to optimize data movements and memory management.

### 5.1 Analysis

In order to identify viable partitioning schemes, the analysis step extracts a set of features exhibited by the code and performs a number of tests that, if failed, dismiss certain partitioning schemes.

Since for now we focus on regular workloads the most important test performed determines the number of indirections when accessing global memory. One indirection corresponds to a direct memory access using a simple index. This index can be the result of a moderately complex calculation as long as

none of the values used in the calculations themselves have been read from global memory. Every time the result of one or more load instructions is used to calculate an index it counts as another level of indirection.

For regular workloads, where we know the data dependencies in advance, any level of indirection that is greater than one dismisses the workload for partitioning. Although this might seem like a massive limitation, a number of workloads can be implemented using only one level of indirection. Examples include matrix multiplications, reductions, stencil codes, and n-body (without cluster optimizations).

If the code is a regular workload, certain features that help deciding on a partitioning scheme are extracted. Useful information includes:

- Maximum loop nesting level
- Minimum and maximum values of indices
- Index stride between loop iterations
- Index stride between neighboring threads
- Number and index of output elements

For the partitioning schemes we are currently exploring, we require the output elements to be distinct for each thread (i.e. no two threads have the same output element). Without this restriction, access to output data would have to be kept coherent between devices.

The index stride between neighboring threads on input data is used in order to determine partition shapes and sizes. For UVA and Input Replication this is not relevant, but streaming kernels with partitioned input data should only be partitioned across "block-working-set" boundaries. The analysis of the index stride between loop iterations gives a deeper insight into the nature of the workload. As an example, in a non-transposed matrix multiplication the loop stride in the left matrix is 1 while it is the matrix width in the right matrix.

All extracted features will be used to train a classifier that later provides hints for proven-good partitioning schemes.

### 5.2 Transformation

The transformation phase is highly dependent on the chosen partitioning scheme. Host code of the kind of CUDA applications we are looking at usually follows the following formula:

1. Read input data
2. Initialize device(s)
3. Distribute data
4. Launch kernels
5. Read back results
6. Produce output
7. Clean up

The relevant parts of this code are steps 2 through 5. Iterative workloads have the same structure, but repeat steps 3 through 5.

Host code modifications currently consist of replacing the regular CUDA calls with custom replacements that act on several GPUs instead of a single one. These are the functions that are implemented differently depending on the partitioning scheme. As an example, for 1D UVA based partitioning, cuMalloc allocates memory only on a single GPU, but enables peer-to-peer access between that and all other visible GPUs. In contrast, for 1D Input Replication based partitioning, cuMalloc allocates memory of the same size on all available GPUs. In all cases the kernel configuration is modified to account for the possibly multiple device buffers and the new partitioned grid size.

For device code we currently employ a trick that greatly simplifies 1D partitioning for UVA and Input Replication schemes. The regular thread grid is embedded in a larger "super" grid that spans the complete workload on all devices. The super ID corresponds to the device number of a GPU within the super grid and the super size corresponds to the size of a single partition. These additional parameters are passed as extra arguments to the kernel. With this abstraction, transformations on the device code are limited to augmenting the function to accept these arguments as well as replacing calls to get_global_id. All calls to get_global_id that query for the dimension we are partitioning along, get replaced with the following computation: `super_id*super_size + get_global_id(<original arguments>)`. This way, no index recalculations have to be performed, as the kernel is essentially the same just with the grid shrunken on and shifted along the partitioned dimension. In order to distribute data for the input replication scheme, regular CUDA memcpys are employed. So far this does not pose a problem, since all GPUs involved are part of the local system.

## 6 Early Experiments

In order to proof-of-concept our ideas, we did preliminary experiments with the first implementation of our toolstack that implements automatic 1D UVA and Input Replication partitioning from section 5. The workload in question is a relatively naive square matrix multiply with the only optimization being the use of 32x32 tiling. It has been chosen due to its regular nature and high computation to communication ratio.

The experiments have been executed on a single node system equipped with two Intel Xeon E5-2667 v3 running at 3.20Ghz, 256GB of DDR3 RAM, and a set of 8 NVIDIA K80 GPUs (each combining 2 GK210 GPUs). The systems runs on Ubuntu 14.04.

As can be seen in figure 3, even with the high compute to memory ratio of a matrix multiply, a UVA based partitioning does not perform well and always results in a speedup of less than one. This can be attributed to the high latency for memory accesses on peers that can not be hidden even by the very high amount of parallelism exposed by this kernel.

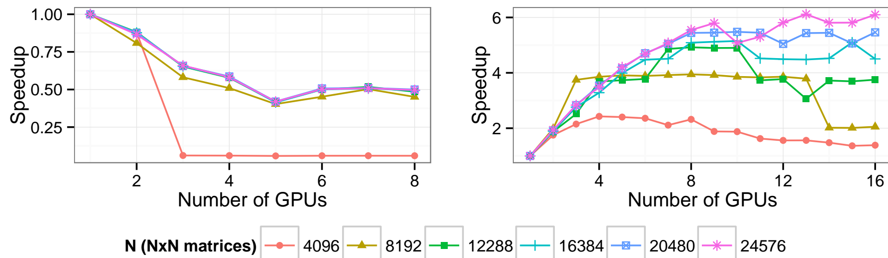


**Fig. 3.** Measured speedup for 1D UVA partitioning vs Input Replication.

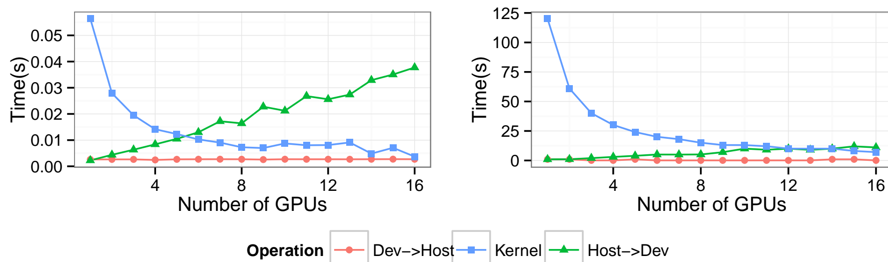


**Fig. 4.** Runtime breakdown of 1D Input Replication partitioning with N=4096 and N=24576.

Input replication, on the other hand, performs reasonably well for larger input sizes. As suspected, figure 4 shows that the initial higher cost of having to distribute the data to all devices is outweighed by the speedup of the kernel execution up to a certain number of gpus. Until the workload hits its problem size dependent point of saturation, the speedup is just slightly less than linear. These initial results indicate that Input based (and possibly Streaming) based automated partitioning might be a promising option to produce high performance GPU code in a productive manner.

## 7 Related Work

There are several projects that focus on simplifying distributed GPU programming without introducing new programming languages or specific libraries that have to be used by the user. Highly relevant are the projects SnuCL from [8] and RCUDA from [13]. They offer a solution to virtualize GPUs on remote notes in a way so that they appear as local devices local (for OpenCL and CUDA respectively) and still require the user to partition the application manually. We consider these projects to be a highly attractive option to scale from a single-node-single-gpu application to a multi-node-multi-gpu application using our techniques.

Several forms of automated partitioning techniques have been proposed in the past. Even though all are similar in principle, the details make them differ

substantially. Cilardo et al. discuss memory optimized automated partitioning of applications for FPGA platforms : while [4] focuses on analyzing memory access patterns using Z-polyhedrals, [5] explores memory partitioning in High-Level Synthesis (HLS) tasks.

The work about run-time systems we examined focus on shared virtual memory and memory optimizations. Li et al. explore the use of page migration for virtual shared memory in [11]. Tao et al. utilize page migration techniques in order to optimize data distribution in NUMA systems in [14]. Both of these works are a great inspiration for the virtual shared memory system we intend on using in order to support irregular workloads. ScaleMP is a successful real-world example of a software based virtual shared memory system.

A mix of compile-time and run-time systems (similar to our approach) has been used for various work: 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 [12]. Lee et al. use kernel partitioning techniques to enable a collaborative execution of a single kernels across heterogeneous processors like CPUs and GPUs (SKMD) [9], and introduce an automatic system for mapping multiple kernels across multiple computing devices, using out-of-order scheduling and mapping of multiple kernels on multiple heterogeneous processors (MKMD) [10].

Work on memory access patterns has a rich history. Recent work that focuses on GPUs include Fang et al., who introduced a tool to analyze memory access patterns to predict performance of OpenCL kernels using local memory [6], which we find very inspiring for our work. Ben-Nun et al. are a very recent representative of various work that extends code with library calls to optimize execution on multiple GPUs by decisions based on the specified access pattern [3]. Code analysis and transformation has also been used to optimize single-device code. In [7], Fauzia et al. utilize static code analysis in order to speed up execution by coalescing memory accesses and promoting data from shared memory to registers and local memory respectively. Similary, Baskaran et al. focus on automatically moving memory between slow off-chip and faster on-chip memory [1].

## 8 Discussion

In this paper, we presented our initial work on GPU Mekong, a tool that simplifies multi-GPU programming using the LLVM infrastructure for source code analysis and code transformations. We observe that the use of multiple GPUs steadily increases for reasons including memory aggregation and computational power. In particular, even NVIDIA's top-notch Tesla-class GPU called K80 is internally composed of two K40 connected by a PCIe switch, requiring multi-device programming techniques and manual partitioning. With GPU Mekong, we gear to support such multi-GPU systems without additional efforts besides good (single-device) CUDA/OpenCL programming skills.

We observe that a dense matrix multiply operation can be computed in parallel with a very high efficiency, given the right data distribution technique. It seems that UVA techniques (load/store forwarding over PCIe) is too limited in terms of bandwidth and/or access latency. Depending on the workload, it might be useful revisiting it later to support fine-grain remote accesses.

For irregular workloads with more than one level of indirection, our current approach of statically partitioning data and code is not going to work. We see virtual shared memory based on page migration as a possible solution for these cases. Given the highly structured behavior of GPU kernels, in particular due to the use of shared memory optimizations (bulk data movement prior to fine-grained accesses), we see strong differences to page migration techniques for general-purpose processors like CPUs. Also, even though it is a common belief that irregular workloads have no locality, recent work has shown that this is not true [2].

As multi-device systems show strong locality effects by tree-like interconnection networks (in particular for PCIe), we anticipate that a scheduling such data movements correctly is mandatory to diminish bandwidth limitations due to contention effects. We plan to support this with a run-time that intercepts block data movements, predicts associated costs, and re-schedules them as needed.

Besides such work on fully-automated code transformations for multiple GPUs, we are envisioning multiple other research aspects. In particular, our code analysis technique could also highlight performance issues found in the single-GPU code. Examples include detecting shared memory bank conflicts, or global memory coalescing issues. However, we still have to find out to which extend these "performance bugs" could be automatically solved, or if they simply have to be reported to the user. Similarly, we are considering exploring promoting global memory allocations automatically to shared memory for performance reasons. Such a privatization would dramatically help using explicit level of the memory hierarchy.

## 9 Acknowledgements

We gratefully acknowledge the sponsoring we have received from Google (Google Research Award, 2014) and the German Excellence Initiative, with substantial equipment grants from NVIDIA. We acknowledge the support by various colleagues during discussions, in particular Sudhakar Yalamanchili from Georgia Tech.

## References

1. Baskaran, M.M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 1–10. PPoPP '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1345206.1345210

2. Beamer, S., Asanovic, K., Patterson, D.: Locality exists in graph processing: Workload characterization on an ivy bridge server. In: Workload Characterization (IISWC), 2015 IEEE International Symposium on. pp. 56–65 (Oct 2015)
3. Ben-Nun, T., Levy, E., Barak, A., Rubin, E.: Memory access patterns: The missing piece of the multi-gpu puzzle. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 19:1–19:12. SC '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2807591.2807611
4. Cilardo, A., Gallo, L.: Improving multibank memory access parallelism with lattice-based partitioning. ACM Transactions on Architecture and Code Optimization (TACO) 11(4), 45 (2015)
5. Cilardo, A., Gallo, L.: Interplay of loop unrolling and multidimensional memory partitioning in hls. In: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition. pp. 163–168. EDA Consortium (2015)
6. Fang, J., Sips, H., Varbanescu, A.: Aristotle: a performance impact indicator for the opencl kernels using local memory. Scientific Programming 22(3), 239—-257 (Jan 2014)
7. Fauzia, N., Pouchet, L.N., Sadayappan, P.: Characterizing and enhancing global memory data coalescing on gpus. In: Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 12–22. IEEE Computer Society (2015)
8. Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., Lee, J.: Snucl: An opencl framework for heterogeneous cpu/gpu clusters. In: Proceedings of the 26th ACM International Conference on Supercomputing. pp. 341–352. ICS '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2304576.2304623
9. Lee, J., Samadi, M., Mahlke, S.: Orchestrating multiple data-parallel kernels on multiple devices. In: International Conference on Parallel Architectures and Compilation Techniques (PACT). vol. 24 (2015)
10. Lee, J., Samadi, M., Park, Y., Mahlke, S.: Skmd: Single kernel on multiple devices for transparent cpu-gpu collaboration. ACM Transactions on Computer Systems (TOCS) 33(3), 9 (2015)
11. Li, K., Hudak, P.: Memory coherence in shared virtual memory systems. ACM Transactions on Computer Systems (TOCS) 7(4), 321–359 (1989)
12. Pai, S., Govindarajan, R., Thazhuthaveetil, M.J.: 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. pp. 33–42. ACM (2012)
13. Peña, A.J., Reaño, C., Silla, F., Mayo, R., Quintana-Ortí, E.S., Duato, J.: A complete and efficient cuda-sharing solution for {HPC} clusters. Parallel Computing 40(10), 574 – 588 (2014), http://www.sciencedirect.com/science/article/pii/S0167819114001227
14. Tao, J., Schulz, M., Karl, W.: Ars: an adaptive runtime system for locality optimization. Future Generation Computer Systems 19(5), 761–776 (2003)