# Automated Partitioning of Data-Parallel Kernels using Polyhedral Compilation

Alexander Matz
alexander.matz@imc.com
IMC Trading B.V. *
Amsterdam, Netherlands

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab
Chicago, Illinois, USA

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

## Abstract

GPUs are well-established in domains outside of computer graphics, including scientific computing, artificial intelligence, data warehousing, and other computationally intensive areas. Their execution model is based on a thread hierarchy and suggests that GPU workloads can generally be safely partitioned along the boundaries of thread blocks. However, the most efficient partitioning strategy is highly dependent on the application's memory access patterns, and usually a tedious task for programmers in terms of decision and implementation.

We leverage this observation for a concept that automatically compiles single-GPU code to multi-GPU applications. We present the idea and a prototype implementation of this concept and validate both on a selection of benchmarks. In particular, we illustrate our use of 1) polyhedral compilation to model memory accesses, 2) a runtime library to track GPU buffers and identify stale data, 3) IR transformations for the partitioning of GPU kernels, and 4) a custom preprocessor that rewrites CUDA host code to utilize multiple GPUs. This work focuses on applications with regular access patterns on global memory and the toolchain to fully automatically compile CUDA applications without requiring any user intervention.

Our benchmarks compare single-device CUDA binaries produced by NVIDIA's reference compiler to binaries produced for multiple GPUs using our toolchain. We report speedups of up to 12.4x for 16 Kepler-class GPUs.

***Keywords:*** Multi-GPU, Polyhedral Compilation, LLVM, Static Analysis, Code Generation, GPU Communication, Runtime Systems

## 1 Introduction

GPUs are prime examples for massively parallel processors and have gained significant traction in the computing landscape. They have established themselves in many domains requiring high bandwidth or high computational performance. They excel in their high computational power and their energy efficiency in terms of performance-per-Watt.

The execution model of GPUs follows the Bulk Synchronous Parallel (BSP) programming paradigm [1], which is designed around creating programs with many more parallel tasks than the underlying hardware can execute simultaneously. This "excess" parallelism is called slackness and allows latency hiding and high portability between different processor architectures, classes, and generations, which might differ in their number of execution units. The data-parallel programming languages implementing the BSP model (e.g. OpenCL and CUDA) facilitate this, hiding architectural aspects from the user while providing consistent performance, independent of actual hardware configurations.

However, performance portability is only observed as long as accesses from processors to memory are equidistant. Individual GPUs today have such an equidistant (or symmetric) memory architecture, but a multi-GPU system qualifies as a non-uniform memory architecture (NUMA), breaking this assumption. We anticipate that future (single) GPU architectures will shift towards NUMA, as technology constraints might require techniques like multi-chip modules, hierarchical memory systems or heterogeneous memory [2].

Multi-GPU programming requires modifications throughout host and device code. These orchestration efforts are completely incompatible with the single-device programming model, whether these GPUs are within one machine or multiple machines. Most high-level optimizations on GPUs aim to reduce stress on the memory subsystem by explicitly caching or reordering memory accesses. Introducing multiple GPUs adds another layer to the execution hierarchy, but not to the memory hierarchy, requiring locality optimizations across multiple GPUs to use different communication methods than within a single GPU.

In this work we introduce an automatic, compiler-based GPU partitioning concept, which allows for a simplified scale-out of single-device GPU programs to almost any number of GPUs. It transparently integrates multiple GPUs into the single-GPU execution and memory model, hiding the complexity of inter-GPU communication and work partitioning from the user. Since no user intervention is required, we can maintain the simplicity and efficiency of single-GPU computing, while providing scalable multi-GPU performance.

* This work was done while the author with the Institute of Computer Engineering at Heidelberg University.

Our hybrid optimization scheme relies on both static and dynamic program analysis. To minimize overhead at execution time, a polyhedral model of the program is analyzed and used to generate optimized transfers between GPUs at run-time.

In particular, this paper makes the following contributions:

- An application model and toolchain that enable automatic partitioning of GPU applications with regular memory access patterns.
- A supporting runtime system that efficiently tracks buffer usage of GPU applications and identifies stale data based on a kernel's memory access patterns.
- Automatic creation of communication and synchronization code using the CUDA Runtime API, orchestrating the execution of partitioned kernels on multiple GPUs.
- An analysis of the performance and run-time overhead of the resulting distributed applications for selected workloads on up to 16 GPUs.

This work provides a detailed description of an automatically partitioning compiler prototype for data-parallel languages that exploits the associated thread hierarchy. Additionally, we analyze the performance of the resulting binaries with a particular focus on the overhead at run-time. We consider this run-time overhead to be of particular importance because even small sequential overheads can severely limit the scalability of a distributed application.

While we initially focus on GPUs within one machine, our automatic communication generation scheme can also be applied to GPU clusters and cloud installations. Our tool stack is based on the gpucc CUDA compiler integrated into the LLVM/Clang compiler framework [3]. CUDA and gpucc were chosen for pragmatic reasons and we see no conceptual difference when replacing CUDA with OpenCL, for instance.

The remainder of this paper is structured as follows. Section 2 provides background information required for the rest of this work, followed by an overview of the compilation toolchain in Section 3; The polyhedral application model used for the generation of optimized communication code is described in Section 4. Sections 5 and 6 explain the host code transformations and polyhedral code generation, respectively. The kernel partitioning mechanism is described in Section 7. Section 8 presents the runtime support system for the kernel orchestration. Performance results are evaluated in Section 9. Relevant work is discussed in Section 10 and the work then concludes with Section 11.

## 2 Background

In this section, we shortly review the current state-of-the-art of GPU architecture and execution models, multi-GPU programming, the LLVM compiler framework, and the polyhedral model.

### 2.1 GPU Architecture and Execution Model

The architecture of GPUs is a consequence of a fundamentally different focus than that of CPUs: while CPUs feature a moderate amount of cores that are highly optimized for sequential performance, GPUs consist of many simple cores that are primarily suited for data-parallel workloads. To satisfy the data-requirements for this large number of processor cores, the GPU memory system is optimized for bandwidth instead of latency. The higher memory latencies can successfully be hidden by minimizing the cost of context switches and scheduling an excess of tasks onto the available execution units. Any threads that are waiting on memory are simply scheduled out for threads that are ready to execute.

Parallel performance is further improved by limiting memory consistency between threads. All threads are grouped into independent collections, so-called thread blocks. Reliable communication is only possible within a thread block thereby eliminating synchronization overhead between them.

The execution model of current GPUs is organized around a regular, hierarchical 3D grid. The highest level in the hierarchy is the grid itself, which is a 3D array of thread blocks. All thread blocks are again 3D arrays of threads and all share identical dimensions. The execution grid can be fully described by the grid size and thread block size each for the three dimensions. Threads are uniquely identified by the position of the thread block that contains them and their position within the thread block.

### 2.2 Multi-GPU Programming

Several approaches to simplify multi-GPU with varying levels of abstraction and room for optimizations programming exist.

The CUDA API directly provides support for multiple GPUs, allowing the distribution of tasks between multiple GPUs. The API is low-level, utilizing it compares to using pthreads for multi-core utilization. Writing multi-GPU code this way requires careful manual orchestration of kernels and data movements and tends to be tedious and error-prone.

Slightly higher abstraction is provided by libraries and frameworks that work on the level of compound data types (e.g. vectors and matrices). Such frameworks can be implemented as a BLAS library, with the GPU-specific code being hidden away in the library. This approach is productive and achieves high performance for applications that only use the limited set of operations provided by these libraries.

Other approaches take inspiration from functional programming and focus on the scalable composition of user-provided kernels. A set of computational patterns, such as `map`, `filter`, or `reduce` operations with user-provided kernels are combined to build more complex systems. This approach typically scales well and has been the basis of

Google's MapReduce [4] and Apache's Spark [5]. The copperhead library also follows this approach and implements it for GPUs using the Python programming language [6].

## 2.3 LLVM Compiler Infrastructure

The LLVM project is a collection of reusable compiler components for program analysis and optimization. It is based on a platform-agnostic, assembly-like intermediate representation (IR) that acts as the interface between all components [7]. This well-defined intermediate representation allows easy reuse of existing functionality, e.g. alias analysis or common subexpression elimination, as well as the rapid development of narrowly focused new components.

Generally speaking, there are three stages in modern compilers:

1. The front-end, which translates human-readable source code into a compiler-specific intermediate representation.
2. Optimizers that take code in intermediate representation as input and optimize it for better performance or smaller code size. This is also called the middle-end.
3. The back-end, which translates code from intermediate representation to machine code.

The LLVM projects provides several front-ends for popular programming languages, various middle-end analyses and transformations, as well as a variety of back ends for common architectures.

## 2.4 Polyhedral Model

The Polyhedral model is a program representation for the analysis of control flow and memory accesses. It uses symbolic affine formulas as a concise representation of memory accesses in loop nests [8].

Both loop iterations and memory accesses are described as unions of Z-Polyhedra. Each Z-Polyhedron is in turn described by a set of inequalities in Presburger arithmetic. The points inside the Z-Polyhedron then represent loop iterations or memory accesses.

$$S_1 := \{\, [y, x] \mid 0 \le y \le x \land 0 \le x \le 4 \,\} \tag{1}$$

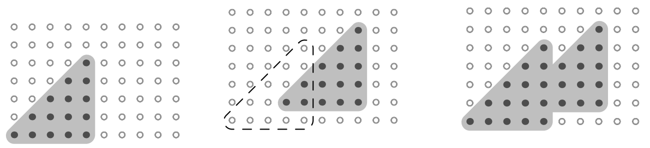
$$M := \{\, [y, x] \rightarrow [y', x'] \mid y' = y + 1 \land x' = x + 3 \,\} \tag{2}$$

$$S_2 := M(S_1) = \{\, [y, x] \mid 1 \le y \le x - 2 \land 3 \le x \le 7 \,\} \tag{3}$$

$$U := S_1 \cup S_2 \tag{4}$$

In Figure 1 the two-dimensional integer sets defined in Equations 1, 3, and 4 are presented. The set $S_1$, defined in Equation 1, is depicted in part 1a. Part 1b shows the image of $S_1$ under the function (or map) $M$, as defined by Equation 2. The resulting set $S_2$ can be described by Equation 3. Figure 1c shows the union of both sets.

Polyhedral optimization consists of three steps: 1) build a polyhedral model of the application 2) transform the model to represent the same computation but with improved performance, and 3) regenerate the application code from the



**(a)** The set $S_1$. **(b)** Translated $S_2 := M(S_1)$. **(c)** Union $U := S_1 \cup S_2$.

**Figure 1.** Visual representation of the integer set $S_1$, its image $S_2$ under the function $M$, thus $S_2 := M(S_1)$, and the union of $S_1$ and $S_2$.

optimized model. Since Z-Polyhedra typically represent loop nests and their memory accesses, the primary task of the code generator is to emit optimized loops that iterate over all points in the set efficiently. In addition, code can be generated to compute polyhedral expressions, for example, the lower bound of a Z-Polyhedron in a particular dimension.

Libraries such as piplib [9], omegalib [10], and *isl* [11] provide implementations for the mathematical concepts underlying polyhedral compilation. For this work, we decided on using the integer set library (*isl*), which is used by the LLVM polyhedral optimizer Polly [12–14]. The *isl* library provides models to represent polyhedral sets and maps and implements many operations on them, e.g. translations, intersections, and projections. Additionally, it provides a code generator for polyhedra and polyhedral expressions, which this work heavily relies on.

# 3 Compilation Toolchain

In this section, we provide a high-level overview of the components making up the toolchain and their interactions.

Our work is built on top of gpucc [3], a CUDA compiler implemented within the LLVM framework. Except for our runtime library, which implements the buffer management, and a source-to-source translator written in lua, all our analyses and transformations are implemented as part of the gpucc pipeline [15]. Note that in contrast to CPU-only applications, GPU applications target two different architectures at once and therefore require two separate compilation paths. To partition GPU applications, our pipeline performs the following five high-level tasks distributed over two passes of gpucc:

1. Analyze the GPU kernels and create high-level application models of their memory behavior (first pass, described in section 4).
2. Apply source-to-source transformations to reference multi-GPU primitives (non-gpucc, described in section 5).
3. Generate communication code from the memory behavior model that dynamically identifies stale and updated kernel data (second pass, described in section 6).
4. Create partitioned copies of the GPU kernels computing partial results (second pass, described in section 7).

5. Link the application against our runtime library that implements the multi-GPU primitives (second pass, described in section 8.

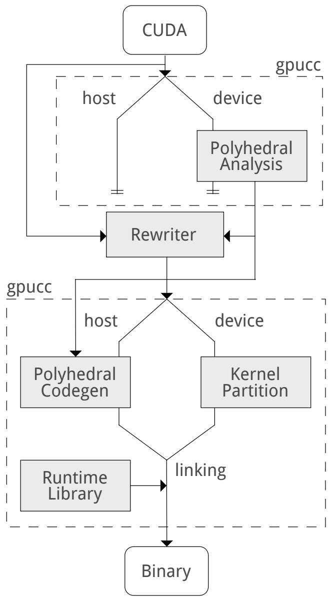


**Figure 2.** Toolchain overview

Figure 2 illustrates how the individual steps are integrated into a single compilation pipeline. The first pass of gpucc is required only to obtain the memory behavior models, other results, e.g. object files, are discarded. After the source-to-source rewriter transforms the application, gpucc is invoked a second time to generate the multi-GPU application. This repeated invocation of gpucc introduces redundant work, resulting in a compile time increase from 1.9× - 2.2× for the tested applications.

## 4 Polyhedral Application Models

In this section, we describe how polyhedral program analysis [16] allows building a model of an application's memory accesses. The analysis is a stand-alone LLVM pass that generates polyhedral memory accesses descriptions similar to the representation used by polyhedral optimizers [14, 17]. However, the pass is designed as a general-purpose analysis and can generate (approximate) results for any reducible control flow graph. While we currently ignore low-level correctness issues, such as potential integer overflows, we can use the same techniques as polyhedral optimizers to ensure soundness [18].

The application model of a kernel describes all memory accesses to externally visible arrays as polyhedral integer maps that map a thread id (in the grid) to zero or more points in each array. Since optional write accesses exist, all relevant memory accesses are first collected and then categorized as either "must" or "may". While available, this information is currently not exploited, optional "may"-accesses are treated as "must"-accesses. This is a pessimistic approximation and utilizing it can improve performance, but it does not impact correctness.

### 4.1 Memory Access Maps for CUDA

The code describes the instructions of an individual thread in the execution grid. Its coordinates are specified in a two-level hierarchy, the thread block index being the first level, specified via `blockIdx.{z,y,x}`, and the thread's position within the thread block being the second level, specified via `threadIdx.{z,y,x}`. Since the grid itself is later partitioned (ref. section 7), memory locations are be expressed in terms of these coordinates. However, as applications can configure arbitrary thread block dimensions (accessible through `blockDim.{z,y,x}` in the kernel code) the global thread position in the grid contains a non-affine multiplication of two variables, which is not directly supported in the polyhedral model. For a dimension $w \in \{z, y, x\}$ of the grid, the global position is typically computed using the following expression:

$$threadIdx.w + blockIdx.w \cdot blockDim.w \tag{5}$$

Since the thread block size is unknown but fixed for one launch of the kernel and the CUDA memory model guarantees the independence of thread blocks, we can introduce a new "block offset" dimension to encapsulate the non-affine multiplication [19]. Thus, with

$$blockOff.w = blockIdx.w \cdot blockDim.w \tag{6}$$

, the global position of a thread in the thread grid becomes the following affine expression:

$$threadIdx.w + blockOff.w \tag{7}$$

Before starting the kernel, `blockOff.{x,y,z}` needs to be initialized accordingly.

For each of the thread grid dimensions $\{z, y, x\}$ we now have three input dimensions *blockOff, blockId, and threadId* that describe the thread's position in the hierarchical grid. This leads to the descriptions of memory accesses in an array having having the form $\mathbb{Z}^9 \rightarrow \mathbb{Z}^d$, with $d$ being the number of dimensions of the array.

The CUDA execution model guarantees thread blocks to be an atomic unit of execution. This allows simplifying the memory access descriptions by eliminating the *threadId* dimension. Before projecting out of all three grid dimensions a constraint is added for each: $0 \leq threadId < blockDim$, which emulates thread blocks. The resulting memory access maps are now a subset of $\mathbb{Z}^6 \rightarrow \mathbb{Z}^d$. They accurately model the memory behavior as seen from outside of a GPU kernel, provided the constraint *blockOff* = *blockId* ∗ *blockDim* is satisfied.

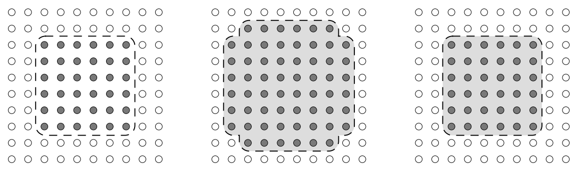

**(a)** Partition **(b)** Read Set **(c)** Write Set

**Figure 3.** Read and written memory locations of a 5-point stencil applied to a partition of the grid.

Figure 3 illustrates the memory access patterns of a 5-point stencil, where each thread computes exactly one element in the 2-dimensional result array. The thread grid partition on the left has read accesses that also include the halo of the target elements. The write accesses, on the other hand, are a 1:1 mapping of the thread grids to array elements.

While read maps can always be over-approximated without compromising correctness, write maps need to be accurate and any over-approximation can lead to incorrect results. Additionally, write maps must be injective, indicating that no two threads write to the same address. Since the CUDA execution model does not impose an order of execution between threads or thread blocks, such writes can occur in any order. Applications with this behavior exist, often as an optimization relying on particular hardware characteristics. Since we cannot replicate these characteristics with multiple GPUs, write-after-write hazards prohibit multi-GPU execution.

After performing these checks, the application model is saved to disk. For each kernel, a record is created that contains the kernel's name, suggested partitioning strategy, and a list of its arguments. The read and write maps of arrays are stored per-argument.

## 5 Host Code Transformations

This section describes the transformations applied to the host code of the application, not including communication code generation, which is explained in section 6. To utilize multiple GPUs, the host code of the application needs to be transformed to use multi-GPU primitives instead of the CUDA API. This transformation can be applied on different representation levels, including plain text, the abstract syntax tree (AST), or the low-level intermediate representation (IR) of the host code. We decided to use text substitutions with regular expressions for the source-to-source transformation. This allows for a simple implementation at the cost of not supporting all possible CUDA applications. Three types of substitutions are made.

The first type inserts information at the very top of the source code file, including:

- An additional `#include` directive for the header file of the runtime library.
- A new, empty definition for each GPU kernel with the same signature as the original except an additional parameter for partition information.
- Declarations (without implementation) for functions describing the kernel's access patterns. Their generation is explained in section 6.

The second type of substitution replaces CUDA API calls with multi-GPU variants in our runtime library. Since the replacements are designed to have identical signatures, the substitution can be made by simply changing the name of the called function. As an example, all `cudaMalloc(&buffer, size)` calls are replaced by `__mgpuMalloc(&buffer, size)` calls.

The last type of substitution alters kernel launches to perform four tasks:

1. Partition the execution grid for the available GPUs.
2. Synchronize all buffers that are read from to contain up-to-date values.
3. Launch each partition of the kernel on its respective GPU.
4. Update the buffer tracker to account for all writes performed by the kernel partitions.

Algorithm 4 shows the pseudo-code that is inserted to replace kernel launches in the original program. The bodies of the three top-level loops correspond to the tasks (2), (3), and (4) in the list above and the grid partitioning from step (1) is integrated into the loops themselves.

In the first loop, multi-GPU primitives defined in the runtime library and the automatically generated code are used to synchronize all buffers between GPUs that are read from by the kernel. A virtual buffer is synchronized by iterating over each partition's memory accesses (determined using polyhedral code generation as explained in Section 6), using the memory tracker to identify the GPU that has most recently written to each location, and copying the data to the local GPU.

The second loop launches a partitioned kernel: a new grid configuration is calculated, then all kernel arguments referring to GPU buffers are replaced with a pointer the partition's local instance, and finally the kernel is launched asynchronously;

The third loop updates the memory trackers of the virtual buffers to reflect the write accesses each partition. This happens concurrently to the asynchronous kernels and relies on multi-GPU primitives from the runtime library.

## 6 Polyhedral Code Generation

This section describes the code generation that enables efficient memory transfers to synchronize buffers contents between kernel launches. Generally speaking, polyhedral maps

```
params = [arg in args | arg is parameter]
for gpu in GPUs:
  partition = model.kernel.partitioning(grid, gpu,
       params)
  reads = [arg in args | arg is array and arg is
       read]
  for array in reads:
    pattern = pattern_for(model.kernel, array)
    buffer_synchronize(array, pattern, partition,
         params)
all_devs_synchronize()

for gpu in GPUs:
  partition = model.kernel.partitioning(grid, gpu,
       params)
  newGrid = partition.max - partition.min
  newArgs = []
  for arg in args:
    if arg is array:
      newArgs += [instance_for_gpu(arg, gpu)]
    else:
      newArgs += [arg]
  partitioned_kernel<<<newGrid, blocks>>>(newArgs,
       partition)

for gpu in GPUs:
  partition = model.kernel.partitioning(grid, gpu,
       params)
  writes = [arg in args | arg is array and arg is
       write]
  for array in writes:
    pattern = pattern_for(model.kernel, array)
    buffer_update(array, pattern, partition, params)
```

**Figure 4.** Pseudo code of the kernel launch replacement that is inserted by the source-to-source rewriter.

are translated into executable code to extract two pieces of information:

1. the elements in the image of an access map, and
2. the dimension sizes of all arrays in global memory.

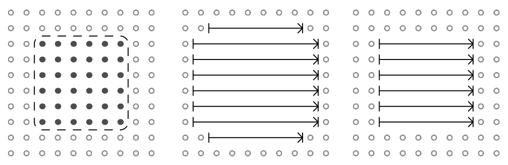
The elements in the image of an access map can be represented in multiple ways. Directly generating a function $f : \mathbb{Z}^6 \to \mathbb{Z}^d$, as described in our model in section 4, would require iterating over all thread blocks of a partition to get its read or write set. This can be optimized into a function of the form $f : P \to A, P \in (\mathbb{Z}^6, \mathbb{Z}^6, \mathbb{Z}^3), A = \{x | x \in \mathbb{Z}^d\}$ that returns all elements $A$ that are in the image of the access map applied to a partition $P$. The partition $P$ is described as a 6-dimensional box spanned between two tuples of $blockOff.\{z, y, x\}$ and $blockId.\{z, y, x\}$ (the thread block dimension need to be provided as well). By constraining the domain of the map to the inside of this 6-dimensional box, the image contains only the elements accessed by a specific thread grid partition. This is exactly the information required from the access maps in order to allow automatic buffer synchronization between GPUs.

## 6.1 Code Generation

The *isl* library provides highly optimized code generation facilities that allow to easily embed this information into the application as native IR functions. It provides an API that generates Abstract Syntax Trees (ASTs) from polyhedral sets and expressions, which can then be translated into LLVM IR. The ASTs generated by *isl* can be directly expressed in a structural programming language, such as C. All nodes in an AST are either control flow or (closed-form) expressions.

Control flow in *isl* is limited to for-loops and conditional branches. Both are basic control flow primitives and can be directly translated into fixed sets of LLVM IR basic blocks and branches.

Polyhedral expressions are also generated as ASTs. Each node in the expression's AST is either a constant value, a variable reference, or an operator with one or more expressions as operands. Since operands can themselves be the result of an operation, complex expressions can be built from simpler ones. Polyhedral expressions are closed-form expressions, meaning they never contain control flow and can be computed in constant time. Virtually every operator in the AST has a direct counterpart in LLVM IR, allowing very easy generation of the appropriate IR for a given expression.



**(a)** Partition **(b)** Read Set **(c)** Write Set

**Figure 5.** Scanning the memory accesses of a 5-point stencil code

Enumerating every single element in the image of an access map (i.e. every accessed array element) is, while straight-forward, prohibitively expensive. Instead, we leverage the fact that CUDA uses row-major order to store multidimensional arrays and enumerate only the first and last element of each row in the image. The first and last set are computed by fixing all but the last dimension of the image to the position of the given row and computing the lexicographical minimum and maximum of the resulting polyhedral set. The result can either be a convex set, in which case this optimization produces exact results, or a union of convex sets, resulting in an over-approximation of the enumerated sets. For a union of sets, the over-approximation can be eliminated by applying this approach to each convex set of the union instead of the union set itself.

## 6.2 Enumerator Interface

The generated code needs to be available to the static runtime library and therefore requires a well-defined interface. Each generated function for a read or write map is given the same name as the kernel, followed by a suffix containing the position of the argument in the kernel arguments and a "read" or "write" parameter.

Input to the functions are the partition of interest and the values of scalar arguments, both are passed using arrays of 64-bit integers to avoid variable numbers of arguments. The partitioning information is a 6-tuple describing the partition as pairs of half-open intervals of thread blocks, one for each of the three thread grid dimensions. The scalar arguments are simply copied into an array from the kernel launch they belong to.

Output of the function is a list of element ranges in the set. Since the number of these are unknown, a callback function is used to avoid dynamic memory allocations. The callback is invoked once for each element range.

Using this approach, the static runtime can easily and efficiently use the information collected during kernel analysis.

## 7 Kernel Partitioning

Transformed kernels should behave as if acting on only a subset of their original thread grid. In this section we describe the transformations required for this. A thread grid partition is a 3-tuple of integer pairs: $((min_z, max_z), (min_y, max_y), (min_x, max_x))$. Each pair describes the start (inclusive) and end (exclusive) of the partition in one of the three thread grid dimensions. In contrast to the generated code in section 6, block dimensions do not need to be included in the partition because the regular block dimensions from CUDA's special registers are still valid.

$$blockIdx.w \longrightarrow partition.min_w + blockIdx.w \tag{8}$$

$$gridDim.w \longrightarrow partition.max_w \tag{9}$$

$$gridConf.w = partition.max_w - partition.min_w \tag{10}$$

$$w \in \{z, y, x\}$$

The first step is cloning the kernel code and appending the arguments for the kernel partition. In this state, the kernel behaves exactly like the original one and would just ignore the additional argument. Next, the two substitution rules from equations (8) and (9) are applied. Rule (8) adds an offset to the block ID so that from the kernel's perspective the thread blocks now start at the start of the partition instead of at zero. Rule (9) replaces the kernel's grid dimension with the end of the partition. Combining both rules results in the kernel executing code only for thread blocks in the half-open interval $[min_w, max_w), w \in \{z, y, x\}$. The correctness of this transformation relies on the grid configuration at kernel launch to be updated according to equation (10).

## 8 Runtime Library

The runtime library contains high-level, static functions that are common to all partitioned applications. These functions do not need to be customized to individual applications and can be implemented and compiled in advance. This allows using a high-level implementation language, such as C++. The library is split into two parts: virtual buffer management and CUDA wrapper functions.

### 8.1 Buffer Management

When kernel partitions distributed over multiple devices, each device needs a copy of its data in a device-local buffer. The coherence protocol between these device-local buffers is similar to cache coherence protocols of GPUs, albeit much simpler due to all synchronization points being known in advance (i.e. memcopies and kernel launches).

Instead of allocating a single buffer on a single GPU, the partitioned application allocates one device buffer per device, creates a tracker component, and bundles them into a "virtual buffer".

The tracker contains a sorted list of non-overlapping segments, each containing a reference to the buffer instance that holds the most recently updated copy of that segment. The tracker is updated by all operations that write to the virtual buffer, namely kernel launches (described in 5) and memcopies (as described in 8.3). This allows the accurate tracking of the distribution of the most up-to-date data contained in a virtual buffer. The segment list is based on a B-Tree map using the start of each segment as the key and the "owner" of the most recent version as the value.

In GPU kernels with regular memory access patterns, locality of thread IDs often directly translates to data locality in the buffers used in the location. This is a result of the execution model encouraging to calculate one output element per thread and to equate the thread ID to the output element index. As a consequence, large contiguous partitions in the thread grid cause large contiguous arrays in memory to be read and written, further limiting fragmentation. The extreme case are GPU kernels with a 1:1 correspondence between the thread ID and the output element index. Kernels with such a write pattern produce a single segment per partition in the tracker of the buffers they are writing to.

### 8.2 Memcopies for Multiple Devices

Memcopies in a single GPU application always have exactly one source buffer and exactly one target buffer. In partitioned multi-GPU applications, however, CUDA memcopies can have multiple source or destination buffer instances. This requires some adaptions to the corresponding memcpy operations. Since the CUDA Runtime API requires its memcopies to specify the direction of the data movement, they can be translated based on the specified direction.

**Host-to-Device** memcopies turn into a *1:n* data movements. The single source buffer is distributed among multiple GPUs. Static analysis can not reliably prove the read pattern applied to the buffer in the presence of unpredictable control flow or kernel configurations. Consequently, data is distributed in a predefined pattern, hoping that this pattern matches the read pattern of the following kernels. Currently, this pattern is a linear distribution among all GPUs. Any mismatches between this pattern and the actual read pattern of the kernels need to be corrected before the kernel launch as described in subsection 8.3.

**Device-to-Host** memcopies are a *n:1* data movement in the translated application. Since all data is gathered into a single host buffer and the tracker has a record of the data distribution of the device buffer, this memcpy is easily translated: 1) The tracker is queried for all contiguous segments and the GPU that segment is located on and 2) the segment is copied from the given GPU to the host buffer.

**Device-to-Device** copies turn into *n:n* data movements. This can be implemented as a combination of the *Host-to-Device* and *Device-to-Host* strategies. Single-GPU applications typically avoid device-to-device copies since it results in duplicated data data on a single GPU. For this reason, Device-to-device memcopies are currently not supported.

**Host-to-Host** data movements are left unmodified.

### 8.3 Synchronization of Virtual Buffers

Enforcing coherence between buffer instances with respect to kernel launches requires the generated code described in section 6. Both synchronizing a virtual buffer and updating its tracker are operations that are specific to a particular partition and therefore need to be repeated for each partition. Currently, each partition corresponds to exactly one GPU.

In order to synchronize a virtual buffer for a given GPU, the read set of that GPU's partition is iterated over using the generated code for that array (as described in 6). For each interval in the read set, the tracker is queried and returns one or more segments pointing to the buffer instance that contains the most recent version of the data in that segment. If the data is already present on the current GPU, nothing is done. Otherwise, an asynchronous memcpy is issued to copy the data from its most recently written to GPU. The tracker of a virtual buffer does not support shared copies, resulting in redundant transfers for applications with large amounts of shared data.

Updating the tracker requires iterating over the write set of a GPUs partition. For each interval in the write set of a given GPU, the tracker is updated so that the interval points to the buffer of that GPU as owning the most recently updated version of the data.

### 8.4 CUDA Runtime Replacement

The CUDA replacement functions have identical prototypes to their CUDA API counterparts to ease code transformation and provide a stable interface. The memory related CUDA API (`cuda{Malloc / Free / Memcpy / MemcpyAsync}`) is replaced by functions dispatching to their virtual buffer implementation. `cudaGetDeviceCount` is replaced by a function that always returns 1 and `cudaDeviceSynchronize` is replaced by a function that synchronizes all available devices. New CUDA replacements are implemented as required.

| Benchmark | Small | Medium | Large | Iterations |
|---|---|---|---|---|
| Hotspot | 8,192 | 16,384 | 36,864 | 1,500 |
| N-Body | 65,536 | 131,072 | 327,680 | 96 |
| Matmul | 8,192 | 16,384 | 30,656 | N/A |

**Table 1.** Configurations of the benchmark applications.

## 9 Experimental Results

In this section, we evaluate the performance of the prototype implementation using three proxy applications.

The system used for the tests is a Supermicro X10DRG equipped with two Intel Xeon E5-2667 Processors, eight NVIDIA K80 GPUs, and 256GiB of DDR4 RAM running at 2133MHz. Single-GPU results are from the reference binary produced by NVIDIA's NVCC compiler version V8.0.61. Multi-GPU applications have been compiled using our toolchain as an external module to the development version of LLVM/-Clang from Jan 8, 2018. NVIDIA driver initialization and host buffer initialization were measured separately and removed from the total runtime.

### 9.1 Workload Evaluation

The selection of workloads the approach can be tested on is limited to those that exhibit memory access patterns that can be accurately predicted using our polyhedral model. The three chosen benchmarks are taken from the computational dwarfs identified by Berkeley in [20], and configurations are summarized in table 1.

*Hotspot* is a 5-point stencil operating on a quadratic grid. The problem size describes the side lengths of this grid and the number of iterations has been fixed to 1500 for the graphs shown. The amount of computation per thread is constant and comparatively low, as are the data requirements per thread. As a result, this benchmark is susceptible to overheads in the distribution process and expected to exhibit only limited scalability. Figure 6 shows that the maximum speedup of about 7.1x is reached with 14 GPUs.

The *N-Body* benchmark is a direct gravitational N-Body simulation, with the problem size describing the number of simulated bodies. Clustering optimizations have not been applied, since the dynamic clusters would result in irregular memory accesses. In this benchmark, computation per thread grows cubic with the problem size, while the data requirements per thread grow only linearly, resulting in excellent scaling behavior. The maximum speedup of about 12.4x is reached using 16 GPUs, as seen in Figure 6.

*Matmul* computes the product of two dense, quadratic matrices, with the problem size being the side length of the matrices. The version used here is a basic tiled implementation. In a square matrix multiply, the total work per thread also grows cubic with the side length of the matrix while the read

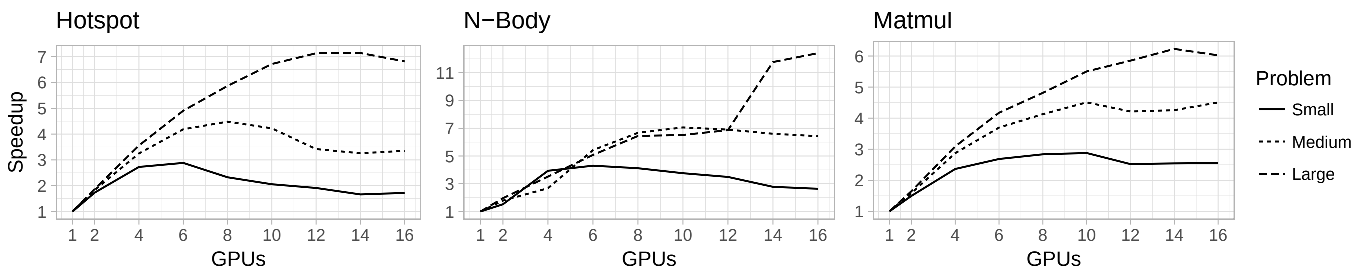


**Figure 6.** Speedup of the benchmarks for up to 16 GPUs.

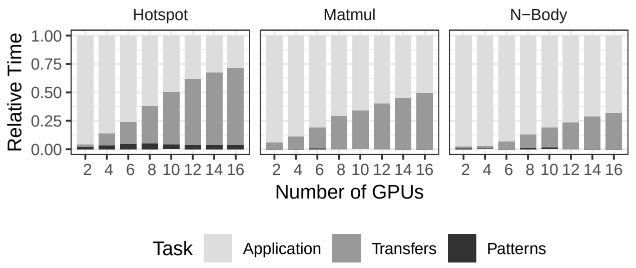


**Figure 7.** Breakdown of the execution time of transformed applications.

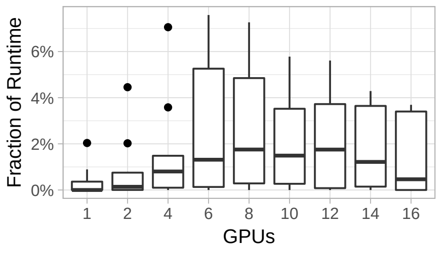


**Figure 8.** Overhead of the runtime system.

set grows quadratic with it. The second matrix of the product is read column-wise by each thread but distributed linearly over all devices (the default distribution pattern). This mismatched data distribution is corrected by the runtime before the kernel starts. The resulting initial overhead together with the lack of iterative execution limits scalability. Figure 6 shows a maximum speedup of about 6.3x for 14 GPUs.

## 9.2 Overhead Analysis

Parallelizing a CUDA application is not only limited by the kernels themselves, but also by the sequential overhead that orchestrates the parallel kernels. The lower bound of these overheads can be measure by executing the partitioned application on a single GPU: across all single-GPU experiments, the slow-down has a median of 2.1 %, with a 25th and 75th percentile of 0.13 % and 3.1 %, respectively.

The next step is a further dive into the different types of overhead introduced by the partitioning of the application. We compute overheads based on direct measurements of the executed applications to avoid instrumentation and parallelism issues. The execution time of each benchmark is measured in three configurations:

- $\alpha$: regular execution of the multi-GPU application
- $\beta$: execution with disabled transfers, but dependency resolution and tracker updates are performed
- $\gamma$: execution with disabled dependency resolution and tracker updates, which automatically also disables transfers

Using these measurements, the following relative times can be computed:

- time spent only in application logic: $T_{Application} = \gamma/\alpha$
- time spent in transfers: $T_{Transfers} = (\alpha - \beta)/\alpha$
- time spent in non-transfer overheads: $T_{Patterns} = (\beta - \gamma)/\alpha$

Figure 7 shows an overview of how these parts of the execution time have been measured for the "medium" sized problems for all three benchmarks. As expected, the relative time spent with overhead increases with larger numbers of GPUs. However, the majority of the overhead is caused by transfers for buffer synchronization that are essential to the partitioning. Non-transfer overheads (mostly caused by the resolution of data dependencies) make up a maximum of 6.8% over all measurements.

Figure 8 gives a more accurate view of the non-transfer overheads over all benchmarks and problem sizes as a box plot. Over all measurements, the 25th percentile of the overhead is at 0.001 %, the 75th percentile at 3.5 % and the median at 0.51 %. The overheads computed in this section already account for highly iterative benchmarks that are sensitive to sequential overheads, such as the Hotspot. Thus, we consider the overhead to be within the acceptable range for an automated solution.

## 10 Related Work

Several forms of automated partitioning techniques have been proposed in the past. Even though all aim to achieve a similar goal, they differ substantially in their concepts and details.

Related work on runtime systems focusses on shared virtual memory and memory optimizations. Li et al. explore the use of page migration for virtual shared memory in [21]. Tao et al. utilize page migration techniques to optimize data distribution in NUMA systems [22]. As opposed to our work, these concepts rely on page migration and perform all tasks at execution time. Instead, we exploit knowledge generated at compile-time to optimize data movements at execution time. However, we see page migration as a possible solution for workloads with dynamic, data-driven memory access patterns like graph computation, sparse linear algebra and similar.

Lee et al. use kernel partitioning techniques to enable a collaborative execution of a single kernel across heterogeneous processors like CPUs and GPUs (SKMD) [23], 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) [24]. However, they focus on scheduling optimizations rather than automatic partitioning. Similar applies to various other works, including [25].

In [26], Lee et al. implement software-based virtual memory management for OpenCL kernels to circumvent GPU memory size limitation. Instead of using static analysis for memory pattern analysis, they create a minimal clone of each kernel that marks accessed memory in a page table, yielding accurate results at the expense of significant runtime overhead.

Related work on memory access patterns has a rich history. Recent work that focuses on GPUs includes Fang et al., who analyze memory access patterns to predict the performance of OpenCL kernels [27]. Ben-Nun et al. are representative of various work that extends code with library calls to optimize execution on multiple GPUs by decisions based on the user-specified access pattern [28].

Bondhugula et al. take a similar approach in their work [29] by modeling the memory access patterns of loop nests and distributing the work and data across a distributed memory system. While influential, their work focuses on solely CPU based systems with no CPU-GPU interplay and relies exclusively on polyhedral compilation, instead of using a dynamic tracker at runtime to allow for the arbitrary distribution of data.

Moll et al. in [19], similar to us, leverage the polyhedral model to reason about memory access patterns and split the input space of data-parallel languages. Since they only model and split a single thread block at a time, it is orthogonal our work. However, we observe that a complete understanding of memory access patterns might require a combination of multiple analysis techniques.

The APOLLO project is an automatic loop-parallelizer for CPU applications that also uses a polyhedral model to partition the code for execution on multiple cores [30]. As opposed to the whole-kernel analysis used in our work, they use instrumented code to profile different partitioning strategies on a small subset of the workload and find the best performing one.

Both SnuCL [31] and rCUDA [32] address the complexity of scaling out GPU applications to multiple nodes by projecting GPUs on remote nodes into the local system, essentially providing GPU virtualization. Although related to our work, GPU kernels and buffers are treated as atomic, liberating both projects from the need to manage coherence within individual device buffers. While extensions [33] optimize scalability by replicating host program execution and data, the task of mapping control and data to these devices is left to the programmer.

## 11 Conclusion

In this work, we presented an LLVM-based toolchain that automatically compiles single-GPU code to multi-GPU applications. Polyhedral compilation is used for the analysis of memory access patterns and communication code generation, and standard code transformation techniques are used to partition host and device code. We introduce the tool stack and the methodology behind it, providing details on the different steps performed at compile time and run time. Three different workloads are used for experiments on a system with 16 GPUs, resulting in speedups of up to 12.4x. A detailed analysis of the runtime overhead, i.e. the non-transfer overhead, has shown it to be very low with a median of 0.51 % of the total application runtime.

These results suggest that automatic partitioning using polyhedral compilation is feasible for GPU programs, and the runtime overhead of the resulting multi-GPU binaries is very low. The primary limitation of this approach is that it requires an accurate model of the kernel's write accesses to global memory. This limitation can be remedied by using instrumentation to collect write patterns, shared virtual memory for result distribution, or annotation of the source code with write patterns by the programmer. Of course, any combination of these and other approaches are also possible, and recent GPU features seem to support such approaches.

## References

[1] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, 1990.

[2] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "Mcm-gpu: Multi-chip-module gpus for continued performance scalability," *SIGARCH Comput. Archit. News*, vol. 45, June 2017.

[3] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt, "Gpucc - an open-source gpgpu compiler," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, (New York, NY), 2016.

[4] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, 2008.

[5] J. G. Shanahan and L. Dai, "Large scale distributed data science using apache spark," in *Proceedings of the 21th ACM SIGKDD International*

*Conference on Knowledge Discovery and Data Mining*, KDD '15, (New York, NY, USA), ACM, 2015.

[6] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: compiling an embedded data parallel language," *ACM SIGPLAN Notices*, vol. 46, no. 8, 2011.

[7] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization*, CGO '04, Mar. 2004.

[8] M. Griebl and J.-F. Collard, "Generation of synchronous code for automatic parallelization of while loops," in *European Conference on Parallel Processing*, Springer, 1995.

[9] P. Feautrier, "Parametric integer programming," *RAIRO Recherche Opérationnelle*, vol. 22, no. 3, 1988.

[10] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott, "The omega library interface guide," 1995.

[11] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *International Congress on Mathematical Software*, vol. 6327, Springer, 2010.

[12] S. Verdoolaege and T. Grosser, "Polyhedral extraction tool," in *Second International Workshop on Polyhedral Compilation Techniques*, IMPACT '12, 2012.

[13] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, 2013.

[14] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly - polyhedral optimization in LLVM," in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques*, vol. 2011 of *IMPACT '11*, 2011.

[15] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes, "Lua 5.1 reference manual," 2006.

[16] J. Doerfert and S. Hack, "Polyhedral value & memory analysis," 2017. 2017 US LLVM Developers' Meeting.

[17] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*.

[18] J. Doerfert, T. Grosser, and S. Hack, "Optimistic loop optimization," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, 2017.

[19] S. Moll, J. Doerfert, and S. Hack, "Input space splitting for OpenCL," in *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, 2016.

[20] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, *et al.*, "The landscape of parallel computing research: A view from berkeley," tech. rep., Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[21] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 4, 1989.

[22] J. Tao, M. Schulz, and W. Karl, "Ars: an adaptive runtime system for locality optimization," *Future Generation Computer Systems*, vol. 19, no. 5, 2003.

[23] J. Lee, M. Samadi, and S. Mahlke, "Orchestrating multiple data-parallel kernels on multiple devices," in *International Conference on Parallel Architectures and Compilation Techniques*, vol. 24 of *PACT '15*, 2015.

[24] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Skmd: Single kernel on multiple devices for transparent cpu-gpu collaboration," *ACM Transactions on Computer Systems*, vol. 33, no. 3, 2015.

[25] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, ACM, 2014.

[26] J. Lee, M. Samadi, and S. Mahlke, "Vast: The illusion of a large memory space for gpus," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, (New York, NY, USA), ACM, 2014.

[27] J. Fang, H. Sips, and A. Varbanescu, "Aristotle: a performance impact indicator for the opencl kernels using local memory," *Scientific Programming*, vol. 22, Jan. 2014.

[28] T. Ben-Nun, E. Levy, A. Barak, and E. Rubin, "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*, SC '15, ACM, 2015.

[29] U. Bondhugula, "Compiling affine loop nests for distributed-memory parallel architectures," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ACM, 2013.

[30] A. Sukumaran-Rajam, L. E. Campostrini, J. M. M. Caamano, and P. Clauss, "Speculative runtime parallelization of loop nests: Towards greater scope and efficiency," *HIPS+ LSPP*, vol. 176, 2015.

[31] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "Snucl: An opencl framework for heterogeneous cpu/gpu clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, (New York, NY, USA), ACM, 2012.

[32] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rcuda: Reducing the number of gpu-based accelerators in high performance clusters," in *Proceedings of the 2010 International Conference on High Performance Computing and Simulation*, HPCS '10, IEEE, 2010.

[33] J. Kim, G. Jo, J. Jung, J. Kim, and J. Lee, "A distributed opencl framework using redundant computation and data replication," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, (New York, NY, USA), ACM, 2016.