# Quantifying the NUMA Behavior of Partitioned GPGPU Applications

Alexander Matz
Heidelberg University
Heidelberg, Germany
alexander.matz@ziti.uni-heidelberg.de

Holger Fröning
Heidelberg University
Heidelberg, Germany
holger.froening@ziti.uni-heidelberg.de

## Abstract

While GPU Computing is pervasive in various areas, including scientific-technical computing and machine learning, single GPUs are often insufficient to meet application demand. Furthermore, multi-GPU processing is a promising option to achieve a continuing performance scaling, given that CMOS technology is expected to hit fundamental scaling limits. We observe that a large amount of work has been done regarding characterizing single-GPU applications. However, characterizing GPGPU applications regarding the NUMA effects resulting from distributed execution has been largely overlooked.

In this work, we introduce a framework that allows analyzing the internal communication behavior of GPGPU applications, consisting of 1) an open-source memory tracing plugin for Clang/LLVM, 2) a simple communication model based on summaries of a kernel's memory accesses, 3) communication and locality analyses from memory traces. Besides characterizing locality of kernels, it allows reasoning about virtual bandwidth-limited communication paths between NUMA nodes using different partitioning strategies. We then apply this framework for a large variety of applications, report initial results from our analysis, and finally discuss benefits and limitations of this concept.

***Keywords*** CUDA, NUMA, Multi-GPU, Memory Tracing, LLVM

**ACM Reference Format:**
Alexander Matz and Holger Fröning. 2019. Quantifying the NUMA Behavior of Partitioned GPGPU Applications. In *General Purpose Processing Using GPU (GPGPU-12), April 13, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3300053.3319420


*GPGPU-12, April 13, 2019, Providence, RI, USA*

ACM ISBN 978-1-4503-6255-9/19/04…$15.00
https://doi.org/10.1145/3300053.3319420

## 1 Introduction

GPUs are massively parallel shared-memory multi-processors, based on a bulk-synchronous parallel (BSP) execution model and excelling in performance for a variety of workloads. While there exists a plethora of related work looking at GPU workload characterization [7, 8, 17], relative little attention was put on analyzing how such workloads would scale with multiple GPUs.

Multi-GPU systems are typically used for workloads with large computational requirements, memory bandwidth requirements, or memory capacity requirements. Examples include high-performance computing [13] and training of deep neural networks [18, 23]. Furthermore, multi-GPU solutions have been proposed to overcome scalability limitations of monolithic GPUs [2, 9]. However, compared to single GPUs, multi-GPU systems show strong Non-uniform memory access (NUMA) effects, as on-card vs. off-card throughput differs by more than one order of magnitude [13, 19], which complicates programming and in particular requires an understanding of the implications of NUMA effects.

Memory access patterns are one of the primary factors in determining suitability and scalability of workloads when scaling to multiple processors. For a given workload and partitioning across multiple processors, a memory access pattern can describe the impact on transferred data volume between different partitions and therefore allows characterizing the implications of NUMA effects. However, they are often declared as out-of-scope, including in the related work presented in section 3.

In this work we propose a framework that produces memory traces from GPGPU applications and analyzes these traces based on a simple communication model to identify NUMA effects arising from the application's internal communication. The framework is based on a compiler plugin that instruments an application and collects traces of its memory accesses for later analysis. A post-processing step reduces the raw traces by summarizing them into read and write sets on a thread block granularity. Using a communication model based on this summary and a virtual partitioning scheme, the data is analyzed with a focus on communication between different partitions.

In particular, this work makes the following contributions:

1. A compiler plugin to track memory accesses via GPU kernel instrumentation and a data acquisition backend to store these traces.
2. A model for simulated partitioning and a communication model based on post-processed memory traces to identify communication.
3. An analysis of the internal communication and NUMA effects of three different benchmark suites for a selection of different partitioning strategies

The rest of the paper is structured as follows: in section 2 we briefly review background on compilation and the GPU execution model. Section 3 discusses related work, before we introduce our analysis methodology including communication model and workload selection in section 4. We follow up by detailing the tracer compiler plugin and related post-processing in section 5. In section 6, we present metrics, apply them to workloads, and discuss results. Section 7 concludes our findings.

## 2 Background

This section provides the background necessary for the understanding of the rest of this work.

### 2.1 GPGPU Programming and CUDA

Hardware design for GPUs generally favors parallelism over single-thread performance and their programming models reflect this fundamental difference from CPUs. We see OpenCL and CUDA as the two mainstream programming languages for GPUs. While in this work we focus on CUDA, all points made in this section also apply to OpenCL under different terminology [16].

A CUDA application contains both regular functions that are executed on the host CPU and specialized kernels that are executed on a GPU by potentially billions of threads simultaneously. This massive amount of parallel threads is managed by grouping them into three-dimensional thread blocks, and in turn arranging these thread blocks into a three-dimensional grid. This is typically used for domain decomposition where each thread is assigned to exactly one element in the problem domain. All threads in a grid execute the same code and use their unique thread ID to switch data locations and control flow. As multiple threads share a single instruction stream, it is encouraged to write highly regular CUDA kernels.

GPUs consist of fast shared memory, which is local to a thread block, and global memory, which is shared among all thread blocks. While threads of the same thread block can communicate (using fast shared memory) and synchronize (using barrier semantics), threads from different blocks are not allowed to interact during kernel execution. As a result, scheduling is highly flexibly, preemption is unnecessary and few guarantees exist about ordering and visibility of operations to global memory during kernel execution.

Global memory can, in contrast to shared memory, be accessed by the host system and is the primary means of communication between these two. For the majority of CUDA applications, changes to global memory are the primary visible side effect of a GPU kernel.

### 2.2 LLVM, Clang, and gpucc

LLVM is a compiler project, aiming for high modularity and extendability. The project is build around a well-defined intermediate representation (IR) that is the input and output of all optimizations [10]. The three main components are:

- the front-end, which compiles a programming language into the IR,
- the middle-end, which optimizes IR code in the form of analysis and transformation passes, and
- the back-end, which generates machine code from the optimized IR.

Clang is the front-end and driver for LLVM for number of C-based languages (C, C++, Objective-C) and, in particular, CUDA. The driver part of Clang acts as the user interface to the compiler and assembles the compilation pipeline. While C, C++, and Objective-C are built using a straight-forward pipeline consisting of front-end, optimizers, and back-end, CUDA requires a different pipeline as it contains code for two different architectures and programming models.

The CUDA pipeline is split into two parts, one for host code and one for device (GPU) code, with each part in itself containing a full compilation pipeline. The device pipeline is executed first and the resulting GPU binary is then fed back into the host pipeline. Using the embedded GPU binary and the kernel prototypes from the GPU parts of the source code, the host pipeline inserts boilerplate code to automatically register variables and kernels with the CUDA Driver [25].

Clang also provides an API to include custom optimization passes into the compilation pipeline (both pipelines in the case of CUDA). This API can be used to perform arbitrary code transformations when the compiler is used, including extensive optimizations or instrumentation.

## 3 Related Work

Communication in between multiple GPUs has received surprisingly little attention yet. Most related works in the context of data transfers and GPU systems focus on the host interface (including [6, 12]). One recent example proposes NUMA-aware GPUs [15], which extends GPU architecture with hardware and software techniques to address those NUMA effects. Proposing a multi-GPU simulation framework, [22] shortly reports cross-GPU traffic for few selected applications. Analysis of inter-thread-block locality analysis is used as motivation for L2 cache improvements by [24] and [14], the former being based on a simulation, and the latter using custom microbenchmarks for direct measurements. The approach most closely related in methodology is [24],

which proposed hardware extensions to optimize accesses to read-only data based on an analysis of inter-thread-block data locality.

The most important related works in the context of multiple GPUs rely on explicitly written multi-GPU applications for an analysis of workload behavior [13, 18, 23]. While we agree that this approach yields the most accurate results, we also see it being limited by the number of available multi-GPU applications. Instead, it is desirable to analyze any given single-GPU application regarding its potential multi-GPU scalability.

Instrumentation software for GPUs exists: SASSI [20] allows similar kernel instrumentation at higher accuracy for non-memory operations, but requires additional tools (e.g. CUPTI) for host-side setup and instrumentation and is closed source.

## 4 Methodology

This section provides an overview of the theory and implementation of the analysis, as well as the benchmarks we applied to it. We will start by defining terms and concepts relevant to the analysis, as some of them are not sufficiently well-defined in the context of GPU programming. Then, we will present our selected method for communication analysis, including out models of communication and partitioning, as well as the chosen mappings used for the partitioning. Finally, we will discuss our workload selection and the reasons behind the exclusion of some benchmarks.

### 4.1 Communications Model and Terminology

The analysis in this work focuses on the communication between thread blocks, which are the atomic scheduling unit for NVIDIA GPUs (see definition 1). GPUs are shared memory systems and, as such, do not provide explicit primitives for communication. Instead, they communicate implicitly by writing to and reading from GPU-globally shared memory (commonly called global memory). We reconstruct communication from traces of memory accesses to global memory that have been collected using instrumented applications. Shared memory (as used in CUDA terminology) is not included in this analysis since its visibility is limited to a single thread block and kernel launch.

**Definition 1.** ***Thread Block.*** *A thread block is a group of GPU threads, executed at the same time during a particular kernel, and is CUDA's atomic scheduling unit. It is identified by its position within a three-dimensional grid, which contains all threads of a CUDA kernel in execution. Identical thread block indices in different kernels refer to different thread blocks.*

**Definition 2.** ***Communication.*** *Two thread blocks A and B communicate if and only if B reads data from an address that was last written to by A. The execution of kernel of thread block B must follow that of A either directly or indirectly.*

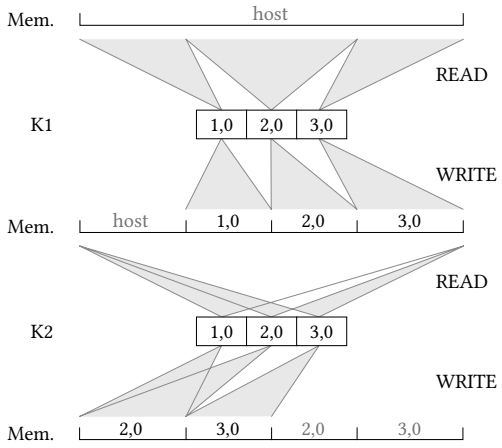


**Figure 1.** Overview of the memory state model, based on the last writer, represented by its thread block index (two dimensions only), and subsequent kernel executions (K1, K2). Note how the conflict of two blocks of K2 writing to the same memory region is resolved to the higher ID. Initial memory state is labeled as "host", as no thread has written yet.

Definition 2 describes the communication model used in this work. If defines inter-thread-block communication based on access to the globally shared memory and the thread block that most recently wrote to a particular address. Multiple writes by different thread blocks to the same address result in a data race; we simply choose the thread block with the highest index as the winner for the communication model. As a result, each byte in global memory can act as a communication channel between exactly one writer and an arbitrary number of readers.

Each CUDA stream in an application acts as a sequential pipeline for GPU commands and generates a separate trace. The trace file of a stream contains an ordered list of kernel launches, each consisting of a list of memory access records. The records of a kernel launch contain information about the thread block issuing the memory operation and the memory access itself. Within a kernel trace, the records for any given thread block are in order, but the ordering of records of different thread blocks is the result of data races.

Reconstructing communication requires walking through the kernels of a trace in the order they are launched and keeping track of the last writer to every address found in the trace by means of a lookup table. Read accesses that are encountered in the trace are then compared against this lookup table. Last writer information, i.e. the lookup table, is only updated at the end of a kernel. Figure 1 illustrates this simulated memory state for a simple application containing two kernels.

Most of the analysis in this work is simplified by analyzing a thread block's read and write sets, each containing the set of all addresses read or written, respectively, by a particular

thread block. This automatically removes redundant accesses and allows using set theory for the analysis.

### 4.2 Analysis Method

We extend the publicly available Clang/LLVM infrastructure by a set of compiler passes that insert tracing calls into GPU kernels when compiling CUDA code. Executing the CUDA application then produces one trace file per CUDA stream that contains records describing all the application's accesses to global memory. This trace file is the basis for all following analyses.

The unpredictable size of traces, which easily reaches tens of gigabytes even for small applications, renders fully buffering traces on the GPU unfeasible. Therefore, all data is pushed to the host as quickly as possible, where it is received, compressed, and stored to disk.

Post-processing then further summarizes the compressed traces into a summary of each kernel's accesses on the granularity of thread blocks. Finally, the post-processed trace summaries are used in an analysis that reconstructs internal communication between GPU kernels and computes both temporal and spatial locality.

We decided against using GPU processor simulators for simplicity and execution speed, as simulators tend to increase runtime significantly, limiting their use to small applications and problem sizes. Additionally, by providing an extension to a CUDA compiler, our approach is automatically compatible with all GPU architectures the compiler supports. Our compilation-based approach allows a use by non-expert users, as a simple recompilation of the application is sufficient.

Still, note that Clang/LLVM do not fully support all features of CUDA, thereby excluding certain applications. The following subsection will show how this influences workload selection.

### 4.3 Partitioning Model

In this work, we partition the thread grid to distribute thread blocks across virtual NUMA nodes. Each partition represents one NUMA node. The communication model is extended to analyze communication between partitions by combining all thread blocks within one partition into one large virtual thread block, allowing to reuse the same analysis.

This type of analysis primarily faces two challenges that result from the nature of kernels found in typical GPU applications:

1. Different kernels within an application exhibit very diverse behavior. Examples are N-Body computations, where one kernel computes velocities and another kernel computes positions. Of the 27 applications analyzed in this work, 17 launched at least two different kinds of kernels and 8 launched at least three different kinds of kernels.
2. Kernels can be started using different thread grid sizes, including different dimensionality. One example of this would be a hotspot simulation where the heat relaxation is computed on a 2D grid and the termination criterion is a 1D reduction that treats the buffer as a flat array. A total of ten out of the 27 analyzed applications launch kernels with different dimensionality.

The lack of uniformity between multiple kernel launches requires a way to reliably and consistently map thread blocks from a thread grid of any size to an arbitrary number of partitions. We split this problem into two parts.

The first part is a mapping

$$M_{\vec{n}} : [0, n_1 - 1] \times [0, n_2 - 1] \times [0, n_3 - 1] \rightarrow [0, 1) \in \mathbb{R}, \vec{n} \in \mathbb{N}^3$$

that maps a three-dimensional thread ID to a real number from zero (inclusive) to one (exclusive), with $\vec{n}$ describing the size of the thread grid. The second part is a mapping

$$P : [0, 1) \rightarrow \{0, 1, \ldots, p - 1\}, p \in \mathbb{N}$$
$$u \mapsto \lfloor u \cdot p \rfloor$$

that maps any real number of the interval produced by $M$ to one of $p$ partitions, numbered 0 through $p - 1$. By chaining both functions together, any thread block in any kernel of an application can be mapped to a partition consistently.

The particular choice for the first mapping $M$ significantly influences the result of the following communication analysis. A mapping has two important properties: the distribution of points in the mapping's image, and preservation of locality. Generally, a uniform distribution in the image and preservation of locality are preferred. The former translates into a uniform distribution of thread blocks into partitions for arbitrary number of partitions. The latter describes how distances in the thread grid translate to distances in the mapping's image, which determines the grouping of thread blocks into partitions. Many GPGPU applications link thread IDs to data locations, creating the expectation that spatial locality is highest between threads that are close to each other in the thread grid.

### 4.4 Selected Mappings

For this work, we choose three different mappings as basis of the following communication analysis.

The first mapping is a **lexicographic** and first assigns each thread block a number in lexicographical order and then divides this number by the total number of thread blocks. It is implemented as

$$M_{\vec{n}}(\vec{i}) = \frac{i_1 + i_2 \cdot n_1 + i_3 \cdot n_1 \cdot n_2}{n_1 \cdot n_2 \cdot n_3}$$

with $\vec{i}$ denoting a thread block ID and $\vec{n}$ the thread grid size.

The distribution of points in the mapping's image is very uniform, resulting in uniform partitions. However, locality is not well-preserved in this mapping. Distance in the mapping's image is dominated by distances in the Z-direction

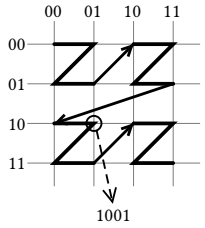


**Figure 2.** A 2D Z-order curve covering 16 points.

first, then Y-direction and lastly the X-direction. A partitioning using this mapping cuts thread blocks across the highest dimension, e.g. partitioning a two-dimensional thread grid results in large horizontal partitions spanning the full width of the grid.

The second, **colexicographic** mapping is the reverse of the lexicographic one. It is implemented as

$$M_{\vec{n}}(\vec{i}) = \frac{i_1 \cdot n_2 \cdot n_3 + i_2 \cdot n_3 + i_3}{n_1 \cdot n_2 \cdot n_3}$$

The distribution in the image is very uniform and locality in each dimension is reversed compared to the lexicographic mapping. Partitioning a two-dimensional thread grid results in vertical partitions spanning the full width of the grid.

The third mapping, using a **Z-order curve**, is a mapping from N-dimensional points to a one-dimensional range that roughly preserves locality across all dimensions. It is used in many areas in computer science, examples are storing multi-dimensional GPU textures in Z-order in order to improve cache hit rates in [5] and efficient indices for multi-dimensional data for use in databases and similar [11]. It belongs to the class of space filling curves and is popular due to its simple calculation [11]. Calculating the one-dimensional index of an N-dimensional vector in Z-order simply requires interleaving the bits of each dimension in lexicographic order. For a three-dimensional point in $\mathbb{B}^b \times \mathbb{B}^b \times \mathbb{B}^b$, $\mathbb{B} \coloneqq \{0, 1\}$, described by three binary numbers of length $b$, the Z-order index can be calculated by interleaving bits of its three components in lexicographic order. This mapping

$$Z : \mathbb{B}^b \times \mathbb{B}^b \times \mathbb{B}^b \to \mathbb{B}^{3b}$$
$$(\vec{x}, \vec{y}, \vec{z}) \mapsto z_b, y_b, x_b, \ldots, z_2, y_2, x_2, z_1, y_1, x_1$$

describes a walk through $\mathbb{B}^b \times \mathbb{B}^b \times \mathbb{B}^b$ in the shape of multiple Zs, giving the curve its name. The Z-order curve is defined for arbitrary numbers of dimensions and always calculated by interleaving the bits of its components in lexicographic order. Figure 2 illustrates the Z-order curve in two dimensions, displaying the distinctive Z-pattern.

Dividing the Z-order index of a point by the maximum number of points in the cube ($2^{3b}$) results in a value in the $[0, 1)$ range. The distribution for thread grids that are not cubic with power-of-two side lengths is very non-uniform , since only a small subset of the $2^b \times 2^b \times 2^b$ cube is populated by the thread grid. For these thread grids, the points in each dimension are scaled to fill the next largest cube with a power-of-two side length. The full formula for the mapping $M$ then results in

$$M_{\vec{n}}(\vec{i}) = \frac{Z(\vec{j})}{2^{3b}},$$
$$\vec{j} = (\lfloor i_1 \frac{2^b}{n_1} \rfloor, \lfloor i_2 \frac{2^b}{n_2} \rfloor, \lfloor i_3 \frac{2^b}{n_3} \rfloor)$$

which maps thread block indices of any given thread grid to the range $[0, 1)$. A three-dimensional index of non-negative integers $\vec{j}$ is interpreted as a three-tuple of binary numbers when applied to $Z$ and the resulting Z-order index is interpreted as a non-negative integer for the division. This approach still produces a very non-uniform distribution for workloads on a 2-dimensional grid. In these cases, an analogue approach using a 2D Z-order curve is followed to produce a more uniform distribution and therefore uniform partition sizes.

An important thing to note about all three of these mappings is that they are exactly identical for one-dimensional thread grids. In this case the mapping function is

$$(i, n) \mapsto i/n, i \in [0, n-1], n \in \mathbb{N},$$

which preserves locality very well and provides a uniform distribution of points in the image. Using these mapping functions, the resulting simulated communication between these partitions can be analyzed.

Last, note that finding an optimal mapping in terms of locality is a hard problem, and we see a high potential in exploring heuristics for this search.

### 4.5 Workload Selection

In order to improve reproducibility and minimize bias, the analysis includes commonly accepted benchmark suites: SHOC [4], Parboil [21] and Rodinia [3]. Related work pointed out that such a subset covers a reasonable set of characteristics for GPGPU applications [1]. Additionally, a few handcrafted benchmarks are used for validation purposes as well as increasing the workload spectrum. Table 1 provides an overview of the resulting set of applications.

The list does not contain all benchmarks from their respective suits. Individual benchmarks had to be excluded for a variety of reasons, ranging from unsupported features in Clang or the tracing plugin to runtime errors. We identified the following classes of errors:

**(F)** Uses features that are not supported by gpucc (e.g. texture memory).
**(K)** Application does not contain regular CUDA kernels.
**(D)** Application only contains a single kernel, resulting in no communication between kernels.
**(L)** Application uses libraries, which are pre-compiled and untraceable.

| Benchmark | Code | Benchmark | Code |
|---|---|---|---|
| Custom - 2mm | C2M | Rodinia - btree | RBT |
| Custom - fft | CFF | Rodinia - dwt2d | RDW |
| Custom - hotspot | CHS | Rodinia - gaussian | RGA |
| Custom - n-body | CNB | Rodinia - hotspot3d | RHS |
| Custom - reduction | CRE | Rodinia - lud | RLU |
| Parboil - cutcp | PCC | Rodinia - myocyte | RMY |
| Parboil - histo | PHI | Rodinia - nw | RNW |
| Parboil - lbm | PLB | Rodinia - particlef. | RPF |
| Parboil - mri-grid. | PMG | Rodinia - srad | RSR |
| Parboil - mri-q | PMQ | SHOC - bfs | SBF |
| Parboil - spmv | PSP | SHOC - reduction | SRE |
| Parboil - stencil | PST | SHOC - scan | SSC |
| Rodinia - backprop | RBA | SHOC - sort | SSO |
| Rodinia - bfs | RBF | | |

**Table 1.** Benchmarks included in the analysis and three letter codes used in figures.

| Benchmark | Error | Benchmark | Error |
|---|---|---|---|
| Parboil - bfs | F | Rodinia - nn | E |
| Parboil - sad | F | Rodinia - pathf. | C |
| Parboil - sgemm | D | Rodinia - streamcl. | E |
| Parboil - tpacf | D | SHOC - fft | E |
| Rodinia - cfd | E | SHOC - gemm | L |
| Rodinia - heartwall | E | SHOC - level 0 | K |
| Rodinia - hotspot | C | SHOC - md | F |
| Rodinia - huffman | E | SHOC - md5hash | E |
| Rodinia - hybridsort | F | SHOC - neuraln. | L |
| Rodinia - kmeans | F | SHOC - spmv | F |
| Rodinia - lavamd | E | SHOC - stencil | E |
| Rodinia - leukocyte | F | SHOC - triad | E |
| Rodinia - mummergpu | F | | |

**Table 2.** Excluded benchmarks and reasons for the exclusion.

**(C)** Compilation failed due to unsupported features in tracing plugin.
**(E)** Execution fails due to bugs in tracing library, deadlocks, or excessive trace size.

The list of applications in table 2 shows all excluded benchmarks and the class of error leading to their exclusion. As an example for the last category, the md5hash benchmark from SHOC searches for collisions for a given hash, which is passed to the kernel using multiple scalar variables instead of global memory. It does not perform any read accesses to global memory and has been excluded as well. As these benchmarks cannot be used for analysis, the total number of available applications is reduced to 27.

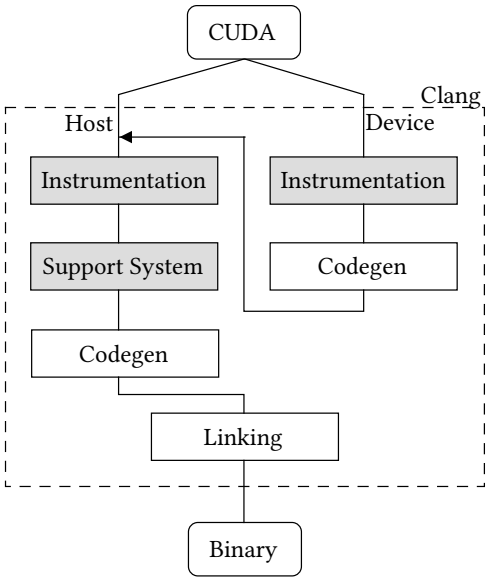


**Figure 3.** Compilation pipeline of Clang (gpucc) with tracing for CUDA. Components added for this work are highlighted in gray.

## 5 GPU Memory Tracing

This section provides a detailed explanation of the framework that is used to generate traces from CUDA applications. First, the compiler plugin implementing the tracing logic and its integration into the compilation pipeline are explained. Then, the post-processing steps to convert the raw traces into compressed application summaries are detailed.

### 5.1 CUDA Instrumentation using LLVM and Clang

Different methods exist to extract memory access patterns from a GPGPU application: *Static analysis* uses the compiler to extract access patterns purely based on the source code of the application, without executing it. Although quick, established approaches quickly become inaccurate for non-static or data-dependent control flow. In *simulations*, the application is run on a virtual model of the GPU and memory patterns are extracted from the virtual GPU. This approach is the slowest and typically provides the most accurate and detailed information. *Instrumentation* executes the application on real hardware but instructs the compiler to inject additional code that extracts information about the application at runtime. This work focuses on instrumentation for simplicity and improved performance.

For the instrumentation used in this work, we extend the Clang/LLVM compiler toolchain [1]. We decided on this toolchain because it is open source, able to compile CUDA source code, and designed in a way that makes it easy to extend. Instrumentation is separated into the three essential

[1]Source code available on Github at https://git.io/fjfyl.

parts 1) device code instrumentation, 2) host code instrumentation, and 3) host code support. Figure 3 shows how they are integrated as LLVM passes into the Clang pipeline for CUDA compilation described in [25]. The components required for tracing are highlighted in gray and roughly translate to one LLVM pass per component. The result is a trace file that contains records with detailed information about every access to global memory performed by a kernel, namely

1. the kernel identifier,
2. the type of each access (load, store, atomic),
3. the address accessed, and
4. the identifier of the thread block issuing the operation.

Instrumenting the device side is comparatively straightforward. First, all function calls are force-inlined. If any function calls are left after this step, the kernel is either recursive or it utilizes dynamic parallelism and cannot be instrumented, causing the compiler to abort compilation. Next, a list of all memory accesses is collected and a very simple approach is used to infer address spaces: the pointer argument of each memory operation is traversed and all encountered address spaces are merged, with ambiguities resulting in a failure. Finally, for all memory operations, information about the type of access, its address and the thread block identifier is collected and pushed into a host queue.

The first part of the host code transformation is the instrumentation of kernel launches. This requires first locating all kernel launch sites and extracting the name of the kernel. Using this information, code is inserted to configure the host queue and device variables that the instrumented kernel requires to access the queue.

The second part of the host code transformation is the inclusion of the runtime system that provides the queue and queue consumers. It is included by simply linking it into the application. The queue is backed by a linear buffer that works in two alternating phases:

- In the first phase, the GPU code pushes into the queue until a watermark is reached, after which a queue-full flag is set. The host does not access the queue during this phase.
- In the second phase the host clears the queue. This involves reading all records from it, compressing them and writing them to disk, after which the queue-full flag is cleared. The GPU does not write during this time.

An entry in the queue encodes the information listed above for every memory access on the granularity of a thread-warp (i.e. 32 threads). Consecutive entries are compressed in a simple run-length encoding that combines entries that are equal in all properties but the access' base address. Although very simple, the regular nature of most GPU kernels allows this scheme to achieve compression ratios of up to $25:1$ in our test cases.

Note that as neither Clang nor the plugin are fully compatible with all CUDA features, future engineering efforts might be necessary.

### 5.2 Post-Processing of Application Traces

The compressed trace files are still large, rendering exploratory analysis very time-consuming. As a solution, they are further compressed by creating a summary for each thread block of any kernel. This summary contains two sets of unique addresses, touched by loads and stores, respectively. We are treating atomic accesses as read-write operations and fold them into both sets. Whereas the trace files still contain information about the data type that was accessed, in the form of a size for each access, this information is lost in the summary, e.g. a 4 byte access to address 8 will is expanded to the set $\{8, \ldots, 11\}$. The summary of an individual thread block simply consists of its read and write sets in this form.

As an example, given a trace file that contains the following four read accesses of size 4 starting at the address 8, 10, 12, and 20, the resulting read set for that thread block is the union $\{8, \ldots, 15\} \cup \{20, \ldots, 23\}$. The union merges adjacent and overlapping sets into a single, potentially larger one. When performed for the whole kernel, this results in a map

$$\begin{aligned} K_{\vec{n}} :&[0, n_1 - 1] \times [0, n_2 - 1] \times [0, n_3 - 1] \to \mathcal{P}(\mathbb{N}_0) \times \mathcal{P}(\mathbb{N}_0) \\ &(x, y, z) \mapsto (R(x, y, z), W(x, y, z)) \end{aligned}$$

that provides the read set $R$ and the write set $W$ for a given thread block with the ID $(x, y, z)$, part of the thread grid $\vec{n}$. The read and write set are sets of non-negative integers. This is repeated for all kernels of the application in the order they were launched and the resulting sequence of kernel summaries constitutes the application summary.

This kind of summary is in-line with the CUDA consistency model, which for this analysis allows to ignore the order of loads and stores during a single kernel execution. The resulting application summary is then stored in a database for further analysis.

## 6 Trace Evaluation

This section details the analysis of the summarized trace files using several metrics. For each metric, we provide an explanation before presenting results computed on the applications selected for this work.

All metrics presented in this work are based on unique accesses per thread block and can be computed from the application summary described in 5.2.

The first part of the analysis focuses on the communication between thread blocks, as defined in section 4.1, while the second part focuses on results computed using the partitioning model from 4.3.

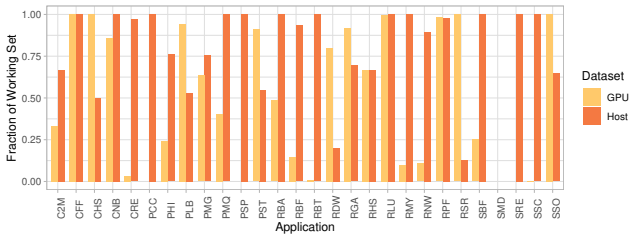


**Figure 4.** Origin of read accesses over complete applications, normalized to the size of the working set. Host refers to an address that is previously unwritten by a GPU (host data set), while GPU refers to an address that was written by a previous kernel (GPU data set).

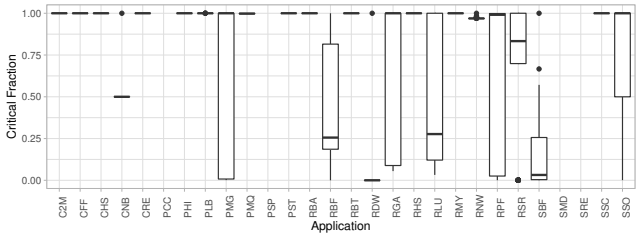


**Figure 5.** Critical volume of the GPU data set per kernel, normalized to the kernel's full GPU data set.

### 6.1 Kernel-Level Communication

The number and origin of read accesses is an indicator into the data intensity and potential communication of a GPGPU application. If a read access targets data that was last written by the GPU, its address is put into the application's "GPU data set", otherwise it is put into the "host data set". Note that an address can (and often will) appear in both categories. For this analysis, the union of the "GPU data set" and the "host data set" constitutes the application's "working set". Figure 4 shows the size of the GPU and host data set, normalized to size of the full working set. The primary insights from this metric are: (1) for the majority of applications (24 out of 27), 50 % or more of the working set are host reads, (2) for almost half (13 out of 27) of the applications, 50 % or more of the working set are GPU reads, and (3) more than one third of the applications have at least 50 % overlap between both sets, meaning the data was first provided by the host and then updated and read by the GPU.

Another interesting property of read accesses is the age of the data being read, i.e., the distance in kernel launches between the producing write and the consuming read. Assuming that a multi-GPU implementation of an application would provide a cache for read-only data, the amount of data that is written by one kernel and immediately read by the subsequent one can provide insights into the urgency of communicating a particular piece of data. It is computed by calculating the critical GPU data set of each kernel (set of read accesses originating from the previous kernel launch), and normalizing its size to the kernels full GPU data set (set of read accesses originating from any prior kernel). This metric is illustrated in figure 5 and shows that the overwhelming majority of applications has a very high amount of critical reads. For 19 out of 27 applications, more than 75 % of GPU-originating data is part of a critical path. Our key insight from this metric is that multi-GPU implementations of these applications are likely to benefit more from a high bandwidth between the different GPUs than from a sophisticated caching solution.

### 6.2 Partition-Level Communication

While kernel-level communication focuses on the overall communication volumes between kernels, partition-level communication helps the identifying NUMA effects an application is subject to and is based on the partitioning model from section 4.3. For the analysis, we interpret every partition as a NUMA zone of the system, the partitioned application is executed in.

The first partition is the fraction of write accesses that result in inter-partition communication. Together with the data from figure 4 and the applications total working set size, this metric allows estimating the amount of communication that would be generated by partitioning the application. The metric has been computed for all three mappings from 4.3: lexicographic, colexicographic and the Z-order curve. Figure 6 reports the results for 16 partitions, which at the time of writing is a usual upper limit for the number of GPUs installed in high-end server systems. Outliers are denoted by crosses. The identical behavior of many applications can be attributed to the common use of one-dimensional kernels, for which all three mappings are identical (see 4.3). For the other applications the Z-order mapping produces the best median results for all but one application. We attribute this to its tendency to preserve locality relatively well in all dimensions. This is supported by the application C2M, two chained matrix multiplications, with simple memory access patterns. This workload is the most susceptible to the mapping choice: while a lexicographic mapping provides the best and a colexicographic one the worst results, Z-order mapping is right in the middle. For workloads that do not have such a clear preference for one particular mapping, choosing the Z-order mapping avoids the worst case of either mapping and produces acceptable results in the absence of specific knowledge about the memory access patterns of an application.

The second metric using the partitioning model is the scaling behavior of total communication volume when varying the number of partitions. Due to the high number of combinations between mappings and the applications, we will focus on Z-ordering only and a subset of the applications. First, all applications without communication between partitions (perfect locality), are excluded. The rest of the applications

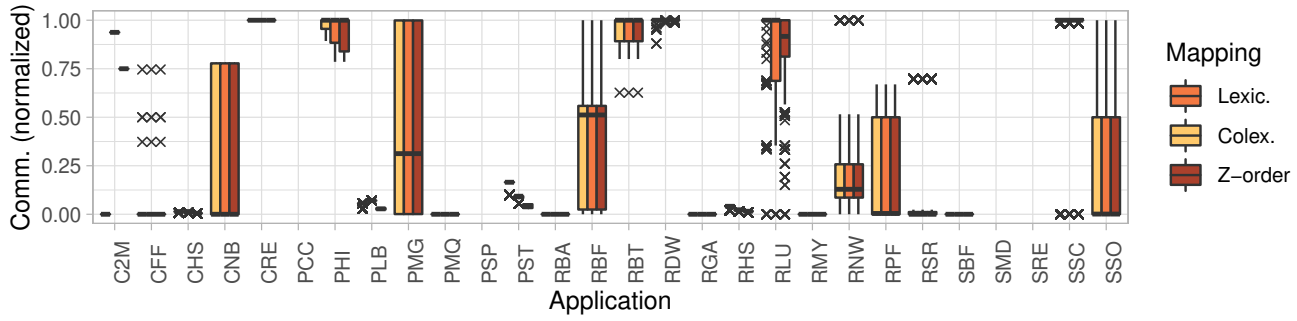


**Figure 6.** Communication volume between partitions, normalized to the application's full GPU data set (16 partitions).

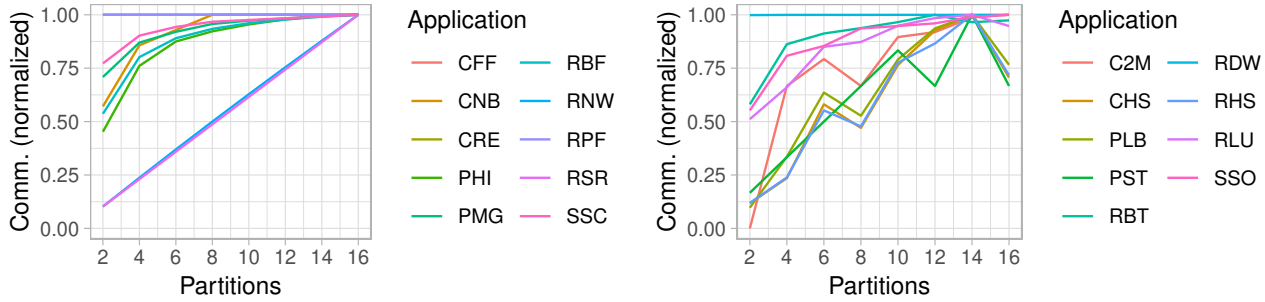


**Figure 7.** Sum of all communication between partitions, normalized to maximum value per application.

is divided into two groups, shown in figure 7. Note that all results are based on strong scaling: the problem size is kept constant.

Figure 7's left chart contains applications where communication scales "perfectly" with either $a + b \cdot p$, with $p$ being the number of partitions, $a$ a constant overhead and $b$ the scaling factor for partitions, or with limited growth $a - a/(b + p)$, $a$ being the maximum amount of communication and $b$ a static offset. Most applications in the left chart contain communication that behaves like limited growth. Note the odd behavior of the CFF and RSR workloads, where the communication is made up entirely by the outliers from figure 6.

The right chart of figure 7 contains all other applications, with many resembling an irregular version of limited growth. Local minima for partitions that are powers of two are likely caused by the way the Z-order mapping is normalized for non-power-of-two thread grids. The normalized thread grid contains points that do not correspond to any real thread block in the application, creating a non-uniform distribution of values in the mapped image of the mapping. If the number of partitions is a power of two, the non-uniform distribution still produces virtually perfectly uniform partitions; if split into another number of partitions however, partitions tend to be slightly non-uniform as well. We suspect locality suffers as well in these cases.

## 7 Conclusion

The goal of this work was providing a framework that allows the analysis of GPGPU applications to quantify the possible NUMA effects they suffer from due to their internal communication. This analysis is based on GPU memory traces created by an instrumented binary of the application in question.

The proposed plugin for the Clang/LLVM compiler is able to collect traces of all accesses to global memory from a GPU kernel, without modifying the source code. The analysis is able to reconstruct communication from memory operations using a simple communication model. It provides metrics for the applications working set as well as temporal and spatial locality on the granularity of thread blocks given a certain mapping to partitions. Interpreting each partition as a NUMA node provides insight into the NUMA effects the application is subjected to. The mapping choice for spatial locality analysis is robust and, intuitively, a mapping that preserves locality in all dimensions also results in the least communication in the cases it applies.

We also see the framework and analyses as a sound approach to evaluate both the overall communication behavior of applications, and the effectiveness of different thread block mappings in terms of communication overhead. We anticipate that much better mappings can be found on a per-application basis, and that such a selection can hopefully be correlated with abstract application characteristics.

We believe these initial results to be of value for the community, and that the methodology and tool are good starting points when porting single-GPU applications to multi-GPU systems or pinpointing performance bottlenecks caused by data transfers.

## Acknowledgements

We thank the GPGPU 2019 reviewers for their valuable feedback. We thank our our fellow researchers at Heidelberg University for their helpful input, especially Peter Zaspel and Dennis Rieber. This work was supported, in part, by the BMBF and Google.

## References

[1] V. Adhinarayanan and W. Feng. 2016. An automated framework for characterizing and subsetting GPGPU workloads. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 307–317.

[2] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 320–332.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54.

[4] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*. ACM, New York, NY, USA, 63–74.

[5] M. Doggett. 2012. Texture Caches. *IEEE Micro* 32, 3 (May 2012), 136–141.

[6] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Edahiro. 2013. Data Transfer Matters for GPU Computing. In *Proceedings of the 2013 International Conference on Parallel and Distributed Systems (ICPADS '13)*. IEEE Computer Society, Washington, DC, USA, 275–282.

[7] N. Goswami, R. Shankar, M. Joshi, and T. Li. 2010. Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications. In *2010 IEEE International Symposium on Workload Characterization (IISWC 2010)(IISWC)*, Vol. 00. 1–10.

[8] A. Kerr, G. Diamos, and S. Yalamanchili. 2009. A characterization and analysis of PTX kernels. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 3–12.

[9] G. Kim, M. Lee, J. Jeong, and J. Kim. 2014. Multi-GPU System Design with Memory Networks. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 484–495.

[10] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[11] Jonathan K Lawder and Peter JH King. 2000. Using space-filling curves for multi-dimensional indexing. In *British National Conference on Databases*. Springer, 20–35.

[12] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 451–460.

[13] A. Li, S. L. Song, J. Chen, X. Liu, N. Tallent, and K. Barker. 2018. Tartan: Evaluating Modern GPU Interconnect via a Multi-GPU Benchmark Suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 191–202.

[14] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-Aware CTA Clustering for Modern GPUs. *SIGPLAN Not.* 52, 4 (April 2017), 297–311.

[15] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, and David Nellans. 2017. Beyond the Socket: NUMA-aware GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 123–135.

[16] Aaftab Munshi. 2009. The opencl specification. In *Hot Chips 21 Symposium (HCS), 2009 IEEE*. IEEE, 1–314.

[17] M. A. O'Neil and M. Burtscher. 2014. Microarchitectural performance characterization of irregular GPU kernels. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. 130–139.

[18] Shaohuai Shi and Xiaowen Chu. 2017. Performance Modeling and Evaluation of Distributed Deep Learning Frameworks on GPUs. *CoRR* abs/1711.05979 (2017). arXiv:1711.05979 http://arxiv.org/abs/1711.05979

[19] Kyle Spafford, Jeremy S. Meredith, and Jeffrey S. Vetter. 2011. Quantifying NUMA and Contention Effects in multi-GPU Systems. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4)*. ACM, New York, NY, USA, Article 11, 7 pages.

[20] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. 2015. Flexible Software Profiling of GPU Architectures. *SIGARCH Comput. Archit. News* 43, 3 (June 2015), 185–197.

[21] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. 2012. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *IMPACT Technical Report* (2012).

[22] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Rafael Ubal, Xiang Gong, Shane Treadway, Yuhui Bao, Vincent Zhao, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. 2018. MGSim + MGMark: A Framework for Multi-GPU System Research. *arXiv e-prints*, Article arXiv:1811.02884 (Oct. 2018), arXiv:1811.02884 pages. arXiv:cs.DC/1811.02884

[23] Nathan R. Tallent, Nitin A. Gawande, Charles Siegel, Abhinav Vishnu, and Adolfy Hoisie. 2018. Evaluating On-Node GPU Interconnects for Deep Learning Workloads. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, Stephen Jarvis, Steven Wright, and Simon Hammond (Eds.). Springer International Publishing, Cham, 3–21.

[24] Lu Wang, Xia Zhao, David Kaeli, Zhiying Wang, and Lieven Eeckhout. 2018. Intra-Cluster Coalescing to Reduce GPU NoC Pressure. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE.

[25] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. 2016. Gpucc: An Open-source GPGPU Compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. ACM, New York, NY, USA.