Master Thesis

Characterization of GPU Communication

Name:	Dennis Sebastian Rieber
Program:	Computer Engineering
University:	Heidelberg University
Department:	Department of Physics and Astronomy
Institute:	ZITI, Computer Engineering Group
Supervisor:	JProf. Dr. Holger Fröning
Date:	October 9, 2017

Contents

Chapter 1

Motivation

- GPUs are not CPUs. Different design philosophy leads to different application optimizations
- GPU's concurrent hardware continues to scale
- therefore optimizations of data movements, consistency etc. are required to harness compute power of new hardware generations
- Complex Applications have non-trivial communication patterns in data parallel kernels, which are not researched yet.
- This work is aimed to generate understanding of communication patterns to help build and optimize Tools (Mekong etc.) and Applications
- Generate Traces with compiler instrumentation because
- ...Process simulators to slow for real applications and only support outdated Architectures (GPUSim only supports up to Fermi)
- ...analytical modelling usually lack the accuracy to capture exact patterns in complex communication since they are based on simplifications

1.1 Goal

The Goal of this Thesis is to develop instrumentation for dynamic global memory tracing in GPU applications. The instrumentation will happen at compile time using custom plugins for Clang and LLVM to transform the source code of the original application. With this setup, traces

can be generated with any application that provides source code and across different generations of NVidia hardware. A loss in performance resulting from the traces will be tolerated.

The generated traces will be used to analyse how CTAs and kernels communicate during the execution of an application. The communication will be analysed on a qualitative level by classifying how the data is exchanged and on a quantitative level with metrics like volume, frequency and density to describe kernel and CTA interactions.

1.2 Outline

Chapter 2 presents technological background information. It will introduce BSP, GPUs, LLVM, SSA and the compiler techniques used to transform the code of the original application.

Chapter 3 discusses work related to this Thesis. It will explore work in the field of code instrumentation and GPU performance analysis.

Chapter 4 defines what communication in the context of this work and explores the analysis space.

Chapter 5 presents how the tracing is realized including the compile stack, AST manipulation and deeper code analysis in LLVM's intermidiate representation (IR).

Chapter 6 presents the set of applications that will be analysed, the different metrics are explained in detail and the results of the analysis are discussed in length.

Chapter 7 concludes and summarizes the results of this Thesis and discusses possible future directions this project could take.

Chapter 2

Background

2.1 Bulk Synchronous Parallel

Bulk Synchronous Parallel (BSP) is a bridging model introduced by [**?**] in ... . A bridging model describes a conceptual understanding of hardware for the purpose software development. The most basic of these models is the von Neumann architecture, which is still used as the basic understanding of sequential hardware in modern CPUs. In the wake of parallel computing, BSP was proposed to fulfil the same roll for the understanding of parallel hardware. The model does not specify how any of the elements should be realised in hardware or software, since it's purpose is to provide a common ground for hardware and software developers. BSP fundamental principal is the separation of computation and communication. A computer modelled after BSP requires

1. a set of components or execution units, performing the computational work of the algorithm.
2. a router to convey information between the components.
3. a facility to synchronize some or all of the components in steady intervals.

Each component performs local computations and then shares it's results with the other components via the rounter. The synchronization guarantees that all components completed computation and communication. The sequence of computation, communication and synchronization is called a superstep. One or more supersteps are used to implement an algorithm. The model allows excluding components from synchronization, if the algorithm allows or requires it.

Although all computations are supposed to be local, BSP allows concurrent read and concurrent

write access to a shared memory between the components. But this is only allowed, if the underlying memory system guarantees coherent and consistent resolution of access conflicts.

2.2 General Purpose GPU Compuing

GPGPU computing utilizes GPUs for computations other that graphics. With CUDA, NVidia offers a proprietary language to write application for NVidia GPUs. GPUs are so called "many-core" processors, with thousands of compute units (CUDA cores). CUDA cores are clustered in Streaming Multiprocessors(SM), executing 32 threads concurrently. Code executed on a GPU is capsuled in a kernel. Figure 2.1 is an overview of the features detailed in the next sections.

2.2.1 Execution Model

Threads is CUDA are hierarchically organized. Groups of up to 1024 threads are organized in a 3D block, called "Collaborative Thread Array" (CTA). The execution of one compute kernel consists of many CTAs, organized in a 3D grid. During a kernel execution, the CTAs are scheduled to a SM in a non-deterministic fashion, based on available resources. Once a CTA is scheduled, it can not be pre-empted and executes until it is finished.

The threads inside a CTA are organized in groups of 32, called warp. All threads in a warp execute the same instruction stream. If the one or more threads in a warp have a branch in the CFG, all threads in the warp execute the branch, but only the ones supposed to work create side-effects. The shared instruction stream continues if all threads finished a branch.

A kernel usually consists of more CTAs than a GPU can execute concurrently. The non-deterministic scheduling on the GPU prevents predictions on the CTA execution order, making CTA interaction in a kernel a deadlock hazard. Therefore interaction of threads belonging to different CTAs are not allowed. The execution model allows threads inside the same CTA to interact using Shared Memory, which is located in the SM and private for each CTA.

Kernels are executed in the order in which the kernel calls are issued by the application. CUDA streams allow concurrent execution of multiple kernels and data movement, given the resources are available on the GPU.

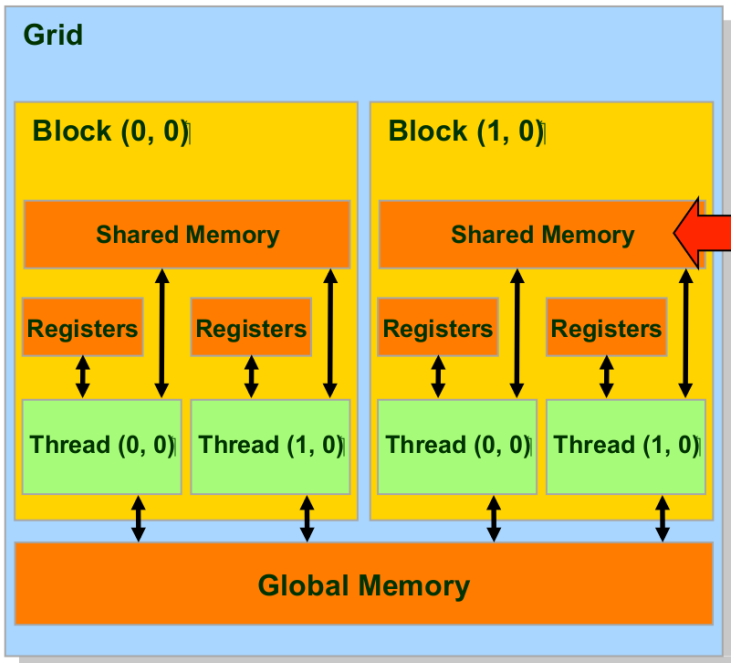


Figure 2.1: GPU thread and memory hierarchy

Memory	Scope	Access	Bandwidth	Latency	Capacity	Location
Register	Thread	RAM	high	low	32bit	SM
Shared	CTA	RAM	high	low	< 48k	SM
Global	Kernel	RAM	high	high	< 20GB	device
Texture/Constant	Kernel	ROM	high	high	< 20GB	device
Host-Mapped	Kernel+CPU	RAM	low	high	> 20GB	host

Table 2.1: GPU Memory Types, move to Background chapter

2.2.2 Memory Model

GPUs have a hierarchical memory model, with different address spaces. Table 2.1 lists all available memories and their properties. For this work we will focus on global memory, as it is the only memory allowing modification by a kernel, which persist beyond the kernel completion boundary. Host-Mapped memory falls in the same category as global, and for our purposes is treated as global memory.

It is important to note, that CUDA only guarantees a consistent state of global memory, after the kernel completion boundary. There are no guarantees on global memory consistency during the execution of a kernel, which is another reason CTA interaction at kernel run-time is not intended.

2.3 LLVM

LLVM is a compiler project providing tools for full-stack compiler development. While originally written for C/C++, nowadays front-ends for many languages are provided and back-ends for many different architectures are available. The C/C++ compiler using the LLVM tool-chain is called Clang. Since ... Clang and the LLVM tool-chain support CUDA code and can produce PTX files for GPU execution. LLVM uses an intermediate representation (IR) for code analysis and optimization (more on this in 2.3.1). Code compiled with Clang is translated into IR and then further processed.

For this work, we will distinguish the tool-chain into three major components:

- The **front-end** takes care of pre-processing, lexical, and syntactical analysis of the original code. Clang uses an Abstract Syntax Tree (AST) to represent the original code. This AST is accessible via an API and additional plug-ins can be added to the tool-chain. After the lexical and syntactical analysis are complete, the code is translated into IR.
- The **optimizer** uses the IR to analyse and then transform the code. The IR is accessible with an API and allows the introduction of additional modules for analysis or transformation. These modules are called a "pass" (details in section 2.3.3).
- The **back-end** includes the linker and back-end architecture depended code generation. This part of the toolchain is not relevant for this work and is only mentioned for completeness.

This overview gives an sufficient understanding of the LLVM stack for this work and in the following sections the concepts of IR and passes are explored in more depth.

2.3.1 LLVM IR Program Representation

LLVM's IR is a typed RISC instructions set with a load/store memory architecture. The IR is architecture agnostic and provides an infinite set of virtual, typed registers. The available primitive language-independent types are: (un)signed integer (8-64 bit), single and double precision floating point, and Boolean. Derived from these primitive types are the aggregate types pointers, arrays, structures, and functions. In LLVM it is possible to transform form any type into an arbitary other type, using the `cast` instruction.

Any kind of pointer arithmetic and access to aggregate types is performed by the `getelementptr` (`gep`) instruction, which preserves type information, is machine and language independent, and

```
1 a = 5
2 b = a + 1
3 a = 2
4 c = a + 1
```

```
1 a1 = 5
2 b1 = a1 + 1
3 a2 = 2
4 c1 = a2 + 1
```

Figure 2.2: The left hand side is normal program text, on the right hand side the corresponding SSA representation. It is customary to use a counting index for successively defined variables in SSA. Variable a's value is changed in line three. Therefore, a new definition is used in the SSA form

returns a pointer. Using `gep` allows the IR to keep load/store instructions clean and only use the direct address to access element.

One important part of LLVM's code transformations are canonicalization and lowering, which happenin the front-end. One example for lowering this is the `gep` instruction, that brings any kind of language intrinsic for address access into the same form. Canonicalization happens for logical constructs in the programs Control Flow Graph (CFG). For example, any loop in a program is brought into the same canonical form before any transformation on the loop is performed. The reason for this is the simplification and optimization of the passes analysing and transforming the program.

2.3.2 Static Single Assignement

LLVM's IR is represented in Static Single Assignement (SSA) form. Informally, it can be defined as "A program is defined to be in SSA form if each variable is a target of exactly one assignment statement in the program text" **??**.

ϕ Functions

A simple example of a program and for the corresponding SSA form is displayed in figure 2.2. This example has no branches and would result in a strictly sequential CFG. Figure 2.3 shows how a branching program creates divergence in the CFG. This creates additional basic blocks, which then merge back together at the end of the branch. The program used in 2.3 is not in SSA form, as the variable `b` is defined at two different places in the program text. One of the most integral concepts of SSA is used to resolve situations like this. The ϕ-function (sometimes called ϕ-node), is a "pseudo assignment function" [**?**] which resolves multiple incoming values at the location a CFG merges back together by defining a new variable that is used in the later program flow.

```
1 if (a = 5)
2     b = 3
3 else
4     b = 5
5 print(b)
6
```

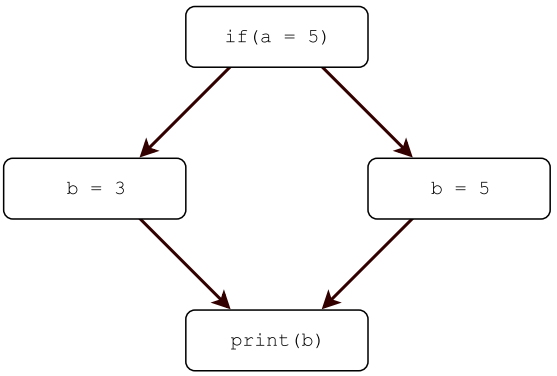


Figure 2.3: Branching program and resulting CFG. Not in SSA form.

```
1 if (a1 = 5)
2     b1 = 3
3 else
4     b2 = 5
5 b3 = phi(b1,b2)
6 print(b3)
7
```

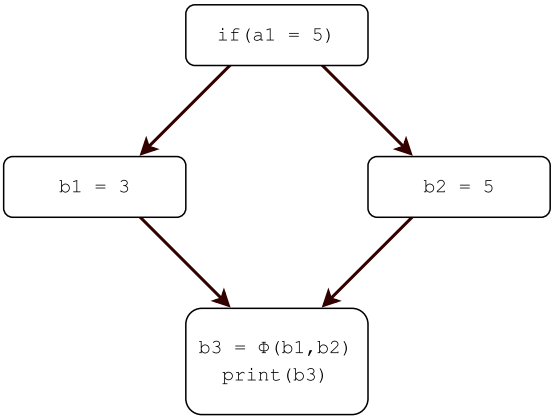


Figure 2.4: Branching program and resulting CFG. Now in SSA form, with ϕ-function resolving merging CFG paths

The ϕ-function has n parameters for the n incoming edges to the basic block. For the definition of the new variable, it selects the value belonging to edge actually coming in at runtime. It is important to keep in mind that ϕ-node are a helper, that is not actually present in the final program. After analysis and transformation are finished by the optimizer, the SSA form is deconstructed by the back-end. Now, as values are mapped to registers and memory locations that can be overwritten, the ϕ-nodes and SSA form are no longer necessary.

Def-Use and Use-Def Chains

An important data structure in compiler analysis are def-use and use-def chains. The former is list of every use a definition has. Use-def chains point backwards to all definitions of variable, from the perspective of a use.

Due to the single definition variables of SSA, use-def chains are a single name, or element. Def-use chains can be build easily because they can be constructed through the single element

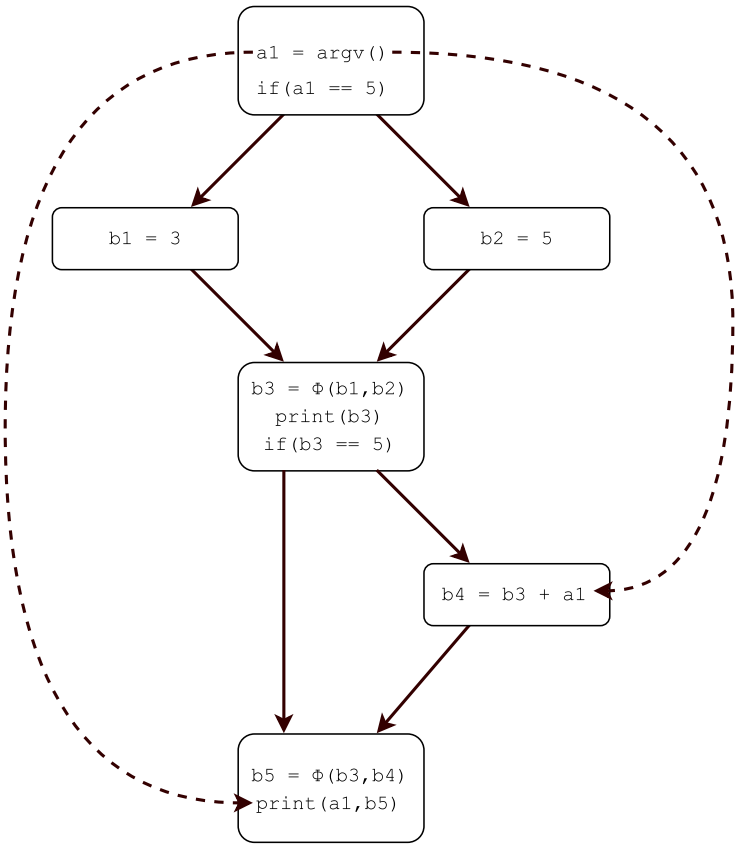


Figure 2.5: Def-Use connections of variable a1 displayed as dotted lines.

use-def chains. We look at each use in the program and add the user to the definition's def-use list. Def-Use connections enable fast forward travelling through a CFG, because we are not bothered with code that is not part of the def-use chain. Figure 2.5 shows an example of def-use connections for the variable `a1`. The use-def connections would be the inversion of the dotted arrows.

2.3.3 LLVM Pass

A pass in LLVM is piece of software responsible for an analysis or transformation of IR. Passes are distinguished into said categories. An analysis pass creates or computer some information on the piece of IR it is running on, which later on can be used by other passes. It does modify the IR. As one pass may depend on the result of another analysis, passes are scheduled accordingly by the pass manager. A transformation pass alters the IR in place and does not produce any results. A transformation might trigger the re-run of an analysis as changes to the IR might invalidate the original results. Opposed to an analysis, transformations are not allowed to rely on results of other transformations.

Scopes

A pass can run various scopes of IR. The pass can not access IR outside it's context and in the same pass, information can not be conveyed from one context to the next. For example: If a pass analyses functions, each function in the IR is analysed individually, with no knowledge about other functions.

- **Module** scope contains the whole compilation unit with all it's functions and definitions. It is largest scope available.
- **Function** scope looks at every function individually. As the pass is limited to a function definition, it has no calling-context information.
- **Loop** scope only provides access to the loop head and body, including loops nested into the loop.
- **BasicBlock** can only access a single basic-block. No CFG Manipulation possible.

2.4 Compiler Techniques

This section will briefly introduce terminology for two types of analysis and transformation a compiler can perform, which are relevant for this work.

2.4.1 Data Flow Analysis

Data flow Analysis is a broad field, with some shared terminology that is described here. Generally, such an analysis traverses the code, search for specific facts and optimizing the code according to the analysis results. Examples are constant propagation or zero value propagation. It is also closely related to pointer analysis, in terms of terminology and overall structure.

- **Property Space** is a set of analysis facts. Represented as partially ordered sets with anti-symmetric, transitive and reflexive relation.
- A **Transfer function** classifies each visited analysis object, and determines to which set of the property space an object belongs to. To avoid loops, the transfer function needs to be monotonic.
- **Program Representation** is how the program is represented during the analysis. Usually, it is the CFG, but more sparse representation like def-use connections can be used as well.

- **Context Sensitive** describes the attribute, that the analysis of a function is sensitive to the context the function is called in. For example, the function takes the functions arguments into regard.
- **Flow Sensitivity** defines that the sequence of instructions is importance for the analysis.

2.4.2 Function Specialization

Specialization (sometime procedure cloning) create distinct versions of a functions for a specific purpose at compile time. The technique was first introduced in [**?**]. It resembles templates in C++, creating a function for every template used argument. A specialized function relies on information gathered during the analysis to optimize create a specialized version. An example for this are functions with differently unrolled loops, depending on the calling context of the function.

It differs from inlining, because it doesn't interfere with the original code structure and does not inflate the source code size. The difference to traditional data-flow analysis based optimizations is that it's possible to create multiple specialized versions for multiple data-flow facts, that vary depending on the calling context.

Chapter 3

Related Work

3.1 GPU Application Characterization

Since GPGPU computing exists, there are publications characterizing GPU application. This happens usually on two levels. Either, the whole application, with all it's kernels and iterations is examined, or low-level features like branch divergence or cache misses are subject to the research.

Works like [], [] or [] broadly analyse whole applications on a superficial level, without looking at the interactions of different elements inside the application. Metrics used are ...

Publication like [] or [] focus on the impact of architectural changes on applications. The paper [] focusses on the impact of the dynamic parallelism on GPU workloads. However, to the best knowledge of the author, there are no publication researching intra-application communication.

3.2 GPU Code Instrumentation

Instrumenting existing application with addition code for memory tracing is essential for this work. A framework for code instrumentation is presented in "Lynx: A Dynamic Instrumentation System for Data-Parallel Applications on GPGPU Architectures" [**?**] by ... . Lynx is a framework including a C-to-PTX JIT, that extends CUDA code on PTX instruction level and a runtime, displayed in 3.1. The user writes instrumentation specification in a C-style language. The specification is translated to PTX and inserted into the original application.

The application can be instrumented on kernel, basic block and instruction level. An API offers access to CUDA constructs like Thread and CTA IDs, CTA barriers and instruction counts. To

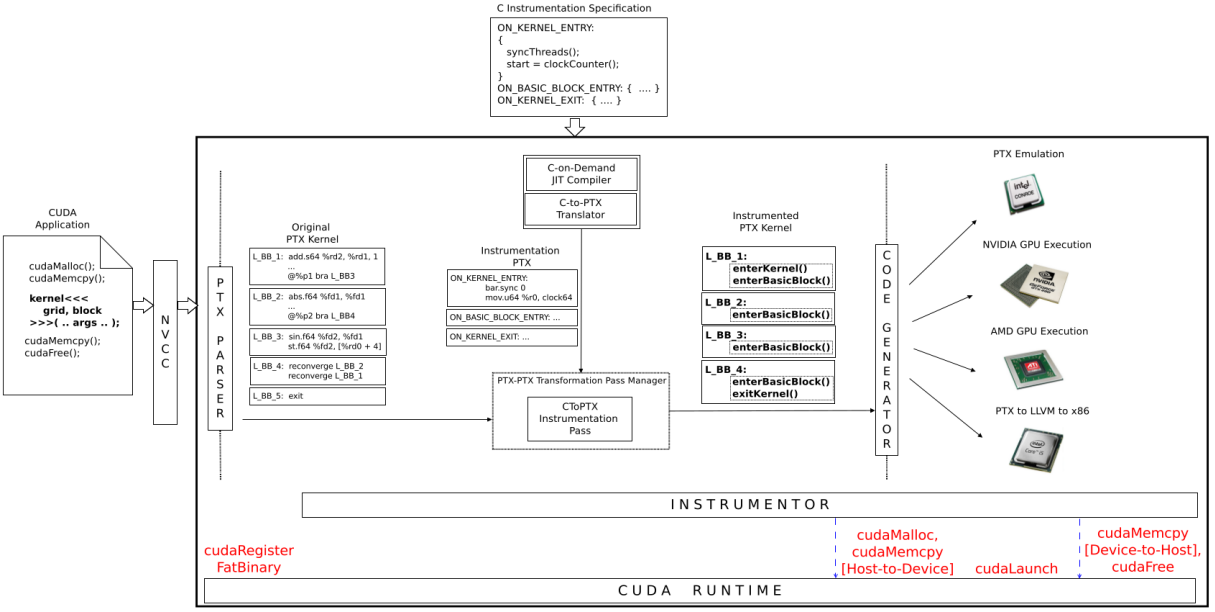


Figure 3.1: Lynx framework with all components. The original application is instrumented on PTX level, with code generated by a C-to-PTX JIT from instrumentation specification. Generated data is moved and form the device by the runtime.

increase efficiency, it is possible that only one thread in a warp performs the specified trace operation. It is possible to access shared and global memory space. The runtime fetches all generated data from the device after the application has finished.

There are two reasons, Lynx is not suitable for this project. First, while it is possible to instrument global memory operations, we found no way to access details about the instrumented instruction, like target address or type size. However, address information is crucial for this work. The second reason is the runtime handling the data buffers. As the generated trace data can be very big, device memory would not suffice to hold all the data, and creating a dynamic producer-consumer buffer using Lynx seemed not feasible.

Chapter 4

Methodology

This chapter explores how GPUs convey information using a shared memory and defines what communication means in the context of this work. Using this definition, the analysis space is described and explored.

4.1 GPU Shared Memory Communication

Other than the explicit communication routines of MPI like send and receive, GPUs convey information implicitly via the global memory on the device. The CUDA execution model and the given guarantees can be mapped to the Bulk-Synchronous-Parallel (BSP) bridging model, explained in **??**.

- **Computation** is a kernel executed on the GPU. The local processors in the model are represented by CTAs on a GPU, each processing it's own workload without the ability to directly interact with other CTAs.
- **Communication** in this step data generated in the computation step is placed in a location where it can be accessed by the processor using the data in a following superstep. For GPUs this location is global memory, because it is modifiable by a kernel during execution and it's modifications persist beyond kernel completion boundaries. In other words, global memory is a memory area allowing visible side-effects of a kernel execution.
- **Synchronization** on the GPU happens implicitly by the kernel completion boundary. It guarantees visibility of all global memory operations by the kernel. This implies that follow-up supersteps can see the memory modifications. Just as the barrier in BSP makes sure the communication step has been completed by all processors.

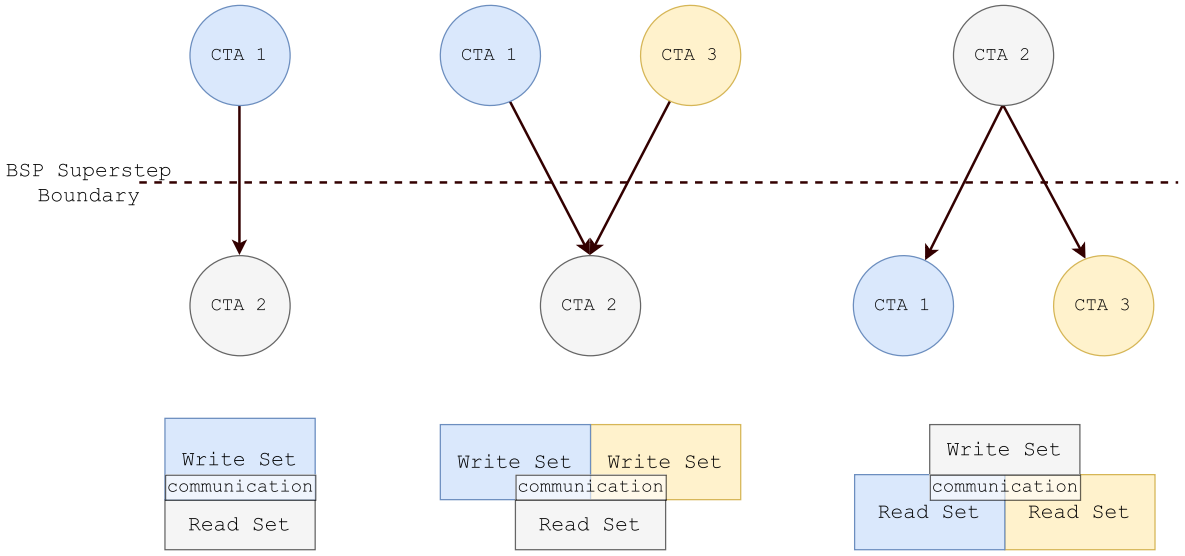


Figure 4.1: Communication of CTAs across a BSP superstep boundary, and how this corresponds to read and write sets of sinks and sources. Edges are logical representations of communication, which is actually the overlap of write and read addresses in different BSP supersteps. Different colors represent different sink/source entities.

Not all of the side-effects mentioned above however, can be classified as communication. In the context of this work, communication is data that is written to global memory by one thread and read by another. The CUDA consistency model allows reliable communication only across kernel completion boundaries. We only consider elements that are communicated reliably.

To exemplify the definition above, figure 4.1 shows how communication between CTAs across a BSP superstep boundary translates to overlaps of write- and read-sets. The nodes above the superstep boundary are data sources, the nodes below sinks. The accumulated edge-weight is the size of the overlapping area in the write- and read-sets. The left-most graph is a simple point-to-point connection, the middle one resembles a gather collective and the right-most graph can be a scatter or multicast, depending on the overlap. More on collectives in 4.2.2. This representation works for all levels of granularity in the hierarchy of the GPU execution model, from a single thread to an application using multiple kernels and streams.

4.2 Analysis Space Exploration

The analysis space for this work can be separated into spatial and temporal dimensions. The spatial dimension includes the layered hierarchy of threads, CTAs and Kernels. Each each layer is the superset of the its components. For example, the total data volume written by a kernel is the sum of all its CTA's writes. Granularity determines how detailed the spatial dimension is

analysed. The coarsest granularity would be a complete application with multiple kernels and iterations, and the finest a single thread. For this work however, CTAs are used as the finest granularity. The spatial dimension can help characterize an application, but by itself cannot describe communication.

The temporal dimensions describes the series of executed BSP supersteps. Without this dimension, there is no communication to analyse, because by our definition, communication requires at least two subsequent supersteps. Again, different granularity can be applied to the analysis. Granularity decreases, as the term to describe the analysis in this dimension becomes more and more specific. The coarsest granularity ignores individual kernels and exclusively looks at the interaction between the existing supersteps. At finer granularities, the number of supersteps in the analysis may change, because only the interaction of two distinct kernels $k1$ and $k2$ across time t is of interest.

4.2.1 Analysis Methods

Based on the dimensions and granularities, different levels for the analysis can be defined.

I Kernel behaviour across all supersteps

II Kernel interactions across two supersteps

III CTA interactions across two supersteps

While the granularity gets finer, all levels try to use accumulated analysis results, rather than mean oder median values.

4.2.2 Caveats

- This work uses CTAs as the fundamental entities of communication source and sink, not threads.
- While MPI offers collective communication with clear definitions such as scatter, gather or multicast, the implicit memory communication of GPUs often can not be categorized as clearly. However the in- and out-degree of a communication entity might indicate tendencies towards a certain kind of collective. For the sake of the analysis, each collective communication ({n:1} {1:n} or {n:n}) can be viewed as a set of point-to-point communications, happening at the same time.

- A kernel execution is a BSP superstep and is serialized with follow-up kernel iterations, or supersteps. Each stream is treated as an individual sequence of BSP supersteps during the execution. Stream interactions can be recreated in the follow-up analysis.

Chapter 5

Dynamic Instrumentation Using Code Analysis And Transformation For Detailed GPU Communication Analysis

This chapter describes how the instrumentation to generate trace data is realized. Goal of the instrumentation is to modify the original source so it generates data at run time that can be used to identify communication between CTAs in an application. As global memory is the only address space allowing side-effects beyond kernel kernel completion boundaries (KCB), global memory operations are the target of the instrumentation. Data is generated by creating a trace record every time a global memory operation is performed. Before a detailed explanation of the steps, an overview of the involved components is given. The instrumentation happens during the build of the program and is handled in Clang and LLVM. They are used to insert code and functions for the data generation.

Communication across BSP steps, honouring the guarantees CUDA's execution model gives, is not bound to any timing or ordering during the execution of a single compute kernel. Negative performance impacts by the instrumentation is therefore tolerated, because in a well-written program the negative impact does not impede the correctness.

As libraries present external code not in source or LLVM IR, code using libraries can't be reliably instrumented.

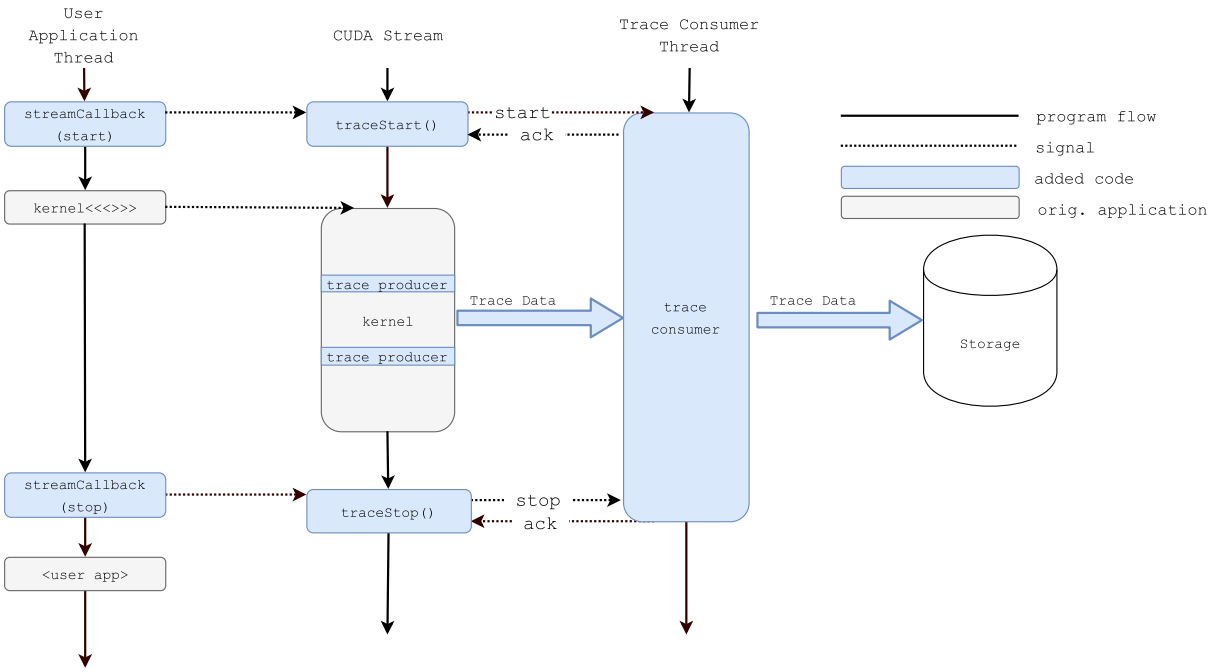


Figure 5.1: Structure of an instrumented application. Segments added during instrumentation are highlighted in blue.

5.1 Trace Setup

Figure 5.1 shows at which points the original application is extended. The instrumented kernel generates data, which is consumed by a thread running on the host. This consumer thread forwards the data to a storage, in this case a simple file. Stream callbacks are used to encapsulate the kernel between two functions controlling the trace consumer. This guarantees isolated traces for every kernel in a stream. CUDA streams allow concurrent execution of kernels and DMA data movement on one device. A stream also allows executing multiple kernels at the same time, as long as resources are available. Multiple kernels simultaneously executed using the same buffer lead to unordered data that is not clearly assigned to a stream or kernel. Post-morten separation of data is infeasible. Multiple consumer threads, each managing one stream and private buffers and storages for each of them resolve this issue. Interaction between streams can be tracked by cross-referencing the stream files in the later analysis of the data.

5.1.1 Build Process

Host side modification happens in a Clang front-end plugin. This plugin extends kernel and function declarations and definitions and includes functions used by the trace. This generates a new file that is used instead of the original source . The instrumentation to trace memory opera-

```
|32 bit |4 bit   |28 bit |64 bit |64 bit              |
|191 160|159  156|155 128|127  64|63   32|31  16|15  0|
|SMID   |Type    |Size   |Address|CTA.x  |CTA.y |CTA.z|
```

Figure 5.2: Trace record format

```
<size of one record, in bytes> //stored in a single byte
<kernel name> // name of kernel, followed by LF
<records> // all trace records, as one binary blob
000000000000000000000000 // EOT
<kernel name>
<records>
000000000000000000000000
<EOF>
```

Figure 5.3: Trace file format

tions in the kernel is done by an LLVM optimization pass and happens during the compilation of the file augmented by Clang. After compilation and instrumentation, the created object files are linked together with the rest of the application and object files used by the trace. Clangs task is detailed is section 5.2, LLVMs process is elaborated in section 5.3 and section 5.4 covers details on how data is buffered and transferred from device to host at kernel run time.

5.1.2 Trace Format

A trace record consists of 192 bits of data as detailed in code snippet 5.2, stored in LSB order. It contains the executing SM, CTA ID, size of the data type, type of memory operation and the operation's address. The record format allows to track the interactions between CTAs across KCBs on a per-address basis. Which subsets of the interactions contain actual communication is explained in chapter 6. The format of a complete trace file is shown in listing 5.3. The first byte in a file is the record length, followd by a `LF`. The next line contains the traced kernel's name. Because all records have the same size, they are stored without a separator. End of trace (EOT) for a kernel is marked by a record consisting of only zeroes. The next line contains the name of the follow-up kernel.

5.2 Source to Source Compilation in Clang

The first instrumentation step extends the host code and adds includes that are used by the kernel. It uses a Clang front-end plugin to transform a file and generate a new, augmented source code

that is then used in the build process. The plugin uses the `ASTVisitor` pattern in clang's API to traverse statements and declarations of interest. After the AST modification, a new file is written and two headers are added.

- `TraceUtils.h` contains buffer and stream management, trace consumer control and storage.
- `DeviceUtils.h` contains `__device__` functions used for the trace. The file is included in all files containing kernel code because the following LLVM optimization pass will create calls to these functions in the original kernel.

AST Declarations contain kernel and `__device__` function declarations. They are extended with the following arguments:

- **Reservation Index Array:** Array of indices for buffer space reservation. Because each buffer is segmented into various bins, each bin needs it's own index.
- **Acknowledgement Index Array:** Array of indices for write acknowledgements. Because each buffer is segmented into various bins, each bin needs it's own index.
- **Buffer Pointer:** Base pointer of the buffer.
- **Bin Size:** Max. number of elements per bin.
- **Bin Count:** Number of available bins.

The mentioned bins are a logical division of buffer space due to reasons explained in 5.4. They are not be mistaken for the private buffers of each stream! Each stream buffer is divided into bins.

AST Statements contain the calls kernels and `__device__` functions. The augmentation of the latter is trivial. The kernel's trace arguments are passed to the function. The former however needs more manipulation.

First, the stream is extracted from the kernel parameters. Two statements are inserted before the kernel call. The first one allocates stream buffers, creates the storage and spawns the consumer thread in a waiting state. This only happens once per stream. Next,1 the function starting the consumer thread is issued to the stream, using `cudaStreamAddCallback()`. Now the kernel call is extended using the buffers allocated for this stream. After the call the function

stopping the trace consumer is issued to the stream. It guarantees that the buffered data is consumed and stored before starting another trace. An explicit stream synchronization is therefore not necessary.

5.3 Code Transformation in LLVM

Instrumenting the application to generate trace data for global memory operations on the GPU happens inside an LLVM IR optimisation pass. The original source is analysed and instrumented at the locations of loads, stores and atomic operations targeting global memory, so that during the execution data is generated. Before the call kernel

Global memory is the only address space of CUDA in which side effects that are visible across kernel completion boundaries are possible. These side effects can carry information from a kernel to another and so memory operations using this address space need instrumentation. Finding loads and stores targeting global memory is not trivial in LLVM IR. Clang performs lowering of all memory access operations during the generation of LLVM IR and in the process eliminates the address space information at location of the memory operation. An example for the lowering is displayed in listing 5.4. The `getelementptr` instruction is used for address calculation, for example to access a specific element in an array. However, before this an `addrspacecall` casts the pointer form address space 3 (shared memory) to address space 0 (generic address space). The pointer used in the `store` instruction in line two does not contain information about its address space. This lack of information means the LLVM pass can not visit loads and stores exclusively to determine the address space and a more complex approach is necessary.

PTX does not necessarily need address space information for a memory operation since it can be resolved dynamically during runtime. While PTX has distinct load (`ld`) and store (`st`) instruction for each address space, a generic version is also available. This generic instruction maps to global memory, if it does not fit into a window for the other address spaces [**?**]. This lookup however comes at the cost of a minor performance decrease [**?**]. So, while beneficial it is not necessary to have the address space information to build a correct application. For the tracing process, however, we need to have this information to prevent the generation of trace data for parts of the address space that are not interesting for this work.

There is an experimental LLVM optimizer pass which tries to infer the address space by analysing the steps that have been introduced by the lowering. However, it works on a function scope which is not sufficient for our purposes. Address space information might not be existing in the function argument because of lowering of the argument previous to the function call. And due to the pass' function scope, information on the calling context is not available.

```
1 %arrayinx1 = getelementptr inbounds (float, float* addrspacecast (float
     addrspace(3)* @_ZZ5saxpyfPfS_iPlS0_E2_x to float*), i64 0, i64 2)
2 store float 1.000000e+00, float* %arrayinx1
3
```

Figure 5.4: Example for address space lowering in LLVM IR

Because the pass can fall back to the generic address space, it is not critical if the address space cannot be determined reliably. This is not sufficient for the needs of this work because the address space information needs to be definite. Therefore an analysis is performed, that generates definitive information, whether a memory access targets global memory or not. If generating this information is not possible, the pass fails. The reason for an unsuccessful execution is explained in section 5.3.1, after more details of the actual analysis have been explained.

LLVM widest analysis scope is 'Module', which roughly corresponds to one translation unit in C. Interactions of the original source that reach beyond this Module scope cannot be analysed, and therefore not instrumented.

5.3.1 Address Space Analysis

The basic principle of this analysis rests on the limited number of sources in a kernel, where a global address space pointer can be obtained from.

- direkt kernel parameters
- indirekt kernel parameters (struct etc.)
- global variables
- pointer type load from global memory

Collecting base pointers form all these sources provides starting points for a sparse analysis of the program. From the def-use chain of each pointer and recursively following each users new definitions, a usage tree for each of base pointers forms. And only memory operations that are part of def-use trees which root in the global memory pointers, can be a candidate for instrumentation.

The used algorithm performs an its analysis on the sparse program representation of def-use chains, created by the SSA form of LLVM IR. Because the the algorithm follows the order of instructions through the program, because only with proper instruction order we can obtain the address space information correctly, the analysis is 'flow sensitive'. The location and arguments

of function calls matters for our analysis, for reasons described in 5.3.1. This classifies our algorithm as 'context sensitive'. The property space of the analysis describes sets of instructions in the context of global memory operations that are defined as following:

Definition 5.1 $MOp = \{$*LoadInst, StoreInst, CallInst*$\}$

MOp contains all types of instruction, that are interesting for tracing. During the analysis they are marked for later instrumentation. LLVM maps CUDA's atomic operations to functions, which is why CallInst is part of this set. Of course they are only considered a memory access, if the callee is an atomic function.

Definition 5.2 $POp = \{$*CallInst, GEPInst, CallInst, PHINode, SelectInst*$\}$

POp contains operations that define new SSA variables which can be used in operations interesting for tracing. $CastInst$ because of the lowering and $GEPInst$ since it is the generic way in LLVM to perform address arithmetic. Their users need to be visited later. $PHINode$ and $SelectInst$ are special cases. Only instructions that resolve pointers and all arguments can be related to a global memory base pointers are a part of this set. The reason for this is explained in section 5.3.1. This set also includes function calls. However, in the case of a function not the return value definition, but the argument passed into the function marked for later analysis. More on the return values of functions in section 5.3.1.

Definition 5.3 $NMOp = \{inst : inst \notin POp, inst \notin MOp\}$

NOp contains all instructions that are of no interest for the analysis and are ignored.

To obtain a list of all instruction which require instrumentation, each of the orginal base pointers is handled by Algorithm 2 to create list of all descendant memory operations of the original by applying the transfer function $f(x)$ to every instruction processed. $f(x)$ classifies each instruction, assigning it to one of the three sets MOp, POp or NOp. After this, a list with the instructions for tracing of each function has been generated. This algorithm only proves if a memory operation uses a pointer originating from a original base pointer, and therefore uses the same address space, which can be achieved without the need to analyse cycles in the CFG. As a consequence the algorithm always terminates once each relevant instruction is visited. By keeping track of each function that was already visited, recursion is not an issue during the analysis.

```
Data: orignal pointer
Result: List of global memory operations based on original pointer
add original pointer to workstack;
while workstack not empty do
    get element form workstack;
    forall users of element do
        /* apply tranfer function f(x) to user:                    */
        if type(user) ∈ MOp then
            add user to list for tracing;
        else if type(user) ∈ POp then
            add new parent to workstack;
        else if type(user) ∈ NOp then
            continue with next user;
    end
end
```

Algorithm 1: How to find global memory operations based on a input pointer

ϕ-Node Limitations

Static analysis has some limitations that can cause an unsuccessful address space analysis. This particular issue is caused by the SSA's need to define a new version for a variable each time said variable changes. This results in the need of ϕ-functions and `select` statements, which is a LLVM IR instruction to handle small `if-else` branches without the need for multiple basic blocks and a ϕ-node. Figure 5.5 shows a minimal code example that can lead to a situation which is not analysable by our static analysis, because it requires resolving pointers to different address spaces at runtime. The code sample on the left creates a situation where a ϕ-node is required to resolve the edges of the incoming basic blocks. Each of these basic blocks defines a surrogate pointer for `d`. For this example we will call them `ds1` and `ds2`. Based on `ds1` and `ds2` the ϕ-node defines a pointer `df` that is used for the following operations performed on `d`. Therefore the ϕ-node resolves which address space `df` points to.

This is illustrated by graphs on the right hand side of the figure. Every block preceding the ϕ-node carries an address space, represented by a colour. The left graph has to resolve different address spaces and therefore the address space for `df` is ambiguous. This leads to an unsuccessful analysis. The right graph shows a special case the implemented pass can resolve by proving that `ds1` and `ds2` both point to the same address space, which concludes that `df`'s address space is definite. The proof is implemented as a table counts how often the analysis pass visits every phi-node. After the analysis is finished the value in the table is compared to the ϕ-nodes incoming edges. If the counted value and number of edges match the pass proved the

```
__global__ float *gx;
__shared__ float *sx;
float *d;
if (a > b)
    d = gx;
else
    d = sx;
```

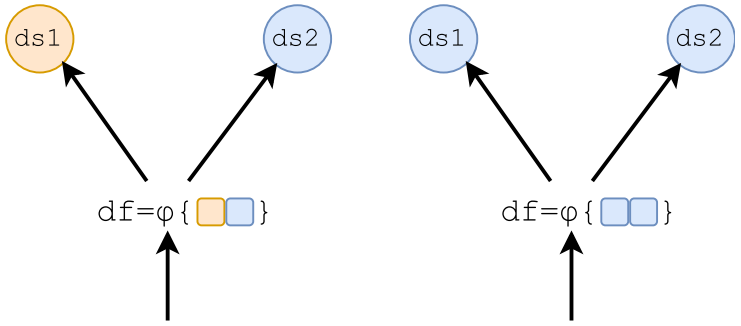


Figure 5.5: Example for static analysis limitations. Different colors represent different address spaces. The left graph represents the generic problem case and right graph a special case that is resolvable by the analysis.

special case, otherwise we have ambiguity in the address space of `df`, which leads to aborting the instrumentation pass.

Ambiguous Function Address Space

Another issue is possible ambiguity in the address space of pointers that are passed to functions. To illustrate this issue, figure 5.6 provides on the left hand side a code sample of a coalescing copy function. CUDA allows calling this device function with pointers to every address space and every combination of address spaces. The kernel on the right hand side then uses the function twice, once to copy data from the global memory into shared memory and a second time to move the data back to the global memory.

Now the instrumentation pass analyses the kernel, finds the first function call and sees that a function is called and the first parameter is a global memory pointer. It enters the function and marks the `load` instruction for tracing. The pass continues and later finds a call to the same function, enters the function again and this time marks the `store` for tracing, because the pointer led the pass to it. Now both memory operations are marked as global memory operations and generate trace data, although they are not always used with a global address space pointer. This issue is the reason, our analysis and the added instrumentation needs to be context sensitive. Potentially, each calling context has unique pattern for tracing.

Context sensitive tracing is implemented using function specialization. During the analysis a table keeps track of each function call, and which of its arguments belongs to global memory addresses. While gradually working through the initial list of global memory pointers and tracking their descendants during the analysis, a function call might be visited multiple times and each time a different argument (and thus different pointer) led the analysis there. These multiples visits form an argument pattern for each function call. After the analysis, all function calls are

```
void copy(float* src, float* dst){
    dst[tix] = src[tix];
}
```

```
__global__ void k (float* gx) {
    __shared__ float* sx;
    copy(gx, sx);
    // kernel does stuff
    copy(sx, gx);
    return;
}
```

Figure 5.6: Example for ambiguous function address spaces

reduced to a pattern list for each function. Every pattern contains a combination of arguments that belongs to the global memory address space. A pattern is stored as an unordered tuple with the length corresponding to number of global memory space pointers and each element is the index in the function's argument list. For our example the analysis creates two patterns, **{0}, {1}**. A third option would be **{0,1}**, which would indicate both pointers belong to the global memory address space. There can be no empty patterns because the analysis would never visit a function call where no global memory pointer is involved.

Now that we know which patterns exists, the original function is cloned once for each pattern. Therefore, the example generates two clones, `copy_0` and `copy_1`. Every created clone is assigned to one function call pattern. Following that, every function call in the created table gets his callee replaced with clone matching the pattern of the call. Based on the call's arguments the analysis is executed again for each function call, now marking the clone's instruction for tracing. During this process, the original function's instructions are removed from the tracing list.

After this process has finished, multiple versions of a function have been created and each of them only creates trace data if they actually required. Specifically, for the example two functions would have been created, one that augments the memory operations on `src` and one for `dst`. The respective calls to `copy` would have been replaced with `copy_0` and `copy_1`. It is possible that more than one parameter is used for tracing.

5.3.2 Instrumentation

The optimizations pass' last step is the actual instrumentation. Some preparation is performed once in every function, before the actual tracing instrumentation takes place.

1. Fetch the arguments added by the source-to-source compilation. These arguments contain the trace buffer information.

2. Add instructions to fetch SMID during runtime. The call

```
asm("mov.u32 %0, %smid;" : "=r"(sm) )
```

Data: Function Call Table
Result: List of global memory operations in cloned functions
forall *function calls in table* **do**
    get argument pattern of call;
    get or create clone for pattern;
    replace callee of call with clone;
    get new argument entrance pointers;
end
forall *new clone pointers* **do**
    re-analyse all new argument pointers;
end

Algorithm 2: Algorithm for function specialization

copies the value from the special read-only `%smid` register into the register that is accessed by variable `sm`.

3. Add instructions to calculate trace buffer bin from arguments:

```
nSlots = 1 << PowerOf2OfNumberOfSlots // 1000
slotMask = nSlots - 1 // 0111
bin = slotMask & sm
```

After the preparation, every instruction marked for tracing in this function is instrumented for tracing by adding a call to the producer function. The producer function was added by the Clang front-end plugin. The call is added before the actual memory operation.

1. Get `Type` descriptor of memory operation. `Type` specifies the kind of memory operation specifically and is defined in table **??**.
2. Fetch pointer operand that is relevant for tracing
3. Calculate the size of the memory operation in bytes. Complex data types like `structs` are recursively explored and the total size of the element is calculated.
4. Insert instructions to generate left-most part of a trace record:

```
64bitDesc = sm << 31
64bitDesc |=  Type
64bitDesc |=  SizeInBytes
```

5. Insert call to trace producer function right before the actual memory operation. This function builds the last part of the trace record, describing the CTA and then writes the data into the trace buffer selected by `bin`. The details of details of how the data is written into the buffer are explained in section 5.4.1.

Memory Operation	Descriptor
Load	1 « 28
Store	2 « 28
AtomicAdd	3 « 28
AtomicSub	4 « 28
AtomicExch	5 « 28
AtomicMin	6 « 28
AtomicMax	7 « 28
AtomicInc	8 « 28
AtomicDec	9 « 28
AtomicCAS	10 « 28
AtomicAnd	11 « 28
AtomicOr	12 « 28
AtomicXor	13 « 28

Table 5.1: Memory operation descriptor table

This is performed for every function that requires tracing. After the pass finishes, the application can be built and linked.

5.4 Tracing Process

Many applications have non-deterministic executions, for example no predetermined number of kernel calls because of varying input data. Examples are graph algorithms depending on implementation and data or numerical solvers, which iterate until a convergence condition is reached by the residual. Because of this, a static buffer that is filled up once and is cleared at the end of the application does no suffice for reliable and dynamic tracing.

Therefore a producer-consumer queue for the generated trace data between GPU and host is used that ensures the application never runs out of buffer space for the tracing data. The GPU generates the data and the host evacuates full buffers, storing the data. The buffer containing the generated data is host-mapped memory, which is separated into several bins. The bins are used to reduce pressure on the memory system that is generated by the atomic index accesses of the producer-consumer setup. The number of bins is always a 2^n and a scheduled CTA uses the n least significant bits of the SM's ID it is scheduled on, to select a bin. Therefore the number of bins should be selected as the closest power of two, to the number of SMs.

Writing the data to disk after they have been cleared from the buffer by the consumer, showed to be a performance critical aspect during the trace. Per default, Linux directs `writes()` to a page cache. Smaller, periodic writes (64Kbyte to 1Mybte) to the disk empirically showed to be

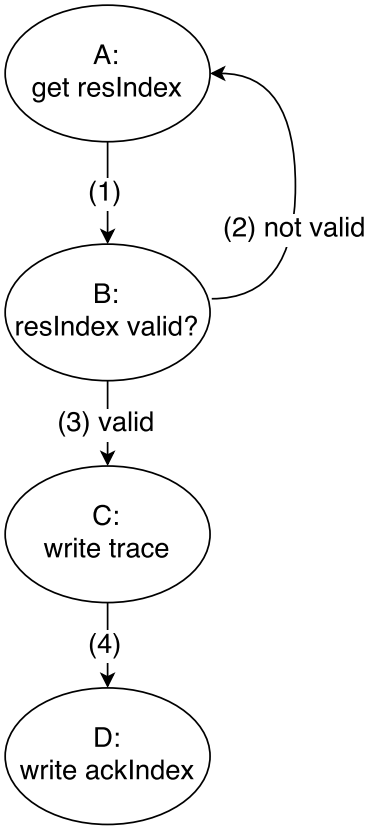


Figure 5.7: Producer CFG

more efficient. The author assumes this is caused by overlapping between cache management and queue management.

5.4.1 On-Device Producer

This section focusses on the producer that is running on the device, making sure that data is only written if there space left in a buffer. The producer-consumer setup uses a head and a tail index to handle reservation of buffer space and write acknowledgements to signal that the write has completed. Code sample 5.8 shows the pseudo-algorithm that is performed by the producer. First the required buffer space is reserved for writing by incrementing the reservation index by the number of required slots (line 1). Then the data generated by the trace is written to the buffer (line 2) and the write acknowledgement is incremented by the same number of buffer slots (line 3).

This is implemented on a warp scope, which means either the whole warp continues or waits, to prevents deadlocks during indices fetching. A deadlock can occur because of how GPUs manage branch divergence inside a warp. If the instruction stream of a warp branches, all member threads

```
1 while((ackIndex > maxIndex) or (id = atomicAdd(resIndex, increment) >
     maxIndex));
2 buffer[id] = traceData;
3 atomicAdd(ackIndex, increment);
```

Figure 5.8: Device Producer, naive approach

of this warp execute both branches, and the group of members that should not be executing this part of the code is masked out. After all branches completed execution, the combined execution of both groups continues.

A deadlock can occur, if each thread fetches it's `resIndex` individually and there is not enough available buffer space for all threads inside the warp. Referring to figure 5.7 now, all threads will execute edge (1) leading from (A) to (B) by fetching `resIndex`. If all threads get a valid index (B), the warp will not branch and the trace can complete successfully. If the check in (B) fails for one or more threads in the warp, a branch is created. The branch without valid IDs will use edge (2), returning to node (A) and retry fetching a valid `resIndex`. The branch with valid `resIndex` has to wait with progressing on edge (3) until the group without valid `resIndex` is also ready to continue on edge (3). But because the warp can not progress over edge (3) to and C and from there over (4) to D, until all threads have a valid `resIndex`, the threads in the warp that already have a valid `resIndex` can not reach node D write `ackIndex`, which would lead to a evacuation of the buffer on by the host and a reset of the `achIndex` and `resIndex`. As a result of this, a deadlock occurs because one group of threads waits for a resource that can not be released by the other group of threads due to the warp execution model.

This can be resolved by handling reservation and write acknowledgement on a warp level. One thread makes the reservation and write the acknowledgement for all threads inside the warp. Now either the whole warp can finish the trace or the whole warp stalls. This way we can use the warp execution model for our purposes, because one thread that waits for the IDs can stall the whole warp with an if branch and no other means of synchronization. This is implemented by using CUDA warp intrinsics, which offer efficient communication inside of warps. Little extra code is needed shift everything to a warp-scope management, as seen in Figure 5.9. Because it is possible that a trace occurs on a point during the execution, where the warp is already branched we have to make sure that the indexes are managed correctly and only the currently active threads participate in the trace. The warp intrinsics `__ballot`, `__popc` and `__ffs` are used to determine how many threads are active, and which one has to perform the atomic operations to handle the producer synchronization. Then `__rlaneid()`, which is user code, calculates the relative lane id for each warp.

While the lowest thread fetches the ID for the buffer using `atomicAdd`, the whole warp stalls

```
active   = __ballot(1); // get bitmap of active threads
nActive  = __popc(active); // get count of active threads
lowest   = __ffs(active)-1; // check which thread has to perform sync
rlane_id = __rlaneid(active, lane_id); // each thread gets its relative Lane
    id, based on number of active threads

if (lane_id == lowest)
    while( ackIndex >= maxIndex-maxWarpWrite
        || (id = atomicAdd(resIndex, nActive)) >= maxIndex-maxWarpWrite
    );
// Warp stalls here until all branch is completed

idx = __shfl(id, lowest) + rlane_id; // distribute id
buffer[id] = traceData;
if (lane_id == lowest )
    atomicAdd(ackIndex, n_active);
```

Figure 5.9: Device Producer, warp scope

at the end of the if clause because of the warp execution model. Once a valid id is fetched, it is distributed among all threads using the `__shfl` intrinsic. Then the trace is written and the lowest thread increments the write acknowledgement by the number it increased the reservation.

Notice that in this case, the conditions for fetching the id now takes into account that data is written in blocks and has to check if there is still space left for one complete block.

Assuming a warp is fully populated warp at the moment a trace happens and thus always reserve enough buffer space for a whole warp can create gaps in the trace data. To prevent old data from lingering, the buffer would need to be wiped after every evacuation of the generated trace data. Additionally, these gaps would require additional handling in the analysis of the data. Therefore, this more complex approach of handling the consumer was chosen, which only reserved as much space as required.

5.4.2 Host Consumer

The consuming thread on the host side is much simpler. A single thread is iterating over all bins in the buffer, checking if the `ackIndex` is at the threshold for evacuation. This threshold is reached if there is not enough space for a complete warp to write a trace. The buffer is evacuated by writing the data into a file handle and `resIndex` and `ackIndex` reset to the beginning of the buffer.

Chapter 6

Evaluation and Analysis of GPU Communication

This chapter examines anylsis results of the traces generated with the technique described in chapter 5. The analysis will focus on aggregated statistics, rather than averaging.

6.1 Evaluated Applications

Set of Functions that can display a communicating behavior, tend to be iterative kernels

- Histogram (CUDA Suite)
- Stencil (Rodinia 2D, 3D)
- NBody (CEG)
- BFS (Rodinia)
- Pathfinder (Rodinia)

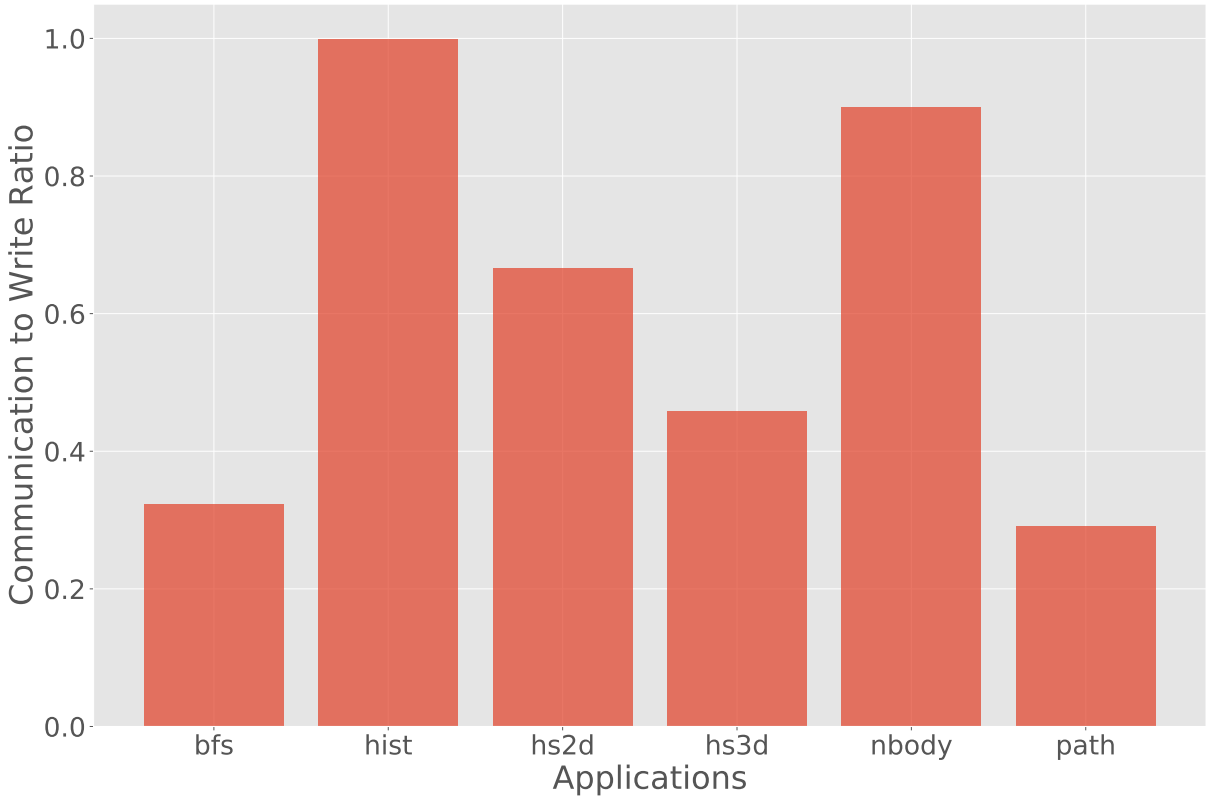


Figure 6.1: Ration of communication to total of all writes.

6.2 Tier I Analysis

6.2.1 Communication Fraction

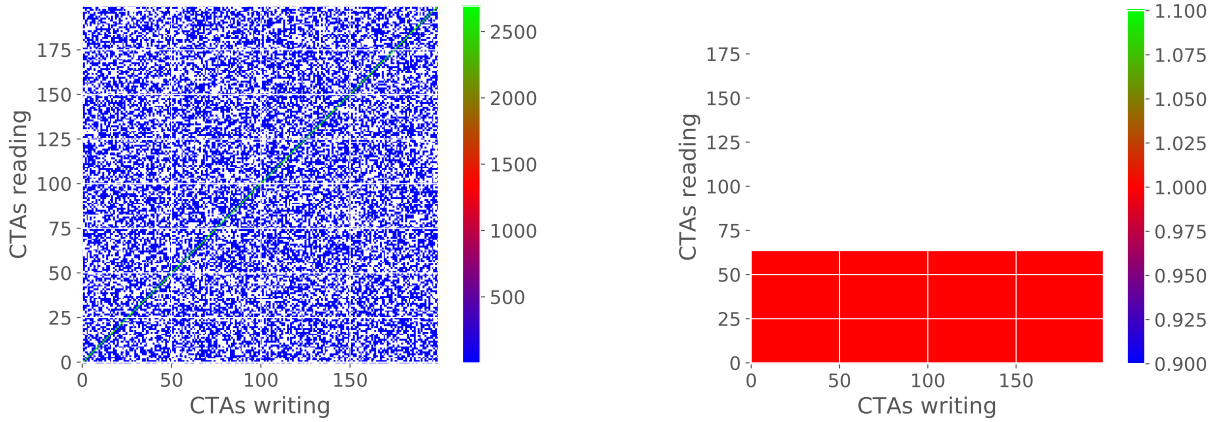
6.2.2 Density Map

6.2.3 Volume Map

6.2.4 Msg-size Histogram/CDF

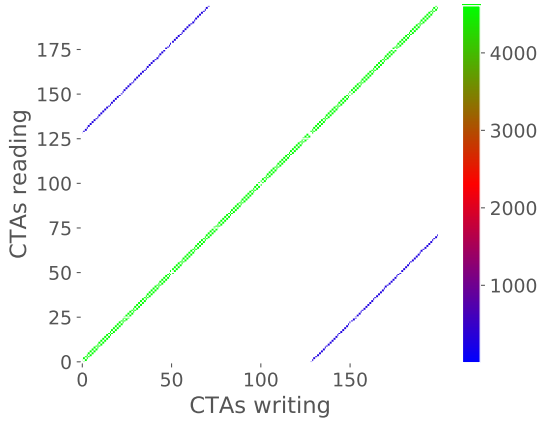
6.2.5 CTA In/Out-Degree

6.2.6 Bisection Volume

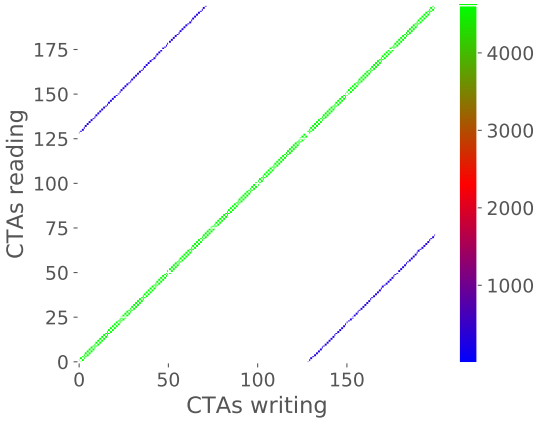


(a) BFS

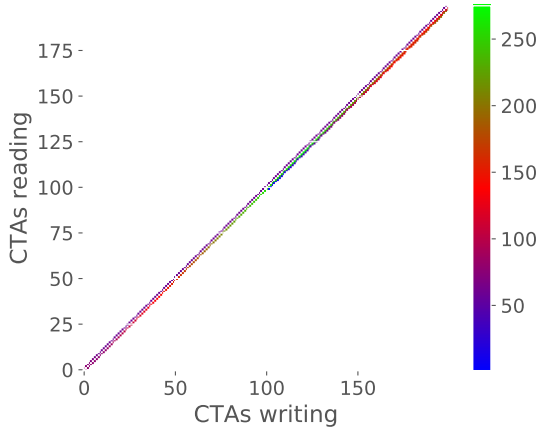
(b) Histogram



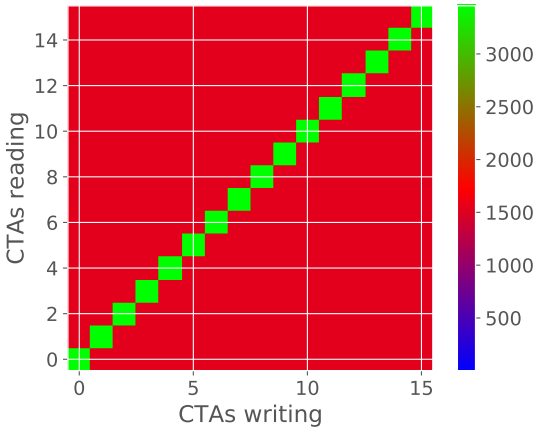
(c) Hotspot 2D



(d) Hotspot 3D

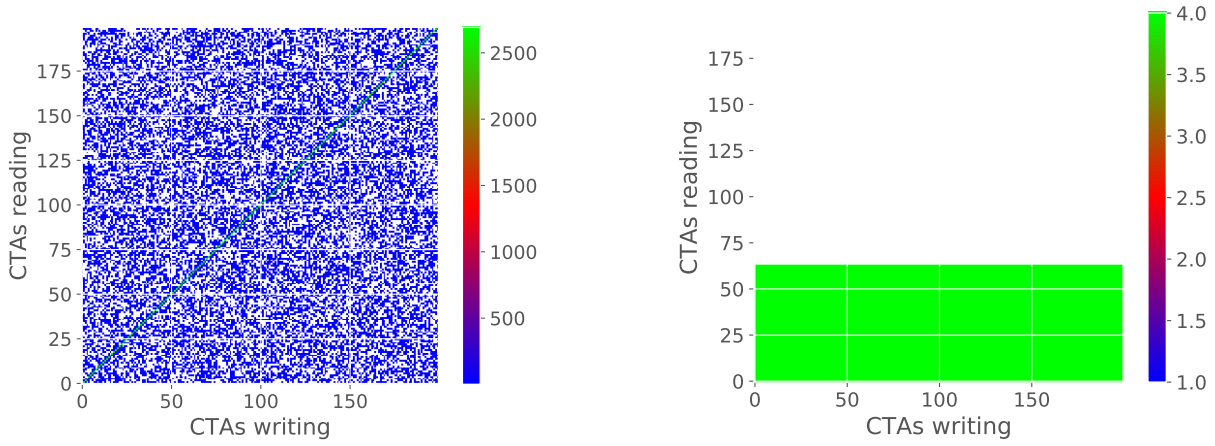


(e) Shortest Path



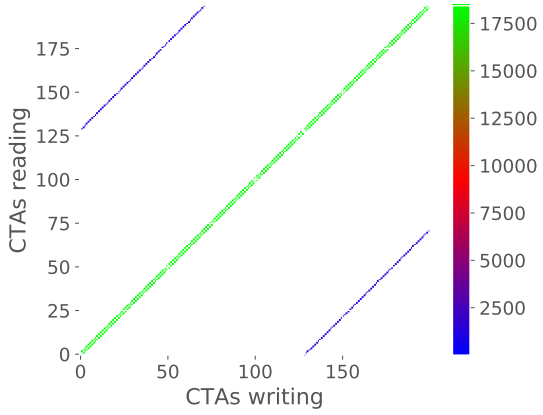
(f) NBody

Figure 6.2: Transfer Density Plots

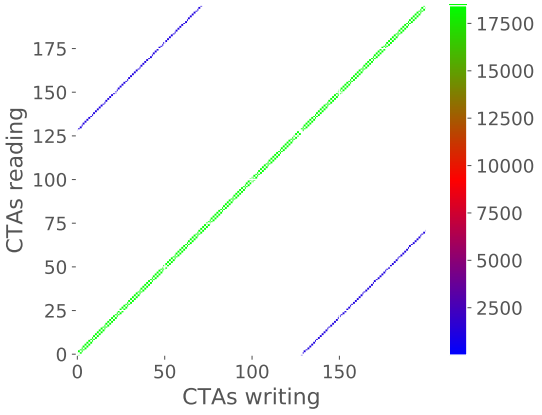


(a) BFS

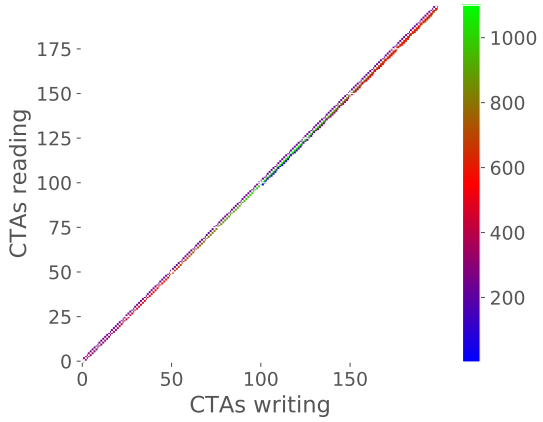
(b) Histogram



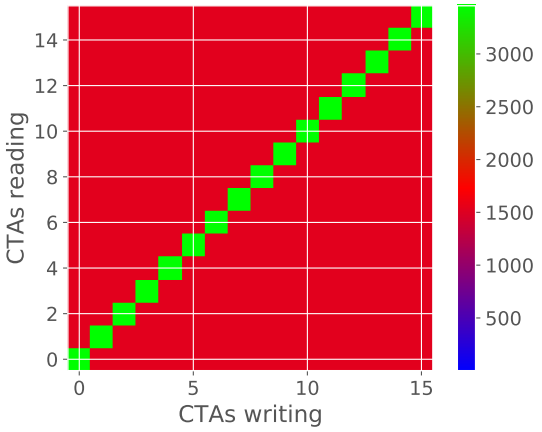
(c) Hotspot 2D



(d) Hotspot 3D

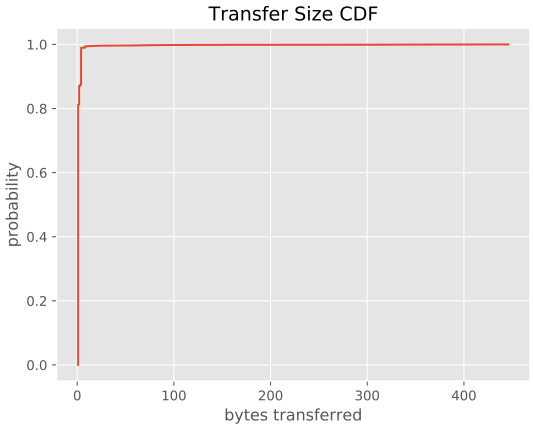


(e) Shortest Path

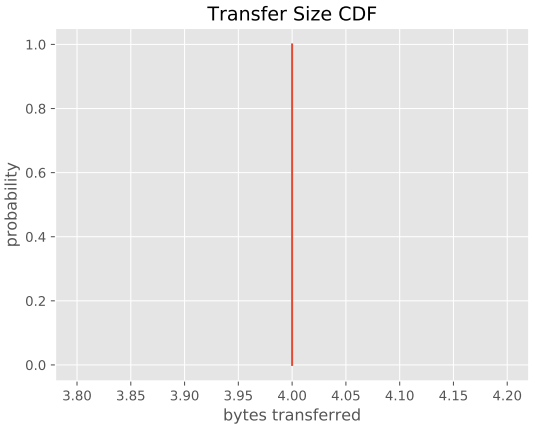


(f) NBody

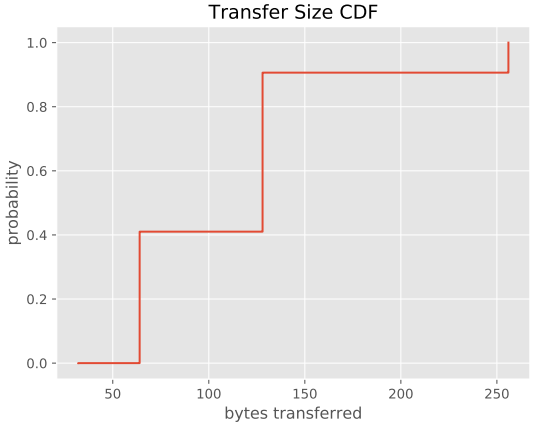
Figure 6.3: Transfer Volume Plots



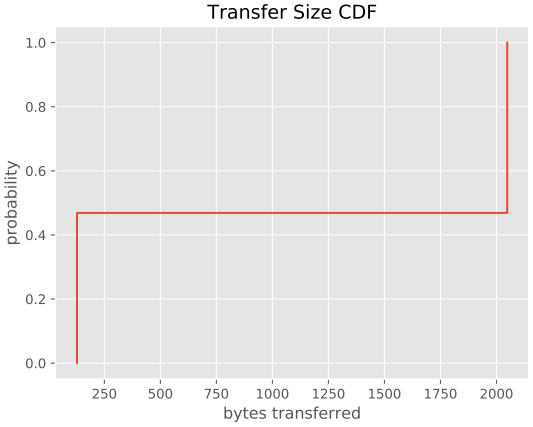
(a) BFS



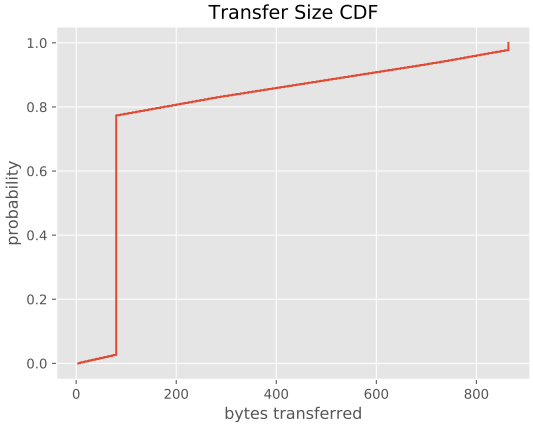
(b) Histogram



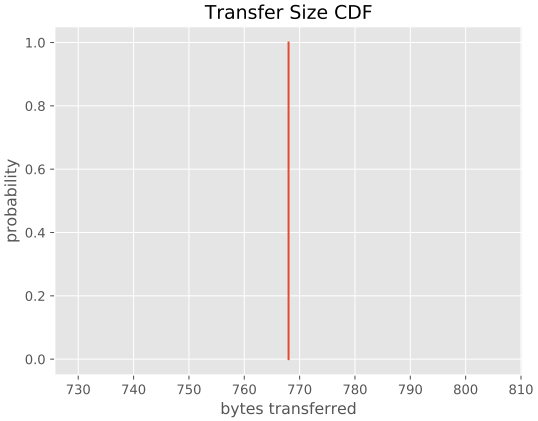
(c) Hotspot 2D



(d) Hotspot 3D

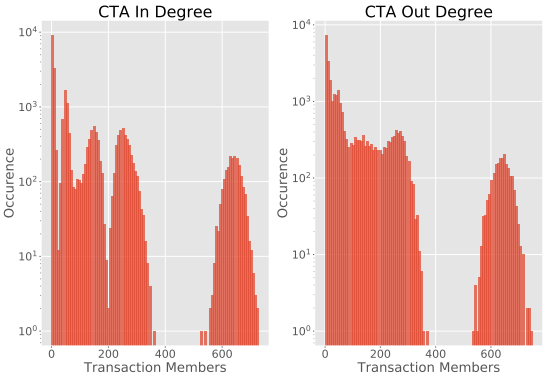


(e) Shortest Path

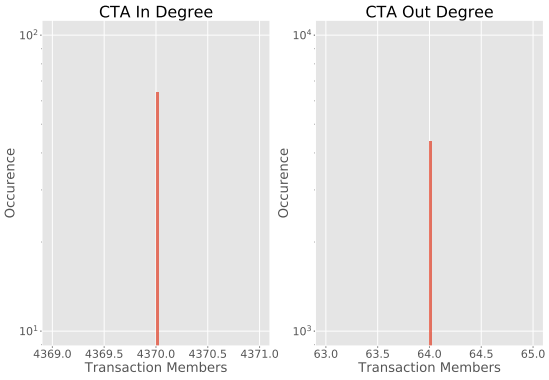


(f) NBody

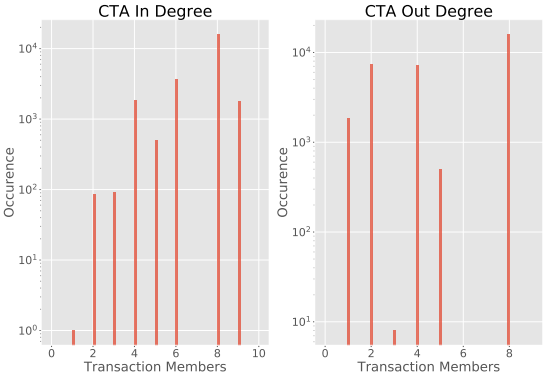
Figure 6.4: Message Size CFD Plots



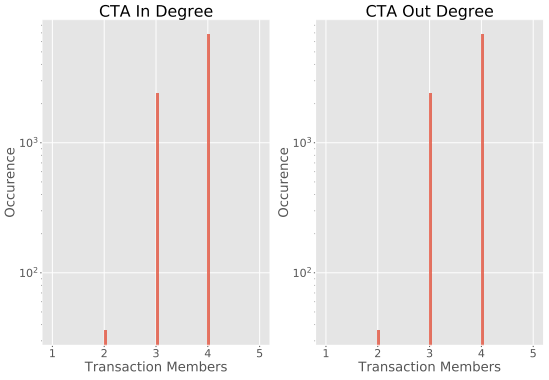
(a) BFS



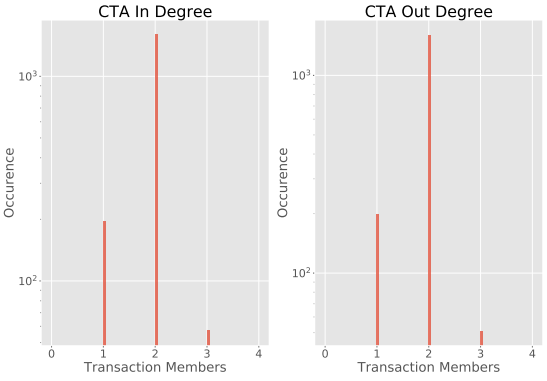
(b) Histogram



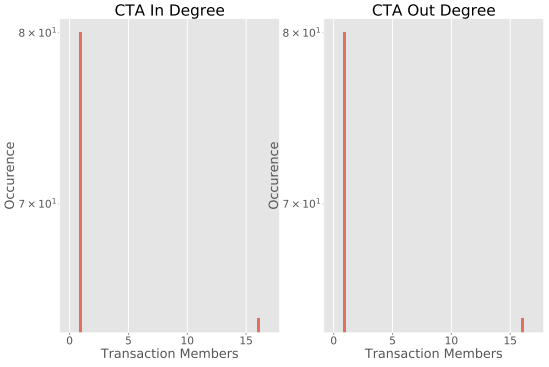
(c) Hotspot 2D



(d) Hotspot 3D



(e) Shortest Path



(f) NBody

Figure 6.5: CTA Degree Histogram

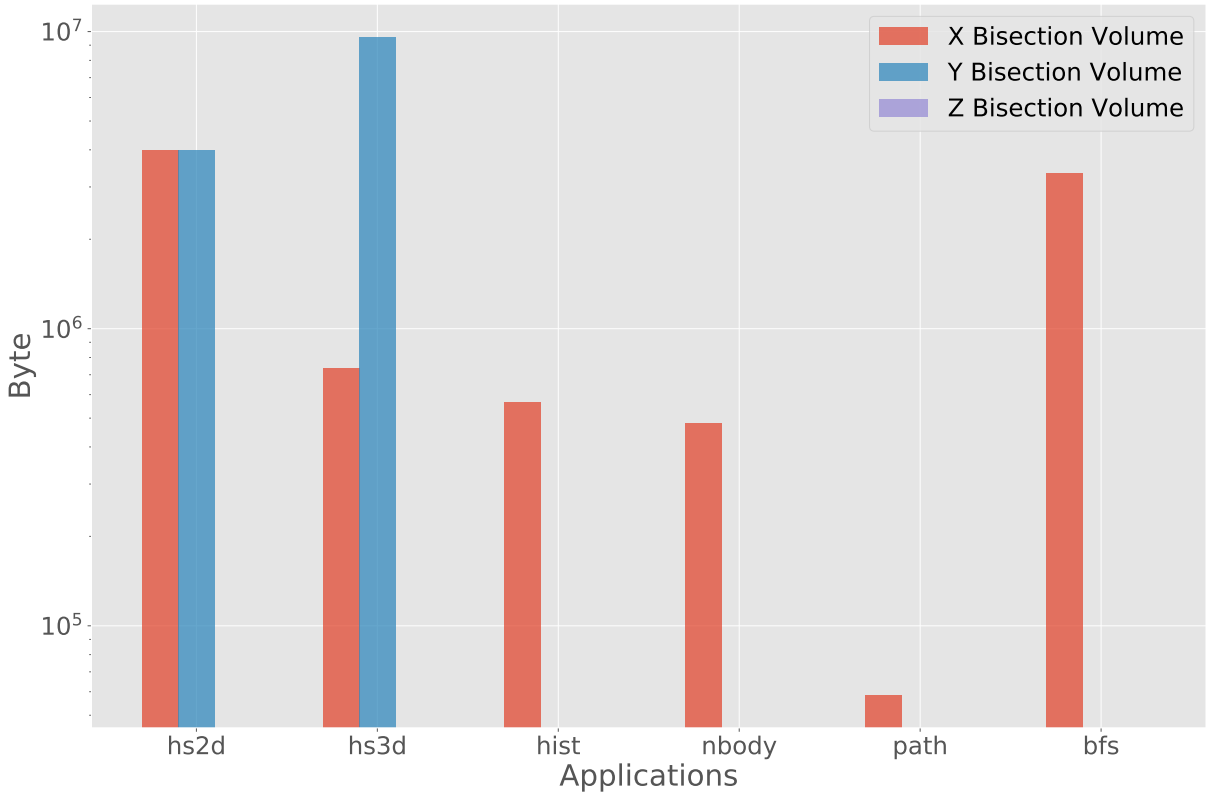


Figure 6.6: Bisection Volume

6.3 Tier II Analysis

6.3.1 Kernel communication Evolution

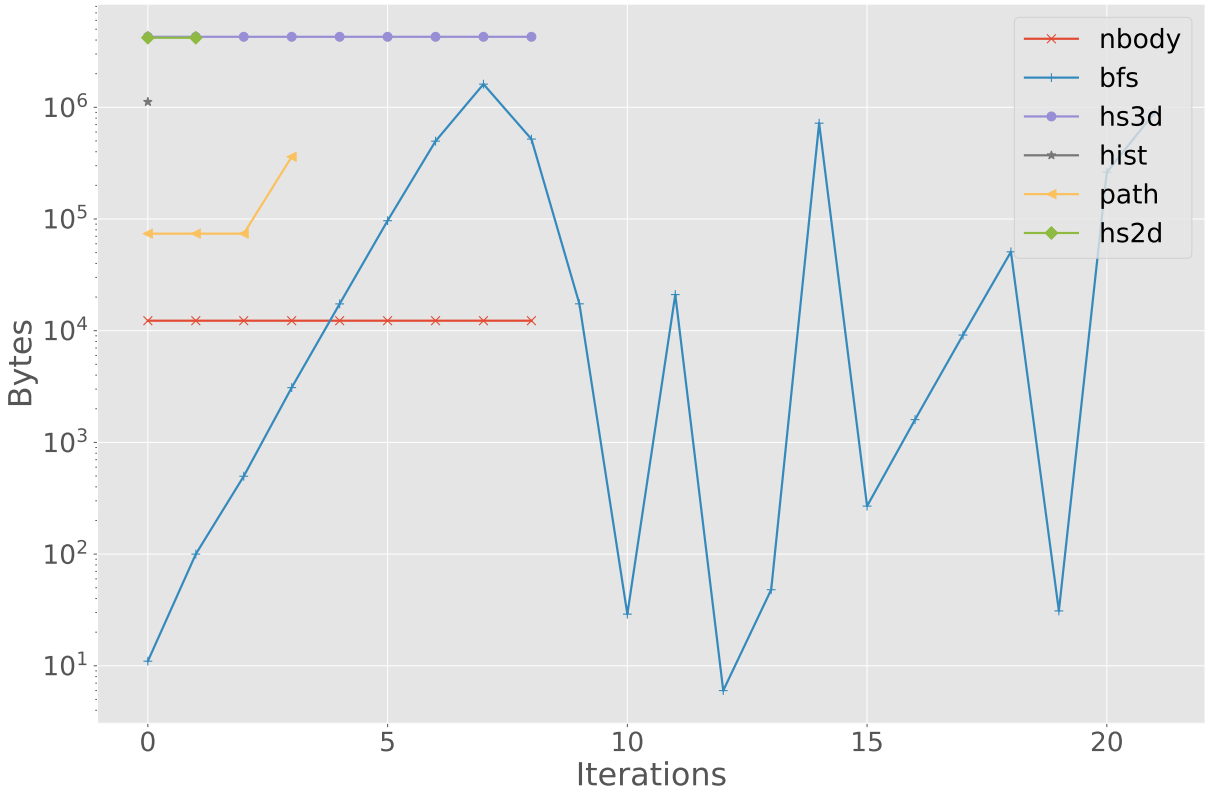


Figure 6.7: Volume of communication writes of each iteration

6.4 Tier III Analysis

6.4.1 Philandering

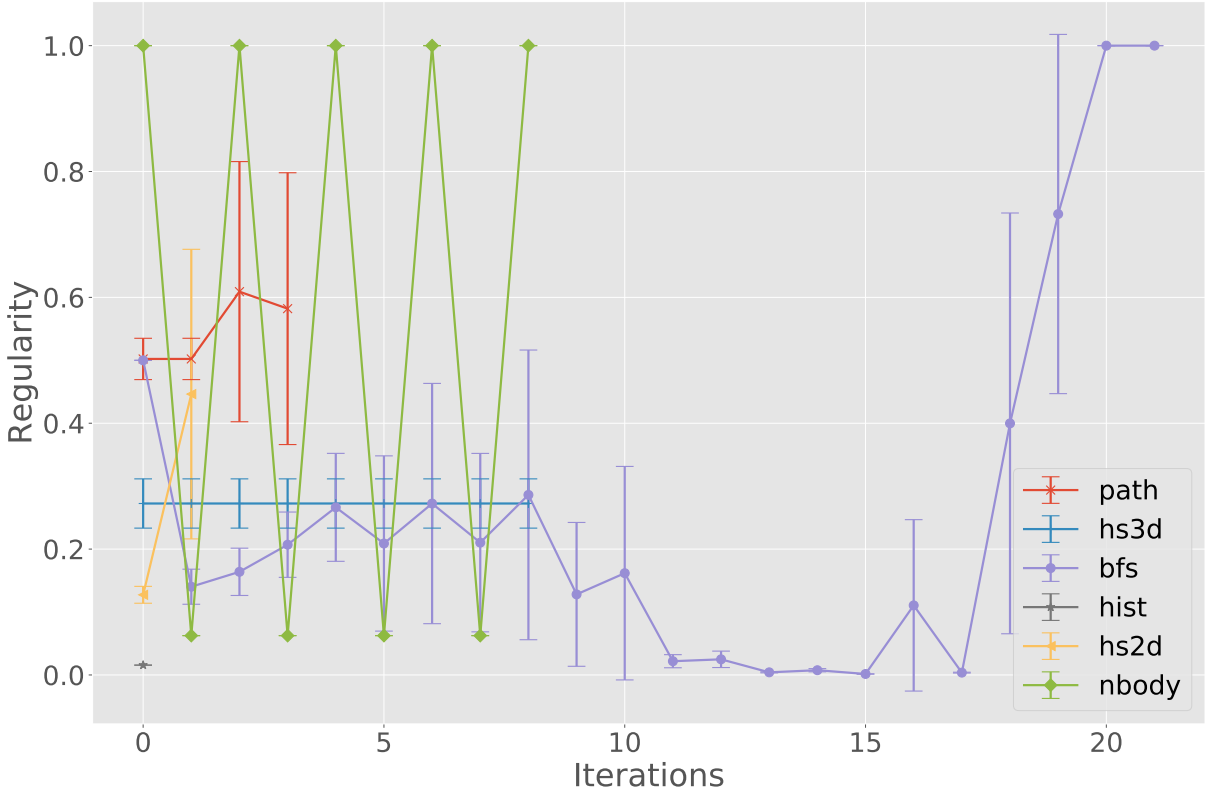


Figure 6.8: Regularity of CTA communication, per iteration partners

Chapter 7

Conclusion

Bibliography