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:	September 30, 2017

Contents

1 Motivation **1**
1.1 Goal . . . 1
1.2 Outline . . . 2

2 Background **3**

3 Related Work **4**
3.1 GPU Application Characterization . . . 4
3.2 GPU Code Instrumentation . . . 4
3.3 MPI Communication Traces . . . 4

4 Methodology **5**
4.1 GPU Shared Memory Communication . . . 5
4.2 Analysis Space Exploration . . . 6
4.2.1 Premises and Simplifications . . . 7
4.2.2 Collective Communication . . . 7

5 Dynamic Instrumentation Using Code Analysis And Transformation For Detailed GPU Communication Analysis **8**
5.1 Trace Setup . . . 8
5.1.1 Build Process . . . 9
5.1.2 Trace Format . . . 10
5.2 Source to Source Compilation in Clang . . . 10
5.3 Code Transformation in LLVM . . . 11
5.3.1 Address Space Analysis . . . 12
5.3.2 Instrumentation . . . 16
5.4 Tracing Process . . . 17

5.4.1 On-Device Producer 19
5.4.2 Host Consumer 20

6 Evaluation and Analysis of GPU Communication 21
6.1 Evaluated Applications 21
6.2 Analysis Parameters 21
6.2.1 Communication Classes 21
6.2.2 Communication Metrics 22
6.3 Analysis of Communication 22
6.4 HW Architecture Impact 22
6.5 Conclusions for MultiGPU systems 22

7 Conclusion 23

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- ...Process simulators to slow for real applications and only support outdated Architectures (GPUSim only supports up to Fermi)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 NVivida hardware. A loss in performance resulting from the traces will be tolerated.

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

1.2 Outline

Chapter 2 presents technological background information. It will focus on 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, communication patterns and GPU performance analysis.

Chapter 4 presents how the tracing is realized including the compile stack, AST manipulation and deeper code analysis in LLVM's intermidiate representation (IR).

Chapter 5 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 6 concludes and summarizes the results of this Thesis and discusses possible future directions this project could take.

Chapter 2

Background

- ? General GPU Introduction?? (not really a fan of that...)
- ? Clang AST (What happens in clang is trivial...)
- SSA/LLVM
- Compiler Techniques...
- ... Pointer analysis
- ... Versioning

Name	Scope	Access	Bandwidth	Latency	Capacity	Location
Shared Memory	CTA	RAM	high	low	low (48k)	Streaming Multiproces
Global Memory	GPU	RAM	high	high	medium	device
Texture/Constant	GPU	ROM	high	high	medium	device
Host-Mapped Memory	GPU+CPU	RAM	low	high	high	host

Table 2.1: GPU Memory Types, move to Background chapter

Chapter 3

Related Work

3.1 GPU Application Characterization

- Usually see Application not as a set of different kernels and ignore kernel "interactions"...
- ... or go down to instruction/warp
- ... or look at architectural impacts on applications
- A Characterization and Analysis of PTX Kernels Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili: Broad overview on many applications. Simulated PTX execution, Memory Intesity, branch divergence etc.
- Most other Papers analyse on similar levels, focussing on different Application or Architecture Aspects

3.2 GPU Code Instrumentation

- Works on PTX Level
-

3.3 MPI Communication Traces

- Not the same because Message Passing != Shared Memory
- Communication Patterns: Rolf Riesen: Paper on Communication Analysis and Patterns, not focussing on special Applications or Architectures. Provides useful metrics to discuss communication

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. With 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.

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.

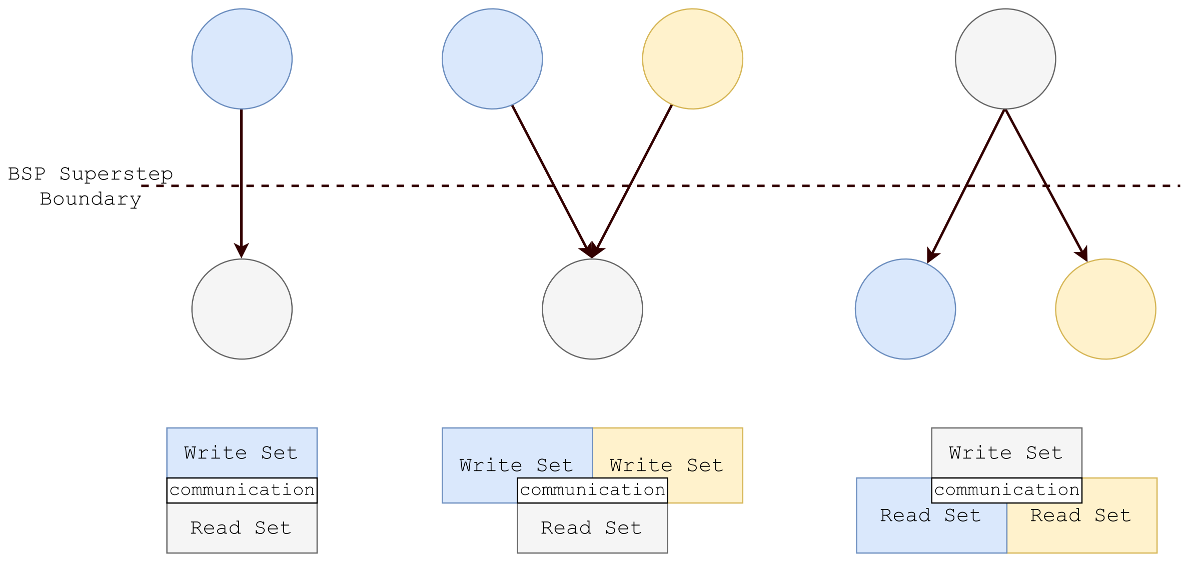

Figure 4.1: Communication across a BSP superstep boundary, and how this corresponds to read and write sets of sinks and sources. Edges are logical representations of communication, which is actually the overlap of write and read addresses in different BSP supersteps. Different colors represent different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

To exemplify the definition above, figure 4.1 shows how communication between partners 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. 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 is multi-dimensional, created by the hierarchical execution model and BSP design. The dimensions consist of:

- Threads
- CTA
- Supersteps
- Different Kernels

At first it seems, these points may be a stepping of granularity, rather than different dimensions. The following four vectors describe interactions of dimensions during different analyses.

$$a = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \qquad b = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad c = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \qquad d = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{4.1}$$

Vector (a) describes an anlysis that only takes kernel interactions into consideration, for example the communicated volumes across as BSP superstep boundary. Vetor (b) describes how different kernels interact

4.2.1 Premises and Simplifications

This work uses CTAs as the fundamental entities of communication source and sink. A kernel execution is a BSP superstep and is serialized with follow-up kernel iterations, or supersteps. Each stream is treated as an individual BSP entity during the execution. Stream interactions can be recreated in the follow-up analysis.

4.2.2 Collective Communication

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 node can give indications on tendencies towards a certain kind of collective. For the sake of the analysis, each "collective" communication can be viewed as a set of point-to-point communications, happening at the same time.

- BSP-Boundary oriented. What happens 'across' boundaries on the GPU. Ignore individual kernels for now
- ignoring programs not "well-written" (CTA interaction inside the kernel)
- How is communication realised in shared memory: bobbels to read/write sets
- On collectives: no hard collectives, rather tendencies toward collective behaviour

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.

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

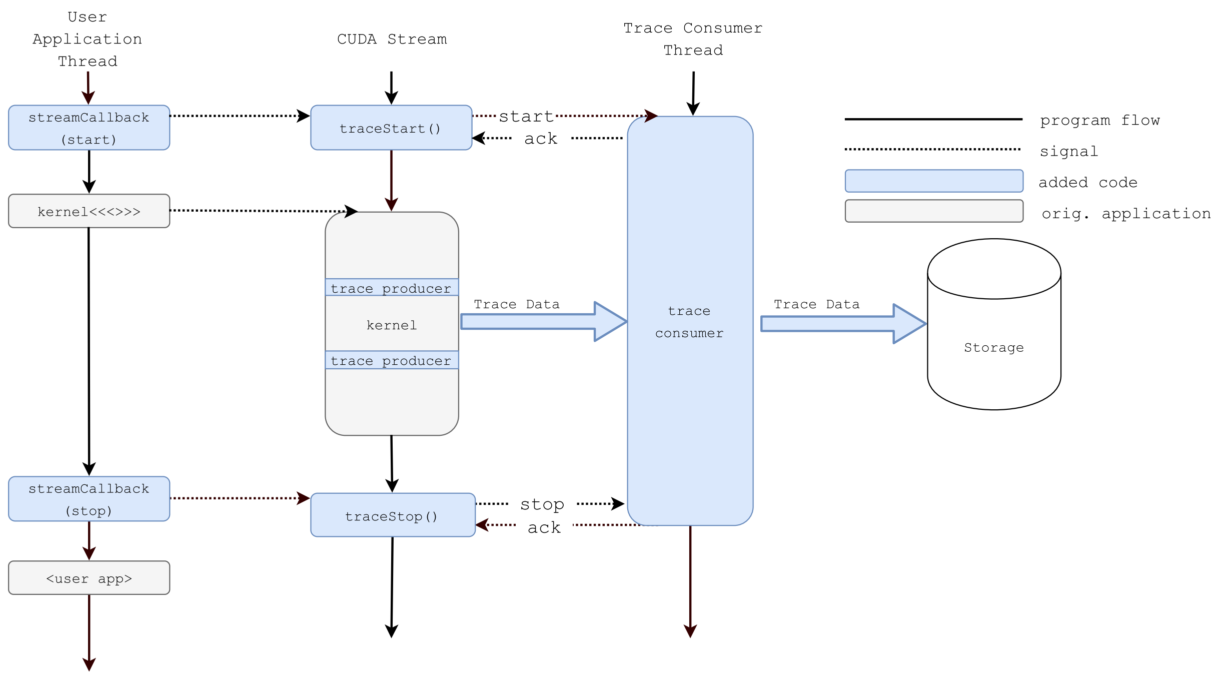

Figure 5.1: Structure of an instrumented application. Segments added during instrumentation are highlighted in blue.

```
    |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

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 operations 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.

```
    <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

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.

```
%arrayinx1 = getelementptr inbounds (float, float* addrspacecast (float
   addrspace(3)* @_ZZ5saxpyfPfS_iP1S0_E2_x to float*), i64 0, i64 2)
store float 1.000000e+00, float* %arrayinx1

```

Figure 5.4: Example for address space lowering in LLVM IR

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. 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, because only memory operations that are part of def-use chains rooting in these base pointers, can be a candidate for instrumentation.

The used algorithm performs a 'must'-analysis on the sparse problem space that is represented by the def-use chains created by the SSA form of LLVM IR. Because the the algorithm follows the order of instructions through the program, it can be described as 'flow sensitive'. Function calls are only analysed if they happen in a 'global-memory context' which in our case means that one of arguments used in the call is a pointer to global memory. 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.

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

```
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 $type(user) \in MOp$ then
            add user to list for tracing;
        else if $type(user) \in POp$ then
            add new parent to workstack;
        else if $type(user) \in NOp$ then
            continue with next user;
    end
end
```

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

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 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.

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

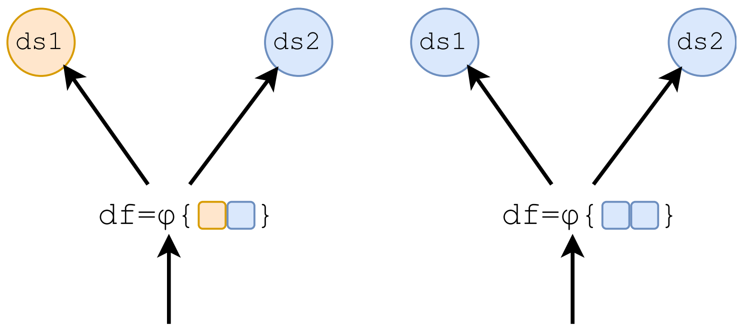

Figure 5.5: Example for static analysis limitations. Different colors represent different address spaces. The left graph represents the generic problem case and right graph a special case that is resolvable by the analysis.

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 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

```
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

```
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

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) )
   ```

   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
   ```

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

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 5.3.2.
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.

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

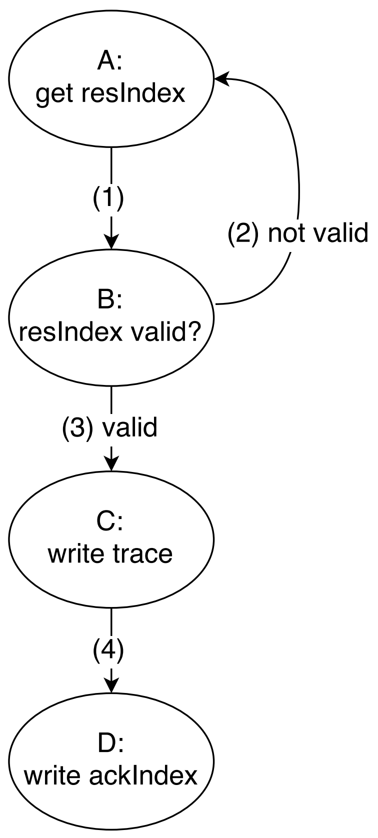

Figure 5.7: Producer CFG

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

```
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

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 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

```
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

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 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

Define what communication means in our context. (Write->Read between different CTAs etc...)
—» Explain hierarchy of analysis. Highest level watches as application as a whole.
going deeper into varius dimensions:
Different Kernels
BSP Boundaries
CTAs

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)

6.2 Analysis Parameters

6.2.1 Communication Classes

- P2P: One Writer, one Reader

- Scatter: One Writer, Multiple Readers
- Gather: Multiple Writes, one Reader
- Synchronization: Atomic access to one Address by multiple CTAs

6.2.2 Communication Metrics

- Write-to-Read ratio
- Transfer Size Histogram: Data Transferred in one Communication between two CTAs
- Transfer Density: Number of Communications(one Write->Read relation) between CTAs in whole application
- Volume Density: Number of bytes transferred in whole application
- Sparcity: Stride in data Communicated via Scatter/Gather

6.3 Analysis of Communication

6.4 HW Architecture Impact

6.5 Conclusions for MultiGPU systems

Chapter 7

Conclusion

Bibliography