MCL: A library for supporting multi-GPGPU programming

Finn-Haakon Ellingsrud Tuft

Thesis submitted for the degree of Master in Informatics: Programming and Networks
60 credits

IFI
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2017
MCL: A library for supporting multi-GPGPU programming

Finn-Haakon Ellingsrud Tuft
Acknowledgments

I would first like to thank my adviser Xing Cai for providing me with an interesting topic for my thesis. I would also like to thank the other member and master student of the HPC department at Simula. I had the pleasure to work as a TA for 2 semester in inf3151, so a great thanks to the staff and the other TAs for a great time and for making a great course. Last but not the least I must thank my wonderful girlfriend Caroline.
Abstract

In the current HPC landscape multi-GPGPU computation has established itself as a highly performant hardware choice for many workloads. A challenge is to ease the programming burden. In this thesis we present a library using a novel 3 layered architecture for supporting communication between multi-GPGPUs. This library allows a user, with little effort to communicate between GPUs and CPU using communication strategies giving state of the art performance. Further on we evaluate the performance of the library using simulated workloads and a real scientific code. This thesis should be of interest for readers concerned with easing the programming of multi-GPGPU.
A  Panfilov implementation  79
B  mcl.h  83
C  Test codes for bandwidth measurement  85
List of Figures

3.1 The layered architecture of MCL ........................................ 19
5.1 Device to Host transfer on Lizhi ....................................... 58
5.2 Device to Host transfer on Al-Khwarizmi ......................... 59
5.3 Host to Device transfer on Lizhi ...................................... 60
5.4 Host to Device transfer on Al-Khwarizmi ......................... 60
5.5 Host to Host transfers on Lizhi ....................................... 61
5.6 Host to Host transfer on Al-Khwarizmi ............................. 61
5.7 Host to Host transfer on Al-Khwarizmi, different topology level .................................................. 62
5.8 Host to Host Host to Device transfer,(0-1,0-H) , on Lizhi 63
5.9 Host to Host Host to Device transfer,(0-1,1-H) , on Lizhi 63
5.10 Host to Host and Host to Device Host to Device(0-1,1-H, H,H-0) transfer on Lizhi .......................... 64
5.11 Pairwise and Left-Right Al-Khwarizmi ......................... 64
5.12 Broadcast on Al-Khwarizmi ......................................... 65
List of Tables

2.1 Max memory copy of NCCL collectives, on 4 GeForce GTX Titan X GPUs [18] . . . . . . . . . . . . . . . . . . . 13
2.2 Left-Right approach performance measurement [23]. . 15

5.1 Hardware specifications for Lizhi . . . . . . . . . . . . . 57
5.2 Hardware specifications for Al-Khwarizmi . . . . . . . . 57
5.3 Values Device to Host . . . . . . . . . . . . . . . . . . . . 59
5.4 Values for Host to Device . . . . . . . . . . . . . . . . . . . 60
5.5 Value for Host to Host . . . . . . . . . . . . . . . . . . . . 62
5.6 Values Halo-Exchange for Left-Right and Pairwise approach, and for Broadcast . . . . . . . . . . . . . . . . . . . 65
5.7 Performance results for 2-GPU Panfilov code, hand-coded and MCL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

A.1 Panfilov performance comparison . . . . . . . . . . . . . 82
# List of Algorithms

1. Make a pattern ........................................... 21
2. Add path to pattern ..................................... 21
3. Remove path from pattern .............................. 21
4. Execute a communication pattern .................... 22
5. All communication generator ........................... 22
6. Round communication generator ....................... 23
7. Left-Right communication generator ................. 23
8. Pairwise communication generator ..................... 24
9. Broadcast communication generator ................... 25
10. Communication engine .................................... 26
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>NCCL example code for AllReduce collective</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>NVSHMEM example code of use of float_get, library is called on device</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>A simple MCL example</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Source code for all communication generator</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>The source code of the all communication generator</td>
<td>40</td>
</tr>
<tr>
<td>A.1</td>
<td>Compute part of the naive cpu code</td>
<td>79</td>
</tr>
<tr>
<td>A.2</td>
<td>Boarder exchange kernel for Panfilov code</td>
<td>80</td>
</tr>
<tr>
<td>A.3</td>
<td>Panfilov singel GPU compute kernel</td>
<td>80</td>
</tr>
<tr>
<td>B.1</td>
<td>The mcl.h header file</td>
<td>83</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Summary

System with one or more GPUs are an attractive option for users wishing for high performance on many workloads, as most of this GPUs are external accelerators. Communication between the GPU and CPU, or GPU to GPU, can be a hindrance for the desired performance. And while APIs as CUDA runtime do make it relatively easy to do the communication the library still has some technical nuances that can be hard to master. In this thesis we will try to show that a library laying atop of the CUDA runtime can help the user, both by making it easier, but also automate the use of complex strategies for speeding up communication.

1.2 Problem Statement

This master project aims to develop a simple-to-use library that helps the programmers to handle data communications that will arise from using a typical GPU-enhanced supercomputer. Through using various techniques for improving communication performance, the newly developed library should contribute to improving the achievable performance on such machines.

This thesis want to:

- Test out various techniques for improving communication performance in the context of GPU computing
- Develop a simple-to-use library for data communication that support communication pattern that typically arises in scientific code
- Verify the correctness and efficiency of the developed library.
1.3 Document organization

Background In the background we will summarize theory available for the thesis and previous work in the area.

Design and Architecture In this section we will specify the design goals using the problem statement. We will then discuss the design. And use this to derive the architecture for the library.

Implementation In this section we will present the implementation of the design we made.

Discussion and Results Here we will present and discuss the performance results of our implementation, in relation to the design goals and previous work. We will also quantitatively and qualitatively evaluate the user interface in relation to the design goals and previous work. Lastly we will present the result of using the library on a real code.

Conclusion and Further Work In this section we will summarize our discussion. Then we will look at ideas for further work. Finally we will conclude in relation to the problem statement.
Chapter 2

Background

2.1 Introduction

In this section we will very briefly look at super computer history and the developments that led to GPGPU programming coming about.

History

The first super computer is bye many considered to be CDC 6600 designed by Seymour Cray [1], it delivers 3 MFLOPs when it first become operational in 1965. Cray later left the company that build the CDC 6600 and founded his own company Cray Research. Which build a line of computers, starting with the Cray 1, first introduced in 1976 that is most associated white vector processors. The performance of the Cray 1 was 160 MFLOPs.

For almost 20 years the empathize was on building faster chips. But starting in the 90’s new systems came out that traded faster chips for more chips. The PC revolution also created opportunities. As commodity chips like the chips of the x86 architecture, but also others, gave designer system that had a higher performance for the same or lower price. A example would be the Paragon(TM) XP/S 150 MP, with the peak performance of 200 GFLOPs. The increase of processor count has led to today’s top super computer systems like the Sunway TaihuLight [Top500.org] with 10,649,600 cores, and a peak theoretical performance of 125,436 TFLOPs.

Moore’s law

Since the early integrated circuits it has been noted that number of transistors in a integrated circuit roughly doubles every 18 months; this is know as Moore’s law. This drove the development of single core chips from the 70 to the early 2000. Designers used the exponentially increase in transistor count, to push up the clock frequency of their chips. A programmer could therefore assume that
his program would have twice the amount of computer power available in roughly two years without changing a bit in the code. This started to falter in the early 2000 because the increase in clock frequency led to ever larger heat loads. Chips designers realized that to use the increasing transistor count, they could no longer rely on increasing clock frequency. Chips started coming out, that integrated several cores on one chip, by keeping the clock frequency the same or lowering it. But putting multiple cores on a chip performance could still keep raising [2].

**Von Neumann bottleneck**

One of the early design decisions of the first computer pioneers was the adoption of the von Neumann architecture. This treat data and programs as the same, and separate the memory and the CPU. As the speed of the CPU increased, the speed of the memory system did not keep pace. This led to what is known as the von Neumann bottleneck, a fast CPU idles waiting for data to be fetched. The solution adopted by CPU chips designers was to add cache to there designs. This has led to more and more chip area to be taken up by on chip cache.

**Start of GPGPU computing**

In the early 90’s graphics heavy games, like Doom, and Castle Wolfenstein become wildly popular among owners of personal computers, to provide higher definition graphics than the on board CPU could provide, several company’s started providing dedicated graphic accelerators to the home computer market, this become know as GPU (Graphics processing units). While the early cards had a hard-coded pipeline, later cards evolved into a programmable design. It was realized that these chips could be used for general purpose computation, this led to GPGPU. The processing of graphics is highly concurrent, this led the designers of GPU to chose a throughput oriented architecture with a massive amount of concurrent cores, that is totals throughput is maximized often sacrificing single task performance [3]. CPUs has started using more and more of the chip area for cache, because of the von Neumann bottleneck. GPGPUs witch do not really on cache for countering the von Neumam bottleneck has with there radical different design, giving a higher FLOP count. Become a attractive option for super computer designers, while GPGPUs only work on problems with large degree of concurrency. Many HPC workloads fit this description, making room for GPGPUs as accelerator in modern Super Computers. As GPGPUs has been adopted, the need for even more computing power has lead to the use of multiple GPGPUs on one node. One such system is the Cray
CS-Storm, of Swiss meteorology service MeteoSwiss, witch delivers up 360 TFLOPS from its 192 GPGPUs. Another interesting system is the NVIDIA DGX-1, witch is one node system with up to 8 GPGPUs, specially made for AI applications.

2.2 Theoretical background

Contestion

Resource contention occurs in many types of computer systems and happens when more then one actor wishes to utilize a resource, this includes buses like PCIe. Looking at contention in a PCIe system two concepts must also be understood, these are path and pattern. A path is simply the route a communication follows through a network, a patterns is one or more paths.

Formally contention is defined on a interconnect graph $G = (V, E)$ where $E$ is the set of edges and $V$ is the set of nodes, $V$ is $V = M \cup S$. Where $M$ is the set of computational unit and $S$ is the set of switches. A communication from node $u$ to $v$ is denoted $u \rightarrow v$. A $Path(u \rightarrow v)$ is the set of edges in the unique path from $u$ to $v$. Two communicators $u_1 \rightarrow v_1 u_2 \rightarrow v_2$ is said to have contention if there exist one or more edge $(x, y)$ such $path(u_1 \rightarrow v_1) \cap path(u_2 \rightarrow v_2 \neq \emptyset)$ a pattern is a set of communication and a contention free pattern is a pet tern where none of the communications are in contention [4].

Speedup

Speedup is measured as the ratio between sequential execution time a parallel execution time [5].

\[
Speedup = \frac{\text{Sequential}}{\text{Parallel}} \tag{2.1}
\]

Little’s law

Little’s law is a result from queuing theory.

\[
L = \lambda W \tag{2.2}
\]

$L$ stand for number of objects in the system, $\lambda$ is the average arrival rate, and $W$ is the average of time an object spends in the system. This result can be used for understanding the performance of memory systems [6]. $\lambda$ is here the bandwidth, $W$ is the latency, this forms the bandwidth-delay product. $L$ is then the total data elements in-flight. To fully utilize the memory system data in-flight must match the bandwidth-delay product. [7].
**Roof line model**

The Roof line model gives a upper bound on performance for a kernel, the key parameters in the model is arithmetic intensity, which is the number of FLOPS divided by the sum of memory reads and writes. In a two dimensional plane with FLOPSs on the y-axis and Arithmetic intensity on the x-axis the maximum FLOP for the bandwidth and maximum FLOP for the system is plotted. By knowing the arithmetic intensity of a kernel, the maximum performance and what bounds it can be predicted on a given architecture [8].

**2.3 Technical background**

**GPGPU**

As we saw in the introductory section, today's supercomputers often contains GPGPU accelerators. In this section we will look closer at GPUs, specifically GPUs from NVIDIA.

**General**

The term GPU (Graphics Processing Units) was coined by NVIDIA in the late 90. The raise of GPGPU (General Purpose Graphics Programming Units) started when programmable pipelines where added to GPUs and some dedicated researchers realist that this GPUs could be used for general purpose computation. But to do this they had to express the computations using graphics primitives as found in among others OpenGL, a graphics API. As is was realized that the GPU had some unique capabilities that could be useful outside graphics computation. But programming the cards thorough the use of graphics primitives was only something for the most dedicated programmer to do, because of this NVIDIA developed CUDA.

**CUDA**

CUDA is a C/C++ family type language (also exist for FORTRAN) and a API developed by NVIDIA for GPGPU programming. Through extensions to the C/C++ it present the CUDA programming model, called by NVIDIA Single Instruction, Multiple Thread, a form of SIMD architecture witch we will look at later. The language in it self compiles to PTX, a assembly style intermediate language, this is later, often at run-time, compiled to machine code for the target architecture. This allows NVIDIA to change the instruction set form generation to generation, while old code is still compatible with newer hardware.
A GPU consist of main memory, with L2 cache, one or more Streaming multiprocessor(SM/SMX), thread block scheduler and host interface consisting of PCIe hardware and copy engines. Blocks of threads are scheduled on the SM by the thread block scheduler, each block run until completion before the next block is scheduled in FIFO order [9]. This contrast to on a regular CPU where scheduling is done in software by the operating system, and the policies are more complex and can even be directly influenced by the running program, through voluntary yielding. The SM (called SMX in Kepler) here for the Kepler architecture, is constructed of multiple CUDA cores and one or more warp schedulers. A warp is 32 hardware thread, so a warp scheduler can schedule 32 hardware threads per cycle. The CUDA cores are of different types: Arithmetic, logic and floating point cores, double cores, memory cores, special functions cores. The number and distribution of type varies between models and different CUDA architectures. For the warp scheduler to schedule a warp, an adequate number of cores must be available, and the operands must be in a register. The last part, data dependencies is mostly handled by a scoreboard, interestingly this architecture is somewhat similar to the first super computer CDC 6600 [1]. A scoreboard is hardware construct that keep tab on with register is ready to read. By scheduling more threads then the hardware is capable of executing simultaneously, the programmer can hide the memory and instruction latency. As the GPU has hardware threads, that is the threads state is permanently stored in register dedicated to that thread. The overhead of thread switching is thereafter negligible, on the other hand this requires a very large register file (64K on Kepler). This is the dominant programming strategy for GPUs, it is also possible to use instruction level parallelism. That is that if there is no dependence between instructions in the execution stream, the next instruction for the same warp can be scheduled. If the programmer is careful he may hide the memory and instruction latency using this strategy, the number of in-flight data needed to hide latency can be calculated with Little's law, but the technical complexity of the is higher then oversubscribing the number of threads [10]. Each SM has its own memory, the memory on the SM is called shared memory and is a form of user controlled cache, a scratchpad memory. The size of this is, is in predefined fractions, controlled by the programmer, the memory not used for shared memory is used as L1 cache. Total size is 64K. There is also texture memory, this is accessible by special instructions or can be generated by the compiler if memory declared as const and restrict by the programmer (keyword restrict is a declaration by the programmer that there is no pointer aliasing). Reading through this can have the advantage that the cache line size is less then for memory read trough L1, lessening the need for alignment. The
proper exploitation of the memory system is important for performance, this include avoiding if possible large strided, that is have a low difference in index of memory access by neighboring threads in a warp. A large stride forced the memory system to fetch much more memory then needed. Ensuring alignment on cache boundary is also important, as the memory system reads/writes using fixed size memory chunks, not proper alignment leads to reading of unnecessary data [11].

The main memory of the GPU is accessible thorough the PCIe interface on the card (some cards can also use Nvlink), this consist on Tesla class cards, of two copy engines that are the hardware units copying data into and out of the cards. Having two copy engines make it possible to have concurrent copying in to and out from the card, giving full-duplex communication. In multi-GPU systems, Tesla class cards can copy from one card to another, called Pair communication. This relives the CPU from handling the communicating, and also makes possible shorter paths if the topology of the interconnect allows it.

CUDA Software Interface

The GPU hardware is exposed to the programmer through two APIs, the lower level Driver API and the Runtime API, witch is built on top of the Driver API. On modern CUDA implementations both can be used at the same time, if the programmer would like it. This APIs give access to query the device for information, give commands to the copy engines and the compute engine. They also offer some functionality on the host side, such as the possibility to register the memory of the host system with the CUDA driver and interact with the parts of the CUDA compiler. Under the Driver API is the CUDA device driver, this interact with the hardware, but is not accessible by the user[11]. Example of some typical CUDA Runtime API, functions.

cudaSetDevice Sets the GPU to use, number from 0- (GPU count -1). This really set the current context for the context for the selected device.

cudaDeviceSynchronize Wait until all task on the currently selected device is finished.

cudaMemcpy Copy’s memory from host to device, device to host or host to host.

cudaMemcpyAsync The same as cudaMemcpy but the function returns do not wait until the memory copy is finish for returning.
cudaMemcpyPeer Memory copy from peer to peer, this is asynchronous to the host.

cudaMemcpyPeerAsync Same but async to device as well

cudaMalloc CUDAS host allocator, allocates memory on current device.

cudaHostRegister Register memory on the host with the CUDA systems.

A fundamental structure for CUDA is the CUDA context, it function as a container for CUDA streams, CUDA events, all memory allocations, both device memory and memory host memory registered with the context. The CUDA context is not accessible in the Runtime API, but is accessible in the Driver API. It is also possible to have multiple context on one device. Of more interest to this thesis, each context contain its own virtual address space, in this device memory and pinned host memory is mapped.

Kernels contains the code that is executed by the GPU, kernels are marked by the keyword __global__. This are compiled by the nvcc compiler into code that can be executed on the GPU. Kernels are either called by the «< »> syntax, or they can be called by the cuLaunchKernel from the Driver API.

Some of the common CUDA run time API functions.

CUDA streams let the programmer exploit concurrency outside of the kernels. Opportunities for this is: CPU/GPU concurrency, the CPU do useful work while the GPU is used. Compute and copy engine concurrency, computation and communication can be overlapped, Tesla class GPU also has two copy engines. Making it possible to transfer data out and into the card concurrently. There is also possible to launch multiple kernel on the same device. And of course multi GPU system operate concurrently. A stream belong to one context, all in all contexts there is a default stream, the NULL stream. If not specified on creation, other stream operations can not overlap with the NULL stream. Streams are FIFO lists, operation are executed in the order they are placed in the stream. By placing operations in two different streams, the programmer states that they can be executed concurrently. In regard to stream it is also naturally to look at events, fundamentally a stream is a FIFO list where the CUDA driver places operations that are in the stream. The system also has a special operation that is called an event, the main purpose of this is signaling, but can also be used for timing. Upon execution of this event, a number is placed in a special variable, if timing is enabled, the time is also recorded. This makes it possible for a programmer to query or block on a stream waiting for this specific event to occur.[11]
MPI

MPI (Message Passing Interface) is a standard for distributed computation, widely used on modern super computers. It concerns itself primarily with communication on systems with distributed memory. It makes point-to-point and collective communications possible. Collective communications is operations like broadcast where one sender sends the same message to several recipients, or scatter where one sender sends a message to several recipients, but that each recipients get their distinct part of the message. Point-to-point operation are simple send from one node to another. The standard provides a common interface above the network protocol on the machines it uses, saving the user from caring about how the actual data transfers happen\[12\].

PCIe

PCI Express (PCIe) is a serialized point to point interconnect standard\[13\]. That offers a relatively high effective bandwidth. The connection between devices is called a interconnect, this consist of one or more lanes. A lane is a full duplex, concurrent signaling in both directions, channel, a PCIe link my consist of 1,2,4,8,16 or 32 lanes. A higher number of lanes, gives a higher bandwidth. The PCI host connects the different devices. The host consist of one root complex, that is connected to the different devices in a tree topology, the root complex being the root of the tree, the leafs are devices. The tree can also contains switches and bridges, this are internal nodes in the tree. A system can also have several root complexes. On Intel systems this can be the case on multi socket systems. As this sockets are one different NUMA domains, inter device communication is then bound by the bandwidth of the QPI. PCIe is a layered architecture with three layers; Transaction layer, Data link layer and Physical layer. The data link layers role is to ensure delivery of packets from the layer above, this are called Transaction Layer Packets (TLP). This is done thorough a ACK (acknowledge) and NAK (not-acknowledge) protocol, ACK and NAK are sent through a special data link packets. These are also used to send flow credits. The physical layer is the devise itself. The transaction layer implement a credit based protocol, this credits are calculated in relation to buffers on the switches. Head-of-Line blocking is when one PCIe transaction is blocked on a switch because on other transaction experience congestion. This reduces the total bandwidth that can be sent on the fabric\[14\].

Memory system

The memory system used on Linux is a virtual memory system. Here is a short explanation of it’s function. The 32 bit system is
used as example, this is for most parts equal in function to a system on 64 bit machines, with the one exception that 64 bits machines has a 48 bit virtual address space while 32 bit machines has 32. A process has the virtual address from 0 to 2**32-1, each process has a separate address space, this ensure the one misbehaving process do not overwrite memory belonging to other processes. In a 32 bit system on Linux each process has a page table of size 4K, one page, this has 1024 entries, a pointer to this page is stored in CR3. The entries point to page directories, if there is memory pages in this part of the address space, this are also of size 4K and have 1024 entries. This entry point to the physical pages. In a memory address, the first 10bits are used to look up in the page table, the 10 next in the page directory. And the 12 last are in the actual page. A page can be in memory, on disk, or for a page never written to the page my not exist. The two last will on attempt on access, trigger a page fault, forcing the OS to assign a physical page to the memory address. The unit that handles virtual to physical memory translation is the MMU (memory management unit). For performance there exist a buffer that directly addresses a page, saving the need to walk the page table on each memory access, this is called TLB, from a performance perspective it is important to remember that the TLB is flushed on context switch, at least partially, on older system fully [15]. If the memory system is NUMA, a physical page may be mapped to one NUMA domain while the neighbor maps to another. As it is up to the operating system to assign physical pages to virtual addresses. In Linux memory is is managed in zones, this is used for several thing, for NUMA machines zones will be created for each NUMA node. If a zone is full, memory from this zone attempted reclaimed and if possible paged to disk. The allocation of memory is determined for witch memory policy used. In Linux there are basics polices, node local and interleaved. Node local is that the memory is allocated in the NUMA domains witch the processor currently is executing, this will be done on a best effort policy. If no memory is available on the node, memory will be allocated on another node. Interleaved is the policy where memory is allocated on the nodes in a round robin fashion, this evens out the penalty of accessing NUMA nodes from the wrong CPU. There also exist a special NUMA library in Linux, this gives access to some more policies, specifically the possibilities to specific witch NUMA node one which to allocate on. This can be used by a programmer needing explicit control over memory allocation[16].

Each CUDA context has it own virtual memory system, in this is stored device memory allocation, but also pinned host memory registered with the CUDA system. Memory need to be pinned as the CUDA system can access thorough the MMU, that is without the need for CPU. If UVA (Unified Virtual Addressing) is enabled, and it is on multi peer system, the CUDA Runtime can identify the device
as the values of the address spaces of the different devices will be non-overlapping \[11\].

### 2.4 Previous work

In this section we look at two libraries, **NCCL** and **NVSHMEM** and some results from the literature that is of interest for this thesis.

#### Libraries

**NCCL** and **NVSHMEM** are both developed by NVIDIA, while **NCCL** makes a **MPI** like user interface, while **NVSHMEM** makes a **OpenSHMEM** like user interface\[?\].

#### NCCL

**NCCL**\[17\] is a new library for multi GPU computation using topology aware collective communicators. It is made by NVIDIA and is released on github\[18\]. It can be used by multi and single process applications. The collective communicators are modelled after the collectives in MPI\[12\] and should be familiar to a MPI programmer.

- **ncclCommInitRank**: Creates a communicator
- **ncclCommInitAll**: Creates a clique of communicators
- **ncclReduce**: Reduces the send buffer into the receive buffer on the root node
- **ncclAllReduce**: Reduces the send buffer into the receive buffer on all devices
- **ncclReduceScatter**: Reduces the send buffer, and scatters the result onto all receive buffers
- **ncclBcast**: Copies a value from the send buffer onto all receive buffers
- **ncclAllGather**: Gathers values from all devices

The library also contains multi processor versions of all primitives and some more house keeping functions. Implementations of this collectives are based on results detailed in\[19\] \[4\] \[20\]. The importance is to take the topology of the interconnect into consideration to avoid contention. This is done by embedding a logical ring in the interconnect topology. Stated formally as, Let \(G = (S \cup M, E)\) be the interconnect topology. \(P\) is the number of machines \(M\) There exist a logical ring pattern \(LP = n_{F(0)} \rightarrow n_{F(1)} \rightarrow \cdots \rightarrow n_{F(p−1)} \rightarrow n_{F(1)}\) if there exist a injection from \(M = n_{0}n_{1}\ldots n_{p−1}\) to \(LP\) that is contention free, this is shown
<table>
<thead>
<tr>
<th>Collective</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcast</td>
<td>10.4 GB/s</td>
</tr>
<tr>
<td>All-Reduce</td>
<td>9.6 GB/s</td>
</tr>
<tr>
<td>All-Gather</td>
<td>9.5 GB/s</td>
</tr>
<tr>
<td>Reduce-Scatter</td>
<td>8.5 GB/s</td>
</tr>
</tbody>
</table>

Table 2.1: Max memory copy of NCCL collectives, on 4 GeForce GTX Titan X GPUs [18]

In [19]. For a tree topology the logical ring is found by first number the elements in $S$ by performing depth first traversal of the tree $S = s_0, s_1, .., s_{|S|-1}$. Then a machine $X_i$ connected to a switch $s_i$ have local numbering $x_0, x_1, .., x_{|X|-1}$ the ordering of the logical ring is then $n_{0,0} \rightarrow n_{0,1} \rightarrow .. n_{0,|X|-1} \rightarrow n_{1,0} \rightarrow .. n_{|S|-1,0} \rightarrow .. n_{|S|-1,|X|-1}$. For the simplest communicator, the all-to-all broadcast the the pattern on the logical ring that is repeated $P - 1$ times is the $LP$ first machines $n_{F(i)}, i \leq P - 1$ sends to $n_{F((i+1)modP)}$ and receives from $n_{F((i-1)modP)}$ data received in previous iteration is forwarded in the next [19]. Internally the library is build around primitives copy, reduce and reduceAndCopy. It transfers data in chunks of 4-16KB between GPUs, using GPUDirect Peer-to-Peer. NCCL uses kernels directly from the devices, this are optimized to achieve maximum bandwidth on low occupancy. This leaves many CUDA threads to do computation, while the communication going on.

Listing 2.1: NCCL example code for AllReduce collective

```c
ncclComm_t comm[4];
ncclCommInitAll(comm,4 ,{0 ,1 ,2 ,3});

//Make cuda streams
//alloc memory for recv and send
//...

for(int i = 0; i<4; i++) {
    cudaSetDevice(i);
    ncclAllReduce(d_send , d_recv , nccDouble , ncclSum , comm[i] , stream[i]);
}
```

**NVSHMEM**

Is a NVIDIA implementation of parts of OpenSHMEM [21] a standard developed by Cray for partitioned address space computing. This is computation by different computational units with separate address space. A unique feature of the NVSHMEM library is that it can initiate communication from within CUDA kernels [22] and can use P2P between the cards.
Initialization

Cleanup

Allocates memory to be handles by library

De-allocates memory

Host barrier

Send data between GPUs, called from inside CUDA

Wait, called from inside CUDA.

Some of the API calls in NVSHMEM.

Listing 2.2: NVSHMEM example code of use of float_get, library is called on device

\[
i = \text{threadIdx.x}; \\
\text{for} \ (\ldots) \ \{ \\
\quad \text{if} \ (i+1 > \text{nx}) \ {v[i+1]} = \text{nvshmem_float_get}(v[1], \text{rightper}) \\
\quad \text{if} \ (i-1 < 1) \ {v[i-1]} = \text{nvshmem_float_get}(v[\text{nx}], \text{leftper}) \\
\}
\]

Theory

In this section we will summarize results from the, somewhat limited, literature of interest to this thesis.

In the guide from NVIDIA, Multi GPU Programming, written by Paulius Micikevicius [23]. Show approaches for 1D ghost exchange: Left-right approach and Pairwise approach. In a 2 and 4 GPU system, with a tree topology. For a 4 GPU ghost exchange the communicating pattern would be, sender-receiver: 0-1 1-0 1-2 2-1 2-3 3-2. Left-right approach is when on a multi-GPU system, the 1D exchange is staged in two, each GPU will send to the GPU on its right in the topology receive from the left, then wait for completion before sending to the GPU on the left and receiving from the one on the right. For a 4 GPU system, this would be, send right, receive from the left: 0-1 1-2 2-3. Send left, receive from the right: 1-0 2-1 3-2.

Pairwise approach in the first stage the even-odd pairs are exchanged, in the second the odd-even pairs.


<table>
<thead>
<tr>
<th>Approach</th>
<th>size</th>
<th>Number of GPUs</th>
<th>Aggregate throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>4 MB</td>
<td>4</td>
<td>11.0 GB/s</td>
</tr>
<tr>
<td>Left-Right</td>
<td>4 MB</td>
<td>4</td>
<td>15.0 GB/s</td>
</tr>
<tr>
<td>Left-Right</td>
<td>4 MB</td>
<td>8</td>
<td>34.0 GB/s</td>
</tr>
</tbody>
</table>

Table 2.2: Left-Right approach performance measurement [23].

For a 4 gpu system this would be, even-odd: 0-1 1-0 2-3 3-2. Odd-even 1-2 2-1.
This gives for both left-right and pairwise approach contention free communication for 4-GPU systems. But for 8 GPU system only left-right is contention free.

There are no results for Pairwise, but the author write that Left-Right approach gives better results then the Pairwise approach for all but the 2-GPU case.
The author also looks at when GPUs are on different host-complexes. As we do not have great interest of this in this thesis, we will just summarize that it gives worst results.

In the article "A PCIe Congestion-Aware Performance Model for Densely Populated Accelerator Servers" the authors using Cray CS-Storm machine with K80 GPUs find that congestion can impact performance in a significant way, cutting bandwidth with up to 50%, the author also develops a performance model and use this to improve the performance of halo-exchange collective operation [14].
In the paper "Topology-Aware GPU selection on Multi-GPU Nodes" the authors show that inter node topology impacts performance and that the communication pattern and the physical characteristics of the system must be considered [24].
Chapter 3

MCL Design and Architecture

3.1 Introduction

MCL is short for Multi CUDA Library. It is a library for making the use of multiple GPGPU fast and easier. The chapter layout is as follows, first section present the design goals for MCL. The second section discusses the design of MCL, then we go through the architecture of MCL.

3.2 Design Goals

In this section the design goals for MCL is discussed. the design goals derive from the problem statement. There were two design goals for MCL:

- Make it easier to program multi GPGPU systems.
- Make it fast, communication overhead can be a serious problem for multi GPGPU programming. So a library that do not ensure fast communication is of limited use. In this context fast means both limiting the lose of performance a library must expect to incur and making optimization strategies not easily used or known available for the programmer thorough the library.

3.3 Design Discussion

This section discuss the design choices made for MCL. As of the time of starting this work, no other libraries were known to do what this library is design to do, there were therefore nothing directly to base the design on. In the process of making this library at least two other libraries have appeared both details in previous work section. In short the design discussions that had to be made where:
• Target user
• What should be handled
• User interface
• API platform

Target user
While strictly not a design choice. Being conscious of the target user is important. In the space of GPGPU computing there are many users at different levels of the software stack and skill level. The target users for this library is C-programmers working in high performance computing.

What should be handled
In deciding this we follow the UNIX design philosophy, "Do one thing, and do it well". That is we only try to do the communication part, and make this as easy an fast as possible. Other more complicated schemes where considered like the making a features in the library for supporting the ghost region/border exchange pattern. This would have resulted in a much more complex library and a loss of generality. Following the UNIX philosophy features like this could be build into its own library, building on top of MCL. For by making simple pieces that do one thing, they can easily be combined or build atop of other for solving new problems. This is also inline with or target user category.

User interface
One of the more important design decisions is the user interface, first for ease of use, but also for speed. As outlined in the theory section, one or more communications forms a communication pattern. A pattern can for example be that one node send data to several other nodes. This becomes like the communication primitives in a standard like MPI. In MCL the basic primitive is the pattern build up by the user by adding or removing communication paths, pre-built patterns like broadcast, scatter and gather, familiar from MPI is also available. This offers a users-interface that is should be familiar for or target users, as it has similarity to communication primitives in MPI. The extra information the user gives be specifying some of the pre-build patterns, gives the library opportunity to optimize this communications.

API platform
We have chosen to use the CUDA Runtime API as the platform to build our library, the alternatives are the CUDA driver API, and
possibly also OpenCL. OpenCL was never really considered. As CUDA is dominant and using it probably would have made the library unusable for CUDA programmer. In the choice between Runtime API and driver API, it must be noted that it is not much of a choice as of in newer CUDA version they can be mixed. While the driver API offer some more control, for a library writer the explicit control CUDA context through the driver API can be attractive, there are very few other benefits. And while the Runtime API is build atop of the driver API, there is very little to no speed benefit of using it. The deciding factor for use of the Runtime API was the inclusion of the `cudaDeviceGetP2PAttribute` which give the possibility to get the PCIe topology level, this is as of now not available in the Driver API. While similar functionality is available in NVIDIA Management Library (NVML). And of course Driver API and the Runtime API can be mixed. And the fact that the driver API gives little benefits while its use is more verbose. Made that the Runtime API was chosen.

### 3.4 Architecture

In this section we will look at the overall architecture of the library, later sections will look at implementation details.

**Overview**

The overall architecture is of a layered architecture where the layers are semi-independent, with the three layers: Frontend, Communication generator and Communication engine.

- Primary function of the frontend is to interact with the user through a API, and build up the communication pattern.
• The communication generator turns the communication pattern, also using information on the PCIe topology, into a linear sequence of commands understandable by the communication engine. There exist multiple generators, and they can easily be chosen at run-time, either specified for specifies predefined pattern or set by the user. This express different strategies for carrying out the communication.

• The communication engine execute the commands in the command sequence it has be given. Example of commands are: asynchronous send from one node to another, wait for event or device synchronize.

While the frontend, in nature of being the interface to the user is not meant to be changed. But both the communication generator and the communication engine can exist in different forms, adapted to the need of the user, of the library.

**Frontend**

In this section we will look at design of the frontend. The main function of the frontend is to let the user build up one or more communication patterns, a pattern is defined in the theory section, is one or more paths. A path is a communication between two units in the system. A path is specified by the user by given the address of the destination, the address of the source and the size of the communication. By specifying a pattern the user says that the communication in the pattern can happen concurrently. There are also defined several pre-defined patterns, like broadcast, scatter, etc. This is makes it easier for the user to define, as less code has to be made, also as the library in this instances know the intent of the user, a appropriate communication generator is assigned to the pattern by the library. The frontend also contain housekeeping functions that initialize the library, reading configuration files etc. from disk. But most importantly the initialization functions can get the topology of the PCIe fabric.

**Data structure**

The primary data structure is the directed graph representing the communication path in the communication pattern. Each node in the graph is called a destination-source device pair (DSDP), is used to store the actual paths, called destination-source address pairs (DSAP). This data structure a linked list. There is two constrains on pattern data structure, a path can not start and begin in the same node, and DSAPs are unique. The operation supported, internally in the library, are add DSAP, and remove DSAP.
**Methods supported by the frontend**

This are the function the fronted at minimum most support:
Make the data structure that contains the pattern.

**Result:** Pattern
Allocate a structure for the pattern;
Initialize the structure;

**Algorithm 1:** Make a pattern

Support add a path operation on a pattern.

**Data:** Pattern,DSAP and size of the transfer
**Result:** Pattern
**if DSAP do not exist then**
| Map the DSAP to the corresponding DSDP;
| Add the DSAP to the node of the DSDP;
**else**
| Return error;
**end**

**Algorithm 2:** Add path to pattern

Support a remove a all ready existing path from a pattern operation.

**Data:** Pattern,DSAP
**Result:** Pattern
**if DSAP exists then**
| Map the DSAP to the corresponding DSDP;
| Remove the DSAP from the node of the DSDP;
**else**
| Return error;
**end**

**Algorithm 3:** Remove path from pattern

Support a operation that turn the pattern into communications on the fabric.

**Communication generator**

In this section we will look at the communication generators, the communicator generator turns the communication pattern, described above, into a linear sequence of commands, scrip bellow. The generators can use information about the topology of the interconnect. There exist a default generator, this can be changed
Data: Pattern
Result: Pattern
if The state of the pattern has been updated then
    Call the command generator of the pattern;
end
Call the communication engine;

Algorithm 4: Execute a communication pattern

by the user. When a communication pattern is allocated it is
assigned the default generator, but this can later be changed if the
user wants to.
The communication generator can be compared to the code
generator in a compiler, therefore the name. It analyses the
intermediate representation, the pattern. And turn out code, the
commands. For the different communication generators we have
found several heuristics for scheduling the communication, some
like left-right and Pairwise approach, described other simpler. We
will in the following subsections, go through a high level view of the
different algorithms for the communication generators.

All generator

The all generator, is called that because it takes all pattern from one
node and turn it into commands before moving on to the next node
in the communication pattern. This is also the default generator.

Data: Communication pattern
Result: Commands
initialization;
while There are more communication topology nodes do
    while There are more communication paths in the node do
        Add asynchronous send to list of commands;
    end
end
Add Device Synchronize to list of commands;
Add Halt to list of commands;

Algorithm 5: All communication generator

Round generator

The idea behind this is to take one path from one node, issue it and
move one to the node with a path. This to try to start as many
communication as possible as quickly as possible. This because
communication from one node, in the communication topology, is
serialized. The name round is because it is somewhat similar to
round robin. And with little effort choose between the communica-
tion strategy that gives best performance.
Data: Communication pattern
Result: Commands
initialization;
while There are more communication topology nodes with non processed paths do
    Add asynchronous send to list of commands;
    Continue to nest topology node;
end
Add Device Synchronize to list of commands;
Add Halt to list off commands;

Algorithm 6: Round communication generator

Left-Right generator

This is the algorithm for carrying out the Left-Right approach described in the background section. Of note is that this is not a general strategy, it only works for ghost exchange patterns. And do not included the CPU.
Data: Communication pattern, list of devices
Result: Commands
initialization;
while There are more nodes communicating to the left do
    while There are more communication paths in the node do
        Add asynchronous send to list of commands;
    end
end
for Devices that participated in the left phase do
    Add Device Synchronize to list of commands;
end
while There are more nodes communicating to the right do
    while There are more communication paths in the node do
        Add asynchronous send to list of commands;
    end
end
for Devices that participate in the right phase do
    Add stream synchronize to list of commands;
end
Add Halt to list off commands;

Algorithm 7: Left-Right communication generator

Pairwise generator

This is a algorithm for carrying out the pairwise approach shown in the background section. Of note is that this is not a general strategy, it only works for ghost exchange patterns. And do not include the CPU.
**Data:** Communication pattern, list of devices  
**Result:** Commands

initialization;

**while** There are more even-odd pairs of devices **do**

| **while** There are more communication paths in the node **do** |
| Add asynchronous send to list of commands; |
| **end** |
| **end** |

**for** Devices that participate in the even-odd phase **do**

| Add Device Synchronize to list of commands; |
| **end** |

**while** There are more odd-even pairs of devices **do**

| **while** There are more communication paths in the node **do** |
| Add asynchronous send to list of commands; |
| **end** |
| **end** |

**for** Devices that participate in the odd-even phase **do**

| Add stream synchronize to list of commands; |
| **end** |

Add Halt to list off commands;

**Pairwise generator**

**Algorithm 8:** Pairwise communication generator

---

**Broadcast generator**

The idea behind this heuristic it to have some of the destination of the broadcast act as senders. As the original sender can only send sequentially, it first send to one of the nodes furthest away in the topology. After this transfer, this two senders do the same until there are no more devices that has a different topology as the senders, when that happens the sender send to all devices that has the same topology difference as them self . Assumptions, the PCIe topology is a tree topology, the communication nodes are index in the order they appear as left in the tree, from the left. The topology level her is the same inside same switch or bridge.
Data: Communication pattern, list of devices, topology levels

Result: Commands

Add Device Synchronize for all devices;
topologyLevel ← the topology level all nodes relative to source node, same topology level is sorted on index;
senders ← add the sender in the broadcast pattern;

while There are more nodes to select from the broadcast pattern do

    forall the senders do
        if dst ← the node in this senders topologyLevel with the highest level but lowest index:
            Add asynchronous send to list of commands, destination dst source this senders;
        then
            There are nodes with different topology level then senders topologyLevel
                forall the Nodes with same topology level do
                    Add asynchronous send to list of commands;
                end
        else
            end
    end

Add device synchronize for all current sender;
Update list of sender with new sender and make own topologyLevel for this new senders;

end

Add Device Synchronize for all devices;
Add Halt to list off commands;

Algorithm 9: Broadcast communication generator

Communication engine

In this section we will look at the design of the communication engine.

The engine is somewhat inspired by a virtual machine, but while a virtual machine in meant to execute virtual machine code, the communication engine execute commands from the CUDA Runtime API, related to communication. The language it look at is a linear language, the terminology her is from compilers and intermediate representations (the alternative for IR would be a graph like structure). Each command is a node, called a command node, in a linked list. A command node contains all the data it needs to execute the command, a command node also identifies its type to the communication engine. In the design great care from the start has been taken to ensure that the library is fast, as this is one of the design goals. A linked list is not a ideal structure, if not care is taken in the implementation.

Type of commands (Not all are listed).
• Record a event in the specified stream
• Wait for specified event to occur
• Wait for list of events to occur
• Copy’s memory host to host using the non-asynchronous function.
• Copy’s memory from device to host using the non-asynchronous function
• Copy’s the memory from device to device.
• Asynchronous memory copy host to device
• Asynchronous memory copy device to host
• Asynchronous memory device to device
• Wait until all scheduled commands on specified device is carried out.
• Halt must be last element of the program, cleans up signaling success.

The communication engine is design as a simple VM (Virtual Machine), there is presently no operation for branching of the instruction flow.

**Data:** Commands
initialization;

**while For all commands in the list do**
  Read the command type;
  Go to appropriate subroutine;
  Execute the current command;
**end**

**Algorithm 10:** Communication engine
Chapter 4

MCL Implementation

4.1 Introduction

In this chapter the implementation of MCL is described in detail. We will first look at mcl.h, the header-file, imported by the user to use MCL. We will then look at an introductory example to show the use MCL, this make a reader have a overview of the library before diving into the technical details. The main part of the chapter follows the architecture of MCL, frontend, communication generator, communication engine.

The API presented to the user, found in mcl.h in the appendix.

- **mcl_init** Must be called before use of the rest of the library
- **mcl_finalize** Cleans up, should be called after use of MCL
- **mcl_alloc_pttr** Allocates the main data store in the library, the communication pattern. This store all data of the communication.
- **mcl_free_pttr** Frees the communication pattern
- **mcl_add_path** The user gives a source and a destination pointer, the addresses has to be registered with the CUDA runtime system as it is queried on the properties of the pointer. The size of the transfer must also be given
- **mcl_rmv_path** Removes the communication with identified with source and destination address.
- **mcl_alloc_gather** Is the gather communication primitive, can not be changed after allocation.
- **mcl_alloc_scatter** Is the scatter communication primitives, can not be changed after allocation.
- **mcl_alloc_1dim_bexch** Do ghost exchange for 1D problems
**mcl_sync_excomm** This is the function that carries out the communication described in a pattern.

**mcl_end_comm_cmpl** Changes the default communication generator.

**mcl_add_comm_cmpl** Changes the communication generator of a pattern.

Here we show a small example highlighting the use of most of the function in mcl.h.

**Listing 4.1: A simple MCL example**

```c
mcl_res_t err = mcl_init();
MCL_ERROR_PRINT(err);

mcl_ptr_t *pt;
mcl_alloc(&pt, 0);

mlc_add_path(dst1, src1, sizeof_path1, pt);
mcl_add_path(dst2, src2, sizeof_path2, pt);

mcl_sync_excomm(mcl_ptr_t *pt);
mcl_rmv_path(dst1, src1, sizeof_path1, pt);
mcl_add_path(dst3, src3, sizeof_path3, pt);
mcl_sync_excomm(pt);
mcl_add_comm_cmpl(MCL_COMM_CMPL_ROUND, pt);
mcl_sync_excomm(pt);
mcl_free_ptrt(pt);
mcl_finalize();
```

Line by line explanation of the example.

- Initialize the library
- mcl_res_t is a typedef enum, the **MCL_ERROR_PRINT** macro prints out a string to stderr, explaining the error.
- Allocated a pattern, the fundamental data store in MCL.
• Add two paths, onto the pattern. All addresses has to be register with the CUDA system, error is return if not. The library uses the Runtime API to interfere where the memory is allocated.

• By calling `mcl_sync_excomm`, the pattern is by the communication generator turned into a linear sequence of commands. This are again executed by the command engine.

• Removes the path added from the pattern. The path is identified by the source and destination address.

• Add new path.

• Executed the updated pattern.

• Changes the communication generator for this pattern given to the function, the generators are represented bye a enum in the user part, internally they map to function pointers.

• Execute the same pattern, but now with a new strategy, which strategy that given best performance change from type of system and type of pattern MCL let the user choose from a list of pre-made communication generators.

• Frees the memory of the pattern.

• Cleans up after the library.

4.2 Front-End

mcl.h

The user-interface is presented in the header-file mcl.h. This is the only file needed to be included by the user. The file gives the function prototypes of the user interface and also error handling. The implementation details of the complex type of the library, exposed to the user, is hidden for the user, through the standard typedef trick. That is the type is declared in a source file, but typedefed in the header file given to the user. The compiler only knows that there is a type of the name, not its size or internal details. This mean, only pointer of this type is allowed in the user program, aiding modularization.

Error handling

The library include basic error handling thorough a own return type `mcl_ret_t` and basic facilities for decoding error messages. Giving useful error messages is important for usability, care has there-for been taken during development to return useful errors. Errors
messages is return by all the functions of the library, they are all defined in the file mcl_err.h (Included in mcl.h), here the error type mcl_res_t is declared, the underlying type of it is enum. The macro MCL_ERROR_PRINT, declared in the header, prints a descriptive error string to stderr.
Part of function returning a const char* to the print function.

```c
switch (res) {
    case MCL_SUCCESS : return "mcl_success";
    case MCL_ERROR : return "mcl_error";
    case MCL_CUDA_ERROR : return "cuda_error";
    .
    .
}
```

**Frontend Implementation**

We will in the following sections go through the most important functions and data structures in the frontend.

**Data Structures**

**struct mcl_common**

The common struct contains the common data needed by the library, it is a global variable in the file mcl.c. Of note is that in the source code communication generator is know as a communication compiler, or short comm_cmpl in the source code. The name is not accurate, but during development it just stuck, more in line is the name communicator generator, from the code generator phase in a compiler. This is used in this thesis.

```c
struct mcl_common {
    /* config */
    char pci_topo_filename[CNF_MAXLEN_PCI_TOPO_FM];

    /* GPU attributes */
    int gpu_count;
    int cpu_count; //1
    struct pci_topo pci_topo;

    /*Standard comm compiler */
    comm_cmpl_func_t comm_cmpl;

    /* Default MCL streams */
    cudaStream_t *up_streams;
}
```
cudaStream_t *down_streams;
}

pci_topo is a struct that contains info on the PCIe topology. comm_cmpl Store a pointer to the default communication generator function. This can be changed by mcl_cng_comm_cmpl. The two pointers up_streams and down_streams are arrays that store the default up and down CUDA streams for the devices, one per GPU.

Pattern
In this part we look at the pattern, the pattern consist of three data structures. The struct pattern itself, the nodes, that is the structure representing the associative array, and the path, this store the actual communication and is stored in a linked list in it’s associated node.

We will lock at this structures in reverse order, first the path:

```c
struct comm_path {
    bool is_sentinal;
    size_t size;
    void *src;
    void *dst;
    struct comm_path *nxt, *prv;
};
```

struct comm_path describes the path from one compute node to another. void *src is the source address, from which communication originates. void *dst is where the communication is destined. size_t size is the size of the communication.

```c
struct comm_topo_node{
    size_t tsize; /* Total size of all paths */
    int count;
    comm_pttr_type_t type;
    struct comm_path top_sentinal;
    struct comm_path bottom_sentinal;
    struct comm_path *stck;

    struct comm_path *stck_view;
};
```
struct comm_topo_node is a element in the associative-array storing the communication pattern. It store a linked list of struct comm_path. The list uses top and bottom sentinels. This is implementation trick, that makes the logic of the supported operation on the data structure simpler, as they do not need to handle a empty list. The list of struct comm_path is stored in the struct comm_path *stck variable, the stck_view variable is used used in the communication generator phase and gives the generator that operate on the linked list a view into the list. It is in this structure to hide the underlying implementation of this data structure from the communication generator, promoting modularity. comm_pttr_type_t type specifies it as a GPU-GPU, CPU-GPU or GPU-CPU type of communication. This done for speeding up the communication generator phase, as it has to select the type of operation to generate.

struct comm_pattr {
    bool is_const;
    /* Total number of paths in the pattern */
    size_t path_count;
    comm_cmpl_func_t cmpl;
    size_t gpu_count;
    cudaStream_t *up_streams;
    cudaStream_t *down_streams;
    struct pci_topo *pci_topo;
    int comm_topo_node_count;
    int comm_topo_x_dim;
    int comm_topo_y_dim;
    struct comm_topo_node **comm_topo;
    struct comm_topo_node *nodes;
    bool is_same;
    struct cmnd *prg;
    struct cmnd _mem_cmnd[CNF_CMND_MAX_PRALLOC_COUNT];
    void *mem[]; //use this, or not.. ?
};

struct comm_pattr is the data structure used for the pattern, in it is everything needed in the process of making a pattern, turning into a linear sequence of commands, and executing those commands by the command engine. We will briefly look at the most important parts.

is_const set if it can not be change by mcl_add_path or mcl_rmv_path, used by the pre-made pattern cmpl is the command generator for this pattern, the underlying type is of a pointer to function with signature. mcl_res_t func(struct comm_pattr *).

The primary data structure in struct comm_pattr is the one stored in comm_topo, it is the associative array storing the pattern. It
is made as a double pointer, if there is no path for a destination-
source device pair it is NULL. If there is one or more paths it is set
to point to the corresponding member in nodes. If a path is re-
moved and it is the last path, it is set back to NULL. Both nodes
and comm_topo is allocated with the same number of elements on
call on mcl_alloc_pttr. Both are accessed by calculating the offset
into the allocated array.

is_same used to not make a new sequence of commands, if it
will be the same. It is maintained by all functions operating on
comm_topo, prg and cmpl.

prg contains the sequence of commands generated by the command
generator in cmpl. The _mem_cmnd is commands that are allo-
cated when the pattern is allocated. This facilitates data-locality,
and makes it possible to initialize the linked list when the path is
allocated. If more is needed it can be allocated, and linked into the
end of the list. But number of elements it contains is larger then
most patterns would need.

_mem this is not used, the trick here is zero size arrays. This is
supported in C99, but was a common hack before. The program-
mer allocates a memory sizeof(struct) + extra. This memory can
be addresses by a zero size array. In struct comm_pttr this can be
used to Alice memory to comm_topo, nodes and also replace _mem
for prg. The performance benefits are somewhat questionable. But
would only make one sys-call, and may help with data-locality. Also
would make it possible to specify on run-time the number of pre-
allocated elements in prg.

Communication pattern functions

mcl_alloc_pttr is the public interface for the internal function
_mcl_alloc_pttr It decoded the opt variable, that the user can use
to specify options as a bitmap. Presently only const type is sup-
ported, rest is for future use.

_mcl_alloc_pttr is the internal function that do the actual allo-
cation of the struct comm_pttr. We will not include it as it rather
long and the hole process is rather routine. The following summa-
rize the function.

• Allocate memory for the struct and the internal data-
structures and sett values.

• Initialize all the struct comm_topo_node in nodes. Setting
the linked list of nodes, by use of the sentinel. And setting the
type

• Initialize command list, by setting up a linked list of empty
commands

33
mcl_add_path is the function used by the user to add paths to the communication pattern.

```c
mcl_res_t mcl_add_path(void *dst, void *src, size_t size, mcl_pttr_t *pt) {

    struct cudaPointerAttributes dst_att;
    memset(&dst_att, 0, sizeof(dst_att));
    CUDA_ERROR(cudaPointerGetAttributes(&dst_att, dst), "cudaPointerGetAttribute");
    const int dst_dev = parse_cudaAtt(&dst_att, pt);

    struct cudaPointerAttributes src_att;
    memset(&src_att, 0, sizeof(src_att));
    CUDA_ERROR(cudaPointerGetAttributes(&src_att, src), "cudaPointerGetAttribute");
    const int src_dev = parse_cudaAtt(&src_att, pt);

    pt->is_same = false;
    /* Make a path add it to pt */
    push_path(dst, dst_dev, src, src_dev, size, pt);

    return MCL_SUCCESS;

ERROR_CUDA:
    return MCL_CUDA_ERROR;
}
```

Summarizing the key operations.

- Get the destination device by the use of `cudaPointerGetAttributes`
- Get the source device by the same method
- Use the internal method `push_path` to add path to the `struct comm_pattr`

The function `push_path` uses the destination and source address to do look-up in the `comm_topo` variable from the communication pattern struct. If this is NULL, meaning there is no path from the destination-source pair, a node is fetched from the nodes variable, and added to `comm_topo`.

```c
const int index = dst_dev + src_dev * comm_topo_x_dim;
```
struct comm_topo_node **node = &comm_topo[index];

/* If there is no path, add node from index */
if(!(*node)) {
    *node = &nodes[index];
}

A new struct comm_path is allocated, and inserted into the linked list like this, note the need for not checking for a empty list, because the use of sentinels.

    cp->nxt = (*node)->stck;
    cp->prv = ((*node)->stck)->prv;
    ((*node)->stck)->prv->nxt = cp;
    ((*node)->stck)->prv = cp;
    (*node)->count++;

mcl_rmv_path is structured much in the same way as mcl_add_path. Only major difference is the use of the internal function del_path. This function will search through the list of paths on the node representing the destination-source device pair, if the path is not found the error message MCL_UNKNOWN_PATH_SIGNATURE is returned to the user.

for( struct comm_path *curr = node->stck->prv;
    curr != &(node->bottom_sentinal); ) {
    if(curr->dst == dst && curr->src == src) {
        /* Extract it from the stck */
        curr->nxt->prv = curr->prv;
        curr->prv->nxt = curr->nxt;
        /* Free the memory */
        comm_path_free(curr);

    if(!(--(node->count))) {
        comm_topo_node_zero(node);
        comm_topo[index] = NULL;
    }
    return MCL_SUCCESS;
    }
    curr = curr->prv;
}
We will only look in detail on one `mcl_alloc_scatter`. The other pre-made patterns are implemented in similar way, and as we will show that using existing functions from the library makes implementing this patterns very easy. Of note is that this functions uses variable length macros defined in `stdarg.h`.

def mcl_alloc_scatter(
    void *src, size_t size,
    mcl_pttr_t *pt, const int dst_cnt, ...)
{
    va_list list;

    struct comm_pttr *cp;
    mcl_res_t err = _mcl_alloc_pttr(&cp,
        common.comm_cmpl, true, true);
    RETURN_ERROR(err);

    va_start(list, dst_cnt);
    for(int i = 0; i < dst_cnt; i++) {
        err = mcl_add_path(va_arg(list, void*),
            (uintptr_t)(src)+i*size, size, cp);
        RETURN_ERROR(err);
    }
    va_end(list);

    return MCL_SUCCESS;
ERROR_RETURN:
    return err;
}

Summery of `mcl_alloc_scatter`.

- Call the internal pattern allocation function. Using the all communication generator
- Call the function `mcl_add_path` destination count number of times, that is call it so all the destination pointer given by the user is read from the parameter list. As scatter spreads a large array onto many remote arrays, the source addresses given to `mcl_add_path` is recalculated for each destination by the use of standard pointer arithmetic.

`mcl_sync_excom` execute synchronously the communication pattern build up manually by the user or from pre-build pattern.

def mcl_sync_excom(
    struct comm_pttr *pt) {

mcl_res_t err = MCL_SUCCESS;

if (!(pt->is_same)) {
    /* Compile the command list */
    comm_cmpl_func_t cmpl = pt->cmpl;
    err = (*cmpl)(pt);
    RETURN_ERROR(err);
}
/* Run the commands */
err = run_cmnd_engine(pt);
RETURN_ERROR(err);

return MCL_SUCCESS;
ERROR_RETURN:
    return err;
}

Summary of the function.

- First check if the pattern has changed in between call on this function. If no change there is no need to call the communication generator. This function as a form of cache.

- If changed or first time, call the communication generator function stored in struct comm_pttr this is the err = (*cmpl)(pt). We will in the next section look at the implementation of this.

- Execute the list of communications commands, stored in struct comm_pttr *pt by the communication generator, synchronously by calling run_cmnd_engine. This calls the command engine which we will look at in the command engine section.

**Housekeeping functions**

**mcl_init** Must be called before all other functions.
- call the function prs cnf with read and parses the configuration file from disk.
- Get and set in the common struct number of GPUs using cudaGetDeviceCount the number of CPUs is also set, this is default to one.
- Parse the PCIe topology file.
- Create the default streams, two streams for each GPU, one down stream and one up stream. For communication going into and out of the GPU.
• Enable peer access between all GPUs, by using `cudaDeviceEnablePeerAccess`

`mcl_finalize` Cleans up after the library.

### 4.3 Communication generator

The communication generator layer is built up of the header file `comm_cmpl.h` for the file `cmpl.c` and at the present 5 different communication generators implemented in there own translation units.

#### Interface

We will first look at key elements in `comm_cmpl.h` and `cmpl.h`, the function of this is to modularize the communication generator layer from the rest of the code, only presenting the interface of the enum and function `cmpl_select` to the rest of the code.

```c
typedef mcl_res_t (*comm_cmpl_func_t)(struct comm_pttr *);

typedef enum {
    MCL_COMM_CMPL_ALL = 0,
    MCL_COMM_CMPL_ROUND,
    MCL_COMM_CMPL_KNAPSACK,
    MCL_COMM_CMPL_LEFTRIGHT,
    MCL_COMM_CMPL_BROADCAST,
    MCL_COMM_CMPL_PAIR,
    MCL_COMM_CMPL_COUNT /*Number of compilers in the enum*/
} mcl_comm_cmpl_t;

mcl_res_t cmpl_select(comm_cmpl_func_t *cmpl, const mcl_comm_cmpl_t ctype);
```

• The first is just a typedef of the function pointer type used by all the communication generator, of note again, at present in the source code a communication generator is know a communication compiler.

• The enum is is used by the rest of library to refer to the internal communication generator.

• Declaration of the function `cmpl_select` which make the `cmpl` variable point to the communication generator type on `ctype`. This function will we now look at in detail.

```c
/* Compiler declarations: */
mcl_res_t all_cmpl(struct comm_pttr *cp);
```
mcl_res_t round_rob_cmpl(struct comm_ptr *cp);
mcl_res_t left_right_cmpl(struct comm_ptr *cp);
mcl_res_t brdc_cmpl(struct comm_ptr *cp);
mcl_res_t pair_cmpl(struct comm_ptr *cp);

mcl_res_t cmpl_select(comm_cmpl_func_t *cmpl,
                      const mcl_comm_cmpl_t ctype) {

    switch (ctype) {
      case MCL_COMM_CMPL_ALL:
        *cmpl = all_cmpl;
        break;
      case MCL_COMM_CMPL_ROUND:
        *cmpl = round_rob_cmpl;
        break;
      case MCL_COMM_CMPL_KNAPSACK:
        *cmpl = NULL;
        break;
      case MCL_COMM_CMPL_LEFTRIGHT:
        *cmpl = left_right_cmpl;
        break;
      case MCL_COMM_CMPL_BRODCAST:
        *cmpl = brdc_cmpl;
        break;
      case MCL_COMM_CMPL_PAIR:
        *cmpl = pair_cmpl;
        break;
      default:
        assert (0); // Programming error
        return MCL_ERROR;
    }

    return MCL_SUCCESS;
}

As we described in the head file, the function just select which communication generator, from the function declarations in the top of the file, that the variable cmpl should be set to.

Communication Generator Implementation

In the following sections we will look at implementations of the communication generator. We will look detailed on the all communicator generator, the implementation of the other are more or less the same, the algorithm is different though. Three important helper functions are init_topo_node_stck_view this all work on
the variable `stck_view` in `struct comm_topo_node`, which gives a private view into the linked list of `struct comm_path` on each device-source device pair. The functions are set up to work somehow as a iterator, where `init_topo_node_stck_view` sets the variable to first element if there is one.

**All communicator generator**

Listing 4.2: Source code for all communication generator

```c
struct comm_path *next_topo_stck_view(struct comm_topo_node *node)
{
    return (uintptr_t)(node->stck_view = node->stck_view->prv) ^
        (uintptr_t)(&node->bottom_sentinal) ? (node->stck_view) : NULL;
}
```

While advance one element, treating the list as a stack, and return the element, if end of list return `NULL`. `set_to_null_topo_node_stck_view` resets `stck_view` to `NULL`.

Listing 4.3: The source code of the all communication generator

```c
mcl_res_t all_cmpl(struct comm_pttr *comm_p) {
    /* Set the first command, we need min one (for halt) */
    comm_p->prg = comm_p->mem_cmnd;
    struct cmnd *const first_cmnd = comm_p->prg;

    const int comm_topo_node_count = comm_p->comm_topo_node_count;
    const int comm_topo_x_dim = comm_p->comm_topo_x_dim;
    const int comm_topo_y_dim = comm_p->comm_topo_y_dim;
    /* One also for halt, not included here + other ops */
    int send_count = 0;
    struct comm_topo_node ** comm_topo = comm_p->comm_topo;

    struct cmnd *cur_cmnd = first_cmnd;

    for(int j = 0; j < comm_topo_y_dim; j++) {
        for(int i = 0; i < comm_topo_x_dim; i++) {
            const int index = j * comm_topo_x_dim + i;
            struct comm_topo_node *node = comm_topo[index];
            if(!node)
                continue;
            init_topo_node_stck_view(node);
            for(struct comm_path *cp = NULL;
               (cp = next_topo_stck_view(node)); ) { 
                assert(!(cp->is Sentinel));
            }
            assert(!cp->is_sentinal);
        }
    }
    return 0;
}
```

40
Should be compiled to \texttt{jmptbl}, so not much loss from having it inside for loop, but ideally should have it outside, but more cleanly this way.

```c
/*
switch (node->type) {
    case COMM_PTTR_GPU_GPU:
        cur_cmnd->type = CMND_ENG_CMND_AP2P_S;
        cur_cmnd->ap2p_s = (struct cmnd_eng_cmnd_ap2p_s) {
            .dst_device = i, .src_device = j
            .sizeof_comms = cp->size
            .src = cp->src, .dst = cp->dst,
            .stream = comm_p->down_streams[j]
        };

        cur_cmnd = cur_cmnd->nxt;
        break;
    case COMM_PTTR_CPU_GPU:
        cur_cmnd->type = CMND_ENG_CMND_AMEMCPY_HD_S;
        cur_cmnd->amemcpy_hd_s =
            (struct cmnd_eng_cmnd_amemcpy_hd_s

            [.device = i, .sizeof_comm = cp->size
            .src = cp->src, .dst = cp->dst,
            .stream = comm_p->up_streams[i] });

        cur_cmnd = cur_cmnd->nxt;
        break;
    case COMM_PTTR_GPU_CPU:
        cur_cmnd->type = CMND_ENG_CMND_AMEMCPY_DH_S;
        cur_cmnd->amemcpy_dh_s =
            (struct cmnd_eng_cmnd_amemcpy_dh_s

            [.device = j, .sizeof_comm = cp->size
            .src = cp->src, .dst = cp->dst,
            .stream = comm_p->down_streams[j] });

        cur_cmnd = cur_cmnd->nxt;
        break;
    default:
        assert(false);
        break;
}
```

set_to_null_topo_node_stck_view(node);
assert (node->stck_view == NULL);
}

/* Synchronize */
const size_t gpu_count = comm_p->gpu_count;

cur_cmnd->type = CMND_ENG_CMND_DEVSYNC;
cur_cmnd->devsync.count = gpu_count;
for(int i = 0; i < gpu_count; i++) {
    cur_cmnd->devsync.devices[i] = i;
}

cur_cmnd = cur_cmnd->nxt;

/* Terminate the command sequence with halt */
cur_cmnd->type = CMND_ENG_CMND_HALT;

return MCL_SUCCESS;
}

The program does the following for all elements in the communication topology, by looping over it in the x and y dimension, stored in the `comm_topo` variable.

- A empty node is NULL node, if it is not empty.

  - It will use the `next_topo_stck_view` to iterate through the linked list of destination-source addresses pairs in `struct comm_topo_node`.

  - The `struct comm_topo_node` is a GPU-GPU CPU-GPU or GPU-GPU. This a optimization, it also make programming simpler. The nodes in associative array that are not GPU-GPU are in the bottom row or the last colon, it could therefore easily be calculated what type of node it is. But explicitly stating it makes it simpler, it also modularize the software as the implementation detail is hidden. Which type of node it is, is important for type of communication command. We will only describe in detail one GPU-GPU.

  - For a GPU-GPU communication, assign it as a asynchronous P2P communication. This is done by setting the `cur_cmnd->type` to CMND_ENG_CMND_AP2P_S, a asynchronous peer-to-peer memory copy. This makes it possi-
ble for the communication engine to interpret the struct that we assign next. The variable `cur_cmnd->ap2p_s` which is part of the anonymous union of `struct cmnd`, is assigned from a compound literal.

After the loop device synchronize is added, such that when it return to the user all operation are complete. The last step is Halt, it tells the command engine that it is the end, and that it can clean up and return.

### 4.4 Communication engine

The communication engine execute the commands generated by communication generator written to the `struct cmnd` pointer in `struct comm_pttr`. We will no brifly go through the main points of the implementation of the command engine, more detailed description follows. The data structur that the engine operate on is a linked list of structs of type `struct cmnd`, this contain the data used by the engine. The data is the type of operation, this can be like asyncronus send or device syncronice, and parameters for this operations. What the engine do is just call the method associated with the type, then advance to the next element in the list.

### Data structur

This is primary data structure for the command engine.

```c
struct cmnd {
    bool allocated;
    cmnd_eng_cmnd_t type;
    union {
        struct cmnd_eng_cmnd_ap2p ap2p;
        struct cmnd_eng_cmnd_ap2p_s ap2p_s;
        struct cmnd_eng_cmnd_ap2p_revent ap2p_revent;
        struct cmnd_eng_cmnd_ap2p_revent_s ap2p_revent_s;
        struct cmnd_eng_cmnd_revent revent;
        struct cmnd_eng_cmnd_revent_s revent_s;
        struct cmnd_eng_cmnd_wevent wevent;
        struct cmnd_eng_cmnd_memcpy_hd memcpy_hd;
        struct cmnd_eng_cmnd_memcpy_dh memcpy_dh;
        struct cmnd_eng_cmnd_amemcpy_hd amemcpy_hd;
        struct cmnd_eng_cmnd_amemcpy_dh amemcpy_dh;
        struct cmnd_eng_cmnd_amemcpy_hd_s amemcpy_hd_s;
        struct cmnd_eng_cmnd_amemcpy_dh_s amemcpy_dh_s;
        struct cmnd_eng_cmnd_devsync devsnc;
        struct cmnd_eng_cmnd_halt halt;
    };
};
```
The function of `struct cmnd` is to store the data needed by the command engine. As there are many different commands, and they need to store different things, an effective way had to be found to store this. To do this the outer struct `struct cmnd` function as a container for the struct that store the data. This does it by having an anonymous union, containing one union of all the possible commands it can contain. The union function such that it set a side memory for the larges type it contain, an anonymous union mean that the element of the union can be addressed without referring to the union. The variable `type` has important function in this scheme, as it gives the type of data stored in the union. Making a user of the it, able to correctly interpret the data stored in the struct. The variable `allocated` signal if the memory underlying is pre-allocated or allocated (requiring to be freed separately), it is presently not used. The `nxt`, is to the next element in the linked list.

We will now briefly look at three of the commands in the `struct cmnd`. There is commands for all type of command contained in the `struct cmnd`.

```c
struct cmnd_eng_cmnd_amemcpy_dh_s {
    int device; //To copy from
    size_t sizeof_comm;
    void* src;
    void* dst;
    cudaStream_t stream;
};

struct cmnd_eng_cmnd_devsync {
    size_t count;
    int devices[CNF_MAX_DEVICES];
};

struct cmnd_eng_cmnd_halt {
    int nothing_her;
};
```

The first is `struct cmnd_eng_cmnd_amemcpy_dh_s` the name means, asynchronous memory copy device to host single copy. The last part means that there is only one copy from the command, non single commands can contain multiple action that shall be carried out, this is meant as a optimization. The elements are `device` the device the commands originate from, `sizeof_comm` the size of the communication. `src` the source address, `dst` address and the `stream` is the CUDA stream to be used. As we see, this is all that the CUDA Runtime API command need `cudaMemcpyAsync(dst,src,sizeof_comm,cudaMemcpyHostToDevice,stream)`. 

44
**struct cmnd_eng_cmnd_devsync** will call device synchronize on all devices in **devices**. The **struct cmnd_eng_cmnd_halt** is just a signal to halt and return, and do not contain any data. This is always inserted at the end.

**Implementation of the engine**

The communication engine itself is build up around executing a list of **struct cmnd**. Architecturally it is build up around a jump table like structure, the entry into this is given from the value of the **type** value of type **cmnd_eng_cmnd_t** in **struct cmnd**. The value stored in the accessed entry is a address to a jump label inside the command engine function. This can be done at the **type**, it is of underlining type enum, that means they have a integer value. By setting the first to 0, the rest of the enum will be numbered from there, giving the correct entry in the **jmp_tbl**. The jump table in the engine.

```c
static const void *jmp_tbl[] = {
    &&SR_AP2P,
    &&SR_AP2P_REVENT,
    &&SR_AP2P_S,
    &&SR_AP2P_REVENT_S,
    &&SR_REVENT,
    &&SR_REVENT_S,
    &&SR_WEVENT,
    &&SR_MEMCPY_HD,
    &&SR_MEMCPY_DH,
    &&SR_AMEMCPY_HD,
    &&SR_AMEMCPY_DH,
    &&SR_AMEMCPY_HD_S,
    &&SR_AMEMCPY_DH_S,
    &&SR_DEVSYNC,
    &&SR_HALT
};
```

The **&&LABEL** syntax is a GNU-C extension, that gives the address of a label.

The engine will use the **goto** to jump to the subroutines the elements from **jmp_tbl** points to, using the goto statement. The subroutine used for is **LOOP_ON_JMPT**, this will also be the first code after the initialization phase entered in the command engine.

```c
LOOP_ON_JMPT:
    prg = prg->nxt;
    goto *jmp_tbl[prg->type];
```
The **LOOP_ON_JMPT** first go to the next element in the list of commands, then using the *type* variable of the current command it get the correct entry into the **jmp_tbl**, by dereferencing this pointer we get the address we jump to.

We will show three examples of subroutines, **SR_AMEMCPY_DH_S**, **SR_DEVSYNC** and **SR_HALT**.

**SR_AMEMCPY_DH_S**: /* async memcpy device to host single*/
{
    struct cmnd_eng_cmnd_amemcpy_dh_s *cmnd = &prg->amemcpy_dh_s;
    int device = cmnd->device;
    size_t sizeof_comm = cmnd->sizeof_comm;
    void *src = cmnd->src;
    void *dst = cmnd->dst;
    cudaStream_t stream = cmnd->stream;
    cuda_err = cudaSetDevice(device);
    if(cuda_err != cudaSuccess) goto ERROR_CUDA;
    cuda_err = cudaMemcpyAsync(dst, src, sizeof_comm, cudaMemcpyDeviceToHost, stream);
    if(cuda_err != cudaSuccess) goto ERROR_CUDA;
}
go to LOOP_ON_JMPT;

**SR_DEVSYNC**: 
{
    struct cmnd_eng_cmnd_devsync *cmnd = &prg->devsync;
    int count = cmnd->count;
    int *devices = cmnd->devices;
    for(int i = 0; i<count; i++) {
        cuda_err = cudaSetDevice(devices[i]);
        if(cuda_err != cudaSuccess) goto ERROR_CUDA;
        cuda_err = cudaMemcpyAsync(dst, src, sizeof_comm, cudaMemcpyDeviceToHost, stream);
        if(cuda_err != cudaSuccess) goto ERROR_CUDA;
    }
}
go to LOOP_ON_JMPT;

**SR_HALT**: /* halt and return */
{
    cap.nxt = NULL; //Need this
    return MCL_SUCCESS;

The subroutines contain all the code to execute the command. The anonymous union in `struct cmnd` is interpreted into the correct type, and the data from it extracted. As seen in the code example, the data extracted from the struct is used to execute various commands from the CUDA Runtime api. The last command is always HALT this returns from `run_cmnd_engine`. For all but SR_HALT, they jump back to LOOP_ON_JMPT.
Chapter 5

Discussion and Results

5.1 Introduction

In this section we evaluate the performance, user interface and implementation of MCL.

This chapter's layout is as follows:

• In the first section we will explain how we will evaluate MCL.
• In the second section we will look at the user interface and the ease of use.
• In the third section we will evaluate the performance of MCL using simulated workloads.
• In the fourth section MCL will be used on a real code. We will evaluate performance.
• In the last section we evaluate the implementation.

5.2 Evaluating MCL

To answer our problem statement we need to evaluate the user interface, performance and correctness. Correctness will only briefly be discussed in this section. For the user interface and performance there are separate sections. The primary purpose of this section is to discuss the evaluation methods.

Correctness

To have a program that is reasonably correct is a prerequisite for evaluating a thing like performance. If a library only transfers 10MB while it should transfer 100MB do invalidate performance measurement. In this section we will in brief explain our testing of correctness.

To test our implementation we developed a small library for testing, the library was initially only to be used for unit testing, but was
adopted it for all testing. The most important thing tested for was that the library transfers the correct data, this was tested for by transferring char arrays, and seeing that they where the same after transfers. Char arrays was used for this for ease of verifying equality. Other test carried out was that correct error values was given on error, during performance testing nvprof was used to see that the correct amount of data was transferred. Valgrind was used for testing for memory leaks and memory errors.

**User-interface**

One of the goals is to make multi-GPGPU programming simpler for the programmer, the user interface is then what we have to evaluate. This can be evaluated quantitatively or qualitatively. To make a quantitative evaluation one or more metrics has to be used, there exist a number of different. Something like cyclomatic complexity\[25\] could have been used, we have in stead chosen the simplest, lines of code. The primary reason for not using other metrics is that we do not really have a form of reliable data for evaluating ease of use. To make a reliable quantitative comparison the data should be from other users of the library to remove potential bias, as we also are targeting HPC users it should cover a large part of potential use scenario for HPC, something like in the line of 7 dwarfs of HPC. As this is not possible to achieve in the scope of a master thesis, we will only make what could be called a illustrative example to show how the library affect lines of code. 

For the qualitative part we will interpret easier as removing the details of the communication, and letting the programmer specify what should be done. A move from imperative to more declarative programming\[26\]. While easier of course is somewhat subjective, looking at the development of computer, letting computers handle more and more is a strong trend, it is also reason for making libraries.

**Performance**

Performance is evaluated in two ways, performance on simulated workloads expressed as different communication pattern over a range of different transfer sizes. This is the establish way of measuring communication performance in the literature\[14\] \[23\] we have evaluated and also gives us a possibility make a comparison to other results. The other way is by using the library on a real scientific code and comparing to hand-coded, this gives potential users a very easy way of seeing the performance impact of the library if any. Further details about this methods are discussed in their relevant sections to make the connection to the measurement easier for the reader.
User-interface

In this section we will illustrate the use of MCL in sample code, and compare this to hand-coding. Showing of the different uses of MCL. This will be in someway in the style of user guide. The purpose is to give an understating of the use of MCL from a users perspective.

#include "mcl.h"

In all programs wishing to use MCL, this must be included, the full header file can be found in the appendix. The library must also of course be linked in during compilation.

mcl_res_t err = mcl_init();

The first statement in any program wishing to use the library, must be mcl_init. The error codes if success is 0 this the same as MCL_SUCCESS, a negative value signals error. A const char * can be read out and printed to stderr with the function _mcl_error_print, or macro MCL_ERROR_PRINT giving a textual description of the error message.

In this part we will look at actual use of the functions which are the main part of mcl: mcl_alloc_pttr, mcl_free_pttr, mcl_add_path, mcl_rmv_path, mcl_cng_comm_cmpl, mcl_add_comm_cmpl and mcl_sync_excomm.

mcl_pattr_t *pt;
mcl_res_t err = mcl_alloc_pttr(pt, 0);
err = mcl_add_path(device1_test0, device0_test0,
                   sizeof_transfere, pt);
err = mcl_add_path(device0_test1, host_test1,
                   sizeof_transfare, pt);
err = mcl_sync_exomm(pt);
%err = mcl_free_pttr(pt);

Here we will go through the code line by line

• Declare a variable of type mcl_pattr_t

• Allocate a communication pattern, no options given.

• Add a communication path from GPU 0 to GPU 1 to the allocated pattern, as the program figure out witch device a memory address belongs to. The user need only give the address.

• This is the same, only that it is from Host to GPU 0.

• Her the communication defined in the pattern pt is executed. As it synchronous it do not return to the caller before all communications has been carried out.
This illustrates a simple use of MCL. The user builds up a pattern, by doing so he is declaring that all communications can be done simultaneously. If there are dependencies, the user must make two or more patterns and put the communications with dependencies in different patterns.

One of the futures of MCL is the possibility for the user to choose which strategy he wants to use for the communication. This is done by changing the communication generator, either for a pattern or for the default. When a pattern is allocated with mcl_alloc_pttr it is automatically assign the default generator. Changing the default, has the effect that later allocations is assigned the new generator. After a pattern is allocated a user can also change the generator assign to that pattern.

Continuing with from the sample code:

```c
err = mcl_add_comm_cmpl(MCL_COMM_CMPL_ROUND, pt);
err = mcl_sync_exomm(pt);
err = mcl_cng_comm_cmpl(MCL_COMM_CMPL_PAIR);
mcl_pttr_t *new_pt;
err = mcl_alloc_pttr(new_pt);
err = mcl_add_path(dst, src, new_size, new_pt);
err = mcl_cng_comm_cmpl(MCL_COMM_CMPL_PAIR);
```

The generator is changed out, to one using a another strategy, in pt. When mcl_sync_excomm is called again. The same communication is executed, but using the new strategy.

```c
err = mcl_cng_comm_cmpl(MCL_COMM_CMPL_PAIR);
err = mcl_alloc_pttr(new_pt);
err = mcl_add_path(dst, src, new_size, new_pt);
```

Changing the default generator, when the new_pt is allocated it will have the MCL_CMPL_PAIR generator as default. In the code there are 6 generator to choose from, MCL_COMM_CMPL_ALL, MCL_COMM_CMPL_ROUND, MCL_COMM_CMPL_KNAPSACK, MCL_COMM_CMPL_LEFT_RIGHT, MCL_COMM_CMPL_BROADCAST, MCL_COMM_CMPL_PAIR five of this we have discussed before. The one not look at is the KNAPSACK this is not implemented (so a bad idea to use this!) but is discussed in future work in this thesis.

If the user wishes MCL to clean up, the function MCL_finalize must be called. Calling MCL after call on MCL_finalize is not allowed.

We will now look at a sample stencil code, the code lacks the kernel, and most other thing not of interest to the comparisons. The purpose of the code is purely illustrative. The different steps of
the code will be compared to the hand-coded code. A stencil code will use the ghost exchange programming pattern for multi GP

```c
/* Allocate memory */

mcl_pttr_t *heat_exchange, *heat_scatter, *heat_gather;

/* Alloc and make pattern for border exchange */
err = mcl_alloc_1dim_bexch(heat_ghost_size, &heat_echange, 6,
heat_right_recv[0], heat_left_send[1], heat_right_send[0],
heat_left_recv[1], heat_right_recv[1], heat_left_send[2],
heat_right_send[1], heat_left_recv[3], heat_right_recv[2],
heat_left_send[2], heat_right_send[2], heat_left_recv[3]);

/* Alloc and make pattern for scatter, of initial values */
err = mcl_alloc_scatter(heat_host, host_heat_size, 4,
heat_device[0], heat_device[1], heat_device[2], heat_device[3]);

/* Alloc and make pattern for gather, of final values */
err = mcl_alloc_gather(heat_host, host_heat_size, 4,
heat_device[0], heat_device[1], heat_device[2], heat_device[3]);
```

In this part of the code we make patterns for halo exchange, the initial scatter, and the final gather.

- The first line declares the three patterns, for scatter, gather and ghost/boarder exchange.

- The second line makes the pattern for the ghost/boarder exchange, this by giving the function `mcl_alloc_1dim_bexch` the left right pairs that the values shall be transferred to.

- Third line makes the scatter pattern, the variable `heat_host` gives the variable to scatter from, and the variables after 4 which is how many variable to scatter to, are the variables to scatter to.

- last line is the gather, `heat_host` is to gather to

```c
err = mcl_sync_excomm(heat_scatter);
```

Here the actual scattering of data from the initialization is done.

Hand-coded code doing the same.

```c
for (int i = 0; i < GPU_COUNT; i++) {
  cudaMemcpyAsync((void*)heat_device[i], (void*)((char*)heat_host + i*heat_host_size),
```
```c
host_heat_size, cudaMemcpyHostToDevice, streams[i]);
```

```c
for(int i = 0; i < GPU_COUNT; i++) {
    cudaMemcpyHostToDevice(i);
    cudaMemcpyDeviceSynchronize(i);
}
```

The hand-coded part is 6 lines of actual source code, the MCL part is 2. For the hand-coded part that is excluding the making of CUDA streams, for MCL that is excluding call on `mcl_init` and declaration of the `mcl_pattr_t` variable.

The next part is the communication related to the computation itself, here as we said before we have excluded the computation. The MCL part.

```c
for(int i = 0; i < loop_count; i++) {
    /*
     * Launch kernels
     * .
     * Device synchronize
     */
    /* Do halo exchange */
    err = mcl_sync_ecomm(heat_exchange);
}
```

Inside the main loop, the halo exchange happens on each iteration of the loop.

```c
for(int j = 0; j < loop_count; j++) {
    /*
     * Launch kernels
     * .
     * Device synchronize
     */
    for(int i = 0; i < GPU_COUNT; i++) {
        if(i != (GPU_COUNT - 1)) {  //Right
            cudaMemcpyDevice(i);
        }
    }
```
cudaMemcpyPeerAsync(heat_left_recv[i+1],
i+1,heat_right_send[i],
i,test_size,streams_right[i]);

if (i != 0) { //Left
cudaSetDevice(i);
cudaMemcpyPeerAsync(heat_right_recv[i-1],
i-1,heat_left_send[i],
i,test_size,streams_left[i]);
}

for(int i = 0; i < GPU_COUNT; i++) {
cudaSetDevice(i);
cudaDeviceSynchronize();
}

The hand-coded part is 10 lines, and the MCL is 3.

The last part of the code, is the gathering of the computed values from the GPUs to the CPU. First the MCL part.

err = mcl_sync_excomm(heat_gather);

The hand-coded part.

for(int i = 0; i < GPU_COUNT; i++) {
cudaSetDevice(i);
cudaMemcpyAsync((void*)((char*)heat_host + i*heat_host_size),
0,(void*)heat_device[i],
host_heat_size,cudaMemcpyDeviceToHost,streams[i]);
}

for(int i = 0; i < GPU_COUNT; i++) {
cudaSetDevice(i);
cudaDeviceSynchronize(i);
}

The hand-coded is 6 lines, MCL is 2.

In summary it looks like MCL gives some saving of lines of code for a multi CUDA stencil code.

Using the mcl_rmv_path a path can be removed from a pattern. This makes it possible to support dynamic communication patterns.

err = mcl_alloc_pttr(pt,0);
err = mcl_add_path(dst[0],src[0],size[0],pt); //Add 10 paths
...
err = mcl_add_path(dst[9], src[9], size[9], pt):
for(int i = 0; i < 10; i++) {
    err = mcl_sync_excomm(pt);
    err = mcl_rmv_path(dst[i], src[i], size[i], pt);
}

for(int i = 0; i < 10; )
    err = mcl_add_path(dst[i], src[i], size[i], pt);
    err = mcl_sync_excomm(pt);
}

While the previous example may not be useful in practice, it shows the possibility for use. In codes where there is unpredictable communicating patterns this possibility can be useful.

Discussion

As we can see from the example, MCL has a potential to reduce the line count of program. But fundamentally the advantage of MCL over hand-coded code, is that in communication becomes more declarative. The programmer specifies what to do, this contrast the imperative programming where the programmer tells the computer step wise how to do what he want it to. Moving from imperative to declarative programming also reduces the change of errors, both correctness but also for performance. One note on the user interface is the use of the library function alloc_1dim_bexch is probably not very useful, building up a normal pattern using mlc_ptr_alloc and mcl_add_path is probably better for building pattern for ghost exchange, as the use may not be to intuitive. If we compare with other libraries, the user interface has many similarities with MPI, and is inspired from this. While MPI, solves a much more complex problem and often operate on much larger count of nodes in the system so it would not be feasible to build you own patterns in MPI. If we look at the new NCCL from NVIDIA, the lack of the ability to build you own pattern make it less flexible then MCL, it also need to be called once for each device.

Performance

One of the design goals is speed, in this part we will evaluate this. This is done on two different system, a 2-GPU Kepler system and a older 4-GPU Fermi system. Summary of there capabilities is presented bellow.

Our 2 GPU Kepler system, called Lizhi.
Hardware specifications for Lizhi

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2 x Intel Xeon E5-2650</td>
</tr>
<tr>
<td>GPU</td>
<td>2 x NVIDIA Tesla K20m</td>
</tr>
<tr>
<td>Memory</td>
<td>32GB DDR3-1600</td>
</tr>
<tr>
<td>HDD</td>
<td>2 x 240 SSD in RAID 0</td>
</tr>
</tbody>
</table>

Table 5.1: Hardware specifications for Lizhi

Our 4 GPU Fermi system, called Al-Khwarizmi. The two physical cards are fused, making it effectively 4 cards.

Hardware specifications for Al-Khwarizmi

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E5620</td>
</tr>
<tr>
<td>GPU</td>
<td>2 x NVIDIA GeForce GTX590</td>
</tr>
<tr>
<td>Memory</td>
<td>24GB DDR3-1066</td>
</tr>
<tr>
<td>HDD</td>
<td>2 x 240 SSD in RAID 0</td>
</tr>
</tbody>
</table>

Table 5.2: Hardware specifications for Al-Khwarizmi

Benchmark

We test different communication patterns, example of test codes can be found in the appendix. The size is selected to get contention effects, in [14] the size 300MB was proposed, all our test has this in their interval. The test are done as from most test from 250KB, this is multiplied 2-exponent up to the value 1024MB for Lizhi and to the value 512MB for Al-Khwarizmi (Not higher because of limited memory).

As we have explained in the design part of this thesis, the purpose of the library is to be a component in a larger solution. The measurement presented here is a part in this, as we try to characterize the abilities of this components, MCL, in the larger system, by showing the performance for over large transfer size domain and different transfer scenario. A somewhat similar test regime was also proposed in [14]. The test of the none complex part is to evaluate potential overhead in the library, the complex strategies is to evaluate potential speedup. This two evaluations are both evaluated in the context of the design goal of a fast library. All times were measured with `omp_get_wtime`, the test code was for Lizhi compiled using NVCC 6.5.16, the library was compiled with GCC 4.8.4. On Al-Khwarizmi NVCC 6.5.12, and GCC 4.6.3 was used, for NVCC no particular options, for GCC O0 was used.

Simple patterns:

- Device to Host.
• Host to Device.

• Host to Host, same topology level.

• Host to Host, different topology level. Only done on Al-Khwarizmi.

• Host to Host and Host to Device. Only done on Lizhi.

• Host to Host and Host to Device. Only done on Lizhi.

• Host to Host, Host to Device and Host to Device. Only done on Lizhi.

• Broadcast. This is sending the same message from one node to all other nodes.

• Halo-Exchange, testing naive versus Pairwise and Left-Right approach on Al-Khwarizmi. This is exchange between GPU0 and GPU1, GPU1 and GPU2, GPU2 and GPU3. As these values can be small in real code, the test start at 62.5 KB, this value is chosen from consideration in [14], where the author has helo-exchange in the range 40-250KB.

We start by looking at device to host for Lizhi and Al-Khwarizmi, in this the default ALL communication generator was used.

Stream test of one Device to Host transfer on Lizhi

![Graph showing the transfer size and total GB/s on PCIe fabric for different transfer sizes.](image)

Figure 5.1: Device to Host transfer on Lizhi
Stream test of one Device to Host transfer on Al-Khwarizmi

![Graph showing total GB/s on PCIe fabric against transfer size, MB]

Figure 5.2: Device to Host transfer on Al-Khwarizmi

Numerical values for Device to Host.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Size</th>
<th>Hand-coded</th>
<th>MCL</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lizhi</td>
<td>512GB</td>
<td>6.711983 GB/s</td>
<td>6.707273 GB/s</td>
<td>0.9992</td>
</tr>
<tr>
<td>Al-Khwarizmi</td>
<td>512GB</td>
<td>6.126603 GB/s</td>
<td>6.126528</td>
<td>0.9999</td>
</tr>
</tbody>
</table>

Table 5.3: Values Device to Host

The measurement show that there is very little overhead in the use of MCL for Device to Host.

We will now look at Host to Device transfer for Lizhi and Al-Khwarizmi.
Stream test of one Host to Device transfer on Lizhi

![Graph showing Host to Device transfer on Lizhi](image)

Figure 5.3: Host to Device transfer on Lizhi

Stream test of one Host to Device transfer on Al-Khwarizmi

![Graph showing Host to Device transfer on Al-Khwarizmi](image)

Figure 5.4: Host to Device transfer on Al-Khwarizmi

Numerical values for Host to Device.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Size</th>
<th>Hand-coded GB/s</th>
<th>MCL GB/s</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lizhi</td>
<td>512GB</td>
<td>6.185398</td>
<td>6.187009</td>
<td>1.0002</td>
</tr>
<tr>
<td>Al-Khwarizmi</td>
<td>512GB</td>
<td>5.484580</td>
<td>6.126528</td>
<td>1.1170</td>
</tr>
</tbody>
</table>

Table 5.4: Values for Host to Device

60
The value from MCL are actually better than the hand-coded.

In this section we will look at Host to Host.

Stream test of one Host to Host transfer on Lizhi

![Graph showing performance comparison between Hand Coded and MCL for Host to Host transfers on Lizhi.]

Figure 5.5: Host to Host transfers on Lizhi

Stream test of one Host to Host, same topology level, transfer on Al-Khwarizmi

![Graph showing performance comparison between Hand Coded and MCL for Host to Host transfers on Al-Khwarizmi.]

Figure 5.6: Host to Host transfer on Al-Khwarizmi
Stream test of one Host to Host, different topology level, transfer on Al-Khwarizmi

Figure 5.7: Host to Host transfer on Al-Khwarizmi, different topology level

Numerical values for Host to Host.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Size</th>
<th>Hand-coded GB/s</th>
<th>MCL  GB/s</th>
<th>Difference</th>
<th>Topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lizhi</td>
<td>512GB</td>
<td>5.305972 GB/s</td>
<td>5.304670 GB/s</td>
<td>0.9997</td>
<td>–</td>
</tr>
<tr>
<td>Al-Khwarizmi</td>
<td>512GB</td>
<td>6.656105 GB/s</td>
<td>6.659059 GB/s</td>
<td>1.0004</td>
<td>Same</td>
</tr>
<tr>
<td>Al-Khwarizmi</td>
<td>512GB</td>
<td>5.309100 GB/s</td>
<td>5.308047 GB/s</td>
<td>0.9998</td>
<td>Different</td>
</tr>
</tbody>
</table>

Table 5.5: Value for Host to Host

Again we show very little difference between hand-coded and MCL. Of note is the high bandwidth for Al-Khwarizmi for same topology level, this is probably because the cards are connected through a bridge and the transfer do not involve the root complex [13] [23].

The rest of the test are more complex, there are more then one transfer going on at the same time, we will do test mostly on Lizhi. For test where contention can be an issue, and where the more complex strategies, like Left-Right, can be an advantage we will do that on Al-Khwarizmi.
Stream test of Host to Host and Host to Device (0-1,0-H) transfer on Lizhi

Figure 5.8: Host to Host Host to Device transfer,(0-1,0-H) , on Lizhi

Stream test of Host to Host and Host to Device (0-1,1-H) transfer on Lizhi

Figure 5.9: Host to Host Host to Device transfer,(0-1,1-H) , on Lizhi
Stream test of Host to Host and Host to Device Host to Device(0-1,1-H,H-0) transfer on Lizhi

**Complex strategies tested on Al-Khwarizmi**

In this section we look at complex strategies linked non-naive broadcast, pairwise and left-right ghost-exchange. This were only done on Al-Khwarizmi as we need 4 a GPU system to see the effect of this.

Stream of test of 0-1 1-0 1-2 2-1 2-3 3-2, Al-Khwarizmi

Figure 5.10: Host to Host and Host to Device Host to Device(0-1,1-H,H-0) transfer on Lizhi

Figure 5.11: Pairwise and Left-Right Al-Khwarizmi
Stream of test of 0-1 1-0 1-2 2-1 2-3 3-2. Al-Khwarizmi

![Graph showing Total GB/s on PCIe fabric vs Transfer size, MB]

**Figure 5.12:** Broadcast on Al-Khwarizmi

**Benchmark for Al-Khwarizmi. Test size 500.000000 MB.**

<table>
<thead>
<tr>
<th>Type</th>
<th>Benchmark</th>
<th>MCL</th>
<th>Difference</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcast</td>
<td>5.688743 GB/s</td>
<td>8.858456 GB/s</td>
<td>1.55</td>
<td>–</td>
</tr>
<tr>
<td>Halo-Exchange</td>
<td>12.361629 GB/s</td>
<td>17.083918 GB/s</td>
<td>1.38</td>
<td>Pairwise</td>
</tr>
<tr>
<td>Halo-Exchange</td>
<td>12.361629 GB/s</td>
<td>11.621079 GB/s</td>
<td>0.94</td>
<td>Left-Right</td>
</tr>
</tbody>
</table>

Table 5.6: Values Halo-Exchange for Left-Right and Pairwise approach, and for Broadcast

Looking at the result for Halo-exchange we see that for small values all strategies give the same result. The probable reason for this is that the PCIe system do not experience contention. Either because the communications are so small that one is finished before there is a competition for resources or that the built in buffers handle small messages without performance penalty. As we do not have access to read the actual packets of the PCIe system we can only speculate. But for higher values, the Pairwise approach show itself to be the superior strategy for or system. We had expected this to be the Left-Right approach. We suspect this is because of the use of bridges to connect the two cards. Comparing with the results from the literature [23], 15GB/s was reported for message size of 4MB, 4 GPU system, using Left-Right approach, we see 15.869142 GB/s for Pairwise [2.2], the author do not report values for Pairwise but say it is slightly slower then Left-Right. This put our results in the same range of performance.

For broadcast our approach gets lower performance then the approach in NCCL [2.1], this is expected as the NCCL approach is...
shown to be near optimal [19].

Summary and Discussion

From out result we can see that there is little or no overhead with the use of MCL, for some simple patterns MCL seems faster. This also seems to be a trend for low transfer sizes, something we would have liked to explore further to find out if it is not a fluke. Our test are over a relative large transfer sizes, and for this sizes we can conclude that our design goal to minimize overhead of use of the library is met. Our complex approach expect the Left-Right approach gives superior result to naive and comparable result to literature studied summarized in the background [2,2]. Our broadcast approach gives better then naive, but not as good as NCCL [2,1]. As the approach in NCCL is shown to be near optimal, this is as expected. To support an algorithm like the one used in NCCL, the communication engine and the linear language it support probably would need to be expanded. NCCL also send from the device, and we currently do not have a communication engine implemented to run on the device. But we can conclude that for the ghost-exchange MCL support the state of the art approach, fulfilling our design goal of supporting approaches that gives faster communication.

5.3 Real code

In the real code part we look at the Panfilov code, implementation of the single code described in the appendix, the code was adopted for two GPUs by dividing evenly between the cards the computation. Two buffers for the ghost region was made, on each device, on out buffer an one in buffer. When the main computation is done, a kernel copy s the needed results into the out buffer. This is then transferred to the other GPUs by the use of the asynchronous peer copy cudaMemcpyPeerAsync. This is then copied into the kernels main computations memory again. While this is not the optimal approach for this code, it is the most general approach and should work for all other similar problems. Of course, the best approach would have been to launch two kernels on each card, the first on the part that has dependencies with the other card, and while the computation is carried out on the non-dependent part, do the communication. But if the size of the computation is sufficient to hide the computation this would have made the measurement not relevant for evaluation of the library. For the MCL part the ALL communicator generator function was used.
The measurement of the FLOPS was carried out by the use of `omp_get_wtime` from the OpenMP library, implemented for GCC. Bandwidth was measured with `nvprof` using the `--print-gpu-trace` option. The test code is compiled with `NVCC` V6.5.16 with `-03`, ECC enabled. The best of three for all measurement was used. There is a small difference of around 1.5 GFLOPs between the hand-code, and MCL, the reason for this is unknown, but from our other characterization of the MCL it seems to give slickly better performance then the hand-coded for lower transfer sizes, it would therefore not be unreasonable that it gives better performance. We would have liked to do the same measurement on Al-Khwarizmi, but technical problems prevented this.

### 5.4 Implementation

In this section we will evaluate the implementation. A big design goal was speed, this was also a great concern during the implementation. Our test show that the library perform as good or in some instance better than hand-coded solution. One problem with the implementation was of course that one of our test system Al-Khwarizmi is not in optimal technical condition. This made it not possible to get the topology level automatically, using `cudaDeviceGetP2PAttribute` on this system. This lead us to hand-code this into the implementation. Currently it is possible to add multiple paths from same memory addresses. There is no good reason to be able to do this, and in the library is design under the assumption that there exist only one unique destination-source address pair in a node, as a transfer will therefore happen more then one time, hurting performance. The original idea was to have a hash-map to do look-up in. But combining hash-map and linked list becomes a complex solution, of course there is the very easy solution to just do linear search in the linked list. The worst case for this is O(N), which probably will give satisfactory performance given that the list only contains a few elements, an alternative could be a skip list. While the worst time is the same as linked list, average time is O(log(N)). In the allocation of `struct comm_path` is done standard `malloc`, this is not ideal as malloc is famously a killer of performance.
A better solution had been to allocate a memory pool per `struct comm_pttr`, for the user known as `mcl_pttr_t`. If a large number of elements is used in this pool, a need for call on `malloc` had been at least greatly reduced. The round (robin) communication generator was found to be not very useful, and this because the all communication generator gives good result, it was initially thought that the all generator would struggle to give good results as it schedules all communications on one destination-source device pair before moving one to the next. It still might, but this would be for patterns with large amounts of paths.

The communication engine contains several commands that are not used, like `CMND_ENG_CMND_AP2P`. This has the capability to execute several CUDA commands from one `cmnd_eng_cmnd_t`. As it was found that the `cmnd_eng_cmnd_t` was less complex to use, and that there was little or no performance hit by that, none of this were used for the communication phase. The communication engine always return to the first subroutine `LOOP_ON_JMPT`, if performance had been an issue this could have been removed, and each subroutine could have contained the same functionality, but this would have given a more complex solution.

We will finish this evaluation with a short look at what state we think the library is in.

In the current state MCL would using standard software realise cycle terminology, be in the beta phase. The software is feature complete and passing tests. For transition to realise software, several things are lacking. The software has only been tested on two system and only on old architectures, Fermi and Kepler. More testing on a larger swat of the hardware space would be necessary. Also testing on larger systems, both engineering test, and most critically performance testing to characterize the system for more then 4 GPUs.

There is also no proper install script, all testing has been done by linking in object files. Of most importance is test of other users then the author to get feedback on existing features and features user’s think could be useful for them.

### 5.5 Summary

In this section we will summarize our evaluation of MCL.

In this chapter we have shown that MCL achieve performance comparable to hand-coded code for the universal patterns built up by the user. For communication patterns like broadcast and ghost-exchange the none naive approach we have presented in this thesis show superior result to the naive. For the Pairwise approach we get result comparable with the literature, surprising the Left-Right approach did not give good performance, our broadcast approach gives lower than what is reported as best in the litterateur, but this
is as expected. We have given example of and argues qualitatively and quantitatively for that MCL makes the programming easier.
Chapter 6

Conclusion and Further Work

6.1 Introduction

This final chapter contains three sections.

- Final discussion
- Future work
- Conclusion

6.2 Final discussion

Starting this work, there existed very little work on the problem dealt with in this thesis. This allowed us to not be constrained by previous work, but to explore our own solution to a somewhat novel problem. On the negative side, coming up with the design required a substantial amount of time and work. And resulted in two other libraries which we found to solve our problem in a non-satisfactory way. While the code used in MCL is novel, the design came about from these previous efforts. And the design is also what we considered the main contribution from this work. A challenge for making APIs is not to constrain the user, by basing our design around a communication pattern the user maintains full flexibility. From this follows the rest of the design. The pattern has to be turn into communication, it could have been interpret directly, but by adding a layer that can express different strategies for turning the pattern into communications, we make changing strategy easy. The architecture maintaining a large degree of flexibility, we believe is a good solution and our implementation shows it to deliver good performance.

At present the communication generator is constrained by the types of operations supported by the communication engine and the IR language it supports. In the future work section we specify several
new ideas for new engines and generators, and some changes to the frontend. It must be determined if the present IR language is capable of supporting this. Specially the algorithms used as basis for NCCL, as they are proven to give good performance. The present IR is more or less follows the Runtime API, we believe it probably should be at a slightly higher abstraction level.

Also of question may be the future need of libraries like this or other, a growing trend in HPC is the construction of DSLs[27]. DSL give application programmer a tool for getting performance from the ever more complex hardware. Some recent examples of DSL would be Halide [28] for digital image processing, which have achieved state of the art performance, and Liszt[29], a DSL embedded in Scala, used for solving mesh-based PDE. But DSL are not at the time generally used, and of course there are many domains not covered by these languages. As have been seen in the literature [24] [14] for large GPGPU systems, scheduling effective inter-GPGPU communication becomes difficult to do by hand and should be done with the help of programs. Meaning that libraries as MLC and compilers can be of help.

In our design we specified the target users of MCL as a C programmer, as the programming model used by CUDA is more complex, the addition of concurrency would be one example, then the standard C programming model. And to do multi-CUDA programming you can expect the programmer to at least modify existing kernels. We would say that the person that can have use of our library, should not find it to difficult to use.

6.3 Further Work

In this section we will very briefly describe some ideas for extending MCL. The section roughly follows the architecture of MCL.

Frontend

In this section we will look at some ideas for the frontend.

More predefined communication primitives

The current primitives do not cover the hole range given by something like MPI, the ones made are more of a demonstration of how to make primitives in MCL. The primary reason for MCL is for the user to make on runtime their own primitives. But making of new predefined primitives is something that definitively should be done in future version of MCL, specially as NCCL has demonstrated near optimal performance for several of the MPI style primitives. To
start with, proper reduction primitives should be made. This would also necessitate the making of reduction CUDA kernels. This would also require some changes to the frontend, as variable type and type of operation also has to be specified. To launch kernels Driver API `cuLaunchKernel` most be used, as the library is written in C.

**Communication generator**

In this section we present several ideas for new communication generators.

**Knapsack**

In the thesis we have shown that contention is a problem for many communication patterns. While we know about heuristics for specific classes of patterns. To our knowledge there do not exist a solution on this, for a general pattern. But we believe the problem of contention in a network can be thought of as a general knapsack problem, to be specific a form of the Precedence-Constraind Knapsack problem \[30\]. There exist exact and heuristics for solving this, as the graphs are small the rather bad (NP) worst-case runtime of exact solution should not be a problem, more involved caching schemes could also be developed.

**Performance model as basis**

An other idea would be to explore the possibility of using a performance model like the one in \[14\] to find new heuristics for communication generators or using a performance model to find optimal communication strategies on runtime.

**Generators from NCCL**

As discussed in previous sections. In NCCL, and the literature it support it self on, the method of embedding a logical ring on the network topology and using this to generate contention free communication primitives would be of interest. This would require new communication generators, and possible new instruction to the command language of the communication engine.

**Communication engine**

In this part we will look at some ideas for new communication engines.
Synchronous communication engines, OpenMP

Using one OpenMP thread per GPU to drive each GPU is a common pattern [11]. We could support this by letting each thread scan the list of commands, looking for communication that it should execute. Or a extra linked list pointer could be added to the command structure, this could be used to create a linked list for each OpenMP thread, while maintaining the old linked list for the non-parallel engine. This would make the list executable by both a OpenMP enhanced engine and a normal. But it is a little dirty, as we modified the command struct to facilitate one specific type of engine.

Synchronous communication engines, MPI

A common programming pattern for multi-GPU programming is the use of one MPI process per device. This is something that should be looked at supporting. This would necessitate the use of inter-process communication. This is also supported by NCCL.

Asynchronous communication engine

The communication engine in MCL is synchronous, this is that control to the user is not return before all activities on device is completed. But to hide the communication of a kernel, asynchronous communication is advantageous. A asynchronous engine, for fully exploiting communication hiding. Must have the possibility to take in streams for each device from the users. While it could be possible to make communicator generators that output asynchronous commands by not inserting synchronizations command. Another possibility is to have a engine that just launch CUDA kernels, the kernels execute the communication by accessing other devices memory. That is the communication engine implemented as a kernel function, this is also what is done in NCCL. The communication kernels would then be launched on the user provided streams for each device.

Tests

We have only tested the system on a 2 multi-GPGPU Kepler and 4 multi-GPGPU Fermi systems. While Kepler only is 2 generations behind Pascal, and there has not been that a big change in PCIe and communication systems, it would be interesting to test the system on newer generation hardware, specially on those equipped with NVLink. While 2 and 4 GPGPU probably are more common then larger systems, testing on larger system would be very interesting. As we have touched on before, more codes need to be ported to MCL.
**Implementation**

We have not looked at systems with several root complexes. While the communication functions in the CUDA Runtime API has fall back functionality, as a PCIe device do not communicate with other PCIe devices on other root nodes, we may have to update our implementation.

In Pascal cards it is possible to get NVLink, a new high-speed interconnect technology from NVIDIA. This offers for P100 cards up to 160 GB/s of directional bandwidth. NVLink also has the possibility for new interconnect topology like hybrid cube mesh [31]. Supporting this is something to look into, of course we would need access to a NVLink equipped machine. Preferably one with 8 P100s or more, so something like a DGX-1 would be nice.

**6.4 Conclusion**

We have developed a library MCL, that presents the user with the abstraction of a communication pattern, built up of communication paths. This hides the details of data transfer in a CUDA GPGPU equipped system, with negligible to no impact on speed. From the literature we have found that for large multi-GPGPU system, contention is a large hindrance for performance [14]. From the literature [23] we have identified the Left-Right and Pairwise approach as two heuristics that can alleviate this problem. By layering our system in three layers, strategies like this can be used with little user effort. From our result section we have shown that the pairwise approach gives the best aggravated bandwidth on our 4 GPU test system. Tests of our library on a real test case, confirms that there is no loss of performance on using MCL on real problems. Our user-interface should be simple to use for our target users. In conclusion we have shown that it is possible to build a library that supports communication in multi-GPGPU systems, for all possible communication patterns , that is easy to use and is fast and that support complex techniques for improving communication performance.
Appendices
Appendix A

Panfilov implementation

Introduction

The Alive-Panfilove is a model of the propagation of electric signals in cardiac tissue. It consists of a 3D-Diffusion equation coupled to two ODEs. The diffusion PDE describes the exchange of ions across the cell membrane during. The ODE describes the kinetics of the reactions. A GPU accelerated solution is described in [32] for the 2D case and in [33] for the 3D case.

Naive CPU compute code

The 3D solution consists of the PDE part, which is solved as a 3D Laplacian with ghost region. And the ODE part. For each time step, border exchange is carried out.

Listing A.1: Compute part of the naive cpu code

```c
/* PDE computation part */
for (k=1; k<=n; k++)
    for (j=1; j<=n; j++)
        for (i=1; i<=n; i++) {
            index = k*offset_z + j*offset_y + i;
        }

/* ODE computation part */
for (k=1; k<=n; k++)
    for (j=1; j<=n; j++)
        for (i=1; i<=n; i++) {
            index = k*offset_z + j*offset_y + i;
            E[index] += -dt*(kk*E[index]*(E[index]-a)*(E[index]-1.0)
```

79
+ E[index]*R[index]);
(-R[index] - kk*E[index]*(E[index]-b-1.0));}

**Optimized CUDA kernel**

We stream thorough the z-dim, use loop unrolling, const restric,
the ODE and PDE part is combined, all compile time constants are
given as numeric constants, we use if statements for part of
the boarder region to ensure aligned read and write.

Listing A.2: Boarder exchange kernel for Panfilov code
__global__ void noFluxFront(double * __restrict__ e_prev) {

const int i = threadIdx.x + blockIdx.x*blockDim.x;
const int j = 1 + threadIdx.y + blockIdx.y*blockDim.y;

const int index = 0*K_OFFSET+ j*K_OFFSET + i ;
#ifdef DEBUG
if ((i < NX +1) && (j < NY +1)) {
#endif
    e_prev[index] = e_prev[index + 2*K_OFFSET];
#ifdef DEBUG
}
#endif
}

Listing A.3: Panfilov singel GPU compute kernel
__global__ void solvPanf3d(double const *__restrict__ const e_prv,
double *__restrict__ e_cur, double *__restrict__ r_cur) {

const int i = threadIdx.x + blockIdx.x*blockDim.x;
const int j = 1 + threadIdx.y + blockIdx.y*blockDim.y;
const int k = 1 + blockIdx.z*BLOCK_DIM_Z_FOR;

int index = k*K_OFFSET + j*K_OFFSET + i ;

register double e_left;
register double e_right;

#ifdef DEBUG
if ((i < NX +1) && (j < NY +1) ) {
#endif
register double center = e_prv[index];
register double north = e_prv[index + K_OFFSET];
register double south = e_prv[index - K_OFFSET];
if(i == 0) {
    e_left = e_prv[index + 1];
} else {
    e_left = e_prv[index - 1];
} if(i == (NX - 1)) {
    e_right = e_prv[index - 1];
} else {
    e_right = e_prv[index + 1];
}

e_cur[index] = center + ALPHA*(e_left + e_right - 6.0*center +
e_prv[index + J_OFFSET] + e_prv[index - J_OFFSET] + north + south);
e_cur[index] += -DT*(KK*e_cur[index]*(e_cur[index] - A) + e_cur[index] - 1.0) + e_cur[index]*r_cur[index];
r_cur[index] += DT*(EPSILON + M1*r_cur[index]/(e_cur[index] + M2))*
(-r_cur[index] - KK*e_cur[index]*e_cur[index] - B - 1.0));

#pragma unroll
for(int __k = 1 ; __k< BLOCK_DIM_Z_FOR; __k++) {
#ifdef DEBUG
    if((__k + k) < (NZ + 1)) {
#endif
        index += K_OFFSET;
        south = center;
        center = north;
        north = e_prv[index + K_OFFSET];
    if(i == 0) {
        e_left = e_prv[index + 1];
    } else {
        e_left = e_prv[index - 1];
    }
    if(i == (NX - 1)) {
        e_right = e_prv[index - 1];
    } else {
        e_right = e_prv[index + 1];
    }
    e_cur[index] = center + ALPHA*(e_left + e_right - 6.0*center +
e_prv[index + J_OFFSET] + e_prv[index - J_OFFSET] + north + south);
e_cur[index] += -DT*(KK*e_cur[index]*(e_cur[index] - A) + e_cur[index] - 1.0) + e_cur[index]*r_cur[index];
r_cur[index] += DT*(EPSILON + M1*r_cur[index]/(e_cur[index] + M2))*
(-r_cur[index] - KK*e_cur[index]*e_cur[index] - B - 1.0));
#ifdef DEBUG
    }
#endif
#endif

}  
#endif DEBUG

}  
#endif

For comparing the performance of our code, we used the Hand-coded GPU verification codes from [33], which were provided by the author, in the table called Panda. The code had some small bugs, the result presented here is from the debugged code. The code uses plane sweeping and loop unrolling for performance.

We used a 512x512x512 grid, standard compiler options, with ECC turned on.

<table>
<thead>
<tr>
<th></th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panfilov</td>
<td></td>
</tr>
<tr>
<td>Naive CPU</td>
<td>3.177504 GFLOP</td>
</tr>
<tr>
<td>Naive GPU</td>
<td>50 GFLOP</td>
</tr>
<tr>
<td>Panda GPU</td>
<td>65.885577 GFLOP</td>
</tr>
<tr>
<td>Optimized GPU</td>
<td>121.261477 GFLOP</td>
</tr>
</tbody>
</table>

Table A.1: Panfilov performance comparison

From the result we see that we get 2.42 faster than naive code and 1.8 faster than the state of the art. Developing this code, we also tried several approaches that failed to give top performance: ILP at different levels, doing this we found that X-macros was useful, using the driver API, trying to use CUDA vector type, only one kernel for ghost-exchange and several variations of this.

As our test code can be considered reasonably optimal, while we do not include it here, calculation has show it to be near the roofline limit, we have code for testing our library that can help us evaluate the performance of our library. A common problem in HPC is that coded used for evaluation often are not optimal, weakening the conclusion drawn from them.
Appendix B

mcl.h

The header file that need to be included to use MCL.

Listing B.1: The mcl.h header file

```c
#ifndef MCL_H
#define MCL_H

#include "mcl_err.h"
#include "comm_cmpl.h"
#include "pttr_opt.h"

typedef struct comm_pttr mcl_pttr_t;
typedef struct comm_pttr mcl_const_pttr_t;

mcl_res_t mcl_init(void);
mcl_res_t mcl_finalize(void);

mcl_res_t mcl_alloc_pttr(mcl_pttr_t **pt,uint32_t opt);
mcl_res_t mcl_free_pttr(mcl_pttr_t *comm);

mcl_res_t mcl_add_path(void *dst,void *src,sizeof_t size,mcl_pttr_t *pt);
mcl_res_t mcl_rmvl_path(void *dst,void *src,sizeof_t size,mcl_pttr_t *pt);

/*/ Predefined patterns */
mcl_res_t mcl_alloc_bcast(void *src,sizeof_t size,mcl_pttr_t **pt,
        const int dst_cnt,...);
mcl_res_t mcl_alloc_gather(void *dst,sizeof_t size,mcl_pttr_t **pt,
        const int src_cnt,...);
mcl_res_t mcl_alloc_scatter(void *src,sizeof_t size,mcl_pttr_t *pt,
       const int dst_cnt,...);
mcl_res_t mcl_alloc_1dim_bexch(sizeof_t size,mcl_pttr_t **pt,int pair_cnt,...);
```
/* Execute the communication pattern */
mcl_res_t mcl_sync_excomm(mcl_pttr_t *pt);
mcl_res_t mcl_sync_omp_excomm(mcl_pttr_t *pt);

/* Changes the default compiler */
mcl_res_t mcl_cng_comm_cmpl(mcl_comm_cmpl_t cc);
mcl_res_t mcl_add_comm_cmpl(mcl_comm_cmpl_t cc, mcl_pttr_t *pt);
/* Add own comm compiler function */
//mcl_res_t mcl_add_raw_comm_cmpl(void *fp, mcl_pattr_t *pt);

#endif
Appendix C

Test codes for bandwidth measurement

Test code for measuring bandwidth.

```c
#if STREAM_TEST4 /*DEVICE TO HOST*/

for(size_t stream_size = 250000; stream_size <= 102400000; stream_size *= 2) {
    printf("TEST\nSTREAM\nDEVICE\nTO\nHOST\n\n") ;
    trans_count = 1;
    time_omp = 0.0;
    time_omp -= omp_get_wtime();
    src = 0; dst = 1;
    CUDA_ERROR(cudaSetDevice(src), "cudaSetDevice");
    CUDA_ERROR(cudaMemcpyAsync(test1_host[src], test1[src],
    stream_size, cudaMemcpyDeviceToHost), "cudaMemcpyPerrAsync");
    for(int i = 0; i<GPU_COUNT; i++) {
        CUDA_ERROR(cudaSetDevice(i), "cudaSetDevice");
        CUDA_ERROR(cudaDeviceSynchronize(), "cudaDeviceSync");
    }
    time_omp += omp_get_wtime();
    fprintf(stdout, "time: %f, stream_size: %f MB, Total transfer on fabric: %f GB/s
" , time_omp , (stream_size) *(1e-6) , (trans_count *( stream_size )/time_omp) *1e-9);
}

/* END OF THIS THIST */
#endif

#if STREAM_TEST4_MCL /* DEVICE TO HOST*/
{
    mcl_pttr_t *pt1 ,*pt2 ;
    err = mcl_alloc_pttr(&pt1, 0);
    for(size_t stream_size = 250000;
stream_size <= 1024000000; stream_size *= 2 ) {

err = mcl_add_path(test1_host[0], test1[0], stream_size, pt1);

printf("TEST STREAM DEVICE TO HOST MCL\n");
trans_count = 1;
time__omp = 0.0;
time__omp -= omp_get_wtime();
err = mcl_sync_excomm(pt1);

time__omp += omp_get_wtime();
fprintf(stdout,"MCL: time: %f, stream size: %f MB, Total transfer on fabric: %f GB/s\n", time__omp, (stream_size)*(1e-6), (trans_count*(stream_size)/time__omp)*1e-9);

err = mcl_rmv_path(test1_host[0], test1[0], stream_size, pt1);
}
#endif


