Relaxations for High-Performance Message Passing on Massively Parallel SIMT Processors

Benjamin Klenk, Holger Fröning *
ZITI, Institute for Computer Engineering, Heidelberg University
Mannheim, Germany
{*benjamin.klenk,holger.froening*}@*ziti.uni-heidelberg.de*

Hans Eberle, Larry Dennison
NVIDIA Corporation
Santa Clara, CA
{*heberle,ldennison*}@*nvidia.com*

***Abstract*—Accelerators, such as GPUs, have proven to be highly successful in reducing execution time and power consumption of compute-intensive applications. Even though they are already used pervasively, they are typically supervised by general-purpose CPUs, which results in frequent control flow switches and data transfers as CPUs are handling all communication tasks. However, we observe that accelerators are recently being augmented with peer-to-peer communication capabilities that allow for autonomous traffic sourcing and sinking. While appropriate hardware support is becoming available, it seems that the right communication semantics are yet to be identified. Maintaining the semantics of existing communication models, such as the Message Passing Interface (MPI), seems problematic as they have been designed for the CPU's execution model, which inherently differs from such specialized processors.**

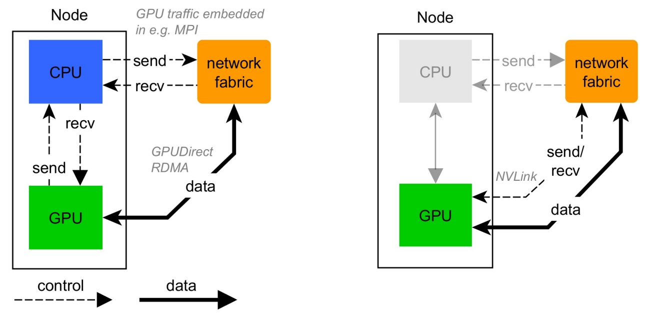
In this paper, we analyze the compatibility of traditional message passing with massively parallel Single Instruction Multiple Thread (SIMT) architectures, as represented by GPUs, and focus on the message matching problem. We begin with a fully MPI-compliant set of guarantees, including tag and source wildcards and message ordering. Based on an analysis of exascale proxy applications, we start relaxing these guarantees to adapt message passing to the GPU's execution model. We present suitable algorithms for message matching on GPUs that can yield matching rates of 60M and 500M matches/s, depending on the constraints that are being relaxed. We discuss our experiments and create an understanding of the mismatch of current message passing protocols and the architecture and execution model of SIMT processors.

***Keywords*-Heterogeneous systems, GPU Computing, Communication Models, Message Passing**

I. Introduction

The accelerated processing model, in which CPUs are the main processor and accelerators are used as an offload device for compute-intensive tasks, is changing. Accelerators are becoming peer devices, bearing more responsibility and operating on the same level as CPUs. A prime example is Intel's Knights Landing many-core processor (KNL), which connects to the QuickPath Interconnect (QPI) and is able to boot an operating system (OS) [1], and NVIDIA's Pascal GPU architecture with its hardware-supported unified virtual memory (UVM) [2]. With this new paradigm, the user has to explicitly deal with interactions between accelerators themselves, in addition to accelerator and CPU interactions. Suitable abstractions are inevitable to make large systems programmable in an efficient and performant way.

*This work was performed while the first two authors were affiliated with NVIDIA Corporation.



(a) CPU-centric communication (b) GPU-centric communication

Figure 1. Comparison of a traditional CPU-centric system and a system, in which CPUs and GPUs are peers in the network.

Many High-Performance Computing (HPC) systems are comprised of clustered compute nodes, connected by high-performance networks. Thus, communication is key to high performance. In the aforementioned CPU-centric model, the CPU is the only device that sinks and sources traffic, as shown in Figure 1(a). With accelerators becoming peers, illustrated in Figure 1(b), the need for them to control communication is becoming more and more important. For example, Intel's KNL embeds an OmniPath network interface controller (NIC), and NVIDIA just released the NVLink-capable Pascal GPU. While the hardware support is now available, the question what the right communication semantics for these highly-parallel processors are remains open.

While there are various message passing models for assorted application domains, the Message Passing Interface (MPI) for HPC may be the most dominant one. In typical cluster systems MPI has become the de-facto standard for communication across OS boundaries. Messages that are sent across the network fabric are placed in queues on the receive side. Receive operations take messages from queues and forward the data to the user application. MPI's semantics are rich, making it a user-friendly abstraction. For example, when receiving data the user only has to specify the source of the message and MPI guarantees in-order delivery. Additionally, the user can apply a wildcard to the source specifier to receive messages from any source process.

To receive the right message, the MPI system needs to match incoming messages with receive requests. The matching is complex as it needs to deal with wildcards and ordering. As a result, most MPI implementations use lists for this purpose, which have to be traversed for every incoming message or receive request. Message matching is key to high message rates and also to the application's performance. For example, the Fire Dynamics Simulator (FDS) application has been accelerated by 3.5x just by improving MPI's matching process without touching the application itself [3].

While list-based approaches work fairly well for latency-optimized processors, massively parallel processors need to exploit parallelism to perform well. Implementing MPI-like messaging on GPUs seems difficult and rather inapplicable. Additionally, due to their highly parallel nature, GPUs could be expected to exchange significantly more messages than CPUs. A node's CPU generally runs tens of processes, while GPUs run grids of thousands of cooperative thread arrays (CTAs), each being independently executed. It seems fair to presume that many of these CTAs need to send and receive messages. Thus, the matching of messages becomes a major limiter for high message rates.

The design space for GPU-centric communication is still wide open and we want to create an understanding of implications that result from processing messages on GPUs, particularly message matching. We present an algorithm for message matching on a GPU that is fully compliant with MPI and analyze its performance and limitations. We further relax some of MPI's constraints, allowing for more parallelism and assess the feasibility of these relaxations based on a comprehensive analysis of exascale proxy applications. With this work we make the following contributions:

- We present a trace-based analysis of exascale applications with regard to characteristics that have an impact on the matching problem.
- We introduce a tag matching algorithm that complies with MPI-like semantics and report performance on the three latest generations of GPUs. We also identify parts of the semantics that make it hard to map the matching task to a GPU.
- We propose optimizations and trade-offs that relax MPI's guarantees to increase the matching rate. We show how these relaxations improve performance and their applicability to exascale applications.

Note that our results and insights are not limited to MPI; they are also applicable to other message passing paradigms. We understand that MPI serves as a proxy here, as its messaging system provides many features. Similarly, even though we evaluate the proposed relaxations for GPUs, the insights are also interesting for CPU-centric messaging, including both general-purpose and many-core architectures.

We also understand that GPUs are not going to run a full MPI stack, as GPUs struggle with single thread performance and lack dynamic in-kernel memory management. However, we want to leverage insights from MPI to design better suited communication models for GPUs.

The remainder of this work is structured as follows: Section II provides the background on GPUs and MPI. Section III elaborates on related work. Section IV presents our analysis of exascale applications regarding their communication behavior. Section V presents our algorithm for MPI-like tag matching on the GPU, followed by relaxations we propose to increase performance. Section VII discusses the results and insights and Section VIII concludes.

II. Background and Methodology

This section presents an overview of current GPU architectures and their programming and execution model. MPI is introduced as a prime example for message passing interfaces. At the end of the section, we present our methodology to evaluate the matching process.

A. *GPUs and their role in the network*

General-purpose GPU (GPGPU) computing has been highly successful in accelerating compute-intensive applications. NVIDIA's GPUs implement a great number of streaming multiprocessors (SMs), ranging to more than 60 in the recently released Pascal architecture. Each SM comprises hundreds of simple compute cores, making the GPU a massively-parallel processor. However, unlike CPU processors, GPUs are optimized for throughput rather than latency and cores execute instructions in order at much lower clock rates.

GPUs offer a throughput-oriented memory system with high-bandwidth off-chip memory. Each SM provides an explicitly managed scratch-pad memory (called shared memory) and a large register file. Accesses to these local resources are fast, though an extensive use reduces the number of threads that can be grouped together on a CTA.

NVIDIA GPUs are programmed by CUDA (Compute Unified Device Architecture). A compute kernel comprises thousands of CTAs with each CTA consisting of up to 1024 threads. Multiple CTAs can map to a single SM. The code is executed in groups of 32 threads in that the same instruction is dispatched to the whole group, called a warp. The scheduler picks available warps and dispatches their instructions to the hardware. Warps that wait for memory accesses are tracked by a scoreboard and noted as inactive until the operation is finished. Having plenty of warps ready to be scheduled allows to hide long instruction latencies.

To operate efficiently, all threads within a warp have to follow the same control flow. Results from diverging threads are simply masked off. Within warps, CUDA provides synchronization primitives that allows data to be communicated (warp shuffle) or evaluated (warp vote) efficiently. For example, the *ballot* intrinsic takes a condition and returns a 32-bit vector wherein the least significant bit (LSB) represents the first thread of the warp and is set if the condition evaluates to true. Additionally, bit functions like finding the position of the first set bit (*ffs*) or counting leading zeros (*clz*) are also supported by the hardware.

Traditionally, GPUs have been perceived as plain accelerators, attached via PCIe and controlled by the CPU. However, this has also been the main bottleneck since the GPU needs data to be available in its high-bandwidth on-device memory. That is why NVIDIA introduced Unified Virtual Memory (UVM), for sharing CPU and GPU memory, and NVLink, a high-bandwidth interface. The role of the GPU is changing now that multiple GPUs can be clustered with NVLink, and GPUs are becoming more and more autonomous and independent. Through NVLink, GPUs can access remote memory directly, spanning a virtual address space across the network. Communication between GPUs in traditional networks has always been performed by the CPU, requiring running kernels to be interrupted and control returned back and forth between GPU and CPU. With GPUs being peers in the network, the need for sourcing and sinking network traffic in the GPU becomes inevitable.

B. MPI

The Message Passing Interface (MPI) has become the standard communication method for clustered systems with distributed memory. The receiver of small messages has to either buffer the data of the message within the MPI framework, or if a matching receive has already been posted, the data is copied directly to the buffer in user space. For large messages, the target matches the message with a receive request and then initiates the data transfer from the source directly to the user buffer on the receiving side.

Message matching is based on a tuple containing the message source (*src*), a *tag*, and a communicator (*comm*). The user can use wildcards for *src* (MPI_ANY_SOURE) and *tag* (MPI_ANY_TAG). In addition, MPI guarantees that messages sent between the same process-pair are matched in order.

Incoming messages that have been received, but could not be matched with any receive request that has been posted so far, are kept in the so-called Unexpected Message Queue (UMQ). Any new incoming receive request has to look for a message in that queue first. If no match can be found, the receive request is appended to the Posted Receive Queue (PRQ). The PRQ is the counterpart to the UMQ, where any incoming message looks for already posted receive requests. Common MPI implementations implement UMQ and PRQ as lists since elements can be easily removed without reordering other elements.

C. Methodology

In our vision, GPUs communicate autonomously as depicted in Figure 1. Like any other messaging-based system, each GPU implements a message queue and keeps connections to its peers. NVLink and PCIe systems allow GPUs to address a peer's memory directly by spanning a virtual global address space (GAS) across the network. 'Send' operations write messages to queues in remote memory and 'Receive' operations query the local queue for new messages. For this work, we presume that there is one communication kernel running on a single GPU streaming processor (SM) while other SMs are executing the application's grid, similar to network on-loading protocols with dedicated communication cores [4] [5]. With message passing layered on top of the GAS, the user calls send/recv routines inside its application kernel, which is concurrently running along with the communication kernel. The matching and other communication tasks are performed in the background by the communication kernel.

Our first step is to characterize the matching problem by analyzing MPI traces of real applications. We provide an analysis of the characteristics that have an impact on the matching performance. This includes the source rank and tag usage and their distribution, the usage of wildcards, and the depth of UMQ and PRQ. The applications we believe to be representative are the U.S. Department of Energy (DOE) Mini-apps [6]. The DOE has made the traces publicly available and we used Python and R scripts to extract the information needed for our analyses from the trace files. General statistics are collected by parsing the trace files, while others require message queues to be restored any time a matching is attempted. The results help to assess feasibility of the relaxations of the message passing protocol we are going to introduce later.

We present an algorithm that performs *tag*/*src* matching on GPUs and run experiments to assess the performance, based on micro-benchmarks and synthetic scenarios. We have to rely on micro-benchmarks as it is not possible to run the applications on GPUs without supporting a full MPI stack on the GPU itself. Nonetheless, these experiments provide insights into the distinct characteristics of the matching problem as well as performance limits and limitations. We run our experiments on three generations of GPUs: Kepler[1], Maxwell[2], and the recently released Pascal[3]. Note that we do not compare the GPU with the CPU matching performance. CPUs are highly optimized for a high number of instructions per cycle at high clock rates, making them perform well at sequential tasks. Rather, we want to present challenges faced by doing message passing according to MPI semantics on the GPU. Nonetheless, we experimentally assessed the

[1]Tesla K80 (single GPU) CUDA v. 7.0.27, NVIDIA driver v. 346.46
[2]Tesla M40, CUDA v. 8.0.27, NVIDIA driver v. 361.72
[3]GTX1080, CUDA v. 8.0.23, NVIDIA driver v. 367.35

CPU's matching rate with various MPI implementations and found that 30M matches/s can be achieved with short queues. However, this rate drops to below 5M matches/s for queues longer than 512 entries. More detail can be found in [7].

In the last part of this work we relax matching semantics. This allows us to better map the message passing model to the GPU's execution model. We analyze how these relaxations improve the matching rate with respect to synthetic and real application scenarios. Our goal is to show how communication semantics have to look for massively parallel processors, such as GPUs. In addition to performance, we assess the feasibility of the relaxations with respect to our findings from the application analyses.

III. Related Work

UMQ and PRQ lengths were also analyzed by Brightwell [8] and Goudy [9]. They report that applications from the NAS Parallel Benchmark (NPB) suite generate a significant amount of unexpected messages resulting in queue lengths of up to 200 entries for UMQ and up to 140 processes. Furthermore, PRQ is always smaller than UMQ and average search lengths are always less than 30 entries. They ran their experiments on dual Pentium 4 Xeon Processors with a Myrinet-2000 network interface. Of note, the authors stated that it is necessary to analyze real applications. Based on this, Underwood et al. [10] propose a list-acceleration unit for NICs and show benefits as long as the queue fits in on-NIC memory. Keller et al. [11] did a similar analysis of large-scale applications, showing that the UMQ length scales linearly with the process count for a thermodynamics application on the Jaguar and JaguarPF systems. However, this only applies to rank 0 while other ranks do not exceed a queue length of 200. UMQ lengths for other applications are much smaller, ranging between 10 and 30 entries.

Zounmevo et al. [12] propose two new algorithms for message matching on CPUs, both aimed at reducing the memory footprint and enhancing scalability. Their approach partitions the rank-space such that multiple queues can be implemented. Each entry is given a sequence number to comply with wildcards. Their results show significant relative performance improvement for two applications (nbody and radix sort) on a small scale InfiniBand Cluster with AMD Opteron CPUs, but no absolute performance numbers are provided. Flajslik et al. [3] use hashes to address multiple queues and insert so-called marker entries to restore order and support wildcards. Their approach yields 3.5x better performance than traditional, list-based matching algorithms for the Fire Dynamics Simulator with 1,792 processes and 256 queues on a 64-node InfiniBand and Xeon CPU cluster. They also looked at LAMMPS and integer sort (NPBS), but only reported a reduction in match attempts as opposed to overall performance.

Some work also exists regarding direct GPU-GPU communication. Stuart et al. [13] implemented a messaging scheme on GPUs. However, a CPU thread is needed to execute the messaging process. The authors claim that message passing is not possible on GPUs since they do not have access to network devices.

This changed with NVIDIA's introduction of GPUDirect RDMA, which is used in our previous work [14] to implement GGAS, a global address space for GPUs. It is based on an FPGA NIC to forward load and store operations across the network fabric. We then compared this PGAS model with RDMA and MPI and showed that direct GPU communication is favorable for a wide range of communication patterns [15].

Another work [7] presents a more comprehensive analysis of Exascale traces, which comprises point-to-point and collective communication, with a focus on the CPU.

To the best of our knowledge, our work is the first to analyze Exascale traces with regard to UMQ and PRQ lengths. Also, we are not aware of any work that implements message matching on GPUs.

IV. Exascale trace characteristics

This section presents findings from our exascale proxy application analysis. These traces are important as they provide insights on the matching requirements and potential for possible relaxations, both in terms of performance and feasibility. Our findings can also be applied to applications with similar behaviors and communication patterns and are not limited to accelerators.

A. Communication characteristics

In the following we present our observations from the application analyses with regard to the matching process. The applications are introduced in Table I. These proxy applications cover a wide area and are considered representative for current and future HPC systems.

Number of src and tag wildcards used: the first surprising observation is that none of the analyzed applications uses the *tag* wildcard, and only two applications (*Design Forward MiniDFT* and *MiniFE*) apply the *src* wildcard (see Table I). Note that we extract this information from the traces and not from the source code (the *dumpi* trace format contains such information) and there is no information whether all traces also cover the initialization phase, in which wildcards might be more prominently used. Nonetheless, initialization is outside the critical path and can be performed by the CPU before the application is handed off to the GPU. MPI's matching process is complicated by the support for wildcards, which seems quite unnecessary with regard to most exascale proxy applications.

Number of communicators: except *CESAR NEKBONE* (2) and *Design Forward MiniDFT* (7), all applications use a single communicator for point-to-point communication. The communicator is part of the matching criteria and would inherently offer parallelism since no wildcard can be applied.

Application	**Description**	**Comm. Pattern**	***src* WC**	***tag* WC**	***#communicator***	**Peers**	**#tags**
MOCFE (CESAR) *	Neutronics code	Nearest neighbor	no	no	1	4	2944
NEKBONE (CESAR)	Fluid Dynamics code	Nearest neighbor	no	no	2	29	1044
CNS (EXACT)	Compressed Navier-Stokes	Nearest neighbor	no	no	1	72	67
MultiGrid (EXACT)	MultiGrid solver (BoxLib)	Nearest neighbor	no	no	1	37	845
LULESH (EXMATEX)	Hydrodynamic simulation	Nearest neighbor	no	no	1	19	2
CMC (EXMATEX) †	Classic Monte Carlo	Nearest neighbor	-	-	-	-	-
AMG (DF)	Algebraic MultiGrid Solver	Nearest neighbor	no	no	1	79	1
AMR Boxlib (DF)	Adaptive mesh refinement	Irregular (sparse)	no	no	1	35	631
MiniAMR (DF) ‡	Adaptive mesh refinement	Irregular (sparse)	-	-	-	-	-
BigFFT (DF) †	Large 3D FFT	Many-to-many	-	-	-	-	-
Crystal Router (DF)	MPI many-to-many code	Staged all-to-all	no	no	1	6	14
Fill Boundary (DF)	Halo update (BoxLib)	Nearest neighbor	no	no	1	23	24
MultiGrid (DF)	MultiGrid solver (BoxLib)	Nearest neighbor	no	no	1	10	104
MiniDFT (DF) *	VASP electronic structure calculation	Many-to-many	yes	no	7	19	19666
MiniFE (DF) *	Finite element solver	Staged all-to-all	yes	no	1	15	3
SNAP (DF) ‡	Neural particle transport	Nearest neighbor	-	-	-	-	-
PARTISN (DF) *	Neural particle transport	Nearest neighbor	no	no	1	1 §	3444

Table I
APPLICATIONS THAT ARE PART OF OUR ANALYSIS. EACH APPLICATION PROVIDES TRACES WITH VARIOUS NUMBER OF RANKS. THE NUMBERS PRESENT THE AVERAGE ACROSS THE MAXIMUMS OF EACH CONFIGURATION.

* The queue analysis of this application was not possible since rank numbers are renamed, resulting from MPI's cart create.
† The application does not use send/recv operations.
‡ The trace file was too large to process.
§ Only one rank is communicating with all other ranks. Any other rank does not exchange messages with all other ranks.

Unfortunately applications do not allow for much parallelism by only using a single communicator.

Number of peers a rank is communicating with: most applications exchange messages with about 10-30 peer ranks. This is a result from the nearest neighbor communication pattern that can be found in the majority of applications. *EXACT CNS* (72) and *Design Forward AMG* (79) spread their messages across the most ranks, however, this is still only a fraction of the total number of ranks the application is launched with. This makes communication rather local and opens up considerable optimization potential with regard to process mapping and topology.

Distribution of src and tag space: another interesting aspect is how often certain source and tag values are used. We observed that this varies significantly across the applications. For example, *Design Forward MiniDFT*, *CESAR MOCFE*, and *Design Forward PARTISN* use thousands of different tags. On the other hand, *Design Forward AMG*, *EXMATEX LULESH*, and *Design Forward MiniFE* use less than four different tags. We will show later that this has an impact on the choice of data structure as traditional lists can be replaced with hash tables. In addition, none of the applications needs tag values longer than 16 bits. Together with the 32-bit value for the source and some bits for the communicator, the entire header could fit into a single 64-bit word.

UMQ and PRQ length: the most critical characteristic with regard to matching is the length of PRQ and UMQ. The UMQ is depicted in Figure 2 and we omitted the graph for the PRQ due to their similarity. The longer the queues the more time is needed to find the right match.

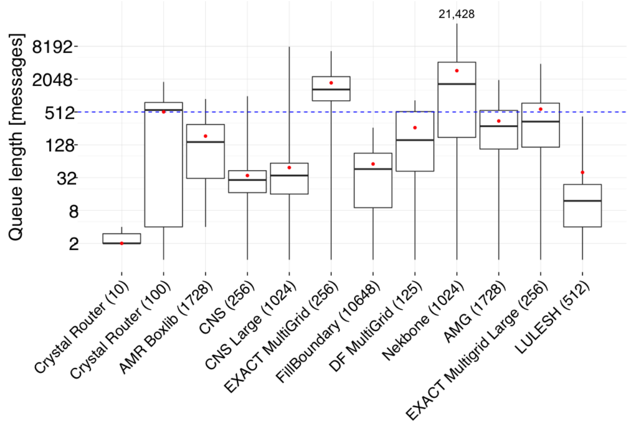


Figure 2. Maximum length of each rank of the UMQ for different applications. The red dot represents the mean, the dotted blue line indicates an arbitrarily chosen reference of 512 messages. The extensions of the boxes mark minimum and maximum.

Based on the trace files, we reconstruct the queues to assess their maximum length at any matching attempt. Our first observation is that UMQ and PRQ show similar queue lengths. Most of the applications' queues range below 512 entries. *EXACT MultiGrid* and *CESAR NEKBONE* have the longest queues with the mean across all ranks being 2,000 (median at 1,500) and 4,000 (median at 1,800) entries, respectively.

V. MESSAGE MATCHING ON GPUS

This section describes how tag matching is performed on the GPU. While CPUs keep message and receive request queues separated from UMQ and PRQ, we unify them in

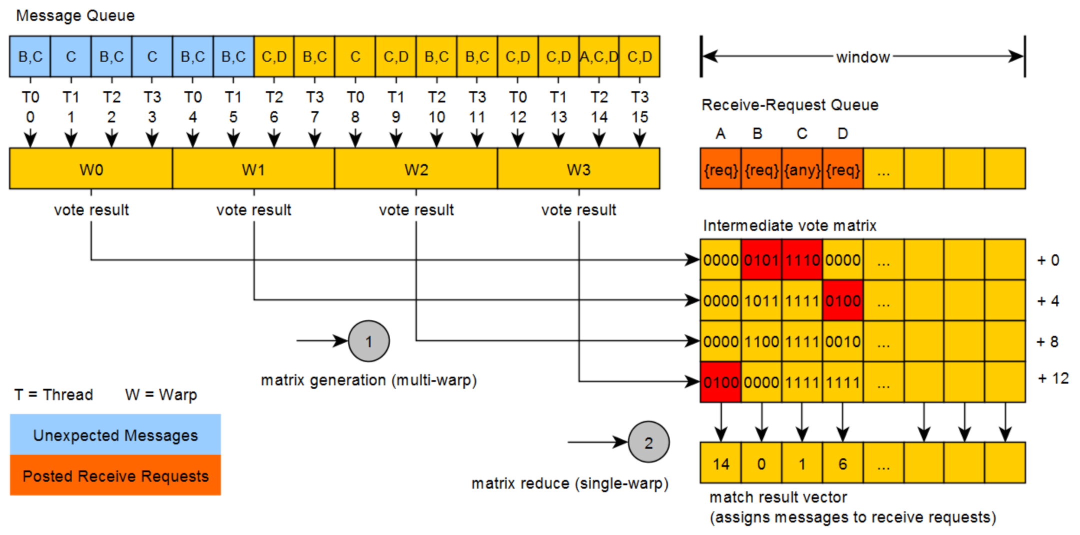


Figure 3. Message matching algorithm on the GPU. The picture shows four warps with a generic warp size of four threads.

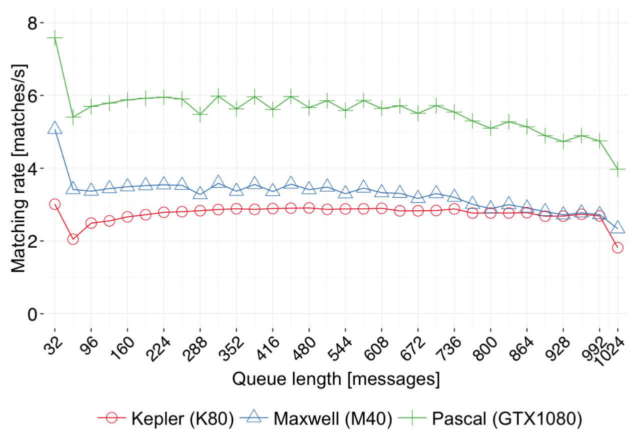


Figure 4. Single CTA matching rate for the GPU algorithm on various GPU architectures.

our GPU implementation. The UMQ is placed at the head of the message queue and the PRQ at the head of the receive request queue, respectively. Both queues reside in global memory on the GPU and we assume that new messages and receive requests are always placed into global memory. Furthermore, since the matching only concerns the endpoint, no messages are actually sent between GPUs in our synthetic scenarios.

A. Algorithm

Our algorithm is divided into two parts: scan and reduce. Scan builds a matrix where matching messages are entered for every receive request. Messages are represented as rows and receive requests as columns. Due to wildcards or identical source and tag specifiers, a message can match multiple receive requests, creating dependencies between elements within a row. Since multiple messages can also match the same receive request, dependencies also exist between elements within a column. The algorithm and setup are depicted in Figure 3. The letters in the message queue indicate which receive requests match with this particular message. For example, *B,C* means that this message matches with receive request *B* and *C*.

The scan is performed by a large number of threads, each matching one message with all receive requests. The number of threads determines the number of rows of the matrix. Due to the limited amount of available shared memory, the scan is performed hierarchically: in the first step, all threads within a warp perform a vote, using CUDA's *ballot* intrinsic. The vote returns a 32-bit vector, each bit representing one thread of the warp. A set bit means the message matched the receive request. The bit-vector is written to the matrix, which comprises as many rows as warps being used. This minimizes the number of rows, because effectively only one bit is stored for every thread. Algorithm 1 describes the procedure in pseudocode. The *window* variable is the number of receive requests being scanned. Instead of reading the entire message or receive request, only *src* and *tag* are being read. The communicator is inherently given since we presume one matching engine per communicator.

The top-left of Figure 3 shows the message queue and warps, where each thread is assigned to one message. On the right, the receive request queue that is scanned in the first phase is shown. As can be seen, each warp writes its intra-warp vote result for every receive request to the matrix. After the scan, the second phase needs to reduce the column vectors to a single match vector.

Due to dependencies between columns, the reduce phase is sequential. Also, as so far all NVIDIA GPUs only support 32 warps per CTA, the matrix height is limited to 32. Consequently, one warp is sufficient to reduce a single column.

The reduce phase is presented as pseudocode in Algorithm

2 and shown in Figure 3. At the first column, each thread starts off with a 32-bit mask that has all bits set, each bit representing a message. The bit-mask is necessary to resolve the dependency within a row, meaning one message can only match one receive request. Whenever a match occurs, the mask is modified and bits are erased. Next, each thread reads one element from the column, where *thread n* reads the *nth* element. The assignment is important since lower IDs belong to messages earlier in the queue. The *ballot* intrinsic is used again to determine threads that found a match. Note that the mask needs to be applied here to avoid re-matching the same message. The *find first set (ffs)* intrinsic returns the position of the first set bit and determines the lowest ID of the warp that placed its vote in the column. Again, lower IDs have higher priority due to ordering. Another *ffs* is required to identify the exact position of the thread within the warp that gets the match. Last, the appropriate bit of the mask needs to be erased and the next column can be reduced.

Looking at Figure 3 again, the first column of the matrix contains only a single set bit. The matching message is determined by the row and the bit position within the element of the matrix. In this example, a warp contains four threads and a total of four warps are used. Only *thread 3* sees a match in its element at position 3. Since the element was written by *warp 3* in the first phase, the matching message can be found at position 14 (warp ID x warp size + bit position - 1). The next column contains several matching messages. The first thread gets the match due to its lowest thread ID. Again, the first bit within the element points to the matching message, which can be found at the head of the queue. This procedure is repeated until all columns are reduced. Note that it also works with wildcards as the third column shows.

The main bottleneck of this algorithm is the sequential reduce phase, while the first phase can be executed by several warps in parallel. However, both phases can be pipelined to overlap execution. After the reduce, the column can be reused and overwritten by the scan phase again. The same applies to the receive requests that can be overwritten by pre-fetching new requests after the scanning is done.

The result of the matching algorithm is a vector that indicates the position of the matched message for every receive request. In real applications, matches cannot be found for all receive requests, leaving *no-matches* in the vector, possibly making it even sparser.

The last step of the matching algorithm is to compact the queues to advance the head pointer and start matching on the remaining requests. The compaction is composed of a prefix scan and memory move operations. In cases when the number of matches is very low, the bubbles can be tolerated and the compaction can be skipped.

Algorithm 1 Multi-warp scan

```
1: SendObj = sendBuffer[ thread:id ]→getObj()
2: for i from 0 to window - 1 do
3:     RecvObj = recvBuffer[ i ]→getObj()
4:     int32 vote = __ballot( SendObj == RecvObj )
5:     voteMatrix [ warp:id * window + i ] = vote
6: end for
```

Algorithm 2 Algorithm to reduce a column-vector, which contains the vote results, to a single match.

```
1: int32 mask = 0xFFFFFFFF
2: if thread:id < warps then
3:     for i from 0 to window - 1 do
4:         int32 vote = voteMatrix[thread:id * window + i]
5:         int32 bidders = __ballot( vote & mask )
6:         if thread:id == __ffs(bidders) -1 then
7:             int32 match = __ffs( vote & mask ) - 1
8:             mask = mask & ∼ (1<<match)
9:             result[ i ] = thread:id * warp:size + match
10:        end if
11:    end for
12: end if
```

B. Synthetic micro-benchmarks

In order to assess the performance of our algorithm we use a synthetic workload on different GPU generations. With our algorithm, up to 1024 messages can be matched in one iteration and unlike list-based approaches the order of receive requests has no impact on the matching rate. The message queues in this benchmark contain random tuples in random order, but all tuples of the message queue match with tuples in the receive queue, thus no elements are left in the queues after the matching. Figure 4 shows the performance for different queue lengths. Note that queues with less than 64 elements are scanned by a single warp and no matrix is generated.

The performance of our algorithm is steady and yields around 3M matches/s for a Kepler-class GPU, around 3.5M matches/s for a Maxwell M40 and 6M matches/s for a Pascal GTX1080. Since the scan has linear time complexity, the higher clock rate of the M40 and GTX1080 yields superior performance. At a queue length of 1024, the performance drops because all warps are required to perform the scan and the reduce phase cannot be overlapped anymore.

Queues that contain more than 1024 elements require multiple iterations and the performance drops accordingly. At this point, the order of the receive requests matters. While an ordered queue would yield the same performance as shown in the graph, a reversed queue would decrease performance. However, as the trace analysis showed, the vast majority of applications have queues smaller than 1024 elements. Nonetheless, we are going to discuss longer queues later

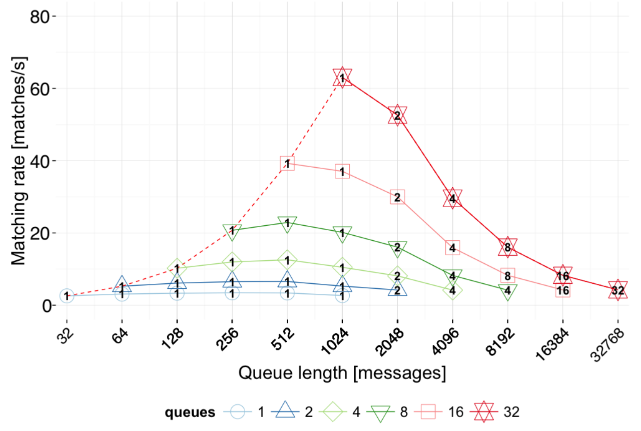


Figure 5. Matching rate for multiple queues on the Pascal-class GTX1080. The numbers represent the number of CTAs, running on the same SM.

when we introduce relaxations.

VI. Relaxations

The previous message matching algorithm completely complies with MPI semantics, but the resulting dependencies limit parallelism and hence the matching performance. However, multiple levels of parallelism are possible. The top level partitions among communicators, as there exist no dependencies. Unfortunately, applications tend to use only a single communicator, as our application analyses revealed.

The next level could partition among ranks, but this is impossible due to wildcards. Consequently, our first relaxation proposal is to *prohibit the use of the any source wildcard*. Note that prohibiting tag wildcards would allow to further partition among tags, but tags are usually not uniformly distributed, resulting in an imbalanced utilization of queues.

Another limitation is caused by unmatched messages, which result from the entire PRQ being traversed with no matching request found. Having all receive requests posted before new messages arrive removes that inefficiency and guarantees that each message matches in one iteration. This brings us to our next relaxation: *no unexpected messages*.

As described earlier, exactly-one matching is guaranteed by naming and ordering. Ordering is another severe limitation for the matching process since it creates dependencies. However, without ordering the user has to take care to identify the right messages, for example, using tags to uniquely identify the right message. On the other hand, in a strict Bulk Synchronous Parallel (BSP) [16] model, tags can be reused after synchronization. This brings us to the last relaxation we are proposing: *no implicit ordering*.

A. Rank partitioning - no source wildcard

Prohibiting the *src* wildcard allows the rank space to be statically partitioned and arranged into multiple queues. The number of queues depends on the number of peer a rank is communication with and messages that are exchanged. Queues should be kept moderate in length in order to yield high matching rates.

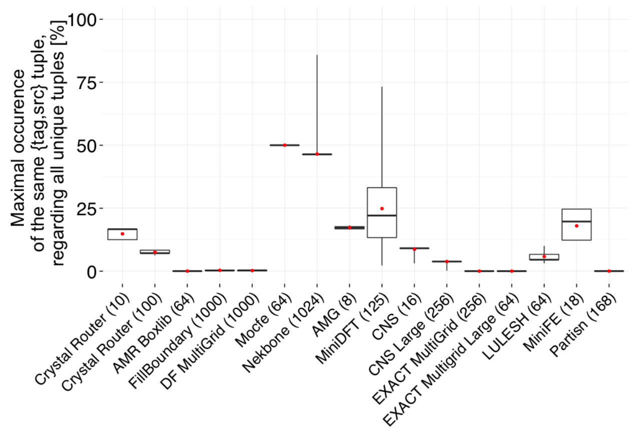
Generally, the more queues are used the better the performance since more parallelism can be exploited. However, this is only valid if each queue contains at least 32 entries in order to efficiently use warps on the GPU. In fact, this is interesting since the single queue approach benefits from less than 1024 entries in the single queue, while implementing many queues requires many messages from different sources in order to equally utilize multiple queues.

Figure 5 depicts the performance for different numbers of queues plotted against the total queue length. In addition, the annotated numbers show the number of CTAs that are required. Note that all CTAs run on the same SM and the graph refers to the Pascal GTX1080. In this experiment, the GTX1080 yields an average speedup of 2.12x over the Kepler K80 and 1.56x over the Maxwell M40, respectively.

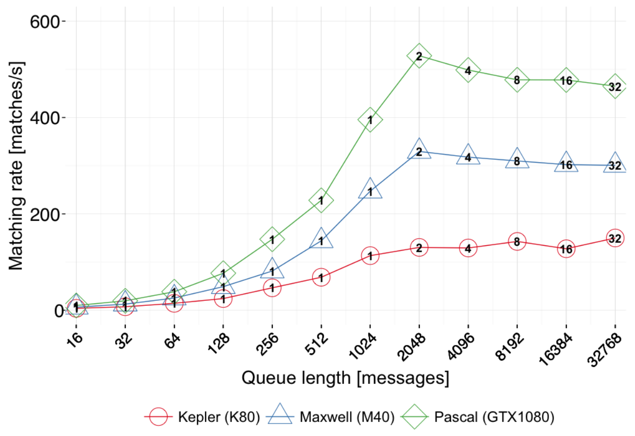
The first observation is that performance scales almost linearly for up to four queues, afterwards it is just below linear due to extra overhead. If queues are split up, each queue becomes smaller and provides less opportunity to overlap both the scan and reduce phases. As a result the pipelining is less efficient. Also, the synchronization required for pipelining applies to all warps and not only to the warps that process the same queue.

If the queue length exceeds 1024 elements, one CTA cannot provide enough threads unless one thread matches more than one message. A single SM is able to schedule warps from up to 16 CTAs concurrently as long as shared memory and register resources are sufficient. As can be seen, increasing the number of CTAs allows for longer queues, but also reduces efficiency and performance. According to NVIDIA's occupancy calculator, this algorithm allows two CTAs to run in parallel. Hence, more CTAs leads to serialization and performance is reduced. Nonetheless, more CTAs provide more threads allowing for larger queues, which are necessary in some cases. If multiple SMs were used, the performance would be increasing linearly since all CTAs would be running in parallel, however, less resources would be available to execute the application.

Besides performance, feasibility of our relaxations is important. The number of peers a rank is communicating with constitutes the maximum number of queues. We have shown that most applications allow roughly 20 queues to be used, while only a few applications allow more. Additionally, it is important to look at the communication pattern. Having multiple queues is only efficient if queues are evenly used. A uniform distribution of ranks that are addressed would be the best case. We analyzed how often a given rank addresses any other rank. While most of the applications show a regular and uniform behavior, *CESAR Nekbone* and *AMR Boxlib* showed a rather irregular communication behavior.



(a) Uniqueness of tuples



(b) Hash table performance

Figure 6. The left graph shows the maximum occurrence of a given {*src*, *tag*} tuple among all tuples and destinations. The lower the better for hash tables, which performance is shown in the right graph.

B. No unexpected messages

Our second relaxation prohibits the use of unexpected messages, which has two performance implications. First, after matching a compaction step is required to remove matched elements. Experiments have shown that this reduces the matching rate by about 10%. Second, non-matching messages still propagate through the entire receive request queue without any progress. In our algorithm performance decreases linearly with the number of matched messages per iteration. For example, if only half of the messages can be matched, the matching rate shown in this paper is reduced by about 50% as well.

While prohibiting the *src* wildcard has no implication on how code is written for most of the applications we have analyzed, not allowing unexpected messages has a much greater impact. It would require the vast majority of applications to be rewritten to remove the synchronization of send/recv calls. Either barriers are required to ensure that receives have been pre-posted or receivers have to be queried to determine if they have posted the receive request already.

C. No wildcards and ordering

The last relaxation we propose is out-of-order delivery of messages, removing dependencies and allowing hash tables to be used as primary data structure. We also prohibit wildcards for the sake of simplicity, but theoretically they could be supported with hash tables as well. The main benefit of hash tables is that they enable constant insert and search time complexity, but for hash tables to perform well the input data set should comprise as many unique values as possible to avoid collisions.

In Figure 6(a) we show the uniqueness of {*src*, *tag*} tuples among all destinations within an applications. For example, a value of 50% means that a single tuple appears in 50% of all messages to a given destination. This would be a bad case for hash tables since many collisions would reduce performance significantly. However, as can be seen, most applications range in single digit percentages, supporting the choice of hash tables.

We implemented a two-level hash table with the primary table being five times larger than the secondary table. First, all threads fetch a receive request from the queue and insert it into the primary hash table. If there is a collision, the receive request is inserted into the secondary table. In case of another collision, the thread holds on to the request for the next iteration. During the second phase, each thread fetches a message, calculates the hash key, and queries the primary table. If no entry matches, the secondary table is queried and if no match can be found again, the match is deferred to the next iteration. The more collisions occur, the more iterations are required to match all elements. We chose Robert Jenkin's 32-bit (6-shifts) hash function [17], which we found to be in wide use. Future work might further investigate various combinations of hash functions and collision resolution policies.

The performance of our hash table approach is depicted in Figure 6(b). Relaxing the ordering opens up a tremendous speedup for the matching process. With 1024 elements and one CTA, a matching rate of 110M matches/s (Kepler) can be achieved, while 32 CTAs yield 150M matches/s (Kepler). An impressive performance is observed on the Pascal GTX1080, which yields about 500M matches/s. This translates into a speedup of 3.3x over Kepler. Note that due to the SM's limited resources the execution of multiple CTAs is serialized. Also note that we chose random values for the {*src*, *tag*} tuple for our experiments.

Although the performance is remarkable, relaxing ordering has significant implications on the user, who now bears more responsibility than with ordered message delivery. The *tag* has to be used to uniquely identify messages from the

Wildcards	Ordering	Unxp. msgs.	Part.	Data structure	Perf.	User implication	Comments
yes	yes	yes	no	Matrix	Low	None	MPI (<6M matches/s)
yes	yes	no	no	Matrix	Low	Medium	$\sim$ 6M matches/s
no	yes	yes	yes	Matrix	High	Low	$<$ 60M matches/s due to compaction
no	yes	no	yes	Matrix	High	Medium	$\sim$ 60M matches/s)
no	no	yes	yes	Hash Table	Very High	High	$<$ 500M matches/s
no	no	no	yes	Hash Table	Very High	High	$\sim$ 500M matches/s

Table II
SUMMARY OF OUR RELAXATIONS AND THEIR IMPLICATIONS. *Part.* SHOWS WHETHER PARTITIONING IS POSSIBLE OR NOT. PERFORMANCE NUMBERS REFER TO THE PASCAL-CLASS GTX1080 GPU.

same source, hence applications have to be rewritten and restructured. We still think this would be applicable in many iterative and BSP-like applications.

VII. DISCUSSION

We focused on the message matching as it is key to high message rates, which again is key to many applications [18]. Furthermore, high message rates are also essential in a PGAS model, which receives noticeable attention on GPUs [19]. Note that the matching performance of our algorithms is summarized in Table II.

A. Communication characteristics

The application analysis showed that point-to-point communication is generally limited to a fraction of the total number of available ranks. Assuming the *src* wildcard was prohibited this allows for 10-30 queues to be implemented in most applications. Furthermore, queues rarely exceed 512 entries and only two applications show queues in the range of 2,000 to 4,000 entries, with both UMQ and PRQ showing similar lengths.

Although we have studied CPU-centric applications we believe these insights are also valuable for GPU-centric communication models. Use cases include porting of existing applications to heterogeneous systems with compiler-assisted mechanisms, preserving the application's behavior and exposing comparable communication patterns to GPUs. Furthermore, communication is part of the algorithmic nature of the problem and should depend only to a limited extend on the architecture the application runs on.

B. User implications through proposed relaxations

While prohibiting wildcards seems applicable in most cases, requiring all messages to be expected needs far more changes in the applications' codes. Additional synchronization becomes necessary, which has to be ensured by the user. However, this is already a widely implemented optimization and well understood by many HPC programmers. LULESH, for example, already posts the vast majority of receive requests in advance.

Giving up ordering guarantees seems to have a great impact since applications needed to be restructured and rewritten, again mandating additional synchronization. However, tags can be used to restore ordering at the user level.

MPI was designed for CPUs to cover a broad range of applications and many features aim to ease programming. However, scientific applications on GPUs are generally well structured and strictly follow the BSP model, so that we believe these features can be spared in the GPU domain. Thus, we consider these relaxations to be feasible.

C. Architectural challenges

The fully MPI-compliant algorithm offers only a limited amount of parallelism and performance is low due to the GPU's low single thread performance. Another issue is the lack of a sufficient number of available warps to hide long instruction latencies. Newer GPU generations show better performance, but only due to higher clock frequencies. We endorse new architectural features like variable warp sizes [20], which helps with the matching of shorter queues, and better dynamic parallelism [21], which allows for adjusting kernel parameters to queue sizes. Furthermore, we hope for more control over scheduling and dynamic memory management within GPU kernels to support further parts of the messaging stack.

VIII. CONCLUSION

In this paper we analyzed exascale proxy applications to understand their communication behavior. We found that send/recv communication is usually limited to a small number of ranks and that message matching queues contain less than 512 entries for most applications. We presented a matching algorithm for GPUs that complies with MPI's constraints. However, the achieved matching rate of roughly 6M matches/s on Pascal-class GPUs seems incompatible with future requirements. We experimentally relaxed MPI semantics to extract more parallelization out of the matching task and were able to report speedups of 10x when prohibiting wildcards, and 80x by allowing out-of-order message delivery. In particular, prohibiting wildcards seems feasible since the vast majority of applications do not use them.

As accelerators are becoming peer processors, it seems inevitable that they will independently source and sink traffic. We observe that even though such accelerators have been available for a long time, there is still little movement towards more suitable and specialized communication models. The insights from our experiments regarding GPU-based matching and relaxations should lead to a better understanding of the performance implications when performing message passing on GPUs. However, many aspects of such communication models are still left open, such as the question of whether send/recv, collectives, put/get, (partitioned) global address spaces (GAS), or some other paradigm is most suitable.

Acknowledgment

We thank the anonymous reviewers for their constructive and insightful comments. We also thank the Networking Research Group at NVIDIA for supportive discussions and feedback, especially Ted Jiang and Matthias Blumrich.

References

[1] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE Micro*, 2016.

[2] NVIDIA, "NVIDIA Tesla P100," Whitepaper, 2016.

[3] M. Flajslik, J. Dinan, and K. D. Underwood, "Mitigating MPI Message Matching Misery," in *Proceedings of the International Conference on High Performance Computing (ISC)*, Frankfurt, Germany, 2016.

[4] A. Ortiz, J. Ortega, A. F. Diaz, and A. Prieto, "Comparison of Onloading and Offloading Strategies to Improve Network Interfaces," in *Proceedings of the Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Washington, DC, 2008.

[5] M. G. F. Dosanjh, R. E. Grant, P. G. Bridges, and R. Brightwell, "Re-evaluating Network Onload vs. Offload for the Many-Core Era," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, Chicago, IL, 2015.

[6] U. DOE, *Characterization of the DOE Mini-apps*, Retrieved July 14, 2016 from https://portal.nersc.gov/project/CAL/doe-miniapps.htm.

[7] B. Klenk and H. Fröning, "An Overview of MPI Characteristics of Exascale Proxy Applications," *To appear in the proceedings of the International Conference on High Performance Computing (ISC)*, Frankfurt, Germany, 2017.

[8] R. Brightwell and K. D. Underwood, "An analysis of NIC resource usage for offloading MPI," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, NM, 2004.

[9] S. Goudy, "A Preliminary Analysis of the MPI Queue Characteristics of Several Applications," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, Washington, DC, 2005.

[10] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell, "A Hardware Acceleration Unit for MPI Queue Processing," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Denver, CO, 2005.

[11] R. Keller and R. L. Graham, "Characteristics of the Unexpected Message Queue of MPI Applications," in *Proceedings of the European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface (EuroMPI)*, Berlin, Germany, 2010.

[12] J. A. Zounmevo and A. Afsahi, "An Efficient MPI Message Queue Mechanism for Large-scale Jobs," in *Proceedings of the IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, Singapore, 2012.

[13] J. A. Stuart and J. D. Owens, "Message passing on data-parallel architectures," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rome, Italy, 2009.

[14] L. Oden and H. Fröning, "GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, Indianapolis, IN, 2013.

[15] B. Klenk, L. Oden, and H. Fröning, "Analyzing communication models for distributed thread-collaborative processors in terms of energy and time," in *Proceedings of the IEEE Symposium on Performance Analysis of Systems and Software (ISPASS)*, Philadelphia, PA, 2015.

[16] L. G. Valiant, "A Bridging Model for Parallel Computation," in *Communications of the ACM*, vol. 33, New York, NY, 1990.

[17] R. Jenkins, *Integer hashing*, Retrieved July 14, 2016 from http://burtleburtle.net/bob/hash/integer.html.

[18] H. Fröning, M. Nüssle, H. Litz, C. Leber, and U. Brüning, "On Achieving High Message Rates," in *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, 2013.

[19] S. Potluri, "TOC-centric Communication: A Case Study with NVSHMEM," in *Proceedings of the OpenSHMEM User Group Meeting*, 2014.

[20] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler, "A Variable Warp Size Architecture," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Portland, OR, 2015.

[21] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Laperm: Locality aware scheduler for dynamic parallelism on gpus," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016.