2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Performance models for CPU-GPU data transfers

B. van Werkhoven
Dept. Computer Science
VU University Amsterdam
Amsterdam, The Netherlands
ben@cs.vu.nl

J. Maassen, F.J. Seinstra
Netherlands eScience Center
Amsterdam, The Netherlands
j.maassen@esciencecenter.nl,
f.seinstra@esciencecenter.nl

H.E. Bal
Dept. Computer Science
VU University Amsterdam
Amsterdam, the Netherlands
bal@cs.vu.nl

***Abstract*—Many GPU applications perform data transfers to and from GPU memory at regular intervals. For example because the data does not fit into GPU memory or because of inter-node communication at the end of each time step. Overlapping GPU computation with CPU-GPU communication can reduce the costs of moving data. Several different techniques exist for transferring data to and from GPU memory and for overlapping those transfers with GPU computation. It is currently not known when to apply which method. Implementing and benchmarking each method is often a large programming effort and not feasible. To solve these issues and to provide insight in the performance of GPU applications, we propose an analytical performance model that includes PCIe transfers and overlapping computation and communication. Our evaluation shows that the performance models are capable of correctly classifying the relative performance of the different implementations.**

***Index Terms*—Performance analysis; GPU Computing**

I. Introduction

Many studies of GPU computing applications focus on individual kernel performance and ignore the fact that many applications have to regularly transfer data over the PCIe bus [1]. In particular large-scale supercomputing applications suffer from this issue, either because the data does not fit into the GPU memory, inter-node communication at the end of each time step requires data to be present at the host, or parts of the application are executed on the CPU. Moreover, as supercomputing systems are large investments even a 10% speedup can often save millions of dollars [2].

PCIe transfers can have a large impact on performance, especially when considering that the bandwidth is much lower than the GPU device memory bandwidth. Gregg and Hazelwood [1] state that GPU kernel execution times can increase to between 2 and 50 times of the original, when PCIe transfer times are included. The de facto way of transferring data to the GPU is by delaying all GPU computation until the entire transfer is complete. An important reason for this is that overlapping computation and communication is challenging and requires a considerabe effort from the programmer and results in much more code [3], [4].

Transferring data to or from the GPU proceeds through explicit memory copy statements or using device-mapped host memory. Overlap between computation and communication can be achieved using either CUDA streams or device-mapped host memory. As we will show in this paper, the most efficient implementations may even require a mix of these different approaches. The different implementations require completely different host codes as well as modifications to the GPU kernel. For example, using mapped memory barely requires any host code, whereas using CUDA streams may require multiple loops of memory copy operations and kernel invocations with advancing offsets. Currently, if the application should run as efficiently as possible the programmer has little choice but to implement all alternatives and run benchmarks to see which method performs best. Implementing all different options is often a large programming effort and not feasible. Knowing which method to apply in advance can save the application developers a lot of time.

In order to solve these issues and provide insight in the performance of GPU applications that require regular transfers across the PCIe bus, we propose an analytical performance model that includes for PCIe transfers and overlapping computation and communication. The model is capable of classifying the alternative implementations with regards to their relative performance. While the parameterized model is specified analytically, it may be instantiated empirically to provide performance estimates for a target hardware platform.

The reason to develop an analytical performance model, as opposed to alternative methods such as simulation, is that analytical models tend to be easier to use and provide more high-level insight [2]. While cycle accurate or model-based simulators can produce extremely accurate projections, the vast amount of data that is generated may not lead to the desired insight within a time frame that is reasonable for application developers. Our goal is to create an analytical performance model with a reasonably small number of parameters. This ensures that the model is easy to use by application developers while performance is characterized correctly.

Using our performance model programmers should be able to quickly answer questions such as: What is the dominant factor for my programs' performance, kernel execution or PCIe transfers? How much can be gained from overlapping computation and communication? Do I have to use memory copy operations or mapped memory? If I use streams, what number of streams is likely give the best performance? How will switching to PCIe 3.0 platforms over PCIe 2.0 impact performance?

The model is presented in two stages. First, we model to what extent computation and communication can be overlapped. Second, we create a model to accurately predict

978-1-4799-2784-5/14 $31.00 © 2014 IEEE
DOI 10.1109/CCGrid.2014.16

PCIe transfer times and combine the two to create a set of performance estimations for different methods of overlapping computation and communication. We show that our performance models can be used to correctly classify the best performing implementation strategy for a small set of kernels and GPUs.

In this paper, we use CUDA terminology [5], although our method can just as easily be applied in OpenCL [6]. The CPU is referred to as the *host* and the GPU accellerator as the *device*, which are connected through PCI Express. We use the term *communication* in this paper to refer to data transfers between host and device.

The rest of this paper is structured as follows. Section II discusses related work. Section III explains what methods for overlapping computation and communication are available and provides upper bounds to indicate performance gain Section IV presents our models for kernels using CUDA streams. Section V shows how PCIe transfers can be modeled accurately. Section VI combines the models from Sections IV and V, presents our model for mapped memory kernels, and provides estimates for the number of streams to use as part of the classification. Section VII evaluates the presented performance models, leading to the conclusions in Section VIII.

II. Related Work

This section discusses related work on performance modeling for GPU applications grouped into three categories, models for kernel execution time, models for GPU cluster applications, and models that consider overlap between computation and communication.

A. Models for GPU kernel execution time

Various performance models for modeling GPU kernel execution times exist in the literature. For example, Zhang and Owens [7] have developed a semi-empirical model to analyze and predict kernel execution times. Hong and Kim [8] and Baghsorkhi et al. [9] presented detailed analytical models to predict the performance of GPU kernels. These models are intended for use in compilers to guide kernel optimization. They require low-level information about the kernel, such as the amount of warp- and instruction level parallelism, which makes them difficult to be used directly by the programmer. Therefore, in this paper we assume the execution time of the kernel is known by the programmer, either by experimentation or by using any of the existing performance models. Modeling the kernel execution time is outside the scope of this paper.

The roofline model [10] is an insightful high-level model that provides an upper bound on performance of parallel software on multi-core architectures. The model abstracts the machine into two performance metrics peak memory bandwidth and peak compute performance. Application kernels are abstracted into a single value (*operational intensity*) that describes the number of floating-point operations per byte of DRAM traffic. The upper bound on the performance of the kernel is then given as the minimum of the peak compute performance and the product of the peak memory bandwidth and operational intensity of the kernel.

For GPU applications, the model can be used assuming that all data fits in the DRAM of the device. In the absense of caches or data reuse, *arithmetic intensity* can used instead of *operational intensity*. In Section III, we explain how we extend the roofline model to also capture the impact of transferring data between host and device.

B. Models for GPU cluster applications

Another class of performance models considers not only the kernel execution time but also the performance in a GPU cluster setting. For example, Schaa en Kaeli [11] describe a model for prediciting the performance of GPU applications for multi-GPU systems. PCIe transfer time is modeled using a constant bandwidth as the only parameter. Overlapping computation and communication is not part of the model.

cudaMPI [12] is a system that provides MPI-like message passing to communicate data stored on device memory in a GPU cluster setting. Lawlor [12] argues against the use of streams and suggests to use mapped memory instead. Bernaschi et al. [13] compare the performance of several other GPU-aware MPI implementations (OpenMPI, MVAPICH2 and APEnet) using the Heisenberg spin glass model as a benchmark application. They argue that the use of CUDA streams is instrumental to achieving the best performance.

Aspen [14] is a domain specific language for developing analytical performance models. Aspen consists of two parts, one that models the application and one that describes the target machine. For example, Aspen is used to develop a performance model for a 3D FFT application in a GPU cluster setting [14]. Aspen assumes a completely connected intra-node topology that operates at a fixed link bandwidth. As such, their abstract machine model currently has no language features for describing the PCIe bus and it is assumed that communication between host and device operates at a fixed data rate of 8 gigabytes per second.

These performance models for GPU cluster applications have to capture PCIe transfer times to some extent. However, it is not their goal to provide the application developer with insight in to what extent computation can be overlapped with communication and how such overlap can be achieved. Therefore, in this paper we develop an accurate model for predicting PCIe transfer times and combine that with models that capture overlap between computation and communication. Finally, our model may be included in performance models for GPU cluster applications that do consider overlap between computation and communication.

C. Models that consider computation and communication overlap

Hoefler et al. [2] have proposed a method for developing application performance models for supercomputing systems. Their method does not explicitly consider the use of GPUs, but it may well be applicable to GPU applications. Through

several different steps their method abstracts the application into three performance metrics, *required memory traffic*, *floating-point instructions*, and *fixed-point instructions*. Step A4 of their method tells the modeler to extract the duration of overlappable serial computation and communication for each kernel. Unfortunately, the paper does not detail how the modeler is to extract this information and step A4 is also omitted in the application example discussed in the paper. The performance models we present in this paper can be used by application developers to perform step A4 of Hoefler's method.

Meswani et al. [15] have developed a framework for predicting the performance of applications executing on accelerators. Using automatically extracted application signatures and a machine profile based on benchmarks they aim to predict the application runtime before the application is ported. PCIe transfer rates are modeled as a function of the size of the data to be transferred between host and device. The PCIe throughput rates reported in their paper are much lower than what we observe, even when considering they assume a hardware setup with two GPUs per node. Unfortunately, their paper does not detail how these numbers were obtained.

Meswani et al. define a performance metric called *data transfer ratio*, which denotes the fraction of the duration of the total data transfer that is to be added to the execution time on the accelerator. For a selected number of kernels they list data transfer ratios, indicating what fraction of the data transfer needs to be overlapped in order for the GPU kernel to outperform the CPU. They conclude that in general 70% of data transfer time between host and device needs to be overcome in order to benefit from using accelerators. This stresses the importance of effectively overlapping data transfer times and computation for HPC applications. However, the analysis of how the overlap between computation and data transfer should be achieved, as well as the methods for achieving such data transfer ratios are outside the scope of their paper.

Gómez-Luna et al. [4] have studied the performance of CUDA streams for the older GTX 280 and GTX 480 graphics cards. In this paper, we do not consider GPUs that predate Fermi cards. They model the execution time of a kernel using CUDA streams including transfers between host and device.

Gómez-Luna et al. model the performance of an application using CUDA Streams on a Fermi GPU using the stream creation time as the only drawback to the use of streams. To include the stream creation time in kernel execution time estimates may be reasonable for applications that consist of a single GPU kernel, such as benchmarks or CUDA SDK examples, but for real-world applications this is not correct. Streams are typically created at the start of the application and destroyed only when the application has finished. Once created, a stream may be used for thousands of kernel invocations. They estimate the stream creation time at 0.03 ms per stream [4], clearly this becomes negligable if the stream can be reused for many kernel invocations. Stream creation times are not a limiting factor in the performance of streamed CUDA applications in general.

When the stream creation time is removed from their performance estimates, no drawback from using additional streams is left in the model. As such, the model suggests that an infinite number of streams results in optimal performance, which is not realistic. We may interpret their model as a *semi-empirical model* that abstracts all per stream overhead into some constant amount of time per stream. This model however does not provide much insight in the working of streams other than that there is some overhead associated with their use. In the following we explain how we model the execution time of streamed CUDA kernels in a way that provides the necessary insight for the application programmer.

III. Overlapping computation and communication

This section explains what methods for overlapping computation and communication are available to application developers and provides upper bounds to indicate how much performance can be gained.

Overlapping computation with communication requires fine-grained control over how data is transferred to and from the GPU. There are several alternative techniques for moving data between host and device in the CUDA Programming model. The most commonly used approach is to simply use *explicit memory copy statements* to transfer large blocks of memory to the GPU, invoke GPU kernels, and copy the results back from the GPU. These explicit memory copy operations can be invoked either synchronously or asynchronously with respect to the host. While the host does not wait for an asynchronous copy to complete, the copy operation will entirely preceed GPU kernel execution and as such, computation and communication are not overlapped.

The *mapped memory* approach uses no explicit copies, but maps part of the host memory into device memory space. Whether this approach is feasible depends on the memory access pattern of the kernel. Typically mapped memory can only be used efficiently if each input and output element is read or written only once by the kernel. This is because every load or store on device-mapped host memory may result in a transfer over PCIe. Although this approach results in very clean host code, requiring no explicit copy statements, it may require more complex kernel implementations with delicate memory access patterns to ensure high performance.

CUDA streams may be used to separate the computation into distinct streams that may execute in parallel. This way, communication from one stream can be overlapped with computation, and with communication in other streams. GPUs with 1 copy engine can only use the PCIe bus in half duplex when data is moved using explicit memory copies. As such, computation can only be overlapped with communication in at most one direction. The only way for these GPUs to use the PCIe bus in full duplex is to use device-mapped host memory instead. GPUs with 2 copy engines, such as Nvidia's Tesla K20, can use the PCIe bus in full duplex using explicit memory copies in different streams. This way, computation and communication in both directions can be fully overlapped using different streams. Streams may also be used to allow

different kernels to execute concurrently, regardless of how data is transferred between host and device. This task-parallel use of streams for exploiting application-level task parallelism is outside the scope of this work.

A. Bounding the performance of kernels with overlap

We can apply the roofline model, outside of its intended scope, for modeling the performance of GPU kernels including PCIe transfer times. For this, we need a metric different from *operational intensity* to filter out all redundant loads and stores to device memory. This is similar to how the roofline model uses operational intensity rather than arithmetic intensity to measure the traffic between cache and DRAM rather than between processor and cache. To model this correctly we need the ratio between the total number of floating point operations performed by the kernel and the amount of bytes transferred between host and device memory. We call this *data intensity*, defined as data intensity = total flops / total bytes transferred.

Using this extended roofline model we can determine whether the execution time of a kernel will be dominated by the PCIe transfers or kernel computation.

$$\begin{aligned}\text{full overlap} = min(&\text{peak compute performance},\\ &\text{peak memory bandwidth} \times \text{operational intensity},\\ &\text{PCIe bandwidth} \times \text{data intensity})\end{aligned}$$

This number gives an upper bound on the kernel performance because it assumes a maximal overlap between computation and communication. However, if the PCIe transfers entirely precede and follow GPU kernel execution then the performance is limited as follows:

$$\text{zero overlap} = \frac{\text{total flops}}{(\frac{\text{bytes transferred}}{\text{PCIe bandwidth}}) + (\frac{\text{total flops}}{RM})},$$

where RM is the upper bound provided by the roofline model.

The two estimates that we have just defined can be used to bound the performance for full or zero overlap between computation and communication. As such, it can be used to indicate how much performance can be gained from overlapping computation and communication. However, this model does not yet express the performance gain that *some* extent of overlap between computation and communication may have, nor does it give any insight into how such overlap can be achieved.

However, the device-mapped host memory approach can achieve an overlap between computation and communication at the highest possible granularity (i.e. warp-level). The mapped memory approach entirely forgoes the use of device memory and as such we cannot use data intensity as a measure for the amount of data that will be transferred. In fact, every request that would normally be served by the device memory will result in a transfer over PCIe. As such, we can directly use operational intensity combined with the PCIe bandwidth to provide an upper bound on performance.

$$\begin{aligned}\text{mapped memory} = min(&\text{peak compute performance},\\ &\text{PCIe bandwidth} \times \text{operational intensity})\end{aligned}$$

The performance bounds introduced in this section can be used to determine performance bottlenecks, but they cannot be used to model the performance that a certain degree of overlap may provide nor do they provide insights in how overlap may be best achieved. Therefore, the following sections introduce new performance models that are specifically created to model the extent of overlap between computation and communication.

IV. Modeling computation and communication overlap for CUDA Streams

In order to model what extent of overlap can be achieved when streams are used, we start using the same notation as Gómez-Luna et al. [4]. Transfer times t_{Thd} (host to device) and t_{Tdh} (device to host) represent the sum of time spent on transfers for all streams combined. The total kernel execution time for all streams combined is written as t_E. The per stream time is written as $\frac{t}{nStreams}$. As some overhead is associated with the use of streams, t_{Thd} and t_{Tdh} will generally be larger than the observed time when no streams are used. As such, the estimates presented in this section are too optimistic and are refined at a later stage.

Based on what GPUs are currently available we identify three categories: (1) Devices with *implicit synchronization* (explained below) and 1 copy engine, examples are GPUs from the Fermi architecture e.g. GTX 480 and Tesla C2050, but also the Kepler GTX 680. (2) Devices with no implicit synchronization and 1 copy engine, for example the GTX Titan. And (3) devices with no implicit synchronization and 2 copy engines, for example the Tesla K20. Figures 1, 2, and 3 show the possible overlap between computation and communication for each category using 1, 2 and 4 streams. The behavior of the kernel is separated into two scenarios, one where kernel execution time is the dominant factor in performance and one where the PCIe transfers dominate performance. The blue bar on the left represents the time spent on data transfers from host to device, the green bar represents kernel execution time on the GPU, followed by another blue bar representing the time spent on transferring data from device to host. From these diagrams we can easily derive simple analytical models for the performance of a kernel that uses CUDA streams.

Implicit synchronization and 1 copy engine. Implicit synchronization delays operations that require a dependency check to see if a streamed kernel launch is complete until all thread blocks of all prior kernel launches from any stream have started executing [5], see Figure 1. In the dominant kernel scenario computation can overlap with transfers from host to device in different streams. In the dominant transfers scenario the transfers from host to device can overlap with computation in different streams. Because of implicit synchronization, practically no overlap between computation and transfers from device to host can be achieved. The extent of overlap between host to device transfers and computation can be modeled as follows.

Dominant kernel $time = \frac{t_{Thd}}{nStreams} + t_E + t_{Tdh}$

Dominant transfers $time = t_{Thd} + \frac{t_E}{nStreams} + t_{Tdh}$

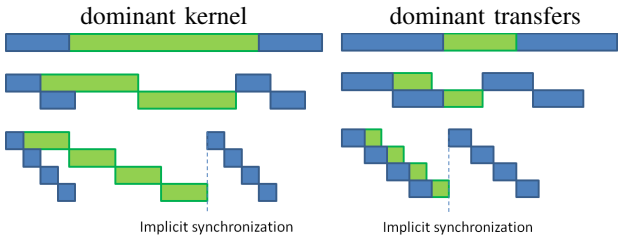

Fig. 1. Diagrams showing possible overlap for devices with implicit synchronization and 1 copy engine when using 1, 2, or 4 streams.

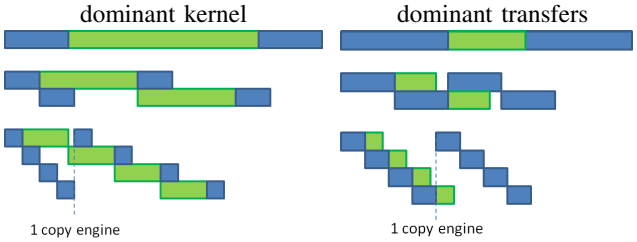

Fig. 2. Diagrams showing possible overlap for devices with no implicit synchronization and 1 copy engine when using 1, 2, or 4 streams.

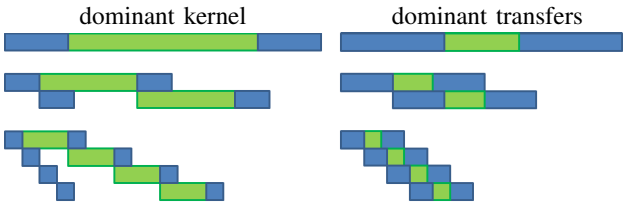

Fig. 3. Diagrams showing possible overlap for devices with no implicit synchronization and 2 copy engines when using 1, 2, or 4 streams.

For devices with implicit synchronization and 1 copy engine, we come to the same conclusions as Gómez-Luna et al. did for Fermi GPUs, although we derive the model from the hardware specifications rather than extensive experimentation.

No implicit synchronization and 1 copy engine. On devices like the GTX Titan, computation can overlap with transfers in either direction in different streams, see Figure 2. However, because there is only 1 copy engine, transfers in the opposite direction can only start once all transfers in the first direction have completed. This means the kernel execution time can be hidden entirely in the dominant transfers scenario, depending on whether the host to device and device to host transfer times are large enough to hide all computation time.

Dominant kernel $time = \frac{t_{Thd}}{nStreams} + t_E + \frac{t_{Tdh}}{nStreams}$

Dominant transfers $time = max(t_{Thd} + t_{Tdh}, t_{Thd} + \frac{t_E}{nStreams} + \frac{t_{Tdh}}{nStreams}, \frac{t_{Thd}}{nStreams} + \frac{t_E}{nStreams} + t_{Tdh})$

No implicit synchronization and 2 copy engines. With 2 copy engines, transfers in one direction can overlap with transfers in the opposite direction in different streams, see Figure 3. This means that all three components can be successfully overlapped with each other. The resulting execution time can be modeled as the largest component plus the per stream time spend on the other two components. Therefore, we provide a single estimate that captures both the dominant kernel and dominant transfers scenarios.

$time = max(t_{Thd} + \frac{t_E}{nStreams} + \frac{t_{Tdh}}{nStreams}, \frac{t_{Thd}}{nStreams} + t_E + \frac{t_{Tdh}}{nStreams}, \frac{t_{Thd}}{nStreams} + \frac{t_E}{nStreams} + t_{Tdh})$

Comparing Figures 2 and 3 we can also conclude that the 2nd copy engine only improves performance if $\frac{t_E}{nStreams} + \frac{t_{Thd}}{nStreams} > t_{Thd}$.

V. Modeling PCIe transfer time

Communication between host and device memory proceeds over PCI Express. PCI Express is a packet-based switched point-to-point interconnect [16]. As such, we decided to use the LogP [17], LogGP [18], and parameterized LogP [19] class of performance models as a base for modeling the performance of transfers over PCI Express. Note that we only consider asynchronous transfers over DMA, this corresponds to either using *cudaMemcpyAsync* on *pinned memory* in a non-default stream or using *device-mapped host memory*.

LogP [17] abstracts the communication of fixed-sized short messages through four parameters: communication latency (L), overhead (o), gap (g), and the number of processors (P). *Latency* is the amount of time it takes each byte to travel from endpoint to endpoint in the network. The *overhead* is defined as the length of time that a processor in engaged in transmission or reception. The *gap* is the reciprocal of the available per byte bandwidth for short messages per processor. LogGP [18] extends the LogP model, by introducing a separate bandwidth for large messages (G). The bandwidth for short messages (g) is kept as a parameter in order to model the startup bottleneck of the network. We take LogGP as a starting point for our model, as PCIe transfers between host and device typically consist of large messages sent over DMA.

To model the performance of a memory transfer between host and device we adapt the LogGP model in the following way.

Under LogGP transferring a single large message of k bytes takes $L + o + (k-1)G + o$ time. For modeling a single data transfer over PCIe we use: $L + o + kG$. The second o in LogGP is for the time spent by the receiving processor. In our case, the transfer proceeds over DMA and data is written directly into the memory at the receiving end, therefore there is no receiving processor overhead. The sending overhead o that is left, can be seen as the time the processor is involved in registering the DMA request with the controller. We replace $(k-1)$ with k, which is only part of the LogGP model to reduce to LogP for $k = 1$. Because all transfers happen over DMA, sending the first byte is not considered to be part of the sending processor overhead. Not including the -1 byte seems more natural.

Sending multiple large messages under LogGP is modeled as $L + o + (k_1 - 1)G + g + (k_2 - 1)G$, where g captures the startup bottleneck of the network. Just as in the LogP and LogGP models we assume a single port model, i.e., only a single transmission can be active at any given time. Therefore, if multiple streams are used to transfer a large message. We assume the per stream transfers happen one after the other. As such, we model the transmission time of large messages

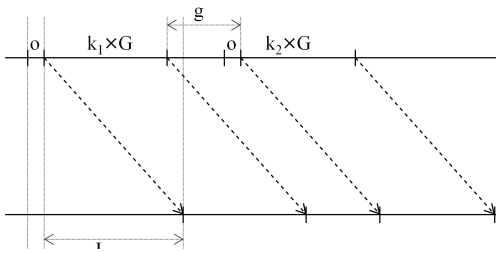

Fig. 4. Transferring data between host and device and vice versa.

Host to Device		Device to host	
L+o	0.009420 ms	L+o	0.009023 ms
G_{hd}	8.318392E-008 ms/byte	G_{dh}	7.924734E-008 ms/byte
g_{hd}	0.002503 ms	g_{dh}	0.002674 ms

TABLE I
OVERVIEW OF MODEL PARAMETERS FOR A PCIE 3.0 SYSTEM

sent using multiple streams in a way similar to LogGP. For example, we model 2 streams each transmitting half of k bytes is modeled as $L+o+\frac{1}{2}k\times G+g+\frac{1}{2}k\times G$, or more generally for $nStreams$ streams $L+o+(\frac{k}{nStreams}\times G\times nStreams)+g\times(nStreams-1)$, which equals to $L+o+k\times G+g\times(nStreams-1)$. Figure 4 presents an overview of how the parameters impact transfer times.

A. Model validation

We validate the model using an Nvidia GTX Titan GPU connected over PCIe 3.0 in a system with 2 Intel Xeon E5-2630 CPUs running at 2.3GHz using CUDA 5.5. The model parameters are obtained as follows. $L+o$ is approximated by measuring the transfer time of a single byte transfer. G is estimated measuring n transfers of size $k_1,\ldots,k_n$ bytes. The sum of transfer times $t_{k_1}+\cdots+t_{k_n}=n(L+o)+G(k_1+\cdots+k_n)$ gives $G=\frac{t_{k_1}+\cdots+t_{k_n}-n(L+o)}{k_1+\cdots+k_n}$.

We have written a small benchmark application that extracts the model parameters using the averages of a large number of runs. The model parameters for our test system are shown in Table I. Because the parameters for host to device transfers and device to host transfers differ, we will write G_{hd}, or G_{dh} respectively, to distinguish between the two. We observe that $g<L+o$, which means that the latency of consecutive transmissions can be overlapped.

We have tested the model accuracy for data sizes ranging from 16 MB to 1 GB and varying the number of streams from 1 to 256. For the host to device transfers our model always underestimates performance with a maximum error of 1.18%. For device to host transfers the maximum overestimation error is 2.47% and maximum underestimation is 0.65%. Figures 5 and 6 show the observed and estimated PCIe transfer times for transfers of 16 MB varying the number of streams. Because of space limitations, we only show graphs for size 16 MB, which produced the largest relative errors. While a more detailed model may be able to produce even more accurate performance estimations, we consider our model to be accurate enough for our intended purposes.

Some devices have 2 copy engines and therefore transfers in one direction can overlap with transfers in the opposite

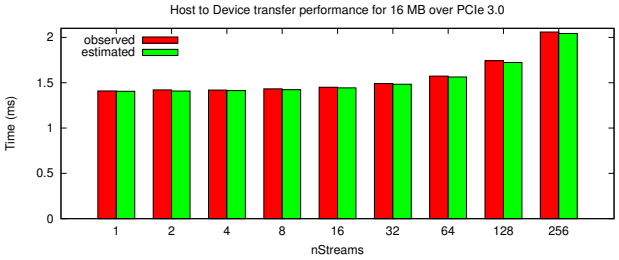

Fig. 5. Performance of transferring data from host to device.

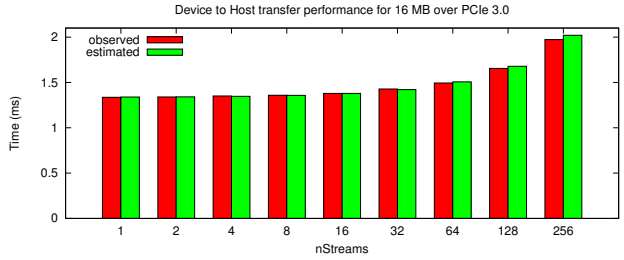

Fig. 6. Performance of transferring data from device to host.

	Host to Device		Device to host	
PCIe 2.0 explicit-explicit	G_{hd}	1.741965E-007	G_{dh}	1.747120E-007
PCIe 2.0 explicit-mapped	G_{hd}	2.491988E-007	G_{dh}	2.236644E-007
PCIe 3.0 explicit-mapped	G_{hd}	1.193386E-007	G_{dh}	1.480396E-007

TABLE II
MODEL PARAMETERS FOR BIDIRECTIONAL TRANSFERS IN MS/BYTE.

direction. The performance of these transfers cannot be expected to be exactly the same as the performance of non-overlapped transfers. While the communication channel can function in full duplex, some resources, including the end points of both transfers, are still shared. As such, we can expect that the transfer time increases while a concurrent transfer in the opposite direction occurs. Similarly, we can expect that the performance of a single large transfer in one direction will degrade when many smaller transfers in the opposite direction occur due to accesses to device-mapped host memory. Therefore, we also provide the G_{hd} and G_{dh} for bidirectional transfers using streams on the Tesla K20 (PCIe 2.0) and for transfers colliding with mapped memory transfers for both the K20 and the GTX Titan, see Table II.

VI. OVERVIEW OF PERFORMANCE MODELS

This section combines the model for PCIe transfers with the models for computation and communication overlap using streams from Sections IV. In addition, we present our model for mapped memory kernels and provide estimates for the what number of streams to use as part of the classification.

Figure 7 shows an overview of all performance estimates we consider in this paper. For streamed applications these are the estimates presented in Section IV with the PCIe transfer model from Section V filled in. Note that the kernel execution time for all streams combined t_E, as well as the per stream kernel execution time $\frac{t_E}{nStreams}$ have to be estimated using any of the existing performance models or be derived empirically. The individual models for streamed kernels have already been discussed in Section IV. Therefore, the rest of this section

Implicit synchronization and 1 copy engine.
Dominant kernel

$$time = L + o + \frac{B_{hd}}{nStreams} \times G_{hd} + t_E + L + o + B_{dh} \times G_{dh} + g \times (nStreams - 1) \quad (1)$$

Dominant transfers

$$time = L + o + B_{hd} \times G_{hd} + g \times (nStreams - 1) + \frac{t_E}{nStreams} + L + o + B_{dh} \times G_{dh} + g \times (nStreams - 1) \quad (2)$$

No implicit synchronization and 1 copy engine.
Dominant kernel

$$time = L + o + \frac{B_{hd}}{nStreams} \times G_{hd} + t_E + L + o + \frac{B_{dh}}{nStreams} \times G_{dh} \quad (3)$$

Dominant transfers

$$\begin{aligned} time = max(&L + o + B_{hd} \times G_{hd} + g \times (nStreams - 1) + L + o + B_{dh} \times G_{dh} + g \times (nStreams - 1), \\ &L + o + B_{hd} \times G_{hd} + g \times (nStreams - 1) + \frac{t_E}{nStreams} + L + o + \frac{B_{dh}}{nStreams} \times G_{dh}, \\ &L + o + \frac{B_{hd}}{nStreams} \times G_{hd} + \frac{t_E}{nStreams} + L + o + B_{dh} \times G_{dh} + g \times (nStreams - 1)) \end{aligned} \quad (4)$$

No implicit synchronization and 2 copy engines.

$$\begin{aligned} time = max(&L + o + B_{hd} \times G_{hd} + g \times (nStreams - 1) + \frac{t_E}{nStreams} + L + o + \frac{B_{dh}}{nStreams} \times G_{dh}, \\ &L + o + \frac{B_{hd}}{nStreams} \times G_{hd} + t_E + L + o + \frac{B_{dh}}{nStreams} \times G_{dh}, \\ &L + o + \frac{B_{hd}}{nStreams} \times G_{hd} + \frac{t_E}{nStreams} + L + o + B_{dh} \times G_{dh} + g \times (nStreams - 1)) \end{aligned} \quad (5)$$

Device-mapped host memory.

$$time = max(L + o + B_{hd} \times G_{hd} + L + o, L + o + t_E + L + o, L + o + L + o + B_{dh} \times G_{dh}) \quad (6)$$

Fig. 7. Overview of performance models. B_{hd} is the number of bytes transferred from host to device. B_{dh} is the number of bytes transferred from device to host.

discusses our model for a kernel using mapped memory as well as estimates for a suitable number of streams.

The model for kernels using mapped memory (Equation 6) reflects that computation as well as communication in either direction can overlap with each other. As such, the performance can be estimated using only the longest of the three components (host to device transfers, computation, or device to host transfers). For example, when the time spent on transferring data from host to device is the longest component, the computations and device to host transfers can occur while additional data is still being transferred from host to device. Therefore, the total time of the kernel can be modeled as the time of the longest component, with the only exception being the latency of the overlapped components. Because the performance of mapped memory kernels is not subject to implicit synchronization or the number of copy engines, no separate models are necessary. It is important to note that when applying the mapped memory model B_{hd} and B_{dh} become larger if elements in device-mapped host memory are read or written multiple times, as multiple accesses may translate to multiple transfers over PCIe.

All estimates for streamed kernels shown in Figure 7 show the possible drawbacks from using additional streams. As such we can also reason how long performance could be improved from increasing the number of streams. The key idea of using more streams is to maximize overlap. However, at a certain point we expect performance to degrade because the cost of using an additional stream is not outweighed by the performance gain of increased overlap.

For devices with implicit synchronization we can take the first derivative of Equations 1 and 2 to find $nStreams_{max} = \sqrt{\frac{B_{hd} \times G_{hd}}{g}}$ for the dominant kernel case and $nStreams_{max} = \sqrt{\frac{t_E}{2g}}$ for the dominant transfer case.

From Equation 4 it would seem from the model that performance can only be increased by decreasing the number of streams. However, $nStreams$ can only be decreased until $\frac{t_E}{nStreams} + \frac{B_{dh}}{nStreams} \times G_{dh}$ or $\frac{B_{hd}}{nStreams} \times G_{hd} + \frac{t_E}{nStreams}$ become larger than $L + o + \frac{B_{dh}}{nStreams} \times G_{dh} + g \times (nStreams - 1)$. As such, the smallest number of streams that still improves performance can be estimated by solving either of the following equations:
if $(B_{hd} > B_{dh})$
$B_{dh} \times G_{dh} + g \times (nStreams - 1) = \frac{t_E}{nStreams} + \frac{B_{dh}}{nStreams} \times G_{dh}$
if $(B_{dh} > B_{hd})$
$B_{hd} \times G_{hd} + g \times (nStreams - 1) = \frac{B_{hd}}{nStreams} \times G_{hd} + \frac{t_E}{nStreams}$

The largest number of streams for devices with no implicit synchronization and 2 copy engines can also be derived from the model in Figure 7. For the dominant transfer scenario we can use the first derivative to find that the optimal number of streams is given by:
if $(B_{hd} > B_{dh})$ $nStreams_{max} = \sqrt{\frac{B_{dh} \times G_{dh} + t_E}{g}}$, or

GPU	CEs	IS	CPUs	Interconnect
GTX 680	1	✓	2 Intel Xeon E562 @ 2.4GHz	PCIe 2.0
GTX Titan	1	✗	2 Intel Xeon E5-2630 @ 2.3GHz	PCIe 3.0
Tesla K20m	2	✗	2 Intel Xeon E5-2620 @ 2.0GHz	PCIe 2.0

TABLE III
OVERVIEW OF HARDWARE USED IN EXPERIMENTS. CES=COPY ENGINES. IS=IMPLICIT SYNCHRONIZATION.

if $(B_{dh} > B_{hd})$ $nStreams_{max} = \sqrt{\frac{B_{hd} \times G_{hd} + t_E}{g}}$.
For the dominant kernel scenario it depends on how t_E increases as the number of streams increases. Modeling kernel execution time is outside the scope of this paper.

For many applications, it is practical to limit the number of streams to one of the input dimensions. However, as part of the classification process one has to choose some number of streams to calculate performance predictions. The estimates presented here may be used to ensure that a realisitic number of streams is used. Unfortunately, we cannot use these estimates to accurately predict the optimal number of streams for a real application, because the kernel execution time is also influenced by the number of streams. However, these estimations may be used to limit the search space for (auto-)tuning the number of streams, once the implementation using streams has been created.

VII. EVALUATION

This section evaluates the ability of our performance models to accurately classify the performance of different kernel implementations for our target hardware platforms. We introduce several real-world examples of kernel implementations that overlap computation and communication to demonstrate how our performance models can be applied to model the execution times of each implementation. For this evaluation we use the DAS-4 distributed supercomputer. DAS-4 is a heterogeneous platform containing many different compute nodes and accelerators. We use three configurations detailed in Table III each using CUDA 5.5. These configurations correspond with the three different categories introduced in Section IV.

We apply our performance models to two different kernels from the Parallel Ocean Program [20]. The Parallel Ocean Program is a global ocean circulation model which solves the three-dimensional primitive equations for fluid motions under hydrostatic and Boussinesq approximations, in which depth is used as the vertical coordinate. We focus on two kernels state() and buoydiff() that are part of the computation of the vertical mixing coefficients [21]. For each function we consider four different versions that we call *Explicit*, *Implicit*, *Streams*, and *Hybrid*. We first describe the four versions in general and then discuss the specific implementations for state() and buoydiff() in detail.

Explicit is a bulk synchronous implementation that uses explicit memory copy statements to copy all the required input data to GPU and from the GPU for the entire three-dimensional grid. The kernel used in *Explicit* creates a two-dimensional array of threads, i.e. one thread for each horizontal grid point, which iterate the grid points in the vertical dimension.

Implicit uses mapped memory and therefore requires no explicit memory copy statements. Instead, data is requested by the GPU directly from the host memory and sent over the PCIe bus. The performance of accessing the memory in this way is very sensitive to the order in which data is requested and care must be taken not to create gaps or misalignments from the mapping between threads and data. Therefore, *Implicit* uses a kernel implementation that creates a one-dimensional array of threads with size equal to the number of grid points in the three-dimensional grid. Each thread then computes its three-dimensional index from its one-dimensional thread ID to direct itself to the correct part of the computation.

The *Streams* implementation creates one stream for each vertical level and uses explicit copy statements to copy the corresponding vertical level of the input and output variables to and from the GPU. If the computation of one vertical level requires input from multiple vertical levels, CUDA events are used to delay the computation until all inputs have been moved to the device and vice versa. The kernel used in *Streams* is similar to the kernel used in *Explicit*, except for the fact that the kernel only computes the grid points of one vertical level. Note that, except for the differences described here, the kernels do not contain any architecture specific optimizations.

The *Hybrid* implementation is similar to the *Streams* implementation in that it uses explicit memory copies and streams to transfer the input data from host to device and for invoking kernels. However, the kernel uses device-mapped host memory to transfer the output from device to host. This may be somewhat counter-intuitive, but the main idea behind this implementation is that input data is often requested multiple times by different threads, therefore using mapped memory for host to device communication is unlikely to give the best performance. Since each output element is written only once, using mapped memory for device to host communication is a good fit. This approach allows overlap between the device to host transfers and host to device transfers in other streams even on devices with only one copy engine and/or implicit synchronization. However, there is no benefit for this method on devices with no implicit synchronization and 2 copy engines. The performance of this implementation for all devices is modeled as the performance of a streamed kernel for devices with 2 copy engines and no implicit synchronization (see Figure 7.5).

The state() kernel computes the water density for each vertical level k based on two variables, temperature and salinity. State() optionally also computes the derivates of the density with respect to temperature and salinity, which totals to three output variables. Our *Explicit* implementation uses explicit copies to move the three-dimensional grid of variables between host and device and creates one thread for each horizontal grid point, which computes all outputs for each level in the vertical direction. This approach is unable to overlap communication to and from the device with GPU computation. However, it is possible to also parallelize the computation of different vertical levels using CUDA streams. Our *Streams* implementation ensures that GPU computation can be overlapped with GPU

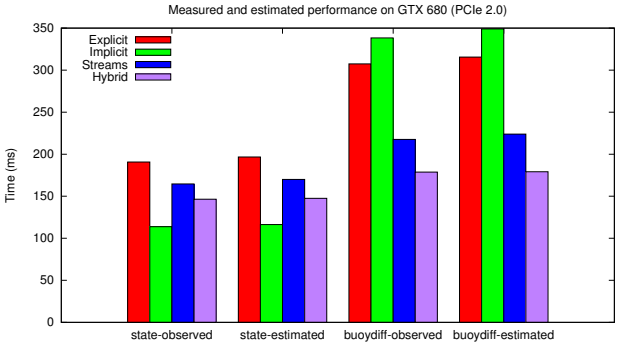

Fig. 8. Performance of state and buoydiff kernels on GTX 680.

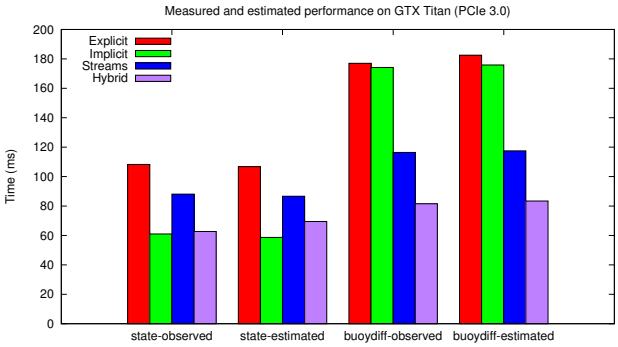

Fig. 9. Performance of state and buoydiff kernels on GTX Titan.

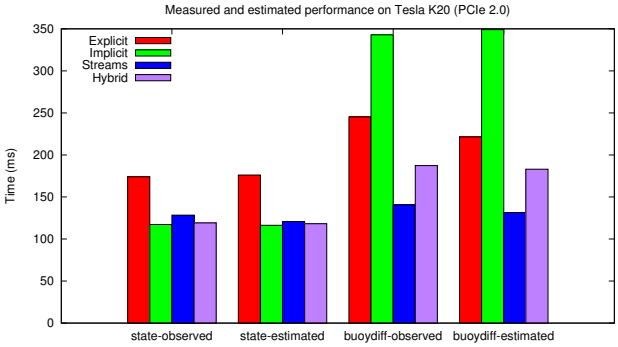

Fig. 10. Performance of state and buoydiff kernels on Tesla K20.

communication of different vertical levels. Because of the simple access pattern in state(), where each input and output element is read or written only once, it is also a good candidate for the highly parallel *Implicit* implementation. The *Hybrid* implementation uses CUDA streams and explicit memory copies to move data from host to device, but uses device-mapped host memory to transfer the output from device to host.

The computation of buoyancy differences at level k requires the density of both the surface level and level $k-1$ displaced to level k, as well as the water density at level k. These values can be computed for each level in parallel as long as all the data is present on the GPU. Overlapping data movement from the host to the GPU with GPU computation and data movement from the GPU to host becomes significantly more difficult, because the tracers for levels 1, $k-1$, and k need to be present on the GPU to compute the buoyancy differences at level k. The *Streams* implementation first schedules memory copies to the GPU for all vertical levels in concurrent streams and then invokes GPU kernel launches for all levels. However, before the execution of the kernel in stream k can start, the memory copies in stream 1, $k-1$, and k need to be complete. The kernel executing in stream k outputs to different vertical levels for different variables. Therefore, some of the memory copies from device to host in stream k have to wait for the kernel in stream $k-1$ to complete. We use the CUDA event management functions to guarantee no computations or memory transfers start prematurely.

We will now evaluate whether our set of performance models is able to correctly classify the different implementations for performance. We have measured the performance of the state and buoydiff kernel on a $1024 \times 1024 \times 42$ domain for each of the hardware platforms in Table III. These observed performance results are compared against the performance predicted by our models in Figures 8, 9, and 10. 42 CUDA streams are used in both the *Streams* and *Hybrid* implementations, one stream is mapped to one vertical level in domain.

For the state kernel the *Implicit* implementation is the most efficient on each of the hardware platforms. This is expected as each input and output element is read or written only once and the *Implicit* implementation is capable of overlapping computation and communication in both directions to a very high extent. This is also what is reflected by our performance models.

As expected, the *Implicit* implementation is not the best performing for the buoydiff kernel. The input data is requested multiple times by different threads, which results in data being repeatedly transmitted. For the buoydiff kernel the most efficient implementation differs from platform to platform. On the GTX 680 and the GTX Titan the *Hybrid* implementations are the best performing. This is because the *Hybrid* implementation enables the kernel to utilize the PCIe bus in full duplex, where this is not possible with using only explicit memory copies. On the K20, which has 2 copy engines and no implicit synchronization, there is no benefit to using a *Hybrid* implementation and as such *Streams* is the best performing. These observations exactly match the classifications provided by our performance models.

We can also look at the performance predictions for the individual implementations. For the *Explicit* implementations of both kernels the performance model estimates with a maximal error of 9.73%, which occurs for the buoydiff kernel on the K20. The *Implicit* implementations are modeled relatively accurately with a maximum error 3.85%. For *Streams* the maximum error is 6.46%. Our *Hybrid* implementations prove to be the hardest to predict, the error goes up to 10.75% for the state kernel on the GTX Titan. While there is some error in the performance estimations produced by the models, they do provide the insight necessary for selecting the best performing method for each kernel on each device.

VIII. Conclusions

PCIe transfers can have a large impact on performance. In particular when the entire data set does not fit into the GPU memory, inter-node communication at the end of each time step requires data to be present at the host, or parts of the application are executed on the CPU. Overlapping computation and communication can be challenging and requires a considerable effort from the programmer and results in much more code. Several different methods for transferring data and overlapping computation and communication are available, but little is known on when to apply which method. The different implementations require completely different host codes as well as modifications to the GPU kernel. Implementing all alternatives is often a large programming effort and not feasible.

In order to solve these issues and provide insight in the performance of GPU applications that require regular transfers across the PCIe bus, we propose an analytical performance modeling approach that includes for PCIe transfers and overlapping computation and communication.

Our evaluation shows that our performance model can be used to correctly select the best performing implementation strategy for two real-world kernels on three different GPUs. The largest error is currently in the estimations for the *Hybrid* implementation. The host to device transfer performance degrades when a large number of mapped memory transfers in the opposite direction occur simultaneously. We observe this on both PCIe 2.0 and 3.0 systems, which is currently captured by the model by using a lower effective bandwidth for the duration of the overlapped transfer. We believe it should be possible to further extend the model to capture this more accurately.

The current set of parameters for the models presented in this paper is a compromise between capturing the main performance characteristics and providing a reasonable framework for algorithm design and analysis. While no small set of parameters can describe all machine aspects completely, analysis becomes more difficult with a large set of parameters.

We believe the performance models presented in this paper open several avenues for future work. First of all, we would like to evaluate the use of the model for more and more different CUDA kernels. The number of parameters may be extended to capture more machine characteristics, for example full-duplex use of the PCIe bus by explicit copies and device-mapped host memory. Such a more detailed model may be used by automatically optimizing compilers and software development tools that automatically generate the host code for CUDA programs. Additionally, the performance models presented in this paper may be extended with a performance model for kernel execution time. Such a unified model can be used to provide a complete overview of the behavior of kernels that overlap computation and communication.

Acknowledgments

This publication was supported by the Dutch national program COMMIT. We would like to thank Prof. H.A. Dijkstra for our close collaborations in the NLeSC eSALSA project.

References

[1] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 134–144.

[2] T. Hoefler, W. Gropp, M. Snir, and W. Kramer, "Performance Modeling for Systematic Performance Tuning," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11), SotP Session*, Nov. 2011.

[3] J. White and J. J. Dongarra, "Overlapping computation and communication for advection on hybrid parallel computers," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 59–67.

[4] J. Gómez-Luna, J. M. GonzáLez-Linares, J. I. Benavides, and N. Guil, "Performance models for asynchronous data transfers on consumer graphics processing units," *Journal of Parallel and Distributed Computing*, vol. 72, no. 9, pp. 1117–1126, 2012.

[5] Nvidia, "CUDA Programming Guide, http://docs.nvidia.com/cuda/," 2013.

[6] Khronos Group, "OpenCL, www.khronos.org/opencl/," 2013.

[7] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for gpu architectures," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 382–393.

[8] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *The 36st International Symposium on Computer Architecture (ISCA), Austin, Texas, USA*. ACM/IEEE, 2009, pp. 1–12.

[9] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for gpu architectures," in *ACM Sigplan Notices*, vol. 45, no. 5. ACM, 2010, pp. 105–114.

[10] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[11] D. Schaa and D. Kaeli, "Exploring the multiple-gpu design space," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[12] O. S. Lawlor, "Message passing for gpgpu clusters: Cudampi," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–8.

[13] M. Bernaschi, M. Bisson, and D. Rossetti, "Benchmarking of communication techniques for gpus," *Journal of Parallel and Distributed Computing*, 2012.

[14] K. L. Spafford and J. S. Vetter, "Aspen: a domain specific language for performance modeling," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 2012, pp. 1–11.

[15] M. R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, and S. Poole, "Modeling and predicting performance of high performance computing applications on hardware accelerators," *International Journal of High Performance Computing Applications*, vol. 27, no. 2, pp. 89–108, 2013.

[16] PCI-SIG, "PCI Express Base Specification Revision 3.0," 2010.

[17] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken, "Logp: A practical model of parallel computation," *Communications of the ACM*, vol. 39, no. 11, pp. 78–85, 1996.

[18] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "Loggp: Incorporating long messages into the logp model for parallel computation," *Journal of parallel and distributed computing*, vol. 44, no. 1, pp. 71–79, 1997.

[19] T. Kielmann, H. E. Bal, S. Gorlatch, K. Verstoep, and R. F. Hofman, "Network performance-aware collective communication for clustered wide-area systems," *Parallel Computing*, vol. 27, no. 11, pp. 1431–1456, 2001.

[20] Los Alamos National Laboratory, "Parallel Ocean Program4, http://climate.lanl.gov/Models/POP/," 2013.

[21] W. G. Large, J. C. McWilliams, and S. C. Doney, "Oceanic vertical mixing: A review and a model with a nonlocal boundary layer parameterization," *Reviews of Geophysics*, vol. 32, no. 4, pp. 363–403, 1994.