Batch Method for Efficient Resource Sharing in Real-time Multi-GPU Systems

Uri Verner, Avi Mendelson, and Assaf Schuster

Technion – Israel Institute of Technology
{uriv,avi.mendelson,assaf}@cs.technion.ac.il

Abstract. The performance of many GPU-based systems depends heavily on the effective bandwidth for transferring data between the processors. For real-time systems, the importance of data transfer rates may be even higher due to non-deterministic transfer times that limit the ability to satisfy response time requirements. We present a new method that allows real-time applications to make efficient use of the communication infrastructure in multi-GPU systems, while retaining the necessary execution time predictability. Our method is based on a new application interface for executing batch operations composed of multiple command streams that can be executed in parallel. The new interface provides the run-time with information it needs to optimize the communication and to reduce the execution time. The method is compliant with common scheduling algorithms, such as EDF and RM, as it provides accurate offline execution time prediction for jobs using their definition and system characteristics.
Experiments with two multi-GPU systems show that our method achieves 7.9x shorter execution time than the bandwidth allocation method, and 39 % higher image resolution than the time division method, for realistic applications.

1 Introduction

In high throughput data processing systems, the computational load is distributed across multiple interconnected processors. To efficiently execute data-parallel parts of the computation, such systems often make use of discrete GPUs and other compute accelerators. The data is collaboratively processed by the CPUs and accelerators, and transferred between their local memories as required by the algorithm.

The data processing operations in many real-time systems are modeled as collections of simple repetitive tasks that generate jobs in a predictable manner. The jobs are executed by offline schedulers, such as Earliest-Dealine-First (EDF) and Rate-Monotonic (RM), which have been extensively studied [1,2]. Such modeling allows execution times to be analyzed and schedule validity to be checked offline. In systems with a multi-processor platform with distributed memories, and particularly in GPU-based systems, the data processing includes data transfer and compute operations.

Modern systems handle communication among different components via a heterogeneous interconnect, which is composed of several linked communication domains, and may be shared by different data transfer operations. As a result, the effective throughput of a data transfer may depend not only on the physical characteristics of the links, but also on the concurrent data transfers that share resources with it. Unless executed

in a controlled way, the resource contention may even cause the communication time to become unpredictable, and the utilization may become prohibitively low.

For real-time systems, where the worst-case execution time is important, the common methods for resource sharing include bandwidth allocation and time division. The bandwidth allocation method calls to divide the resource into a number of smaller portions, and to assign each of them to a task. The time division method calls to split the time domain into time-slots and to assign them to the tasks according to a given priority scheme. Each of these methods allows the system to make the communication time deterministic at the expense of underutilizing resources such as the bus bandwidth, since a resource that is assigned to a task but not fully utilized is not used by the other tasks. Underutilization of the resources could increase power consumption or even prevent the system from meeting real-time deadlines.

In this paper, we propose a new execution method for data transfers and distributed computations between the host (CPUs) and multiple devices (GPUs). The new proposed method is based on a new application/run-time interface that allows the application to define which parts of the multiple command streams of data transfer and compute operations can be considered as a single job (unit of work). From the application's point of view, all the command streams in a job start and complete execution at the same time. We show that using the additional information provided by the new proposed interface the run-time is able to execute data transfers and computations in a way that (1) exploits parallel execution, (2) efficiently utilizes the computation and communication resources, and (3) still provides predictable job execution times. Our measurements on two multi-GPU systems with realistic GPGPU applications show that the tested application achieved up to 7.9x shorter execution time using our method than the bandwidth allocation method, and up to 39 % higher image resolution than the time division method.

2 CPU and GPU Terminologies and Definitions

2.1 System Architecture

In this work, we will focus on a single-node multi-GPU system that includes one or more CPUs and a set of discrete GPUs, each of which has a local memory module and serves as a computational device. Figure 1 illustrates a possible architecture of such a system. The interconnect provides connectivity among the different components such as memories, processors, GPUs, etc., and consists of several communication domains, depicted in the figure by background blocks. Each domain consists of components that use the same architecture and protocol for communication. The domains are bridged by components that belong to several domains.

As the figure shows, the system consists of memory modules (MEM), CPUs, GPUs, and I/O Hubs (IOH). The I/O hubs bridge between the processor interconnect and PCI Express, and provide connectivity between PCI Express devices and the main memory modules; in some architectures, the I/O hubs are integrated into the CPUs. In addition to GPUs, other external devices may be connected via PCI Express; examples include compute accelerators such as Intel's Xeon Phi and high-throughput network cards. In

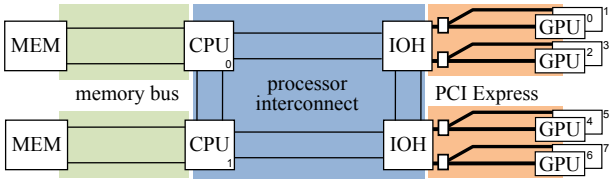


Fig. 1: System architecture of a multi-GPU system

this work, we focus on GPU-based systems, but the proposed method can be extended to be used with other devices as well.

Each CPU accesses its local memory directly via the memory bus, while distant CPUs and I/O Hubs (IOH) access it indirectly via the processor interconnect, which is a set of high bandwidth full-duplex point-to-point links. Examples of such interconnects include Intel QuickPath Interconnect (QPI) and HyperTransport. Data transfers from and to GPU memory are executed by DMA controllers (a.k.a. DMA engines).

2.2 Aggregate Bandwidth

Theoretical bandwidth is the maximum rate at which data can be transferred over a communications network according to hardware specification. *Effective* bandwidth is the maximum measured throughput, which is typically lower than the theoretical bandwidth. In this paper we will also use the term *aggregate* bandwidth which is the total bandwidth over a number of data transfer routes, e.g., the aggregate bandwidth from main-memory to GPU0 and to GPU1.

The bandwidth of a route that traverses several communication domains is limited by the bandwidth in each of the domains. For example, the following table lists the (theoretical) bandwidth of the communication domains and their bus types in a Tyan FT72B7015 system:

Communication domain	Bus type	Bandwidth (GB/s)
memory bus	DDR3	32
processor interconnect	QPI	9.6
PCI Express bus	PCIe 2.0	8

One can see that QPI has higher bandwidth than PCIe 2.0. Therefore, the bandwidth from CPU0 to GPU0 (Fig. 1) is limited by PCI Express. However, the bottleneck for aggregate bandwidth is not necessarily in the bus with the smallest bandwidth. In the example, QPI is the bottleneck for the aggregate bandwidth from the main memory to GPUs 0 and 2 since each GPU uses a separate PCIe port.

2.3 GPU Commands in CUDA and OpenCL

The CUDA and OpenCL programming models provide functions for data transfer and kernel execution. For example, the CUDA function

`cudaMemcpyAsync(dst,src,count,kind[,stream])`

copies `count` bytes from the memory area pointed by `src` on the source device (CPU/GPU) to the memory area pointed by `dst` on the destination device; this function requires CPU buffers to be in page-locked memory [3]. We ignore the `kind` parameter since it is not used on the latest architectures.

CUDA defines the notion of a *stream* (command queue in OpenCL) as a sequence of commands that execute in order. The run-time can execute commands from different streams in any order. Event objects are inserted as bookmarks in a stream for monitoring the progress of execution. The application can query these objects when the execution order reaches the events. In the function above, the application can optionally provide a `stream` object to issue the operation to a stream. For clarity, we use the term *command stream*.

2.4 Sharing Resources in Real-time Systems

The current interfaces for CUDA and OpenCL do not provide mechanisms for real-time execution. The interfaces do not define methods to specify deadlines for commands or command streams, or methods for predicting their execution times. Applications that schedule the program execution in heterogeneous architectures usually do a worst-case static analysis of the execution patterns in order to guarantee that the deadlines are met. The analysis is based on static resource sharing methods, such as static resource allocation or time division, and assumes that the worse case execution time (WCET) is known ahead of time for each job (computation or data transfer). Such a system can be suboptimal in terms of power and utilization, but also may cause the system to miss deadlines, as the following example indicates.

Consider a real-time production inspection system that has a CPU and four GPUs illustrated in Fig. 2. Every period of 50 ms, four images arrive to main memory, are copied to GPU memory, scanned for defects, and, for each image, a Boolean value is returned to main memory. This value denotes if defects were found. The image sizes are 32 MB, 128 MB, 128 MB, and 32 MB, and they are processed on GPUs 0-3, respectively, one image per GPU. The system is required to produce the result in main memory within 50 ms, i.e., before the next period starts. The throughput of a GPU for scanning an image is 10 GB/s (value chosen arbitrarily). The result transfer time is in the order of microseconds; for simplicity, we ignore it in the following time analysis.

Using the resource allocation method (bandwidth allocation) for sharing resources, each task is statically assigned a portion of the communication bandwidth. In this system, the bandwidth bottleneck (to all four GPUs) is 8 GB/s on the CPU-IOH link. Assuming equal bandwidth distribution, each stream is assigned 2 GB/s. Fig. 3a shows a timeline of the expected execution for this method. The processing time is computed as follows:

$$t_{bandw} = \frac{128\,\mathrm{MB}}{2\,\mathrm{GB/s}} + \frac{128\,\mathrm{MB}}{10\,\mathrm{GB/s}} = 76.8\,\mathrm{ms}$$

The resulting time is longer than the 50 ms latency bound, so this approach does not guarantee that the latency constraints are satisfied.

Using time division, the full bandwidth is allocated to tasks for periods of time. The GPUs are allocated separately; hence, kernel executions may overlap. The order in which the system allocates bandwidth to the images is not strictly defined by the method; Fig. 3b shows a timeline for the scenario with the shortest expected processing time. This time exceeds the latency bound. The method does not utilize the bandwidth of the 8 GB/s CPU-IOH bus, but only of a 6 GB/s bus.

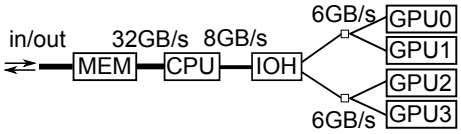
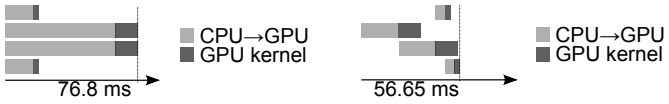


Fig. 2: Communication topology of explanatory example



(a) Bandwidth allocation (b) Time division

Fig. 3: Timeline of data processing for different resource sharing methods

One may notice that the incapability of the system to move allocated resources from one stream to another when they are not used, caused the system to miss the deadline. In the next section we present the batch allocation method that aims to ease this problem.

3 Batch Method for Data Transfer and Computations

In order to allow better utilization of the communication buses (and so, the entire system) while maintaining the ability to bound the execution time, we introduce a new run-time interface to the application (API). The new interface shifts from asynchronous command streams, as defined by CUDA and OpenCL, to batch operations that "bind" segments of multiple command streams together. It allows the system to optimize the execution, taking into account all the resources needed by the batch. From the point of view of the programmer, batch operations are executed atomically in issue order, but internally the run-time schedules the command stream segments concurrently, aiming to minimize the execution time.

Using the new interface, we will show that the run-time can provide two main services to the application: (1) better resource utilization, and (2) execution time prediction for a future batch operation. Such information can be used by the application to better schedule the batch operations in a way that meets system requirements (such as deadlines). It can also be used to dynamically control the execution, hence reducing the gap between the worst-case and actual execution times, and further improving the utilization of the system and its power consumption. Our new model is also compatible with

the task model of classic scheduling algorithms, such as earliest-deadline-first (EDF) and rate-monotonic (RM).

3.1 Interface

We propose the use of the following API for batch operations:

```
    batchOp(dsts,srcs,sizes,kernels,n)
h = batchOpAsync(dsts,srcs,sizes,kernels,n)
b = batchOpQuery(h)
    batchOpSynchronize(h)
t = predictBatchLatency(dsts,srcs,sizes,kernels,n)
```

The API has three types of functions:
– Execution - `batchOp()` and `batchOpAsync()` execute a batch operation that contains `n` command streams. The function `batchOp()` is synchronous, while `batchOpAsync()` is asynchronous and returns a handler object `h` that the application uses to monitor the execution progress. Each command stream is either a data-block transfer, a kernel call, or a data-block transfer to a GPU followed by a kernel call that uses the transferred data on the target GPU.
– Latency prediction - `predictBatchLatency()` returns an execution time prediction for a batch operation. Here, the sources and destinations represent the memory modules, rather than specific addresses, such as GPU0 or MEM0 (local to CPU0).
– Progress monitoring - `batchOpQuery(h)` returns a boolean value indicating the completion status of an asynchronous batch operation, while `batchOpSynchronize(h)` blocks until the operation completes.

Using the API, the application can schedule data transfer operations and kernel calls, providing the run-time with the necessary information to use the interconnect and GPUs efficiently. In this work, we use a restricted model of a command stream: a stream is composed of a data transfer operation followed by a kernel call (both optional). This model does not limit the scheduling options for the application, as it may seem. We discuss this issue in greater detail in the Discussion and Related work section.

3.2 Implementation

This section describes the implementation of the batch method for data transfer and kernel execution. The latency prediction algorithm is presented here and described in detail in the next section.

Execution. Calls to `batchOp()` and `batchOpAsync()` are processed in FIFO order. The execution algorithm schedules the command streams to execute in parallel with the goal of minimizing the execution time. For data transfers, it configures multiple DMA controllers to concurrently transfer data blocks that belong to different streams.

We propose a heuristic algorithm that aligns the *completion times* of the command streams. The algorithm computes when to start executing each command stream, such that the execution of all streams completes at the same time. The algorithm for computing the execution start times is described in the next section.

Two types of batch operations require special treatment:

1. If the batch operation contains command streams that use the same DMA controller, these operations cannot overlap, so the streams are merged by joining their data transfers and their kernels.
2. Data transfers between two GPUs that reside in different PCIe communication domains (e.g. GPUs 0 and 4 in Fig. 1) currently require staging in CPU memory, hence requiring a GPU→CPU and a CPU→GPU data transfer. These transfers may overlap, but the GPU→CPU must begin first. Therefore, after the initial schedule of the batch operations is computed, the algorithm checks whether the order of transfers is inverted and schedules the GPU→CPU earlier if necessary.

The heuristic algorithm limits the execution time to the longest command stream execution time and aims to achieve the highest throughput for this stream's data transfer. While the kernel execution time is independent of execution of other streams, the data transfer can be delayed due to contention. The algorithm delays the data transfers in the other streams to reduce bandwidth contention with the long stream and make them overlap with the kernel execution. Consider the example in Fig. 4; by aligning the completion times, the large blocks are delayed the least by contention with the small blocks, which reduces the execution time.

Fig. 4: Timeline for the batch method

Latency prediction and run-time scheduling. To schedule the execution of data transfers and kernels in a batch operation, the run-time creates a timeline of start and completion events for the component operations (data transfers and kernel calls); the full algorithm is shown in Algorithm 1 in the appendix. This algorithm also computes the total execution time of the operation; i.e., it predicts the execution latency.

The event times are computed iteratively by simulating the execution. Since our execution method aligns the kernel *completion* times, the event times are computed from the batch completion time backwards. For simplicity of discussion, we consider the moment when the batch execution completes as $t = 0$, and count the time backwards.

Computing the start and completion times of kernels is trivial since they all start at time zero (on the reversed timeline) and execute in parallel for fixed amounts of time, given by the user. For data transfers, such computation is more complex, since it needs to take contention into account. It is assumed that the throughput of concurrent data transfers is constant as long as no transfer starts or completes; hence, it is constant between any two events. Next, we present an algorithm for computing the throughput of concurrent data transfers.

Model for bandwidth distribution for concurrent data transfers. We represent the interconnect topology as a weighted directed graph $G = (V, E, b)$, where a vertex $v \in V$ represents a component of the network, such as main-memory, a CPU, a GPU, or a routing component (controller, I/O hub, switch, etc.). A directed edge $e_{ij} \in E$ represents a unidirectional link from component n_i to component n_j, and the positive weight $b(e_{ij})$ represents its bandwidth.

The topology graph can be used to detect the bandwidth bottleneck along a path between two components and across multiple paths. For example, Fig. 5 illustrates the network topology graph of a Tyan FT72B7015 system with four GPUs. In this graph, the M0→GPU0 and M0→GPU1 paths both traverse the CPU0→IOH0 edge; hence the aggregate bandwidth on these two paths is 9.6 GB/s. The throughput of a data transfer is limited by the minimum edge weight $b(e)$ on its path in the graph. It may further be limited by concurrent data transfers that consume some of the bus bandwidth.

We use the *effective* bandwidth for the edge weights instead of the theoretical one; for worst-case analysis, a lower bound on the effective bandwidth is used.

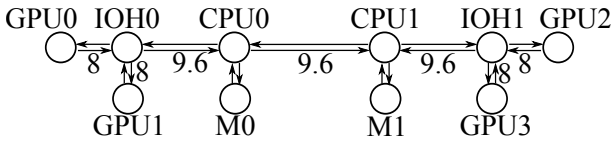


Fig. 5: Network topology graph of a Tyan FT72B7015 system with four GPUs. The edge weights denote the theoretical bandwidth of the link in GB/s.

3.3 Results for the Example in Sect. 2.4

Our example in Sect. 2.4 indicates that using the existing resource sharing mechanisms may lead to sub-optimal utilization of the shared resources and to a failure of the system (deadline miss). In this section, we return to this example and show that our method solves these problems for that case.

The batch method re-allocates the bandwidth during the execution in a predictable manner. Thus, the application is able to utilize the full 8 GB/s bandwidth, while still being able to validate that the execution completes within the given latency constraints. The application binds the CPU→GPU data block transfers and kernel calls into a batch operation, followed by another batch operation for the return values. The run-time executes the command streams concurrently, aiming for a minimum total execution time. Fig. 4 shows a timeline of the expected execution of a batch operation consisting of the data transfers and kernel calls for the four images. Due to the higher bandwidth, the operation completes after 44.8 ms, which is consistent with the given latency constraints. Thus the schedule is valid.

The example shows that our proposed method achieves shorter execution times than the existing methods by using the interconnect more efficiently.

4 Evaluation

To evaluate the batch method, we consider two applications running on two multi-GPU systems. We compare the batch method with two other bandwidth distribution methods: bandwidth allocation and time division.

For each system, we provide the system specification, find and analyze the effective bandwidth, and compare the performance using each of the methods in test case applications.

4.1 Baseline Configurations and Their Effective Bandwidth

In this section we describe two existing systems and use the techniques described in Sect. 3.2 to analyze their effective bandwidth.

Nehalem multi-GPU system

This system is a Tyan FT72B7015 server featuring the following components:
– Two 4-core Intel Xeon 5620 CPUs at 2.4 GHz, based on the Nehalem micro-architecture
– An Intel 5520/ICH10R chipset with a QPI processor interconnect at 4.8 GT/s (9.6 GB/s) and two I/O hubs, each with two PCIe 2.0 ports
– A total of 24 GB RAM in two modules
– Four NVIDIA Tesla C2050 GPUs, each with 3 GB of GDDR5 memory
The system runs Ubuntu 10.04 x64 Linux with CUDA SDK 5.0.

The batch method uses the effective aggregate bandwidth to compute the completion times of data transfers. In Sect. 3.2, we presented a method for computing the aggregate bandwidth from a topology graph with edge weights that represent bus bandwidth. We obtained the edge weights using a series of benchmarks. To find the effective bandwidth of a bus, we transferred 256 MiB (2^{28} bytes) data blocks between CPU0 and the GPUs in host-to-device (H2D), device-to-host (D2H), and bi-directional (Bi) modes on all the paths that traverse this bus, and measured the aggregate throughput for each direction (Fig. 7a). Using these measurements, we determined the effective bus bandwidths in Fig. 6a. For simplicity, we did not include the bi-directional bandwidth measurements in the figure.

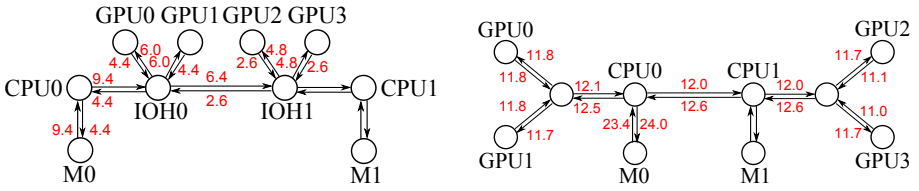
The results show that the bus bandwidth is asymmetric; the effective H2D PCIe bandwidth is between 25 % and 50 % higher than the D2H bandwidth. We also see that the effective bandwidth to the remote GPUs is 20 % lower. Since the QPI bus has higher bandwidth than the PCIe bus, this indicates that the latency of the extra hop over QPI translates into throughput degradation. The H2D aggregate bandwidth scales up for two GPUs by 57 % for local GPUs (saturates the QPI bus), and by 33 % for remote GPUs. In contrast, the D2H bandwidth for two GPUs does not scale. For four GPUs, the H2D bandwidth does not scale further, but the aggregate D2H bandwidth lines up with the bandwidth of the local GPUs. We ascribe the reduced scaling to chipset limitations, except where the QPI bus was saturated. Since the GPUs only have one DMA engine, they are not able to get more bandwidth from bi-directional transfer, yet we see that for the local GPUs a bi-directional transfer is faster than an H2D transfer followed by a D2H.

Sandy Bridge multi-GPU system

This system features the following components:
– Two 6-core Intel Xeon E5-2667 CPUs at 2.9 GHz based on the Sandy Bridge micro-architecture
– An Intel C602 chipset w/ QPI at 8 GT/s (16 GB/s) and two PCIe 3.0 ports in each CPU

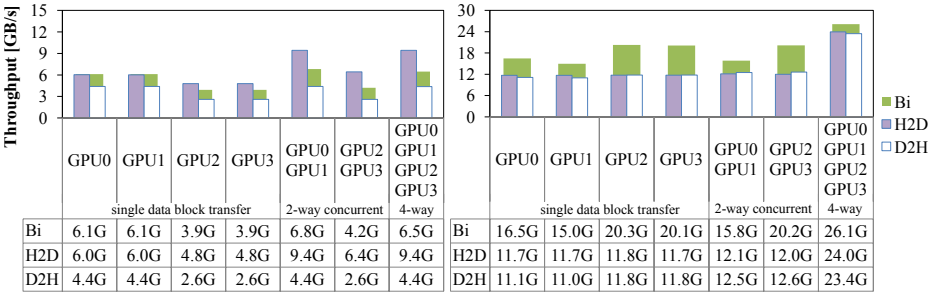
– 64 GB of RAM in two modules
– Two NVIDIA Tesla K10 cards, each with two GPU modules, connected by a PCIe switch, that include a GPU and 4 GB of GDDR5 memory.
The system runs Red Hat Ent. 6.2 Linux with CUDA SDK 5.0.

We determined the effective bus bandwidths shown in Fig. 6b using a series of benchmarks, as described for the Nehalem system. Figure 7b shows the benchmark results for the Sandy Bridge system.



(a) Nehalem (b) Sandy Bridge

Fig. 6: Topology graph showing effective bandwidth in GB/s



	GPU0	GPU1	GPU2	GPU3	GPU0 GPU1	GPU2 GPU3	GPU0 GPU1 GPU2 GPU3
Bi	6.1G	6.1G	3.9G	3.9G	6.8G	4.2G	6.5G
H2D	6.0G	6.0G	4.8G	4.8G	9.4G	6.4G	9.4G
D2H	4.4G	4.4G	2.6G	2.6G	4.4G	2.6G	4.4G

(a) Nehalem

	GPU0	GPU1	GPU2	GPU3	GPU0 GPU1	GPU2 GPU3	GPU0 GPU1 GPU2 GPU3
Bi	16.5G	15.0G	20.3G	20.1G	15.8G	20.2G	26.1G
H2D	11.7G	11.7G	11.8G	11.7G	12.1G	12.0G	24.0G
D2H	11.1G	11.0G	11.8G	11.8G	12.5G	12.6G	23.4G

(b) Sandy Bridge

Fig. 7: Bus bandwidth benchmark

The Sandy Bridge system has higher bandwidth than the Nehalem system as it uses PCIe 3.0. Moreover, the bandwidth in Sandy Bridge is symmetric and is similar for local and remote GPUs, unlike in the Nehalem system. The aggregate bandwidth to the GPU pairs is only slightly higher than for the individual GPUs; this is expected, as the GPUs share a PCIe bus. However, moving further to four GPUs, the bandwidth scales almost perfectly. For a single GPU, bi-directional transfers increase the bandwidth by 32 %-72 % over uni-directional transfers, while for all four GPUs, the increase in bandwidth is only 10 %.

4.2 Application Based Performance Evaluation

The evaluation we are providing in this section is based on two applications, one that represents a solution for a wide range of algorithms in areas such as signal process-

ing and image processing, and the other which is motivated by an application we are developing as part of a consortium for inspection of 450 mm wafers called Metro 450.

Domain decomposition. In many scientific GPGPU applications that work on large datasets, the domain is decomposed into parts that fit in GPU memory. The application we examine uses two GPUs, and follows a common pattern in multi-GPU programming:

```
1: scatter 3.69 GB input data to 2 GPUs
2: repeat 100 times
3:   compute boundary points (1 ms)
4:   in parallel:
5:     GPUs exchange 295.2 MB boundaries
6:     compute internal points (23 ms)
7: gather output from GPUs
```

Table 1 shows the execution time using each of the bandwidth distribution methods. Using the batch method on the Sandy Bridge system, the application runs 7.9 times faster than using the bandwidth allocation method, and 2 times faster than using the time division method. On the Nehalem system, it runs 28 % faster than bandwidth allocation, and 15 % faster than time division.

While the batch method efficiently uses the available bandwidth, the bandwidth allocation splits the available bandwidth and conservatively assigns pieces to each data transfer; hence, when a data transfer is not actually being transmitted its bandwidth is lost. The time allocation method also does not fully utilize the available bandwidth, as without executing several data transfers concurrently, the system cannot fully utilize the available bandwidth.

Wafer inspection. In a real-time wafer inspection system, an image of a wafer arrives to main-memory every 25 ms, and is sent to the GPUs for defect detection. The defects on the wafer are distributed non-uniformly, so in order to balance the kernel execution times, the image is split between GPUs 0-3 as follows: 40 %, 10 %, 10 %, and 40 %. The four GPUs process the image at 10 GB/s.

The system is required to produce a defect report in main-memory every 60 ms, and its size equals that of the image. This requirement limits the resolution of wafer images processed by the system, yet the maximal resolution depends on the bandwidth distribution method. To calculate the maximum resolution for each of the methods, we take into account that the system uses the non-preemptive EDF scheduler, and check the validity of the schedule using the test described by Jeffay et. al. [4]. We use image size as a measure of the resolution.

Table 1 shows the maximum image sizes using each of the bandwidth distribution methods. The batch method achieves the highest average throughput by multiplexing the data transfers. On the Nehalem system, it supports 39 % higher image resolution than that of the bandwidth allocation method, because, when the short data transfers complete, it assigns some of the released bandwidth to the long transfers. It also achieves 37 % higher image resolution than the time division method by utilizing the aggregate bandwidth of multiple routes.

On Sandy Bridge, it supports 22 % higher image resolution than that of the bandwidth allocation method, because, when the short data transfers complete, it assigns some of the released bandwidth to the long transfers. It also achieves 39 % higher image resolution than the time division method by utilizing the aggregate bandwidth of multiple routes.

Table 1: Performance comparison of the different methods

	Domain decomposition (execution time)		Wafer inspection (image size – larger is better)	
	Nehalem	Sandy Bridge	Nehalem	Sandy Bridge
Batch method	20.03 sec	3.18 sec	82.46 MB	156.77 MB
B/W allocation	25.59 sec	25.19 sec	59.28 MB	128.10 MB
Time division	23.10 sec	6.43 sec	60.24 MB	112.58 MB

4.3 Execution Time Prediction

A fundamental part of the batch method is the execution timeline computation algorithm described in Sect. 3.2. This algorithm is used both for scheduling the execution of batch operations and predicting their latency. Hence, precision is essential. We evaluate the algorithm's precision by (1) computing an execution timeline for selected batch operations; (2) executing the operations and recording the execution time of each command stream; and (3) comparing the execution times for each command stream and for the batch in total.

We ran two experiments on each system. On Nehalem, we executed the following batch operations:

Experiment 1 (Batch #1)	Experiment 2 (Batch #2)
256 MiB H2D (GPU0) + kernel (10 ms)	256 MiB D2H (GPU0)
128 MiB H2D (GPU1) + kernel (10 ms)	128 MiB D2H (GPU1)

On Sandy Bridge we executed similar experiments, but with four GPUs; the additional block sizes were 64 MiB and 32 MiB. Table 2 displays the results of the computations and measurements side by side, and shows the prediction error of the total execution time. The prediction error was no more that 3.6 % for all the tested cases. The table in that figure shows that the computed execution times of individual command streams are also similar to the recorded values.

We've shown that our algorithm correctly estimates command stream execution times for batch operations with different data transfer sizes that execute concurrently. Thus, it can be used for efficiently scheduling the execution of command streams and predicting the execution time of batch operations.

4.4 Throughput Computation vs. Measurement

In Sect. 3.2, we presented a method for computing the aggregate throughput using a topology graph of the system. The topology graph represents bus bandwidth as edge

Table 2: Predicted vs. measured execution time for each command stream and for the entire batch

Configuration	Prediction					Measurement					
	S1	S2	S3	S4	Total	S1	S2	S3	S4	Total	Err
Nehalem H2D	60.71	38.46			60.71	60.94	38.64			60.94	**0.4 %**
Nehalem D2H	91.69	61.12			91.69	91.97	61.39			91.97	**0.3 %**
Sandy Bridge H2D	43.74	32.25	18.40	15.60	43.74	43.44	32.08	18.85	16.01	43.45	**0.7 %**
Sandy Bridge D2H	34.11	22.02	8.59	5.72	34.11	32.95	21.52	8.65	5.76	32.94	**3.6 %**

weights, allowing the computation of end-to-end throughput by finding the bottleneck. However, the computed throughput may differ from the actual throughput due to chipset limitations. We evaluated the relative error of our algorithm by comparing the computed values with throughput measurements in a series of experiments with concurrent data transfers on the Sandy Bridge system.

For the experiments in Fig. 7b the error was 0 % for H2D and D2H, and 0 %–(-4) % for Bi. The error in these experiments was low, because the edge weights in the topology graph were set to match these measurements as much as possible. Interestingly, the weights could not fully match the Bi bandwidth measurements, and an error of 4 % was observed. In cases with two GPUs that do not appear in Fig. 7b, the maximum error was as follows: 0.7 % for H2D, 5.4 % for D2H, and 26 % for Bi. The error for the Bi transfers was high because the measured bandwidth did not fit the topology model. We attribute this result to chipset limitations.

To summarize, our algorithm efficiently calculated the expected throughput for uni-directional data transfers, but not for all bi-directional data transfers.

5 Discussion and Related work

We have shown in Sect. 4.2 that our method is efficient in executing workloads of data transfers and computations. The experimental results show that it achieves up to 7.9 times better results than the bandwidth allocation method (execution time) and up to 39 % better results than the time division method (higher image resolution) in realistic applications. In Sect. 4.3, we have also shown that the batch execution time is predictable and that our latency prediction algorithm computes a close estimation of that execution time. Sufficient safety margins on bandwidth and setup time should be used to compensate for prediction errors where strict latency guarantees are required.

Communication scheduling that incorporates contention awareness has been studied in several works [5,6,7,8]. In all these works, the basic unit of communication is a message – a chunk of data that is sent from a source to a destination. In order to overcome bus contention issues when sending multiple messages, the proposed algorithms allocate bandwidth and/or communication time for each message. In this work, we proposed to extend the classic communication model by allowing multiple messages to be sent in one job. This extension allows the run-time to utilize the communication network more efficiently by optimizing the total job execution time, rather than the transfer

time of each message. Our method provides an algorithm to predict the job execution time, which is required as part of the input to the scheduler.

Our method executes batch operations that are a collection of command stream segments containing data block transfers and kernel calls. However, we have limited each command stream segment to contain only a data transfer followed by a kernel call (both optional), forcing the application to divide longer streams into multiple batch operations. However, fine-grained scheduling can be achieved using two techniques: (1) dividing large data blocks into smaller chunks (also suggested in other works [9,10]), and (2) scheduling kernels in parallel with batch operations, without using the API. When using the second technique, the application needs to make sure that the kernel and the batch operation do not use the same GPU for computations. Using the techniques mentioned above, the application can make fine-grained scheduling decisions for data transfers and kernel calls. Our method uses execution time computations assuming no external factors that influence the execution time. Therefore, during the execution of a batch operation, the interconnect must be used exclusively by the execution algorithm.

In Sect. 3.2 we presented an algorithm for computing throughput using a topology graph of the system with edge weights that represent bus bandwidth. The precision evaluation in Sect. 4.4 shows that this method is effective for data transfers from the CPU to the GPUs and in the opposite direction, but not for bi-directional data transfers. Our experiments show that for bi-directional transfers, the results are not consistent with topology graph analysis. In applications with workloads where the throughput computation algorithm is not efficient, we recommend using a lookup table that contains the throughput values for any combination of data transfer routes that may be used in the application. The throughput values can be found using benchmarks. Since the batch operation execution uses the interconnect exclusively, the benchmark results are expected to be consistent with the throughput values during the execution.

RGEM [9] is a run-time execution model for soft real-time GPGPU applications. It divides data transfers into chunks to reduce blocking times and schedules the chunks according to given task priorities. Because this model uses blocking data block transfers, it is based on the time division bandwidth distribution scheme.

Verner et al. [11] presented a framework for processing data streams with hard real-time constraints. The framework abstracts the GPUs as a single device and transfers the data to all the GPUs collectively, thus utilizing the bandwidth of the shared bus. The authors demonstrate the importance of developing a data transfer mechanism that would provide efficient communication and execution time guarantees in order to achieve better scheduling opportunities.

Kato et al. [12] described a technique for low-latency communication between the GPU and other I/O devices. The technique allows the devices to communicate by writing directly to each other's memory, thus decreasing the transfer latency. However, since there is no centralized mechanism that schedules the data transfers, the transfer latency may be influenced by concurrent data transfers.

Several works [11,13,9] predicted the CPU-GPU data-transfer latencies using empirical performance models. The performance models were built upon data-transfer benchmarks for a range of transfer sizes.

6 Conclusion

In this paper, we have presented a new execution method for data transfers and computations in multi-GPU systems. Using this method, a real-time application can schedule and efficiently execute data transfers and kernel calls. The method executes jobs we call "batch operations" that include multiple data transfers and kernel calls that can be executed in parallel. The method also includes an algorithm for predicting the execution time of batch operations, which is made possible by executing them atomically. The method also implements a new and efficient execution algorithm for batch operations. Unlike existing execution methods that provide predictable execution times for individual data transfers and kernels calls at the cost of performance, our algorithm overlaps the execution of multiple data transfers and kernels, thus gaining efficient communication and compact execution.

Acknowledgments. The work was supported by the Metro 450 Israeli national consortium for inspection of 450 mm wafers.

References

1. O. U. P. Zapata and P. M. Alvarez, "EDF and RM multiprocessor scheduling algorithms: Survey and performance evaluation," *Queue*, pp. 1–24, 2005.
2. S. Baruah and J. Goossens, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman Hall/CRC Press, 2004.
3. NVIDIA Corporation, "CUDA API Reference Manual, version 5.0," 2012.
4. K. Jeffay, D. Stanat, and C. Martel, "On non-preemptive scheduling of period and sporadic tasks," in *Real-Time Systems Symposium*, 1991, pp. 129–139.
5. J. P. Lehoczky and L. Sha, "Performance of real-time bus scheduling algorithms," *ACM SIGMETRICS Performance Evaluation Review*, vol. 14, pp. 44–53, 1986.
6. M. Natale and A. Meschi, "Scheduling messages with earliest deadline techniques," *Real-Time Systems*, no. 1993, pp. 255–285, 2001.
7. O. Sinnen, L. A. Sousa, and S. Member, "Communication contention in task scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 6, pp. 503–515, 2005.
8. M. Balman, "Data transfer scheduling with advance reservation and provisioning," Ph.D. dissertation, Louisiana State University, 2010.
9. S. Kato and K. Lakshmanan, "RGEM: A responsive GPGPU execution model for runtime engines," *Real-Time Systems Symposium (RTSS)*, pp. 57–66, Nov. 2011.
10. C. Basaran and K.-D. Kang, "Supporting preemptive task executions and memory copies in GPGPUs," *Euromicro Conference on Real-Time Systems*, pp. 287–296, Jul. 2012.
11. U. Verner, A. Schuster, M. Silberstein, and A. Mendelson, "Scheduling processing of real-time data streams on heterogeneous multi-GPU systems," *International Systems and Storage Conference (SYSTOR)*, pp. 1–12, Jun. 2012.
12. S. Kato, J. Aumiller, and S. Brandt, "Zero-copy I/O processing for low-latency GPU computing," *International Conference on Cyber-Physical Systems (ICCPS'13)*, pp. 170–178, 2013.
13. C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-aware task scheduling on multi-accelerator based platforms," in *International Conference on Parallel and Distributed Systems (ICPADS)*, Dec. 2010, pp. 291–298.

Algorithm 1 Batch run-time scheduling algorithm (MATLAB)

```
function Batch
    %Batch = [ (dst1,src1,size1,kernel1),
    %          (dst2,src2,size2,kernel2),
    %             ...
    %          (dstN,srcN,sizeN,kernelN)]
    N=size(Batch,1)

    % Event times for each command stream
    % (kernel begin, kernel end, copy begin, copy end)
    StreamSched = zeros(N,4)
    StreamSched(1:N,2) = arrayfun(@KernelTime, Batch(1:N,4))
    StreamSched(1:N,3) = StreamSched(1:N,2)

    % Timeline of data transmission start times, which are equal to the
    % kernel completion times. Col #1: time, Col #2: id of command stream
    DataStart = [StreamSched(1:N,3), [1:N]']
    DataStart = sortrows(DataStart,1)   % sort by time

    % t is a time point that moves between event times as they are computed
    % Workload is the set of stream ids that have a data transfer in
    % progress. The initial values are set to the time when the first data
    % transfer starts executing
    t = DataStart(1,1)
    Workload = DataStart(1,2)
    for i = 2:size(DataStart,1)
       Tnext = DataStart(i,1)
       while (t < Tnext)
           [t,Workload,Completed,Batch] =
                ComputeNextEvent(t,Tnext,Workload,Batch)
           StreamSched(Completed,4) = t
       end
       Workload = [Workload; DataStart(i,2)]
    end
    while (~isempty(Workload))
        [t,Workload,Completed,Batch] = ComputeNextEvent(t,inf,Workload,Batch)
        StreamSched(Completed,4) = t
    end
    % makespan is the time difference between first and last events
    makespan = t + SetupTime(Batch) % add setup time (usually very small)
    % reverse timeline
    Sched = makespan - StreamSched(:,end:-1:1)
end

function [CurTime,Workload,Completed,Batch] = ComputeNextEvent(Tstart,Tnext,
     Workload,Batch)
    if (length(Workload) == 0)
        CurTime = Tnext
        Completed = []
        return
    end
    % Get throughput for each data transfer in workload
    T = Throughput(Batch(Workload,:))
    sizes = [Batch{Workload,3}]'
    % Compute earliest completion time
    completion_times = sizes ./ T
    soonest = min(min(completion_times),Tnext-Tstart)
    CurTime = Tstart + soonest
    % Update remaining data to transfer in all active command streams
    sizes = sizes - soonest * T
    for ii = 1:length(Workload)
        Batch{Workload(ii),3} = sizes(ii)
    end
    % Completed is a list of completed data transfers
    Completed = Workload(sizes <= 0)
    Workload = Workload(sizes > 0)
end
```