Scheduling processing of real-time data streams on heterogeneous multi-GPU systems

Uri Verner
Technion – Israel Institute of Technology
uriv@cs.technion.ac.il

Assaf Schuster
Technion – Israel Institute of Technology
assaf@cs.technion.ac.il

Avi Mendelson
Technion – Israel Institute of Technology
mendlson@cs.technion.ac.il

Mark Silberstein
University of Texas at Austin
marks@cs.utexas.edu

Abstract

Processing vast numbers of data streams is a common problem in modern computer systems and is known as the “online big data problem.” Adding hard real-time constraints to the processing makes the scheduling problem a very challenging task that this paper aims to address. In such an environment, each data stream is manipulated by a (different) application and each datum (data packet) needs to be processed within a known deadline from the time it was generated. This work assumes a central compute engine which consists of a set of CPUs and a set of GPUs. The system receives a configuration of multiple incoming streams and executes a scheduler on the CPU side. The scheduler decides where each data stream will be manipulated (on the CPUs or on one of the GPUs), and the order of execution, in a way that guarantees that no deadlines will be missed. Our scheduler finds such schedules even for workloads that require high utilization of the entire system (CPUs and GPUs).

This paper focuses on an environment where all CPUs share a main memory, and are controlled by a single operating system (and a scheduler). The system uses a set of discrete graphic cards, each with its own private main memory. Different memory regions do not share information, and coherency is maintained by the use of explicit memory-copy operations. The paper presents a new algorithm for distributing data and scheduling applications that achieves high utilization of the entire system (CPUs and GPUs), while producing schedules that meet hard real-time constraints.

We evaluate our new proposed algorithm by using the AES-CBC encryption kernel on thousands of streams with realistic distribution of rates and deadlines. The paper shows that on a system with a CPU and two GPU cards, our current framework allows up to 87% more data to be processed per time unit than a similar single-GPU system.

SYSTOR'12, June 4–6, 2012, Haifa, Israel.
Copyright © 2012 ACM 978-1-4503-1448-0…$10.00

1. Introduction

Processing vast numbers of data streams is a common problem in modern computer systems and is also known as the “on-line big data problem.” There are many reasons for the fast increase in the amount of data being stored and transferred over the network: new devices, location based services, social networks and more [1, 23]. Analyzing this data in real-time and the ability to draw smart conclusions from it can be crucial in life-critical and latency-sensitive applications such as medical data processing and traffic control; it can also be very profitable in the stock exchange. Thus, many company resources and significant money are spent in this new fast growing area. The need for meeting (hard) real-time constraints fundamentally changes the system design space. The existing well-known throughput-optimized stream processing techniques may not lead to the expected result for several reasons. These include (1) tight deadlines, which may prevent computations from being distributed across multiple computers because of unpredictable network delay, and (2) the requirement that the scheduler determines in advance whether a given scheduling policy, when applied to a set of streams, will violate their deadline requirements.

A compute engine that aims to manipulate hard real time streams can be as small as a smart phone or as large as a farm of servers or even a computer cloud. It is commonly agreed that an important building block for such engines is a system built out of a combination of CPUs and GPUs, where the OS, scheduler, drivers and applications are running on the CPUs, while the GPUs are used as efficient accelerators for applications that need to manipulate data streams.

Efficiently scheduling operations on heterogeneous systems is considered a very difficult task. Adding hard real-time constraints to the processing makes the scheduling even more challenging. A recent paper [22] addresses the schedulability aspect of a system that contains only CPUs and a single GPU. That work addresses the different difficulties introduced by the unique architectural characteristics of the GPU, such as:

- Low single thread performance – GPUs are optimized for throughput; they multiplex thousands of lightweight threads on a few SIMD units, thereby trading single-thread performance for higher throughput. Thus, guaranteeing deadline compliance of individual streams is challenging.
- Unpredictable communication latency between the CPU and the GPUs
- When using discrete graphics cards, each has its own memory. Data needs to be copied between the CPU's main memory and the GPU's graphical memory. No consistency is guaranteed.

The scheduling methods presented in that paper extend the "classic" works on scheduling under hard real-time processing [4, 5, 8, 18, 19] for dealing with a very simple heterogeneous environment. This work aims to extend the previous work to more common environments, where the system contains several different discrete graphical cards. Such an extension is not trivial and requires the development of new techniques and algorithms.

We develop a fast polynomial-time heuristic for scheduling thousands of streams between the CPUs and the GPUs. The heuristic looks at the 2D space defined by stream rates and the deadlines of the applications that manipulate the streams, and partitions the space to per-device regions, so that all the streams belonging to a region will be scheduled to the respective device (CPU or GPU).

We show how to define these regions and prove through experiments and simulation that this simple heuristic can substantially improve the system utilization and also increase the number of streams the system can manipulate at a given time.

We implement and evaluate our stream processing framework by using AES-CBC encryption of multiple streams. Unlike other works on AES encryption on GPUs [10, 11, 17], our version of the AES-CBC encryption is stateful and requires sequential processing of the data belonging to the same stream. The algorithm forces multiple streams with different rates to be scheduled concurrently on a GPU. The paper describes a method for the static balancing of multiple streams that achieves high GPU throughput even for harshly unbalanced workloads. It then develops the schedulability criterion for AES-CBC processing on a GPU, used in conjunction with the scheduling algorithm described above.

We perform extensive experiments on a variety of inputs with thousands of streams, using exponential and normal distributions of rates and deadlines. The framework is capable of achieving maximum throughput of 22 Gbit/sec even with deadlines as short as 40ms while processing 12,000 streams. We show that our current framework allows up to 87% more data to be processed per time unit than the one in our previous work by taking advantage of a second GPU, and up to 106% increase in GPU-only throughput.

Overall, adding two GPUs to a quad-core machine allows 7-fold throughput improvement over highly optimized CPU-only execution.

2. System model

Our system applies a data-processing application to a set of real-time data streams. It aims to achieve the maximum throughput while satisfying the real-time constraint of each stream.

2.1 Data stream

A real-time data stream s_i is a source of data packets of size p_i that arrive with period t_i, to be serially processed (e.g., encrypted) by the system within a given latency bound l_i of their arrival. Figure 1 depicts the data processing schema of such data stream. The arriving packets (stage 1 in Figure 1) are accumulated in a buffer, until being sent for processing. Periodically, with period T_i, the buffer is observed, and all accumulated packets are removed, packed, and sent to a processing queue as a job (stage 2). A job is described in Figure 2 and denoted by a tuple $\langle w, q, d \rangle$, where w represents the amount of work needed for processing all data within the job, $q = T_i$ is the job inter-arrival period, and d represents a deadline for the computations of the job. The deadline is relative to the arrival time of the first data packet. After the job is formed, it is scheduled for execution.

From the computational engine point of view, a stream s_i is composed of an infinite sequence of jobs J_0, J_1, J_2, $\ldots$ (Figure 2), arriving at times T_i, $2T_i$, $3T_i$, $\ldots$. For a single stream, all jobs have the same characteristics, but jobs that originate from different streams may have different characteristics, $\langle w_k, q_k, d_k \rangle$.

For simplicity, we assume that the amount of computations needed to process a job k (represented by w_k) is proportional to the size of the job, i.e., $w_k = \lambda p_i \frac{T_i}{t_i}$. Thus, we can characterize a job using a shorter representation, $\langle r_k, d_k \rangle$, where $r_k = \frac{p_i}{t_i}$ represents the data rate.

We can describe the system as a compute engine that needs to process a set S containing multiple independent streams, where each stream $s_i : \langle r_i, d_i \rangle \in S$ has its own rate and deadline. Jobs belonging to the same stream need to

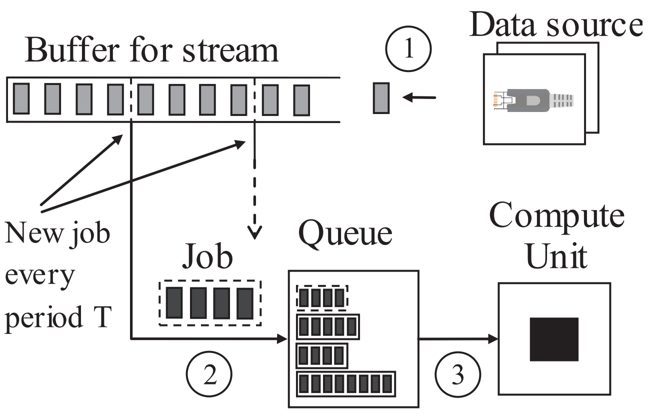


Figure 1: Data processing schema for stream s_i

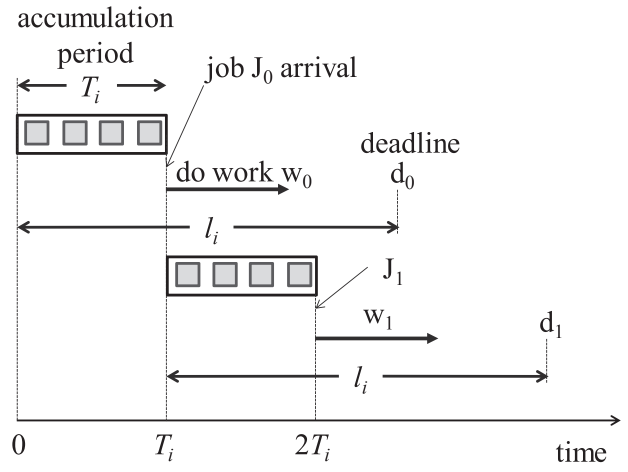


Figure 2: Processing timeline

be processed sequentially while jobs belonging to different streams can be processed in parallel.

2.2 Hardware

This work assumes a heterogeneous system composed of a set of compute units (CU): general-purpose processors (CPUs) and a set of graphics or vector units (GPUs), as depicted in Figure 3. Such systems are commonly used by servers, HPC machines, and many other modern systems. A CPU serves as a master; it runs the OS, performs the scheduling, and executes jobs that were assigned to it. In our model, GPU cards are placed over a PCI express bus, have their own graphical memory, and serve as slave devices (in accelerator mode). A GPU receives a batch of jobs (data and instructions) from a CPU and executes the code (kernel) on data that was previously copied to its local memory. If necessary, the results are then copied back to the main memory, controlled by the CPU. A GPU processes one batch at a time in a non-preemptive manner. Jobs in the batch are scheduled on the GPU's SIMD units by a hardware scheduler. Thus, we have no control over or information about the order in which the jobs will be executed; we assume that the scheduler strives to maximize GPU throughput.

In our model, a CPU and a GPU have separate physical memories, as evident in the use of discrete graphic cards. Data is copied using DMA engines that allow independent data movement between the main memory and the local memory of each graphics card. Note that PCIe buses are organized in a tree structure, where the root of the tree is usually located on the main board. A CPU and a GPU implement the release consistency memory model, whereby the data transferred to and from a GPU during kernel execution may be inconsistent with that observed by the running kernel. This means that consistency is enforced only at the kernel boundaries.

3. Scheduling of multiple streams

We define the scheduling problem as follows: Given a set of input streams $\{s_i : \langle r_i, d_i \rangle\}$, each with its own rate r_i and deadline d_i, find the assignment of streams to compute units, s.t. the obtained total processing throughput equals $\sum_i r_i$, and no data item misses its deadline. This problem has an optimal algorithm for a uniprocessor CPU [16]; that is, it produces a valid schedule for every feasible system. However, scheduling jobs with arbitrary deadlines on multiprocessor systems is a hard, exponential problem [18]. The existing CPU-only scheduling approaches [5, 16, 18] cannot be applied in our setup because they schedule jobs one- or a-few-at-a-time. In contrast, GPUs require batches of thousands jobs to be packed together, raising the question of how to select those to be executed together so that all of them comply with their respective deadlines.

Two main approaches are known for scheduling real-time streams on multiple CUs: dynamic global scheduling and static partitioning [19]. In global scheduling, idle processors are dynamically assigned the highest priority job available for execution. Static partitioning is done in two steps: task allocation to CUs and scheduling tasks on the individual processors.

Multiprocessor systems are known to be tightly coupled and use shared memory for inter-process communication. The time of such communication is low compared to task execution times. Due to this reason, a multiprocessor may use a global (centralized) scheduler. In contrast, the memory in our system is divided to several distant physical units – the main memory, shared by the CPU cores, and DRAM memory on every GPU, shared by the cores of that GPU. Global scheduling is not practical in our system for two reasons:

1. The standard GPU model is bulk synchronous; GPUs are not optimized for dynamic update, of data in GPU memory while a kernel is running. Thus, a batch of jobs being executed by a GPU must be completed before a new batch can be started. Hence, the new batch for the

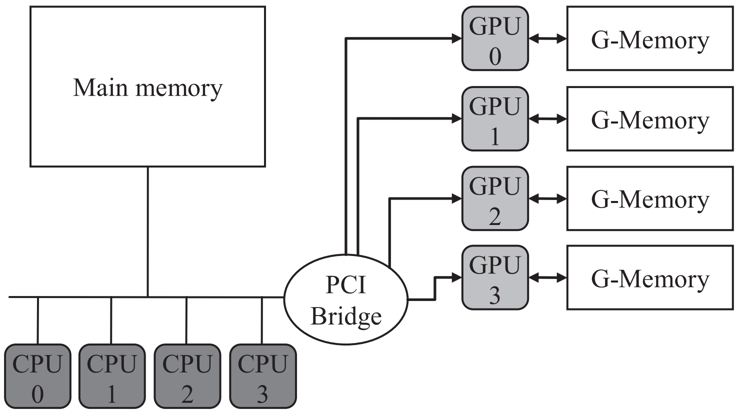


Figure 3: A model of a heterogeneous system composed of CPUs and GPU accelerators

GPU should be statically created in advance on a CPU, under the constraint of timely completion of all jobs in the batch. Even if an efficient mechanism for sending new jobs to a running kernel existed, their completion time would depend on the hardware scheduler and would not be known in advance.

2. The stream state is carried from one job in a stream to the next job in the same stream. The overhead of moving jobs between CUs might become a bottleneck due to slow communication between the CPU and the GPU. Furthermore, since a steady state is assumed, migration can be avoided with proper batch creation.

These considerations led us to choose the static partitioning method, where multiple streams are statically assigned to a particular CU. Streams that are assigned to the CPU can then be scheduled globally, or with another level of static partitioning. Stream scheduling on the GPU uses the hardware scheduler to execute periodic batches of jobs. The main challenge is to select the streams for the GPU to achieve the required throughput under the deadline constraints.

3.1 GPU batch scheduling

Scheduling on the GPU is performed in *batches-of-jobs*, or *batches*. A batch is a collection of jobs from a set of streams. Its execution time depends on parameters such as the number of jobs, distribution of job sizes, and the total amount of work to be processed. Every batch goes through a four stage pipeline: data aggregation, data transfer to local GPU memory, kernel execution, and transfer of results to main memory. These stages are illustrated in Figure 4. The duration of a pipeline stage is at least the time of the longest of the four operations. Processing must be complete before any job deadline is missed, i.e., before the earliest deadline of any job in the batch. Therefore, for a set of GPU-assigned streams S, and $d_{min} = min_{s_i \in S} d_i$, the earliest relative deadline in S, the duration of a pipeline stage is limited by $\frac{1}{4}d_{min}$. This constraint limits the length of jobs in the batch. Given a batch of jobs, we rely on a GPU performance model to calculate the length of each pipeline stage.

Our design for efficient job batching was guided by the following principles:

1. Batch as many jobs as are ready to be executed. Batches with many independent jobs have higher parallelism and provide more opportunities for throughput optimization by the hardware scheduler.

2. For every job, aggregate as much data as possible (this means aggregate as long as possible). This is because batch invocation on a GPU incurs some overhead which is better amortized when the job is large. Moreover, transferring larger bulks of data between a GPU and a CPU improves the transfer throughput. Aggregation time is limited by $\frac{1}{4}d_{min}$.

Distribution of job sizes in a batch affects the load balancing on the GPU. We batch jobs in a way that minimizes their overall execution time, while all batches complete on time. GPU-assigned streams are processed in batches of jobs with a uniform job inter-arrival period $T_i = T = \frac{1}{4}d_{min}$. All jobs arriving at time nT $(n \in \mathbb{N})$ are processed as a batch, with an effective deadline of $(n-1)\,T + d_{min}$.

3.2 Schedulability

Schedulability on the GPU in our execution model is the ability to process job batches not later than their effective deadlines, i.e. the earliest deadlines of the jobs in them. As described in Section 3.1, each batch goes through a four stage pipeline: data aggregation, data transfer to local GPU memory, kernel execution, and transfer of results to main memory. For a set of GPU-assigned streams $S = \{s_i : \langle r_i, d_i \rangle\}_{i=1}^{N}$, with minimum deadline $d_{min} = min_i d_i$, data is aggregated from streams in S and forms a new batch of jobs every $\frac{1}{4}d_{min}$. S is schedulable on the GPU if the longest of the last three pipeline operations does not exceed

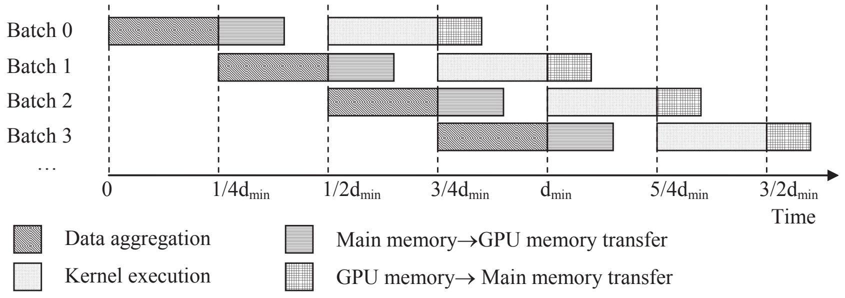


Figure 4: Batches are processed in a four-stage pipeline

$\frac{1}{4}d_{min}$. The duration of each operation can be estimated using a GPU performance model. We test GPU schedulability for S in 3 steps:

1. Calculate job sizes in a batch $W = \left(w_i | w_i = \frac{1}{4}d_{min}r_i\right)$.
2. Estimate T_D, T_K, T_R – the times of data transfer, kernel execution and results transfer – using a GPU performance model.
3. S is schedulable if $max\,(T_D, T_K, T_R) \leq \frac{1}{4}d_{min}$; otherwise we consider it as not schedulable.

3.3 GPU empirical performance model

The GPU performance model, or more specifically, its schedulability function, is used by our partitioning algorithm to estimate the kernel time for processing a batch of jobs. It is not a general model: it was built specifically for our batch execution algorithm. In the model we assume that the jobs in a given batch are distributed among the SMs so that each SM is assigned the same number of jobs, regardless of job size. Such distribution is equivalent to a random one which ignores load balancing considerations. Hence the runtime estimate for this distribution gives us an upper bound on the expected runtime for a distribution which is optimized for load balancing. The problem is thus reduced to estimating the runtime of the jobs on a single SM, and taking the longest one among the SMs as the estimate of the runtime of the whole batch on a GPU.

We now consider execution of jobs on an SM. For convenience, we express the performance of an SM as utilization, namely, the fraction of the maximum throughput achievable by a single SM. The maximum throughput for a given GPU stream processing kernel can be easily obtained by measuring the throughput of that kernel on tens of thousands of identical jobs.

We call *a utilization function* the dependency of the SM utilization on the number of jobs invoked on that SM. The utilization function can be calculated by means of a lookup table generated by benchmarking a GPU kernel for different numbers of jobs. Figure 5 demonstrates the utilization function for the AES-CBC kernel with four threads per data stream. The saw-like form of this graph is due to the execution of multiple jobs together in a single warp.

When running jobs of different sizes the utilization function becomes more complex. The number of running jobs on an SM dynamically decreases during execution, and thus the utilization changes as well. Consider the example in Figure 6. It shows 6 jobs of different sizes invoked at the same time. Utilization at a certain time depends on the number of incomplete jobs. In this example, the execution can be divided into 3 utilization periods. During the execution of job_1, the utilization is $U(6) = 12\%$. The time of this period is $t_1 = 6 \cdot |job_1| \cdot (12\% \cdot FLOPS_{SM})^{-1}$, where the size of job_1 is measured in FLOPs, and $FLOPS_{SM}$ is the maximum throughput of a single SM in FLOPs/sec. Four jobs are left in the second time period. The length of this period is $t_2 = 4 \cdot (|job_3| - |job_1|) \cdot (8\% \cdot FLOPS_{SM})^{-1}$. The length of period t_3 is calculated similarly. SM processing time is the sum of all these periods.

Given a batch of jobs $W = \left(w_i | w_i = \frac{1}{4}d_{min}r_i\right)$, we can now estimate the kernel processing time T_K, as defined in Section 3.2. To estimate the other batch processing times, T_D and T_R, the performance model uses a lookup table that contains a column of data chunk sizes and two columns for achievable transfer rates (one for each direction). Figure 13 shows transfer rate benchmark results for the configuration described in Section 5.1. In our system, incoming data for GPU-assigned streams is automatically stored contiguously in memory. This is achieved by allocating contiquous blocks of memory (256 B) to store incoming data from these streams. Consequently, the data for a whole batch can be transferred to the GPU as a data chunk in a single operation.

We evaluate the precision of our model in Section 5.6.

4. Distribution of workload between CPUs and GPUs

To achieve maximum utilization of the system, the workload is distributed between the computational devices – CPUs and GPUs. Methods for real-time scheduling on multiprocessors (CPUs) have been extensively studied, including global scheduling techniques and task migration [20]. It is thus natural for CPUs to share a workload. For GPUs, such sharing is inapplicable for two reasons: (1) Synchronization between GPUs is done explicitly by a program running on the CPU,

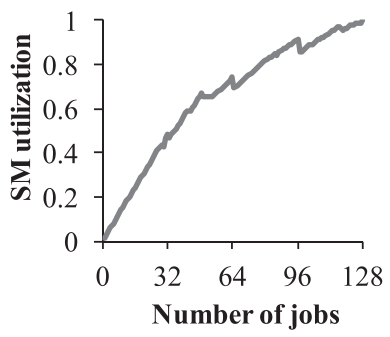


Figure 5: Non-linear SM utilization

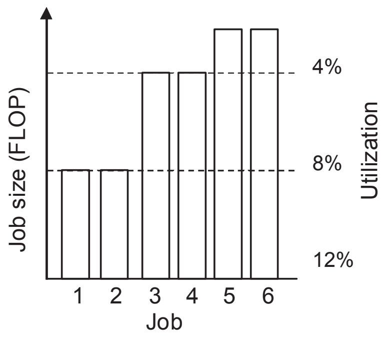


Figure 6: Effective utilization of jobs with different sizes

and (2) GPUs do not share the same device memory. For these reasons, the same limitations that made global scheduling inapplicable for a CPU and a GPU in Section 3 make it inapplicable to multiple GPUs. Consequently, with multiple CPUs and GPUs, the workload is to be partitioned to several sub-sets – one for the CPUs and one for each of the GPUs. In [22], the *Rectangle* method was presented for workload distribution between multiple CPUs and a single GPU. By design, this method does not support multiple GPUs as it partitions the workload to two sets – one for the CPUs and one for the GPU.

We present a method for workload distribution between multiple CPUs and GPUs. First, we reduce the problem to distribution between multiple CPUs and a single GPU. Then, we use the Rectangle method to find a schedulable distribution of the workload to two sub-sets: one for the CPUs and another for a (virtual) GPU. Last, we use scheduling information that we collect from the Rectangle method to further partition the second sub-set to per-GPU sub-sets.

4.1 Rectangle Method

We now briefly explain the Rectangle method. The method assigns all streams inside a contiguous area in rate-deadline space to the GPU, and other streams to the CPUs. In this method, the area is bounded by a rectangle, defined by four

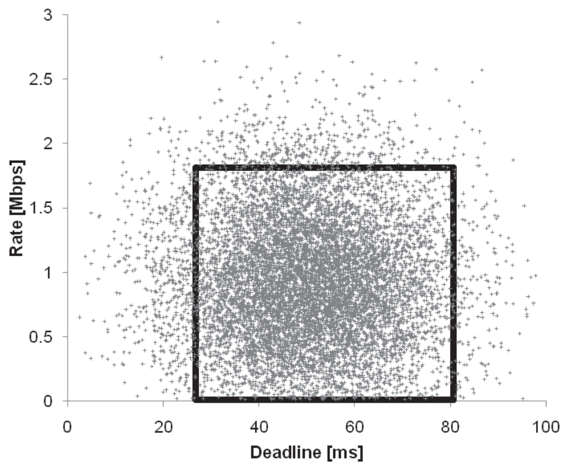


Figure 7: A partition of 10,000 data streams (represented by dots). Streams within the rectangle are assigned to the GPU. Rates are normally distributed with mean $\mu_R = 1$Mbit/sec and standard deviation $\sigma_R = 0.5$Mbit/sec. Deadlines are similarly distributed with $\mu_D = 50ms$ and $\sigma_D = 15ms$.

coordinates: $[d_{low}, d_{high}, r_{low}, r_{high}]$. For example, Figure 7 shows a partition of 10,000 streams with normally distributed rates and deadlines. Each dot represents a stream. All streams inside the black rectangle are assigned to the GPU, while the others are assigned to the CPUs. We see that a stream is assigned to the GPU if its rate is in the range [0bit/sec, 1.8Mbit/sec] and its deadline is in the range $[27ms, 81ms]$. The lower deadline bound $d_{low} = 27ms$ is also an upper bound on the processing time of a batch of jobs on the GPU (d_{min}). The method attempts to find a rectangle, such that the streams inside the rectangle are schedulable on the GPU, and the rest are schedulable on the CPUs. The method uses a performance model to test schedulability (Section 3.2).

4.2 Multiple GPUs

For multiple GPUs, we apply the Rectangle method with a virtual GPU configuration. This virtual GPU represents the unification of the compute and memory resources of all the GPUs in the system under a single GPU device. For example, the unification of two GPUs with 30 SMs each, 1 GByte of on-board memory and a 3 GByte/sec PCIe connection would be represented by a virtual GPU with 60 SMs, 2 GByte of on-board memory and a 6 GByte/sec PCIe connection.

Given some workload, if the Rectangle method produces a valid CPU/GPU partition, then it found the GPU partition schedulable on the virtual GPU. Figure 8 illustrates a part of the schedulability test where streams are partitioned between SMs (mapped to SMs). If the workload is found schedulable, then the streams are partitioned between the CPUs and (non-virtual) GPUs. Each stream in the virtual-GPU partition is

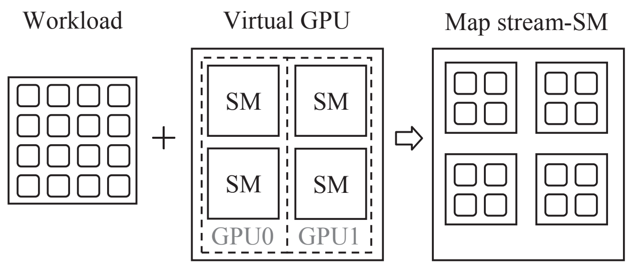


Figure 8: Workload distribution to multiple GPUs. Streams are partitioned between the SMs of a virtual GPU. This maps each stream to an SM that corresponds to a real GPU.

assigned to the GPU that corresponds to the SM it is mapped to.

When jobs are actually processed on a GPU, the produced stream-to-SM mapping is not enforced since work on the GPU is scheduled by a *hardware* scheduler. The estimated execution time could thus be shorter than the actual time. This error could result in a false categorization of a workload as schedulable, which is not acceptable. The performance model avoids producing shorter estimations by using a sub-optimal scheduling policy – it assigns the same number of randomly chosen streams to each SM. In contrast, the hardware scheduler of a GPU dynamically schedules thread blocks for execution when resources become available, and although the scheduling policies of GPUs are not disclosed, we assume that they are superior to the simple scheduling policy of the performance model.

When mapping streams to SMs, the method does not attempt to prioritize SMs of one GPU over another. As a result, the streams with deadlines that are equal or close to d_{min} are distributed among all GPUs. Effectively, on every GPU, the minimum stream deadline is approximately d_{min}. Since the batch collection period is $\frac{1}{4}d_{min}$, batch processing on the GPUs is *synchronous*.

This does not have to be the case. In fact, it might be beneficial to use different batch collection periods for different GPUs. For example, suppose that a workload S with a minimum stream deadline $d_{min} = 8ms$ is partitioned between GPU1 and GPU2. Using the described method, the collection period for both GPUs is $8/4 = 2ms$. However, if all streams with deadlines in $[8ms, 20ms]$ are assigned to GPU1 and all streams with deadlines higher than $20ms$ to GPU2, then the batch collection period will be $2ms$ for GPU1, and $5ms$ for GPU2. The throughput of GPU2 might then be higher in the second case. However, our experiments on two GPUs show that using different batch collection periods in our system is currently not possible. When using different collection periods for the GPUs, the data transfer time varies significantly between iterations. Experimentally, the latency of a data transfer to a GPU, when performed while another data transfer (to another GPU) is being executed, might increase by as much as the latency of the other operation. This problem was observed on two different desktop systems with GeForce 200 and Fermi series GPUs. We leave to future work solving this problem and developing methods that benefit from different collection periods for different GPUs.

5. Evaluation

In this section we evaluate our framework using modern hardware on thousands of streams.

5.1 Experimental platform

Our platform was an Intel Core2 Quad Q8200 2.33Ghz CPU and two NVIDIA GeForce GTX 285 GPU with CUDA 4.1. Each GPU has thirty 8-way SIMD cores and is connected to the North Bridge of an Intel P45 chipset on the main board by a 3GByte/sec PCI-E bus.

The workload is based on the AES-CBC data stream, stateful encryption application. The AES 128 bit symmetric block cipher is a standard algorithm for encryption, considered safe, and widely used for secure connections and classified information transfers (e.g., SSL). Several modes of operation exist for block ciphers. CBC is the most common – it enforces sequential encryption of data blocks by passing the encryption result of one block as the input for the encryption of the following block.

The CPU implementation is Crypto++ 5.6.0 open-source cryptographic library with machine-dependent assembly-level optimizations and support for SSE2 instructions. For the GPU we developed a CUDA-based multiple stream execution of AES. The implementation uses a parallel version of AES, in which every block is processed by four threads concurrently. Streaming data is simulated by an Input Generator that periodically generates random input for each data stream according to its preset rate. During the simulation we dedicate one CPU to the input generator and another to control the GPU execution. Data is processed on two CPU cores, using the GPUs as accelerators.

We also developed a tool that estimates the system throughput for a given type of workload on arbitrary configuration of compute units. We use this tool to (1) compare the performance of our method to simpler heuristics that we consider as baseline in Section 5.3, and (2) estimate how system performance scales for up to 8 GPUs in Section 5.4.

5.2 System performance

We tested the efficiency of our method on workloads with different distributions of stream rates and deadlines in five experiments using one and two GPUs. Each workload consisted of 12,000 streams. In each workload we used the constant, normal or exponential distributions to randomly generate the rates and the deadlines. Such workloads are common in data streaming applications, and test the framework for robustness to large variance and asymmetric distribution. To avoid generating extremely high or low values, we limited generated figures to $[0.1\mu, 30\mu]$ for the exponential distribution, where μ denotes the average distribution value, and to $[0.002\mu, 100\mu]$ for the normal distribution. In each experiment, we measured the maximum throughput (sum of stream rates) for the given workload specification. The average stream rate can be calculated by dividing the maximum throughput by 12,000. For brevity, we will use the term throughput instead of maximum throughput.

Figure 9 shows the throughput of the framework with 2 CPUs and one or two GPUs, on 5 different workloads. The overall throughput with two GPUs was up to 87% higher than with one GPU, and the GPU-only throughput (the total rate processed by the GPUs) increased by up to 106%. An increase of more than 100% in GPU-only throughput may occur, together with a increase of less than 100% in CPU-only throughput, when the total speedup is not higher than 100%.

In the first experiment, equal rates and deadlines were assigned to all streams. The framework achieved the highest throughput for this workload since there is no load imbalance for GPU jobs and a relatively large batch collection period can be used. In the second experiment, the framework achieved a 97% of maximum throughput for a realistic workload of streams with normally distributed rates and constant $40ms$ deadlines. In workloads with exponential distribution of rates (experiments 3 and 5), stream rates were as high as 150 Mbit/sec. The gain in throughput from an additional GPU for such workloads is smaller because the number of very high-rate streams that can be offloaded to the CPUs is limited. Such streams can create great load-balancing problems for the GPU. In the fourth experiment, the framework finds a provably deadline-compliant schedule for a workload that contains streams with sub-millisecond deadlines, with throughput as high as 94% of the maximum.

5.3 Comparison to baseline methods

We compared our method with two baseline techniques: `MinRate` and `MaxDeadline`. `MinRate` attempts to optimize the scheduling of streams with respect to their rates, and `MaxDeadline` does the same with respect to the deadlines. In `MinRate,` streams are sorted by rate, and the streams with the highest rates are assigned to the CPUs, up to its capacity, and the rest are assigned to the GPUs. In `MaxDeadline,` streams are sorted by deadline, and the streams with the shortest deadlines are assigned to the CPUs, up to its capacity, and the rest are assigned to the GPUs. Between GPUs, streams are equally distributed in random order.

We estimated the system throughput of each method by applying a performance estimation tool that we developed. The tool simulates the distribution of workload to a set of compute units and estimates the resulting throughput. Its input is a set of CU specifications (CPUs and GPUs), the desired workload distribution method, and a specification of the workload. The tool generates a large pool of stream configurations following the given workload specification $POOL = \{s_i : \langle deadline_i, rate_i \rangle\}$ and finds the maximum N for which the workload $\{s_1, ..., s_N\}$ has a schedulable distribution (using the method) between the CUs. Internally, the CU specifications are passed to the performance model, which is used in order to determine whether a distribution is schedulable. The tool uses benchmark results and specification data of GPUs to make the performance model as precise as possible. Note that the tool uses a workload specification format in which the number of streams is not specified. Therefore, its throughput results may not match the results in Figure 9.

We ran a series of five experiments with different distribution of stream rates and deadlines for each of the compared methods. Figure 10 shows the throughput of our method (named `Virtual GPU`), `MinRate`, and `MaxDeadline`. The table below the graph describes the settings used for generating the streams.

Our method provided higher throughput in all of the experiments. For the *const* workload, all the methods provide similar throughput since the streams do not differ in deadlines and rates. `MinRate` was never better than `MaxDeadline`. Since this method ignores deadlines, it assigned streams with very short deadlines to the GPUs, thus imposing a very short and inefficient processing cycle. The performance of `MaxDeadline` is better, but the schedules it produces suffer from load-balancing problems in different levels of severity. The performance of `Virtual GPU` was higher by up to 51 % than that of `MaxDeadline`.

5.4 Scaling to more GPUs

We tested how system throughput scales with up to 8 GPUs by applying the performance estimation tool and that was described in Section 5.3. We ran a series of five experiments, the settings for which are shown in the table in Figure 10. We used configurations of two CPU cores and between 1 and 8 GPUs, with the CPU and GPU specifications that were described in Section 5.1. Since the number of CPUs is kept constant, the purpose of these experiments is to show how the throughput of the GPUs scales with the number of GPUs, rather than the total throughput.

Figure 11 shows the throughput of the system for experiment *const*. The throughput scales perfectly in the number of GPUs because every GPU processes the same number of streams as the first. Figure 12a shows the throughput of the system for *exp*. A sample of 200 such streams is illustrated in Figure 12b. The system does not scale perfectly, with x4.3 speedup in GPU-only throughput for 8 GPUs. This might be surprising at first, since streams are drawn from a pool with randomly generated values. Each additional GPU would apparently achieve similar utilization with a similar number of additional streams. However, as shown in Figure 12c, the minimum deadline of streams on the GPUs d_{min} gets shorter as the number of GPUs is increased. As a result, the batch collection period $\frac{1}{4}d_{min}$ gets shorter, and the batching overhead increases (as explained in Section 3.1).

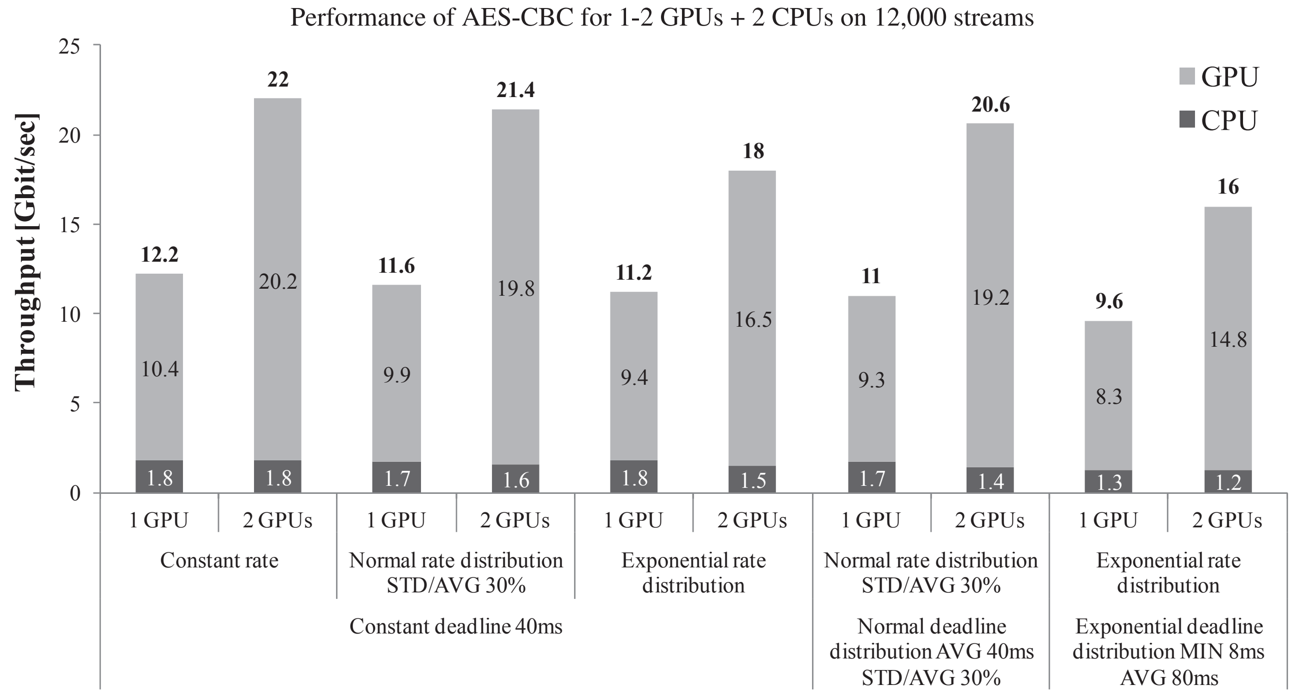
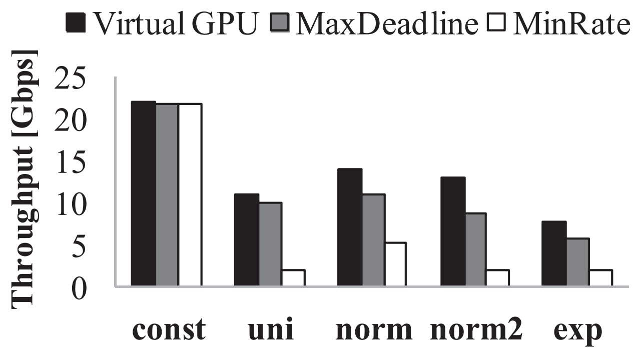


Figure 9: An additional GPU increases the total throughput by up to 87%



	Rate [Mbps]
const	3.1
uni	uniform in [0.2,6]
norm	normal with AVG=3.1 and STD=1.1
norm2	normal with AVG=1.7 and STD=0.9
exp	exponential with STD=1.1, truncated at 6 (max)

	Deadline [ms]
const	20.5
uni	uniform in [1,40]
norm	normal with AVG=20.5 and STD=7.5
norm2	normal with AVG=11.0 and STD=6.2
exp	exponential with STD=7.1, truncated at 1 (min)

Figure 10: Comparison to baseline methods for 2 GPUs

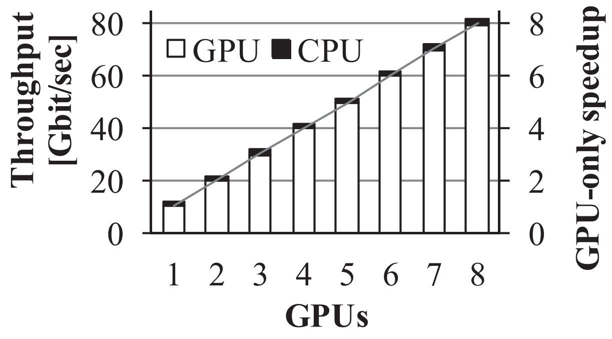
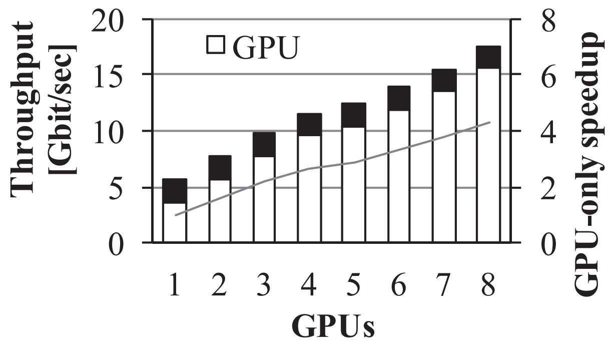


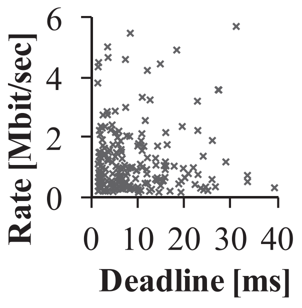
Figure 11: Perfect scaling (gray line): all streams have 20.5ms deadline and 3.1Mbit/sec rate

The reason for the decrease in d_{min} lies in the limited capacity of the CPU. As more streams with short deadlines are drawn from the pool and assigned to the CPU by the Rectangle method, streams with slightly higher deadlines that were previously assigned to the CPU get re-assigned to the GPU due to limited capacity. If the number of CPUs scaled with the number of GPUs, then we'd expect to get perfect scaling of the total throughput. In this experiment, the batch collection period gets as low as 0.4 ms.

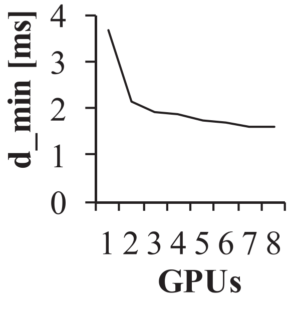
The minimum and maximum scaling observed in all of the experiments is shown in the experiments shown in figures 12a and 11, respectively.



(a) Throughput (bars) and scaling (gray line)



(b) Sample of 200 streams



(c) Min. deadline of streams on GPUs

Figure 12: GPU speedup is sub-linear due to decreasing minimum stream deadline. Deadlines and rates were generated using the truncated exponential distribution.

5.5 Transfer rates

Table 1 presents the maximum transfer rates between the main memory (Host) and the GPUs (Devices), and compares them to the maximum kernel performance for AES-CBC. One can see that the bandwidth of two GPUs is in the order of twice the bandwidth of a single GPU. This is in line with the bandwidth in the hardware model. Namely, each GPU gets half of the 16 PCI-Express lanes. These lanes connect the GPUs to the North Bridge, which is directly connected to the memory. The GeForce series GPU cards have one DMA engine, and cannot work in full-duplex. Hence, the rate of data transfer to the device and back is half the average uni-directional rate. Data transfers are a bottleneck of the system for AES-CBC when the kernel runs at maximum performance.

Figure 13 presents data transfer rates for a range of chunk sizes using one and two GPUs. For chunks of size 4 KiB or smaller the transfer time ranges around $20\mu s$, and the transfer rate is very low accordingly (up to 0.22 GByte/sec). For larger chunk sizes, the rate increases with size, reaching 95% of maximum with chunks of 2 MiB (per GPU).

As opposed to always assuming maximum rate, we use a transfer-rate lookup table in our performance model to predict data-transfer time. This is especially important when deadlines are short. Since such deadlines impose short batch processing cycles, data is transferred in small chunks, and the transfer rates are lower than the maximum.

	D-to-H	H-to-D	H-to-D-to-H	GPU kernel
1 GPU	2.77	3.355	1.517	1.875
2 GPUs	5.31	5.81	2.775	3.75
Speedup	x1.92	x1.73	x1.83	x2

Table 1: Maximum transfer rates and kernel throughput in GByte/sec ($G = 10^9$)

	Kernel		H-to-D copy		D-to-H copy	
	MIN	MAX	MIN	MAX	MIN	MAX
1 GPU	-2%	15%	-2%	17%	-3%	17%
2 GPUs	-2%	21%	-6%	-2%	-6%	2%

Table 2: Performance model estimation error

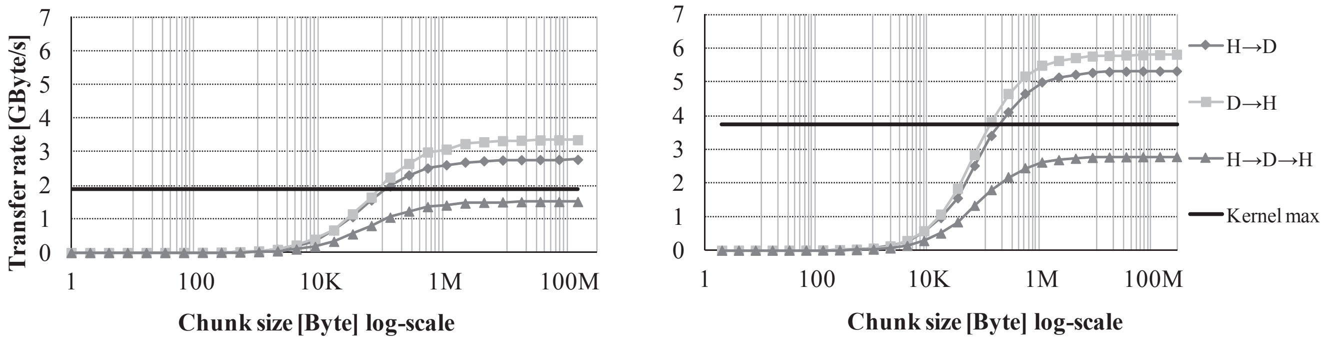
5.6 Accuracy of performance model

The estimation error of our GPU performance model for the workloads described in Section 5.2 is shown in Table 2. A negative error means that the estimation time was lower than the actual time. For example, the kernel estimation time was never lower than the actual time by more than 2%. In the framework, we leave a safety margin of 10% to account for performance model errors. As the figures show, the performance model never estimated any time by less that 94% of the actual time. When the kernel runtime is very short (less than $2\,ms$) and the batch is composed of jobs with a wide range of different sizes (experiment 5), the estimation error is large. This can be explained as follows: (1) inaccurately estimating small constant factors, which results in a large relative error when the runtime is short, and (2) deliberately having a simplistic scheduling policy in the performance model (as described in Section 3.3). Interestingly, there was a 17% error in the prediction of small data transfers with one GPU despite using a lookup table for the expected transfer rate. Apparently, the actual time of the data transfers was longer than expected since it was calculated including a CUDA device synchronization command, while in the benchmark data transfer times were measured more accurately using events.

6. Related work

Data processing of hard real-time streams on CPU/GPU systems was first described in [22]. The work presents a static method for hard real-time workload distribution between CPUs and a single GPU accelerator. A major limitation of this method is that it does not scale to multiple GPUs. Modern data processing systems are required to cope with persistently rising volumes of data, and use multiple GPU accelerators.

Transfer rate on two GTX285 GPUs with P45 chipset



(a) 1 GPU

(b) 2 GPUs

Figure 13: Data transfer rate to the GPU and back is lower than maximum kernel throughput for AES-CBC

The problem of hard real-time stream processing can be mapped to the domain of real-time periodic task scheduling. Elliot and Anderson [9] proposed global scheduling techniques for implicit-deadline streams on CPUs and GPUs. As described in Section 3, global scheduling is inapplicable to hard real-time data streams due to a high costs of CPU-GPU synchronization and state transfer.

A series of studies dealt with different aspects of task scheduling and load balancing in GPU-CPU systems. Joselli et al. [12, 13] proposed automatic task scheduling for CPU and GPU, based on load sampling. These methods are not suitable for hard real-time tasks, as there is no guarantee of task completion times. Dynamic load balancing methods for single- and multi-GPU systems using task queues and work-stealing techniques were developed[3, 6, 7, 21]. In [2, 3] a set of *workers* on each GPU execute tasks taken from local EDF (earliest deadline first) queues. These approaches cannot be used in our case as it is very hard to guarantee hard real-time schedulability in such a dynamic environment. An important work of Kerr et al. [14] describes *Ocelot*, an infrastructure for modeling the GPU. Modeling can be used in our work to create a precise performance model without benchmarking.

Kuo and Hai [15] presented an EDF-based algorithm to schedule real-time tasks in a heterogeneous system. Their work is based on a system with a CPU and an on-chip DSP. This algorithm is not compatible with our system model because of the high latency of CPU to GPU communication. The problem of optimal scheduling of hard real-time tasks on a multiprocessor is computationally-hard [18].

The term *stream processing* is often used in the context of GPUs, and refers to a programming paradigm for parallel computations. As such it is irrelevant to our work. To prevent confusion, we use the term *data stream processing*.

7. Conclusions, discussion and future work

This work presented a framework, scheduler and a performance estimation tool for a compute engine running on a heterogeneous architecture (CPU and GPU) that aims to process vast numbers of data streams under hard real-time constraints. Such an environment is commonly used by many "online big data" environments, which have recently become very popular and very profitable.

The paper presents a framework and algorithms to handle such an environment by mapping the work and deadline requirements of each stream into a common space and defining "regions" within this space that allow the system to schedule all jobs belonging to that region on a specific GPU accelerator or CPU.

We evaluated the algorithm using a real system constructed of four CPUs and two GPUs and ran AES-CBC encryption to manipulate different streams of data with a variety of deadlines. We performed extensive experiments on thousands of streams with different workloads. We used a software tool to estimate the performance of the algorithm on systems configurations with up to 8 GPUs.

Our current framework allows up to 87% more data to be processed per time unit than the one in our previous work by taking advantage of a second GPU, and up to a 106% increase in GPU-only throughput. We also show that by adding two GPUs to a quad-core machine we get a 7-fold throughput improvement over highly optimized CPU-only execution.

In the course of our experiments, we experienced different phenomena that need to be further investigated in future work. We noticed that when performing the same experiment with different combinations of main board chipsets and types of graphical cards, the data transfer time over the buses did not increase as expected, and did not distribute evenly between the cards when moving from a configuration of one card to configuration of multiple cards. For this reason, the results presented here are on a system with a relatively old main board chipset and graphical cards and not on the newest system. We are planning to further investigate this important design parameter and hope it will yield a new set of algorithms and results.

Streams of dynamically varying rates can be treated by further decreasing the complexity of the method. Batching with throughput margins may compensate for the imprecision of the accelerator performance model.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the ENCORE Project (`www.encore-project.eu`), grant agreement n° 248647.

References

[1] IDC Corporation. `http://www.idc.com/`.

[2] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. Data-aware task scheduling on multi-accelerator based platforms. In *IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS)*, 2010.

[3] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Euro-Par*, 2009.

[4] Sanjoy K. Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Systems*, 2006.

[5] Sanjoy K. Baruah, N. K. Cohen, C. Greg Plaxton, and Donald A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 1996.

[6] Daniel Cederman and Philippas Tsigas. On sorting and load balancing on GPUs. *SIGARCH Computer Architecture News*, 2008.

[7] Long Chen, O. Villa, S. Krishnamoorthy, and G.R. Gao. Dynamic load balancing on single- and multi-GPU systems. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010.

[8] UmaMaheswari C. Devi. An improved schedulability test for uniprocessor periodic task systems. *Euromicro Conference on Real-Time Systems*, 2003.

[9] Glenn A. Elliott and James H. Anderson. Globally scheduled real-time multiprocessor systems with gpus. *Real-Time Systems*, 2012.

[10] Owen Harrison and John Waldron. Aes encryption implementation and analysis on commodity graphics processing units. In *9th Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2007.

[11] Dag Arne Osvik Joppe W. Bos and Deian Stefan. Fast implementations of aes on various platforms, 2009.

[12] M. Joselli, M. Zamith, E. Clua, A. Montenegro, A. Conci, R. Leal-Toledo, L. Valente, B. Feijo, M. d'Ornellas, and C. Pozzer. Automatic dynamic task distribution between CPU and GPU for real-time systems. In *Computational Science and Engineering, 2008. CSE '08. 11th IEEE International Conference on*, pages 48 –55, 16-18 2008.

[13] Mark Joselli, Marcelo Zamith, Esteban Clua, Anselmo Montenegro, Regina Leal-Toledo, Aura Conci, Paulo Pagliosa, Luis Valente, and Bruno Feijó. An adaptative game loop architecture with automatic distribution of tasks between CPU and GPU. *Comput. Entertain.*, 7(4):1–15, 2009.

[14] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. Modeling gpu-cpu workloads and systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 31–42, New York, NY, USA, 2010. ACM.

[15] Chin-Fu Kuo and Ying-Chi Hai. Real-time task scheduling on heterogeneous two-processor systems. In Ching-Hsien Hsu, Laurence Yang, Jong Park, and Sang-Soo Yeo, editors, *Algorithms and Architectures for Parallel Processing*. 2010.

[16] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973.

[17] S.A. Manavski. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. In *Signal Processing and Communications*, 2007.

[18] Srikanth Ramamurthy. Scheduling periodic hard real-time tasks with arbitrary deadlines on multiprocessors. In *IEEE Real-Time Systems Symposium*, pages 59–, 2002.

[19] Srikanth Rarnarnurthy and Mark Moir. Static-priority periodic scheduling on multiprocessors. *Real-Time Systems Symposium, IEEE International*, 0:69, 2000.

[20] John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 1995.

[21] Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the gpu. In *High Performance Graphics*, pages 29–37, 2010.

[22] Uri Verner, Assaf Schuster, and Mark Silberstein. Processing data streams with hard real-time constraints on heterogeneous systems. In *International Conference on Supercomputing (ICS)*, pages 120–129, 2011.

[23] Andreas Weigend. The social data revolution(s). http://blogs.hbr.org/now-new-next/2009/05/the-social-data-revolution.html, May 2009.