This is the author's version of the work. The definitive work was published in *Proceedings of the 25th International Conference on Architecture of Computing Systems (ARCS), Munich, Germany, February 28–March 2, 2012.*

Dynamic Task-Scheduling and Resource Management for GPU Accelerators in Medical Imaging

Richard Membarth[1], Jan-Hugo Lupp[1], Frank Hannig[1], Jürgen Teich[1], Mario Körner[2], and Wieland Eckert[2]

[1] Hardware/Software Co-Design, Department of Computer Science,
University of Erlangen-Nuremberg, Germany.
[2] Siemens Healthcare Sector, H IM AX,
Forchheim, Germany.

Abstract. For medical imaging applications, a timely execution of tasks is essential. Hence, running multiple applications on the same system, scheduling with the capability of task preemption and prioritization becomes mandatory. Using GPUs as accelerators in this domain, imposes new challenges since GPU's common FIFO scheduling does not support task prioritization and preemption. As a remedy, this paper investigates the employment of resource management and scheduling techniques for applications from the medical domain for GPU accelerators. A scheduler supporting both, priority-based and LDF scheduling is added to the system such that high-priority tasks can interrupt tasks already enqueued for execution. The scheduler is capable of utilizing multiple GPUs in a system to minimize the average response time of applications. Moreover, it supports simultaneous execution of multiple tasks to hide data transfers latencies. We show that the scheduler interrupts scheduled and already enqueued applications to fulfill the timing requirements of high-priority dynamic tasks.

Keywords: Dynamic task-scheduling, Resource management, GPU, CUDA, Medical imaging.

1 Introduction

The exponential growth of semiconductors doubles the number of computing resources in graphics processors (GPUs) approximately every second year. This trend is reflected by larger bit widths as well as the rapidly increasing number of cores that can be integrated into a single chip. In addition, the programmability of GPUs has steadily evolved, so that they became general purpose computing devices in these days, and now often outperform the performance of standard multi-core processors. Consequently, GPUs have been adopted in many computationally intensive fields like in medical imaging. In these domains, GPUs are used not solely as rasterizers for 2D and 3D image scenes anymore, but also for image preprocessing, segmentation, registration, and other computationally intensive tasks. In particular during invasive treatments that are guided by imaging, soft real-time processing is of highest importance in order to satisfy rigid latencies and very high throughput rates—here, traditionally, multi-DSP or FPGA-based accelerators are deployed. In order to achieve the aforementioned objectives with graphics hardware, scheduling and resource management techniques for GPU accelerators are investigated in this paper.

High-performance architectures with accelerator extensions are often specifically designed for one application. Even for standard, programmable solutions, the mapping and

binding of sub-algorithms to computing/accelerator resources is static and predetermined before run-time. Only few flexible approaches exist that can overcome this limitation. For instance, the Vforce framework presented by Moore et al. in [8] is an environment for multi-node supercomputers with reconfigurable FPGA accelerators. Specific to Vforce is the ability to perform load balancing at run-time by a resource manager. However, the framework does not consider constraints imposed by the application such as timing requirements and does not provide support for GPU accelerators. A framework which accounts also for multiple accelerators such as GPUs and FPGAs was recently proposed by Beisel et al. in [3]. Here, the authors extend the GNU/Linux scheduler to support time-sharing of accelerator resources between multiple user-level applications and provide an API to use their scheduler. This allows fairness between multiple applications competing for the same accelerator resource. However, their scheduler considers no timing constraints such as task execution times and deadlines.

The fined-grained scheduling of operations and blocks [2, 5] to increase the performance of a single application has been studied extensively on graphics processors. For example, using *persistent threads* [2] is an efficient technique for load balancing for single kernels executed on graphics cards in situations where the work performed by single threads is data dependent.

However, scheduling and resource-management of multiple (heterogeneous) graphics cards and applications specified at task-level—possibly with annotated constraints—has been barely considered. Although running applications on the GPU can be interrupted and resumed using checkpointing/restarting tools [9,10], these tools allow to interrupt running applications only at synchronization points and the involved overhead makes them not suitable for time-critical applications such as considered in this paper. To the best of our knowledge, we consider for the first time a scheduling framework for GPU accelerators that considers a) multiple applications at task-level, with b) annotated constraints such as priorities, inter-task dependencies, deadlines, or execution times in order to allow c) dynamic high-priority tasks to interrupt tasks enqueued for execution. Thereby, the scheduler utilizes d) multiple heterogeneous GPU accelerators while exploiting e) the simultaneous execution of multiple kernels (like data transfer kernels and compute kernels) on the same GPU accelerator.

The remainder or the paper is organized as follows: Section 2 defines the requirements to the scheduler with the help of a scenario from medical imaging. The concepts of the scheduling framework are discussed in Section 3, while the scheduler is evaluated in Section 4. Finally, the paper is concluded in Section 5.

2 Scheduling Requirements

In this section, the requirements to the scheduler are explained using a motivating example from medical imaging.

2.1 Motivating Example

Considered is a versatile GPU-based compute server for accelerating computationally intensive tasks. A possible scenario could be an application for image preprocessing that has to be run in soft real-time during an angiography treatment with 25 frames per second (fps). This application may consist of several algorithms that have to be executed in a pipelined fashion for each image as shown in Fig. 1. The example shows a pipeline consisting of four kernels A1–A4, which may compensate defect pixels, temperature

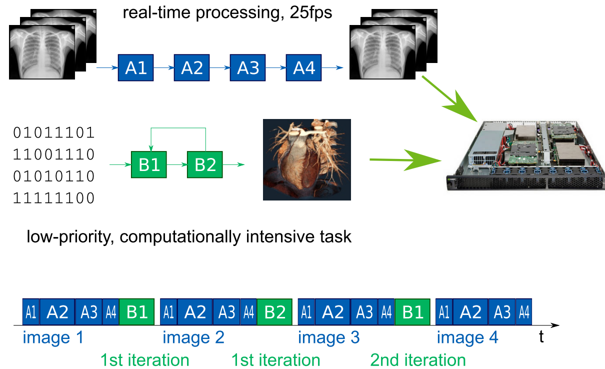


Fig. 1: Two applications *A* and *B* scheduled to a compute server. Application *A* is time-critical and therefore, the calculation of application *B* has to be interrupted to guarantee timing constraints.

effects, apply a filter, or crop the image (see [7]). Besides guaranteeing the timing constraints, the GPU server may process other low-priority tasks in the background, like a volume computation or image registration, which are not time-critical (task B1–B2 in Fig. 1). Such flexible scheduling is not possible using the current scheduling policy of graphics cards. Once tasks (also called *kernels* on the GPU) are submitted for execution to the GPU, they are stored to a queue and processed in a first in, first out (FIFO) manner. New arriving high-priority tasks are also enqueued to the same queue and have to wait until all proceeding tasks have been processed. In order to allow the desired flexible behavior, we introduce a scheduler in software that precedes the hardware scheduler of graphics cards.

2.2 Scheduling Requirements

Looking at the example from Fig. 1, the scheduler has to support at least constraints annotated to applications, the ability to dynamically interrupt tasks already enqueued for execution, and the automatic scheduling. In particular, the following requirements have to be fulfilled by a flexible scheduler in a compute-server:

- Scheduling of independent kernels (task-parallel execution)
- Explicit specification of processing sequence of kernels
- Estimation of execution time of kernels
- Interrupt of current execution and resumption (for dynamic kernels, preemptive scheduling)
- Priority-based scheduling
- Priority-based scheduling with deadlines (real-time requirements)
- Resource management (e. g., devices, memory)
- Utilization of multiple graphics cards (multi-GPU support)

From a user perspective, kernels annotated with execution constraints are passed to the scheduler. To pass kernels to the scheduler, the following aspects have to be considered and provided by the scheduler:

- A set of kernels can be submitted to the scheduler (e. g., in the form of a task graph)
- The data dependencies between the kernels can be modeled (e. g., using edges within the task graph)

– Execution constraints can be annotated to each kernel (e. g., deadlines for real-time requirements or priorities)
– New kernels can be submitted to the scheduler at run-time (i. e., dynamic scheduling)

Since the scheduler has to support data dependencies as well as real-time constraints, the scheduler is based on the *Latest Deadline First* (LDF) scheduling discipline [4]. LDF minimizes the maximal lateness of a set of tasks.

3 Scheduling Framework

This section describes the design of the scheduler and the properties of its components. The implementation of the scheduling framework provides support for CUDA, although the principles are also valid for other GPU programming interfaces. Hence, the framework can be easily extended to support for instance also OpenCL.

3.1 Scheduler Design

Kernels: the basic elements for execution on graphics cards are *kernels*. We introduce an abstract representation based on C++ classes. Each user-defined kernel inherits from a global kernel base class that allows the programmer to describe the properties of the kernel itself as well as properties required for scheduling. The scheduling related properties are as follows: a) priority, b) deadline, c) worst-case execution time (WCET), d) stream, and e) event. The programmer has to specify the priority, deadline, and worst-case execution time of a kernel. This information is used to determine the schedule of all kernels. The stream and event associated with a kernel, are set by the scheduler: *streams* are required to allow overlapped kernel executions and data transfers as well as concurrent execution of multiple kernels on Fermi hardware [1]. *Events* are required to synchronize between different streams in order to preserve data dependencies. The assignment of streams to kernels and the utilization of events is described later on. The invocation of the CUDA kernel itself is specified in the *run()* method of each kernel. This method is called by the scheduler whenever the kernel is scheduled for execution on the graphics hardware. Initialization for a kernel can be expressed in the *constructor* of the kernel class. The kernel can be a compute kernel as well as a data transfer kernel.

Data dependencies: to specify data dependencies between kernels, a directed acyclic graph (DAG) is used as seen in Fig. 2. Each kernel is a node in the DAG and each edge in the DAG denotes a data dependency. In addition, a *no operation* (NOP) node is added as entry point for a tree of kernels. In the example DAG, kernels for data allocation, data transfers, and computations are used.

Streams: graphics cards allow the concurrent execution of data transfer kernels and compute kernels. In order to utilize this feature, the kernels that should be overlapped, have to be in different streams. Streams represent different execution queues on the graphics card. In case there are two kernels from different streams ready for execution (with different type—data transfer and compute), they are executed at the same time. Fermi hardware allows even the same kernel to be executed at the same time (e. g., multiple compute kernels from different streams or data transfer kernels to and from graphics card memory). The scheduler supports streams and assigns streams automatically in order to benefit from concurrent kernel execution. To do so, the scheduler scans the whole DAG from the root to the bottom and checks if data dependencies allow to use different streams. That is, if a node has more than one successor, all successors can be executed

Algorithm 1: Assignment of streams to tasks for concurrent kernel execution.

```
Input : Set of nodes n of task graph T
1  foreach node n of task graph T do
2  |   in_edges ← get_in_edges(n);
3  |   if in_edges > 1 then
   |   |   // use new stream - simplifies synchronization later on
4  |   |   stream ← cudaCreateStream();
5  |   else
6  |   |   p ← get_parent(n);
7  |   |   out_edges ← get_out_edges(p);
8  |   |   if out_edges > 1 then
9  |   |   |   stream ← cudaCreateStream();
10 |   |   else
11 |   |   |   stream ← get_stream(p);
12 |   |   end
13 |   end
14 |   set_stream(n, stream);
15 end
```

concurrently. In Algorithm 1, this is done by checking if the parent node has more than one successor (lines 6–12). Iff this is the case, a new stream gets created and assigned to the node (line 9), otherwise, the stream from the parent node will be used (line 11). In case a node has more than one predecessor, the stream of one of the predecessors can be reused and it has to be ensured that all predecessors have finished their execution (discussed in the next paragraph). In order to simplify the synchronization, the scheduler assigns in this case also a new stream to the node (line 3–4).

While the algorithm assigns streams such that data transfer kernels can be overlapped with compute kernels, synchronization, if required, is added later on. The graphics card scheduler may also use a different schedule and overlap the data transfer kernels with the compute kernels. The scheduler assigns streams so that both possibilities are given to the scheduler of the graphics card.

Synchronization: once streams are assigned to the nodes, synchronization has to be added to nodes where data dependencies exist and different streams are used. Synchronization between two streams requires two steps and is realized using *events* in CUDA. An event is handled like a normal kernel, enqueued into a stream using *cudaEventRecord()*, and executed as soon as all previously enqueued kernels to the stream are finished. Events take just the current time stamp on the graphics card and can be used for timing analysis and for synchronization. In case of synchronization, *cudaEventSynchronize()*—blocking until the specified event was triggered—can be used to ensure a task has finished: the event has to be recorded after the last kernel in a stream has been executed, and the second stream has to synchronize for the recorded event before executing the first kernel. The event has to be recorded whenever a series of kernels assigned to the same stream ends. The first part of Algorithm 2 describes the two cases when this is necessary: a) in case a node has more than one successors (line 3–6) and b) in case the node has one child and a different stream was assigned to the child (line 8–13: this is the case iff the child has more than one predecessor and, hence, a different stream was assigned in lines 3–4 of Algorithm 1).

After the events have been added, the successor nodes in different streams have to synchronize on these events. Algorithm 3 describes how this is done: when a node has more than one predecessor, it synchronizes to the event of each of the predecessors (line 2–6). In case the node has only one parent node, synchronization is only needed if the

Algorithm 2: Assignment of events to tasks for synchronization of streams.

```
   Input: Set of nodes n of task graph T
1  foreach node n of task graph T do
2      out_edges ← get_out_edges(n);
3      if out_edges > 1 then
4          stream ← get_stream(n);
5          event ← cudaEventRecord(stream);
6          set_event(n, event);
7      else
8          c ← get_child(n);
9          if get_stream(n) != get_stream(c) then
10             stream ← get_stream(n);
11             event ← cudaEventRecord(stream);
12             set_event(n, event);
13         end
14     end
15 end
```

parent node uses a different stream (line 9–12: the current node is not the only successor of the parent node).

Algorithm 3: Events synchronization for tasks of streams.

```
   Input: Set of nodes n of task graph T
1  foreach node n of task graph T do
2      in_edges ← get_in_edges(n);
3      if in_edges > 1 then
4          foreach predecessor p of node n do
5              event ← get_event(p);
6              cudaEventSynchronize(event);
7          end
8      else
9          p ← get_parent(n);
10         if get_stream(n) != get_stream(p) then
11             event ← get_event(p);
12             cudaEventSynchronize(event);
13         end
14     end
15 end
```

3.2 Dynamic Scheduling

Once streams are assigned to the nodes of the DAG and synchronization events are assigned, the LDF scheduling is applied and a task list for the schedule is created. This task list and the corresponding DAG is passed from the scheduler to a dedicated host thread responsible for scheduling kernels to the graphics card. This thread starts enqueueing kernels from the task list to the graphics cards, using the assigned stream of each kernel. Before the kernel is enqueued, event synchronization is applied if necessary. Likewise, events are recorded after the kernel has been issued if necessary. If the scheduler enqueues all tasks in the task list to the graphics card, the flexibility to interrupt the execution of the current task graph, for example for a high-priority task—is lost. Therefore, the scheduler issues only kernels that take a certain, adjustable, amount of time for execution.

When a new task or task tree arrives at the scheduler, it can now create a new schedule for the remaining tasks of the old tree and the tasks of the new tree. This is done by

inserting the new tree as a child of the NOP node and calculating a new schedule using LDF. Adding the new tree as a child to the NOP node of the existing tree assures that no data dependencies exist between the old and new task tree.

3.3 Multi-GPU Support

The scheduler supports multiple GPUs: at startup, two threads are created per GPU. One for issuing kernels from the kernel list to the GPU and one for building the task tree, applying LDF to the task tree, and providing the task list (LDF schedule) to the GPU thread. When a task tree arrives to the scheduler, it picks one of the available GPUs for execution. The GPU is chosen so that the average response time (of all task trees) is minimized. Therefore, the execution time for complete task trees is required. This is the sum of the worst-case execution time annotated to each kernel in the tree. For partially processed task trees, the scheduler updates the time required to execute the remaining tree. Since the execution time of a kernel is not the same on different GPUs, the WCET has to be provided for each GPU in the system.

In the current implementation, only complete task trees, which are independent of other task trees, are scheduled to different GPUs. Kernels from within one task tree are always scheduled for execution on the same GPU in order to avoid costly data transfers between multiple devices.

4 Results

To evaluate the scheduler, we use a system with three graphics cards: two high-end Tesla C2050, and one low-end Quadro FX 1800 (required by the BIOS to boot). While the two Tesla cards feature the most recent Fermi architecture, the Quadro card has still the original Tesla design of the first CUDA-capable cards. This gives us a system setup, where the performance of the scheduler for a single GPU, as well as for multi-GPU can be evaluated. In addition, using two different types of GPUs, allows us to show that the scheduler minimizes the average response time of all task trees. For the execution time of kernels, we use the maximal measured execution time of 100 runs rather than the analytically determined WCET.

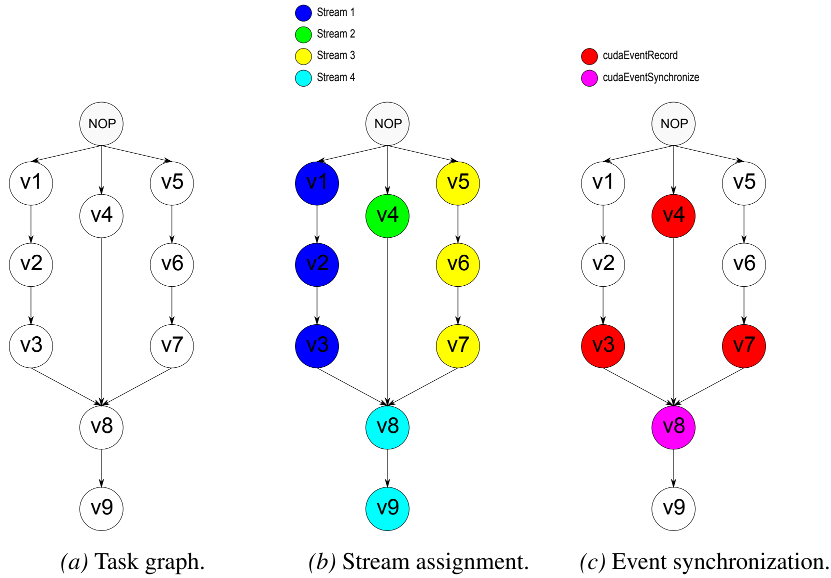
The scheduler can be seen as a compute server, accepting applications that should be executed on a graphics card. Such an application is described by a task tree, which describes the data dependencies between all kernels. While a single application is often optimized for exclusive execution on the available resources (GPU), the scheduler assures that multiple of such selfish applications can be executed at the same time and that the average response time of all applications is optimized.

4.1 LDF Scheduling

To show how the scheduler assigns streams and events to different kernels, we use a simple application multiplying two matrices with a constant, and adding the two matrices afterwards. The task graph of the matrix calculation example is shown in Fig. 2(a). The size of each matrix is 1000×1000 elements.

The kernels associated with each node are as follows:

- *v1*, *v4*, *v5*: cudaMalloc
- *v2*, *v6*: cudaMemcpy(HostToDevice)
- *v3*, *v7*: matrixMul
- *v8*: matrixAdd



(a) Task graph. *(b)* Stream assignment. *(c)* Event synchronization.

Fig. 2: Shown is the task graph for the matrix calculation example (a) as well as the streams assigned by the scheduler (b) and the added event synchronizations (c).

- *v9*: cudaMemcpy(DeviceToHost)

Given the task graph of Fig. 2(a), the scheduler assigns streams to the nodes and adds event synchronization as seen in Fig. 2(b) and 2(c).

In order to determine the schedule using LDF, the execution time of each kernel has to be known as well as the deadline for each task. For priority scheduling, a priority is assigned to each kernel. Table 1 lists the nodes from the matrix calculation example, their measured maximal execution time on the two considered graphics cards, as well as their deadline and priority.

In order to visualize how the scheduler issues kernels, which streams are used, and what time is spent for which kernel, we write the timings of all kernels to a file. The format of that file is in such a way that the profiler from NVIDIA can read the files and display them. By doing that, overlapping data transfers are not visible in the profile, since we synchronize after each kernel invocation in order to get accurate timings. Figure 3

Table 1: Execution times in milliseconds of kernels from the matrix calculation example.

	v1	v2	v3	v4	v5	v6	v7	v8	v9
WCET (Tesla)	0.15	0.70	0.53	0.15	0.15	0.70	0.53	0.60	1.30
WCET (Quadro)	0.12	1.70	1.00	0.12	0.12	1.70	1.00	1.10	1.88
deadline	1	3	5	7	2	4	6	8	9
priority	9	6	5	8	7	4	3	2	1

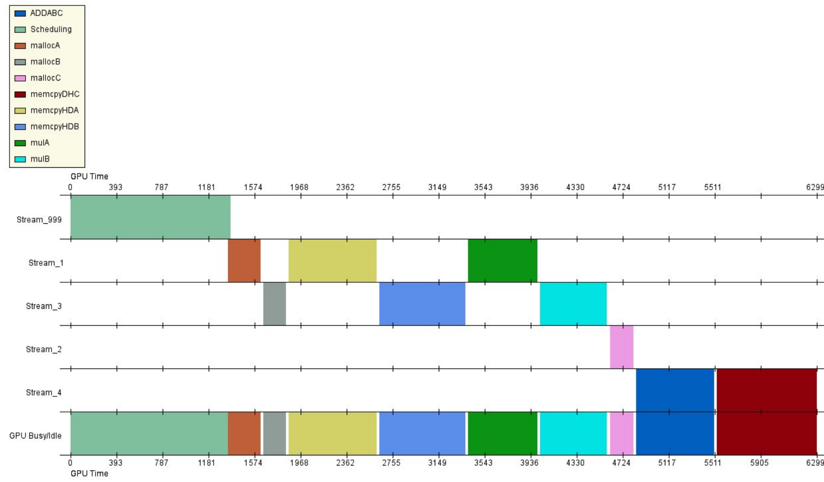


Fig. 3: Profile of the matrix calculation example on a Tesla C2050.

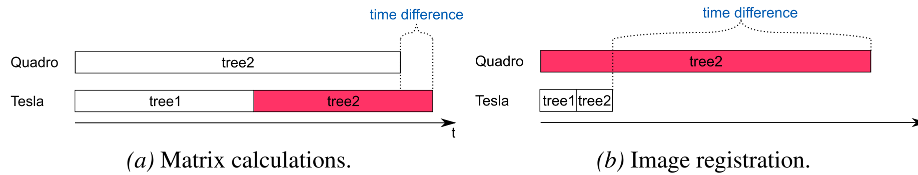
shows that profile for the LDF schedule. We assign always the stream with id 999 and 998 to activities of the scheduler. Each of these activities corresponds to creating the LDF schedule and the task list for one graphics card. That is, when only one of these streams is present, only one GPU is used. The execution time for the task tree of Fig. 3 takes 4.28 ms if no asynchronous data transfers are used. Otherwise, the execution time is reduced to 3.78 ms—0.5 ms less, which is the execution time of the matrix multiplication of node *v3* being executed concurrently with the data transfer kernel of node *v6*.

4.2 Multi-GPU Support

To evaluate the behavior when multiple graphics cards are available, we submit more than one task tree to the scheduler. The scheduler uses the annotated WCET to decide which GPU should be used. The matrix calculations take 4.81 ms on the Tesla and 8.74 ms on the Quadro. We use the 2D/3D image registration from the domain of medical imaging as second application [6, 11]. The image registration is an iterative algorithm that tries to align a 3D volume with a 2D image and we limit the number of iterations to 64 for our experiments. The image registration takes 2107.80 ms on the Tesla and 22199.40 ms on the Quadro. When we submit more than one application to the scheduler, this WCET times are used to decide which GPUs to use.

Figure 4 shows the benefits of using the Quadro in addition to one Tesla card when each of the two application is submitted twice. Although the matrix calculation takes almost double the time on the Quadro (8.74 ms) compared to the Tesla (4.81 ms), it is beneficial to use both cards to minimize the average response time. However, in case of the 2D/3D image registration, the Quadro (22199.40 ms) card takes ten times as long as the Tesla (2107.80 ms), and there is no benefit of using both cards.

Executing exactly these two scenarios, we get the schedule as shown in Fig. 5 and 6. In case of the matrix calculations, two host threads are used for scheduling, and two GPUs are used for execution. The GPU associated to stream 1 is the faster Tesla, and the GPU associated to stream 2 is the slower Quadro. When the two instances of the image



(a) Matrix calculations. *(b)* Image registration.

Fig. 4: Response time when using a Tesla C2050 and a Qaudro FX 1800 for the matrix calculation example (a) and the image registration (b). For the matrix calculations using both cards minimizes the average response time, while using only the Tesla card minimizes the average response time for the image registration.

registration are used, both are scheduled to the faster Tesla and all the scheduling is done by one thread. This takes 4288.73 ms.

Using both Teslas and the Quadro for the two image registration instances, both Tesla cards are utilized as seen in Fig. 6, requiring 2116.76 ms in total.

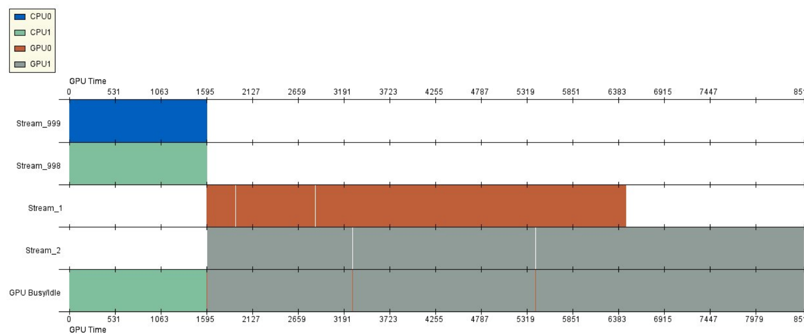
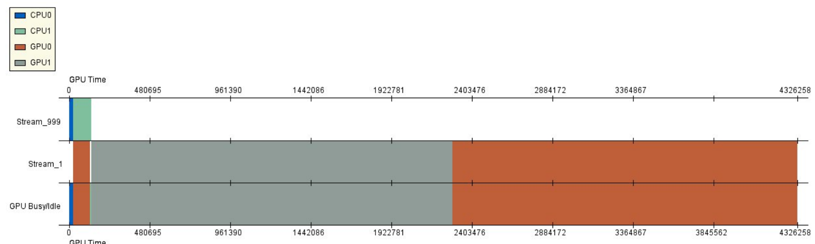


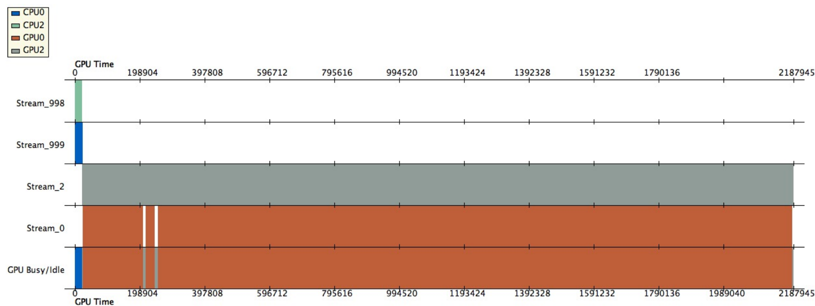
Fig. 5: Profile of two matrix calculation task trees scheduled on a system with one Tesla C2050 and one Quadro FX 1800. The scheduler uses both GPUs to minimize the average response time.

4.3 Dynamic Scheduling

Dynamic scheduling behavior is required when tasks already enqueued for execution are interrupted by a high-priority task. For this scenario, we use the image registration as long-running application, which is periodically interrupted by small matrix calculations. Only one Tesla card is used here, so that the long-running application gets interrupted by the high-priority tasks, rather than scheduling the high-priority tasks to the second Tesla. Figure 7 shows the schedule for this scenario: as soon as the high-priority task arrives, a new schedule is calculated using LDF. While the new schedule is calculated, the GPU thread continues to issue tasks from the old tree to the graphics card. Once the new task list is calculated, the GPU thread switches to the new task list, removes the already processed kernels from this list and continues with the next kernel from the list—kernels from the high-priority task. Almost no time is wasted since the GPU thread continues scheduling kernels of the old task list while the new schedule is created.



(a) 1× Tesla 2050 and 1× Quadro FX 1800.



(b) 2× Tesla C2050.

Fig. 6: Profile of two image registration task trees scheduled on a system with (a) one Tesla C2050 and one Quadro FX 1800 and (b) two Tesla C2050. The scheduler uses in (a) only the Tesla GPU to minimize the average response time, while in (b) both Tesla GPUs are used.

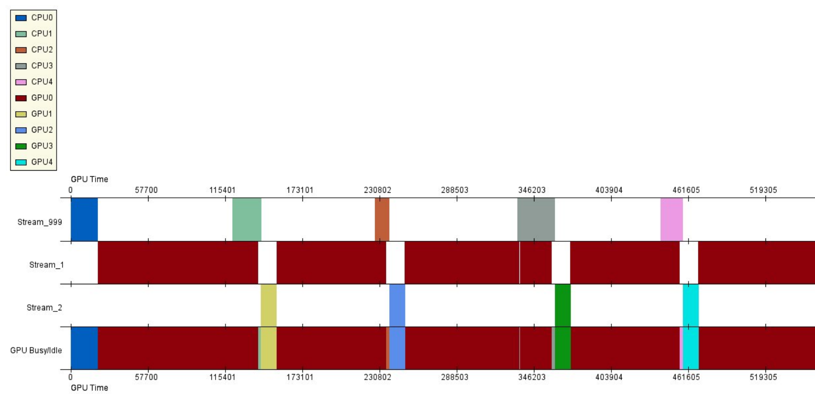


Fig. 7: Profile of the image registration, with periodically arriving high-priority tasks: the scheduler interrupts the image registration and schedules the high-priority tasks and resumes the execution of the image registration.

5 Conclusions

In this paper, we introduced a scheduling framework for GPU accelerators in the domain of medical imaging with timing requirements. The proposed scheduler supports both, priority-based and LDF scheduling with support for high-priority tasks that can interrupt tasks already enqueued for execution. This dynamic behavior is not possible using GPU-only scheduling. In case the system hosts multiple graphics cards, it has been shown that the scheduler assigns incoming applications to the available GPUs so that the average response time is minimized. Assigning streams automatically to independent kernels, the concurrent execution of data transfer kernels and compute kernels is supported, too. To ensure that data dependencies are met, the scheduler uses events to synchronize the assigned streams.

While it has been shown that the proposed scheduler improves the performance of medical applications and supports dynamic high-priority tasks, the scheduler is not aware of other applications running on the GPU, like the X-server. Such applications that bypass our scheduler may have negative impact on the scheduling behavior. Instead of providing the scheduling support as a library to the user, it could be also integrated into the device driver speaking directly to the graphics hardware (see [3]).

References

1. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. White Paper (Oct 2009)
2. Aila, T., Laine, S.: Understanding the Efficiency of Ray Traversal on GPUs. In: Proceedings of the Conference on High Performance Graphics (HPG). pp. 145–149. ACM (Aug 2009)
3. Beisel, T., Wiersema, T., Plessl, C., Brinkmann, A.: Cooperative Multitasking for Heterogeneous Accelerators in the Linux Completely Fair Scheduler. In: Proceedings of the 22nd IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP). pp. 223–226. IEEE Computer Society, Santa Monica, CA, USA (Sep 2011)
4. Buttazzo, G.C.: Hard Real-time Computing Systems. Kluwer Academic Publisher, Boston, MA, USA (1997)
5. Fung, W., Sham, I., Yuan, G., Aamodt, T.: Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (Micro). pp. 407–420. IEEE Computer Society (Dec 2007)
6. Membarth, R., Hannig, F., Teich, J., Körner, M., Eckert, W.: Frameworks for GPU Accelerators: A Comprehensive Evaluation using 2D/3D Image Registration. In: Proceedings of the 9th IEEE Symposium on Application Specific Processors (SASP). pp. 78–81 (Jun 2011)
7. Membarth, R., Hannig, F., Teich, J., Litz, G., Hornegger, H.: Detector Defect Correction of Medical Images on Graphics Processors. In: Proceedings of the SPIE: Medical Imaging 2011: Image Processing. vol. 7962, pp. 79624M 1–12 (Feb 2011)
8. Moore, N., Conti, A., Leeser, M., King, L.: Vforce: An Extensible Framework for Reconfigurable Supercomputing. Computer 40(3), 39–49 (Mar 2007)
9. Nukada, A., Takizawa, H., Matsuoka, S.: NVCR: A Transparent Checkpoint-Restart Library for NVIDIA CUDA. In: Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW). pp. 104–113. IEEE (May 2011)
10. Takizawa, H., Sato, K., Komatsu, K., Kobayashi, H.: CheCUDA: A Checkpoint/Restart Tool for CUDA Applications. In: Processing of the 10th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT). pp. 408–413. IEEE (Dec 2009)
11. Weese, J., Penney, G., Desmedt, P., Buzug, T., Hill, D., Hawkes, D.: Voxel-Based 2-D/3-D Registration of Fluoroscopy Images and CT Scans for Image-Guided Surgery. IEEE Transactions on Information Technology in Biomedicine 1(4), 284–293 (1997)