A Static Task Scheduling Framework for Independent Tasks Accelerated using a Shared Graphics Processing Unit

Teng Li, Vikram K. Narayana, Tarek El-Ghazawi
NSF Center for High-Performance Reconfigurable Computing (CHREC),
Department of Electrical and Computer Engineering
The George Washington University
Washington, DC, USA
{tengli, vikram, tarek}@gwu.edu

***Abstract*—The High Performance Computing (HPC) field is witnessing the increasing use of Graphics Processing Units (GPUs) as application accelerators, due to their massively data-parallel computing architectures and exceptional floating-point computational capabilities. The performance advantage from GPU-based acceleration is primarily derived for GPU computational kernels that operate on large amount of data, consuming all of the available GPU resources. For applications that consist of several independent computational tasks that do not occupy the entire GPU, sequentially using the GPU one task at a time leads to performance inefficiencies. It is therefore important for the programmer to cluster small tasks together for sharing the GPU; however, the best performance cannot be achieved through an ad-hoc grouping and execution of these tasks. In this paper, we explore the problem of GPU tasks scheduling, to allow multiple tasks to efficiently share and be executed in parallel on the GPU. We analyze factors affecting multi-tasking parallelism and performance, followed by developing the multi-tasking execution model as a performance prediction approach. The model is validated by comparing with actual execution scenarios for GPU sharing. We then present the scheduling technique and algorithm based on the proposed model, followed by experimental verifications of the proposed approach using an NVIDIA Fermi GPU computing node. Our results demonstrate significant performance improvements using the proposed scheduling approach, compared with sequential execution of the tasks under the conventional multi-tasking execution scenario.**

Keywords-GPU; scheduling; multi-tasking; resource sharing

I. INTRODUCTION

Modern High Performance Computer architecture has evolved from the traditional homogeneous architecture equipped with only CPUs to the modern heterogeneous architecture incorporating CPUs and hardware accelerators. Due to the rapid technological advancements in the graphics processing field over the past few years, Graphics Processing Units (GPUs) have been increasingly used as co-processors in High Performance Computing (HPC). While the use of GPUs sacrifices programming generality, it provides massively parallel computing and programming architecture as well as significantly improved floating-point computing performance. As a result, a wide range of HPC systems have incorporated GPUs as co-processors to achieve better performance for various applications. Example systems include the latest top ranking supercomputer in the Top 500 list such as Tianhe-1A (Rank 2nd), which is equipped with NVIDIA Tesla M2050 GPUs and able to achieve a sustained 2.57 PFlop/s LINPACK performance [1].

In order to use GPUs within an application, current programming models are based on the use of a separate programming language for the GPU. Execution of GPU tasks therefore relies on function calls in the main application process running on the CPU. Since the main control process resides on the CPU, there are overheads associated with transferring data back and forth between the CPU and GPU memory. Nevertheless, substantial gains in performance can be achieved due to the massive computational capability of the most recent GPU, with long computations on the GPU amortizing the data transfer overheads. Needless to say, the best performance in this mode of computation is obtained when tasks make the use of all the available GPU computational resources or processing cores.

The number of processing cores utilized by a GPU task is directly related to the processed data size, due to the Single-Instruction, Multiple-Thread (SIMT) programming approach followed for GPUs [2]. Although programmers attempt to maximize the use of the available SIMT parallelism, oftentimes applications have tasks that do not utilize all the processing cores in the GPU due to the limited data size they operate on. The sequential use of the GPU for accelerating each of these tasks leads to performance inefficiencies due to the idle GPU resources during the execution of each task. It is therefore important for the programmer to cluster these small tasks together for sharing the GPU; however, the best performance cannot be achieved through an ad-hoc grouping and execution of these tasks. A scheduling framework for carrying out an off-line clustering and sequencing of GPU tasks is therefore essential to be studied and implemented.

In this paper, we explore the problem of GPU task scheduling to allow multiple tasks to efficiently share the GPU and execute in parallel. Our study is based on the currently available NVIDIA Fermi class of GPUs that support the execution of multiple kernels simultaneously [3]. We analyze factors affecting multi-tasking parallelism and performance, and exploit features available in contemporary GPUs, such as concurrent I/O and execution, bi-directional data transfers, as well as concurrent kernel execution. For NVIDIA GPUs programmed using CUDA [2], the use of these capabilities relies on programming constructs called

1521-9097/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPADS.2011.13

IEEE computer society

streams, with each stream essentially taking care of the set of operations required for executing a task. The programmer can make use of CUDA streams to launch independent tasks for parallel execution on the GPU, subject to the availability of GPU compute resources. Since GPU resource constraints prevent the programmer from grouping together all tasks into one cluster, the available set of tasks needs to be broken down into two or more subsets, with each subset making use of streams to launch the constituent tasks for parallel execution. Even though GPU tasks within each subset execute in parallel, they sequentially make use of the same I/O channel for transferring data from the CPU to the GPU (and vice versa), and as a result, the order in which the tasks are launched becomes an important factor in determining the performance of the parallel tasks.

We therefore propose a scheduling framework to aid the programmer in efficiently grouping the available tasks into subsets for parallel execution on the GPU, in addition to sequencing the launching of tasks within each group. To enable efficient scheduling, it becomes necessary to model the parallel GPU execution of the clustered group of tasks, with the performance prediction based on the compute and I/O characteristics of each task. The description of our scheduling framework thus begins with a discussion of a multi-tasking performance model.

The remainder of the paper is organized as follows. Section II highlights the related work available in the literature, followed by a brief background on GPU computing in Section III. Subsequently, Section IV describes the proposed scheduling framework, including the performance model as well as heuristics carrying out the required task scheduling. Section V then validates the performance model through a set of experiments, followed by an evaluation of the proposed scheduling algorithms through synthetic tasks as well as actual application kernels running on the GPU. Finally, Section VI concludes the work and discusses future work.

II. RELATED WORK

There are several research efforts related to task scheduling in the HPC field. One direction of research has looked at scheduling for General-Purpose computation on Graphics Processing Units (GPGPU) [4]. In this direction, previous work has investigated the problem of GPU resource under-utilization by providing task scheduling at various levels either statically or at runtime. Guevara et al. [5] introduced a GPU kernel merging approach, which merges task workloads that underutilize the GPU resource. The work focuses mainly on providing a runtime environment that intercepts kernel invocation as the scheduling inputs and makes scheduling decision by making necessary kernel merges. A similar approach was adopted by Saba et al. [6]. They presented an algorithm for GPU architectures that would adaptively allocate GPU resource matching the goals and loads for large workloads. Their approach focuses on a time-bound algorithm that measures the output quality and optimizes the execution path. While the approach targets a different problem with large runtime workloads, they also employ a similar kernel-merging approach to increase the GPU utilization. Furthermore, Chen et al. [7] proposed a task queue scheme, by utilizing a kernel-merging approach that merges the kernels of the workloads into the persistent kernel, to achieve dynamic load balancing on the GPU systems. Although these approaches improve task parallelism and the GPU utilization, they suffer from several drawbacks. Firstly, [5][6] do not consider the task data transfer metrics in the scheduling approach, which is not negligible since data transfer time is a key factor that determines multi-tasking performance. Secondly, the GPU architecture used by these work are superseded by the latest Fermi architecture, which provides further kernel parallelism support. Thus, with Fermi, kernel-merging approach would not be necessary. Furthermore, kernel-merging approach would require additional programming efforts, which would also pose a programmability problem to be solved.

Another stream of similar research concentrates on task scheduling on heterogeneous architectures [8][9]. [8] presented StarPU, which is a runtime execution model based platform allowing easy programmability and co-scheduling of both GPU and CPU tasks targeting at a general heterogeneous platform. Similarly, [9] discussed a static task partitioning approach for heterogeneous system composed of both CPUs and GPUs with predictive execution model approach. While both work employed execution model prediction approach (combined coarse-grained CPU and GPU model), which is also the approach we use (fine-grained GPU task model) in this paper, our approach draws the difference in tackling a different scheduling problem, which is the scheduling of static GPU tasks by performing a finer-grained task execution modeling and scheduling.

Task scheduling on other types of co-processors such as Field-Programmable Gate Arrays (FPGAs) has also been studied in the literature. For example, [10] described an early attempt by using task scheduling to efficiently share the FPGA with partial run-time reconfiguration. [12] discussed the Reconfigurable Computer(RC) task scheduling with a HW/SW automatic co-design approach. [11] explored the FPGA task scheduling problem with the consideration of inter-task data communication. In [13], Angermeier et al. proposed a virtual area management approach to pack hardware modules with time-varying resource requests efficiently and tackled the problem by designing a series of heuristics. Our work in this paper also defines a similar scheduling problem and uses similar heuristics. However, in this paper, we target at a completely different problem of static multi-tasking GPU scheduling to improve resource utilization, which has not been tackled before. We investigate the problem by considering task I/O and resource profiling, multiple inter-task concurrency scenarios, derived execution model as well as multiple scheduling heuristics.

III. BACKGROUND

We provide a brief overview of CUDA [2] and NVIDIA GPU architecture in this section. Our further analysis will be based on CUDA programming model and NVIDIA Fermi architecture [3]. CUDA is the parallel programming model provided by NVIDIA following master-slave model. The master-slave flow follows that the CPU master process

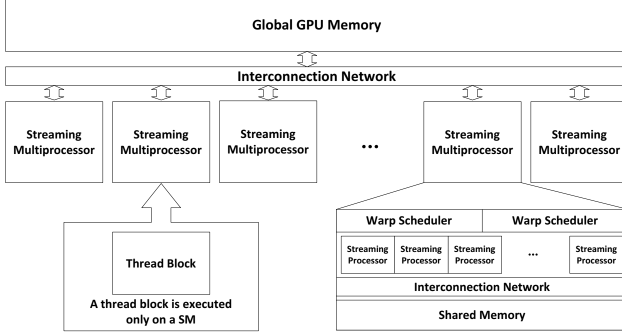

Figure 1. A top-down overview of Fermi architecture

launches the GPU program, sends the data to GPU device memory, waits for the GPU to finish executing and retrieves the result data. Currently, CUDA does not allow the interruption of a GPU program once it starts, limiting the GPU tasks to be atomic. The finest execution unit in CUDA is the thread, which is grouped into two hierarchical levels: blocks and grids. A CUDA program is launched per grid by executing a kernel function. A grid is composed of a number of blocks and each block contains a number of threads. A detailed description of NVIDIA Tesla GPU architecture is provided by [14]. Figure 1 shows the latest Fermi architecture from NVIDIA. From a top-down overview, Fermi is consisted of 16 Streaming Multiprocessors (SMs), each of which can be further decomposed of 32 Streaming Processors (SPs). Each thread block can only run on a single SM. Inside each SM, while each thread is executed on an SP; threads are dispatched as warps (a group of 32 threads) at one time. Moreover, 48 warps (Fermi) can be tracked on each SM allowing up to 1,536 concurrent threads per SM.

Since GPU kernel execution is non-preemptive, the launching of one kernel blocks the execution of other kernels until current one finishes. Thus multi-tasking execution on GPU generally follows sequential execution. Fermi employs concurrent kernel execution through CUDA streams. By using streams, further overlapping such as concurrent I/O and kernel execution and concurrent bi-directional I/O can be achieved. However, the required number of streams need to be allocated statically a priori, and the launching of kernels either sequentially or in an ad-hoc manner through random streams leads to resource waste since no prior scheduling decision has been made; thus this leads us to further discuss the proposed task scheduling and execution framework to tackle the problem by utilizing kernel and I/O concurrency support from Fermi in the following sections.

IV. Scheduling Approach

In this section, three major components of our approach are presented. We first conduct a profiling study of GPU tasks under multi-tasking scenario. As we are interested in improving the performance of multiple independent tasks, tasks are treated equally as non-interruptible GPU kernels. In the meantime, we analyze necessary task timing and resource metrics to be considered in our execution model and scheduling study. In the text that follows, we present the task execution model based on the optimized task execution

Figure 2. A GPU execution cycle for a single task

scenario, which provides the task execution flow and performance prediction of a task sequence. We further define the multi-tasking scheduling problem; analyze the problem based on the execution model and present several scheduling algorithm heuristics as the scheduling problem solution.

A. Task Profiling Analysis and Metrics Definition

The primary goal in conducting preliminary task profiling is to utilize characteristics of each task and derive corresponding performance metrics as a way of performance prediction. We define both timing and resource metrics to be profiled for tasks. While timing metric is mainly used for deriving predicted execution time, the resource metric is defined to analyze the possible inter-task concurrency.

For timing profiles of each task, we define the metrics explained as follow. Since offloading a single task on the GPU involves initializing the required GPU resource such as GPU context, we define the GPU initialize time as one general timing metric. The initialization time is generally a one-time overhead per task can also be referred as a "warm-up" time when the GPU starts to execute the task. For each GPU task, the execution cycle is composed of the following stages. The task sends the input data to the GPU device memory, followed by computing the task and retrieving the result data back from the GPU device memory, as shown in Figure 2 with the timing metric defined for each stage.

When considering GPU resource metrics, we consider the GPU resource utilization of each task (R_{task}) and define it as (*task resource utilization / full resource of the GPU*) in general with details shown in equation 1. The amount of GPU resource utilized by one task primarily depends on the data parallelism of the task kernel. While some tasks with enough data parallelism can fully utilize the GPU, most tasks without being described as embarrassingly parallel can only utilize partial resource. In each task (kernel), threads are allocated with computing resource in the SMs such as registers and special functional units. Thus, the total number of threads of the task determines the fine-grained resource utilization. While one block only runs on an SM, CUDA also allows multiple blocks to run on a single SM. This only happens when the block cannot fully utilize the SM resource. The metric considers the *full point block #* of the task to be the number of blocks that will be used when all the GPU SMs are utilized, which is the product of the number of SMs on the GPU and the number of blocks running on each SM. Thus, the metric defined in equation 1 provides the basis for analyzing multi-tasking resource sharing in the following discussions of the analytical multi-tasking GPU execution model and the scheduling algorithms.

$$R_{task} \begin{cases} = \dfrac{Block\#_{task}}{Block\#_{fullpoint}} = \dfrac{Block\#_{task}}{SM\#_{GPU} \times Block\#_{SM}} \ldots\{\text{if } (Block\#_{task} < Block\#_{fullpoint})\} \\ = 1 \ldots\ldots\ldots\{\text{if } (Block\#_{task} \geq Block\#_{fullpoint})\} \end{cases} \quad (1)$$

B. The Multi-tasking GPU Execution Model

1) The Multi-tasking Execution Scenario

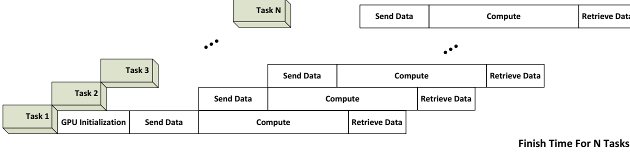

Figure 3(a). A special multi-tasking execution scenario without data retrieving wait, when all tasks have the same profile

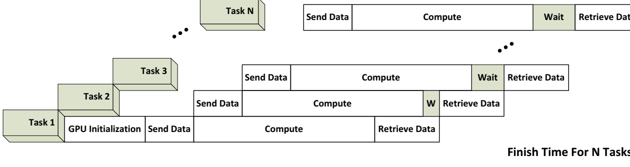

Figure 3(b). A special multi-tasking execution scenario with data retrieving wait, when all tasks have the same profile

The tradition multi-tasking scenario follows sequentially executing each task, where resource can be under-utilized and no inter-task concurrency occurs. The key approach to avoid resource underutilization among multiple tasks is to achieve inter-task parallelism. Thus, we propose a multi-tasking execution scenario on the GPU, which our further execution framework is based on. At the presence of multiple GPU tasks, the execution scenario we propose to efficiently execute all tasks is to divide tasks into task groups and execute each group sequentially. Each task group needs to be able to provide full task parallelism, which requires that each group has enough SM resource for the each task within the group. Therefore, under CUDA architecture, each task needs to be launched in a separate asynchronous stream. By launching tasks through streams, three kinds of inter-task concurrency can be achieved on Fermi: the concurrency of data transfer and task computation, task computations and bi-directional data transfers.

2) The Multi-Tasking Execution Model

When tasks are grouped and executed sequentially with the proposed task execution scenario, the execution model emulates the asynchronous stream execution scenarios for each group with possible inter-task overlapping. It uses profiling metrics from each task and predicts the execution time of tasks in a group. To merely demonstrate the inter-task overlapping, we first show a special execution scenario with identical tasks. Since the task group has been given full task parallelism, all tasks can execute concurrently. As the execution model is used to provide a performance upper bound, we further make the assumptions that single directional data transfers are atomic and cannot be overlapped or interrupted. In other words, single directional data transfers always take the full I/O bandwidth. Thus, as shown in Figure 3, the next task can only start after the first task finishes inbound data transfer. Since CUDA stream execution allows both task execution and inbound data transfer from one stream to overlap with the task execution and outbound data transfer from another stream, both Figure 3(a) and Figure 3(b) demonstrate the concurrencies occurred. By assuming data transfers are atomic, we use Figure 3(a) and Figure 3(b) to show two scenarios when data retrieving wait occurs in Figure 3(b) but does not occur in Figure 3(a).

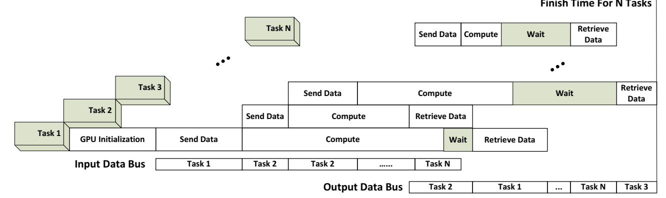

Figure 4. A general multi-tasking execution scenario

TABLE I. PARAMETERS DEFINED

N_{task}	The number of GPU tasks to be executed in the group
T_{init}	The average time to initialize the GPU device and corresponding GPU contexts
$T_{data_in_t(i)}$	The average time for task *i* to transfer the data into the GPU device memory
$T_{data_out_t(i)}$	The average time for task *i* to retrieve the data back from the GPU device memory
$T_{comp_t(i)}$	The average time for the GPU to compute the task *i*
$T_{finish_comp_t(i)}$	The total time it takes for task *i* to reach "finsh_comp" point since the execute model starts
$T_{total_t(i)}$	The total time it takes for task *i* to finish since the execution model starts
$T_{total_grp(m)}$	The total time it takes for the task group *m* to finish, which is the output of the Algorithm 1
T_{total}	The total time it takes for all tasks in all task groups to finish, which is the output of the model
$R_{task_t(i)}$	The GPU resource utilization metric defined earlier for task *i*
R_{GPU}	The GPU full resource point as defined earlier

As the output of the model, the predicted finish time is based on the finishing point of the last completed task in the model.

We further apply a general case to the model when multiple tasks have varied profiles as described in Figure 4. Since each task can achieve a complete overlapping for both "send data" and "compute" with the prior task, we define the point for a task to finish "compute" stage without proceeding to "retrieve data" as "finish_comp" point. Different from the special case with identical tasks that "finish_comp" of each task follows the task starting order, tasks reach "finish_comp" point out of order in the general case. Thus, the "retrieve data" stage for each task also occurs out of order and follows "First-Finish, First-Served (FF-FS)" policy as shown in Figure 4. The last "retrieve data" determines the total time of finishing all tasks as the model output.

Table I defines necessary parameters for both the execution model and the algorithm. As the execution model focuses on the general multi-tasking scenario, we first present an algorithm that generalizes the total time of executing all tasks in a group for the model, as shown in Algorithm 1. The algorithm follows the FF-FS policy for the "retrieve data" stage and performs the sorting based on $T_{finish_comp_t(i)}$. Since "retrieve data" stage is atomic, the sorting enables deriving the finishing time of the last task in the sorted task list, therefore outputs the total time for the model. By using Algorithm 1, given a task group with a defined sequence, the expected total group time can be derived, as shown in Figure 5 by an example. While Algorithm 1 does the task-sorting based on "finish_comp" point, it does not change the task sequence since the sorting is only used for

Algorithm 1: Algorithm for the Execution Model

Input: N_{task} GPU tasks in task group m with profiling information

for i = 1 **to** N_{task}

$$T_{finish_comp_t(i)} = \sum_{n=1}^{i} (T_{data_in_t(n)}) + T_{comp_t(i)};$$

Sort N_{task} tasks according to $T_{finish_comp_t(i)}$ in an **ascending order**, giving $Tlist_{sorted}$;

Using $Tlist_{sorted}$

$T_{total_t(1)} = T_{finish_comp_t(1)} + T_{data_out_t(1)}$;

for i = 2 **to** N_{task} in $Tlist_{sorted}$

j = i-1;

$T_{total_t(i)} = MAX\ (T_{finish_comp_t(i)}, T_{total_t(j)}) + T_{data_out_t(i)}$;

$T_{total_grp(m)} = T_{total_t(N_{task})}$;

Output: $T_{total_grp(m)}$

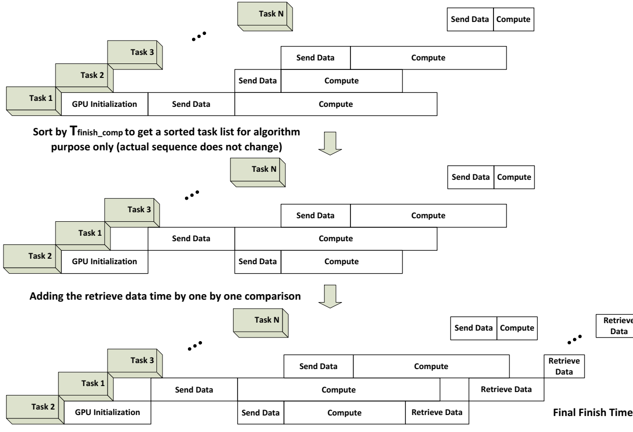

Figure 5. Deriving method of Algorithm 1 by an example

deriving the total time. We also choose not to consider T_{init} in Algorithm 1 and further analysis only because T_{init} is a constant overhead and does not affect the model and further scheduling results with varied task profiles.

In multi-tasking scenario, the execution model targets at all tasks in a single task group. Since many tasks cannot be fit into a single group due to the resource constraint. Different groups need to be executed sequentially, with a barrier between two groups. We simply model the multiple group execution total time, T_{total}, by sequentially adding $T_{total_grp(m)}$ of each group, as shown in Figure 6.

C. The Scheduling Problem and Algorithm Heuristics

As we propose the scenario of executing task groups and the total execution time of all tasks depends on the number of groups, finding the smallest number of groups becomes one of the key scheduling issues. Meanwhile, when executing each task group using the proposed execution scenario and model, different task execution sequences give varied group execution time. Thus, finding the best execution sequence within each group becomes the other scheduling problem. By considering both task grouping and sequencing problems, we define the multi-tasking scheduling problem, which is to find the grouping scenario that minimizes the

Task Sequencing Heuristics: **Algorithm 2** (T_{comp} Prioritized) and **Algorithm 3** (T_{data_in} Prioritized)

Input: N_{task} GPU tasks with profiling information (*Tlist*)

In *Tlist*

find the task m with $MAX\ (T_{comp_t(m)} + T_{data_out_t(m)})$;

swap_task ($Tlist_{(1)}$, $Tlist_{(m)}$);

for i = 1 **to** $N_{task} - 1$ in *Tlist*

for j = i + 1 **to** N_{task}

if (there exist tasks that meet ($T_{data_in_t(j)} + T_{comp_t(j)}$) $\geq$ ($T_{comp_t(i)} + T_{data_out_t(i)}$))

find task k with $MAX\ (T_{comp_t(k)} + T_{data_out_t(k)})$ within all found tasks meeting the previous requirement;

swap_task ($Tlist_{(i+1)}$, $Tlist_{(k)}$);

else

(Algorithm 2) find task k with $MAX\ (T_{comp_t(k)})$;

OR

(Algorithm 3) find task k with $MIN\ (T_{data_in_t(k)})$;

swap_task ($Tlist_{(i+1)}$, $Tlist_{(k)}$);

Output: *Tlist*;

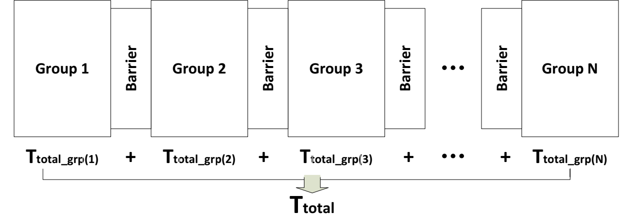

Figure 6. Total execution time deriving of multiple task groups

number of groups while giving out an optimized task sequence with the minimum execution time for each group.

We first consider the sequencing algorithm for tasks within a task group. The sequencing problem can be solved by the brute-force method, which tries every possible sequence permutation and applies Algorithm 1 to find the task sequence with shortest execution time. While the brute-force method gives the optimal time, the time consuming nature of this approach motivates us to find additional faster heuristics. We therefore propose two heuristics for the sequencing problem as part of our initial efforts, as outlined in both Algorithm 2 and 3. Both algorithms are based on a greedy approach to choose the next task within a group of tasks. The main idea of the heuristics is to always achieve the maximum inter-task overlapping. In other words, starting by executing the task with maximum ($T_{comp_t(i)} + T_{data_out_t(i)}$), both heuristics choose the next task with the maximum overlapping with the current task. If the maximum overlapping ($T_{comp_t(current)} + T_{data_out_t(current)}$) can be found among multiple tasks, both algorithms choose the next task with maximum ($T_{comp_t(next)} + T_{data_out_t(next)}$). If not, Algorithm 2 focuses on the priority of $T_{comp_t(i)}$, which guarantees the task with the longest $T_{comp_t(i)}$ to be executed as early as possible, and Algorithm 3 focuses on the priority of $T_{data_in_t(i)}$, which ensures starting the "compute" stage of the next task as early as possible (with shortest $T_{data_in_t(i)}$).

For the task grouping problem, all tasks need to be grouped into a finite number of groups while minimizing the

Algorithm 4: Main Task Scheduling

Input: All GPU tasks with profiling information
Sort all tasks according to R_{task} **in a descending order**, giving the sorted task list $Tlist_{sorted}$;
Using $Tlist_{sorted}$
  Perform bin-packing heuristic(*FirstFit, BestFit*) on $Tlist_{sorted}$, giving m task groups;
  for i = group 1 **to** group m
    Perform the sequencing heuristc(Algorithm 2, 3) ;
    Perform Algorithm 1 giving $T_{total_grp(i)}$;
    $T_{total} = T_{total} + T_{total_grp(i)}$;
Output: m groups of sequenced tasks, T_{total};

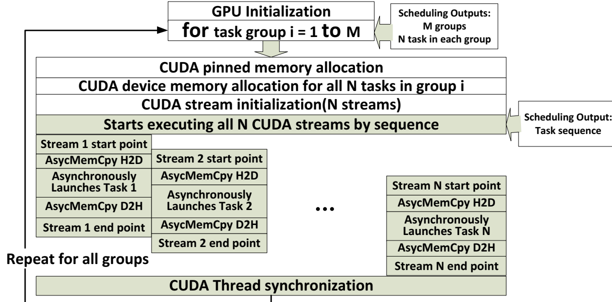

Figure 7. The GPU execution framework

number of groups. As our goal is to provide each group with complete execution concurrency, the grouping policy considers R_{task} of all tasks and constrains that the total R_{task} in each group does not exceed R_{GPU}. Thus, the problem can be categorized as a general NP-hard bin-packing problem [15]. As an initial effort, we consider using the following two bin-packing heuristics: *FirstFit* and *BestFit*. *FirstFit* is the heuristic that places the task in the first group where the task fits. *BestFit* is the heuristic that considers all groups where the task fits and places the task in the fullest group. With both grouping heuristics, we formalize the main scheduling algorithm as shown in Algorithm 4. It first conducts the sorting of all tasks based on R_{task}. This is for better bin-packing results by grouping the task with largest R_{task} first [16]. We apply the two bin-packing heuristics first to group tasks into m groups and perform sequencing to the m groups using the sequencing heuristics. For the derived sequence of each group, we apply Algorithm 1 to derive $T_{total_grp(i)}$ for each group and summarize T_{total}. Algorithm 4 gives both the grouping and sequencing results as well as T_{total} as outputs.

V. Implementations and Results

The proposed execution model and algorithm are based on modern GPU's capability of asynchronous kernel execution and data transfer. In other words, the model emulates the GPU execution scenario when multiple tasks take asynchronous CUDA streams to achieve inter-task overlapping. Figure 7 shows the details of the proposed GPU execution framework, which is used for our further benchmarks. The framework initially creates the required GPU execution resources for the task group such as CUDA streams and host pinned memory for the concurrent data-transfer and execution. It executes each task within a group using a single CUDA stream based on the scheduling sequencing results. The execution performs sequentially for each task group until all groups have been executed.

To demonstrate the efficiency of our approach in executing multiple GPU tasks with scheduling, we start with analyzing the proposed execution model and scheduling framework by conducting several micro-benchmarks. We primarily use micro-benchmarks to evaluate the accuracy of the proposed model with GPU results, followed by evaluating the efficiencies of the proposed algorithm heuristics. We further utilize an application benchmark to show the advantage of our scheduling approach by comparing multi-tasking execution using the proposed scheduling framework with native sequential task execution. The experiments are conducted on our GPU computing node. The node is equipped with NVIDIA Tesla C2070 (Fermi) computing GPU with 14SMs running at 1.15 GHz and 6GB device memory, which allows maximum 16 concurrent running tasks (kernels). The node also has dual Intel Xeon X5570 quad-core hyper-threading CPU running at 2.93 GHz and 48 GB system memory. The CUDA version is 3.2, which runs under Ubuntu 10.10 with 2.6.32-30 Linux kernel.

To demonstrate the prediction accuracy of the model, we first utilize the micro-benchmark, which is composed of a sequence of 14 vector multiplication GPU kernels (1 block of kernel size), to compare the GPU running time versus the predicted time from the model. We choose 14 kernels due to availability of 14 SMs within the chosen GPU platform [2]. The kernel computing intensity has been varied based on the number of vectors used in the computation. By adjusting the kernel computing intensity, we divide our analysis into two categories: compute-intensive and I/O-intensive applications, to analyze computing and I/O behavior separately, as shown in Table II. For each category, we evaluate three types of task (kernel) sequences under the model: in order, reversed order and random order based on the order of $T_{finish_comp_t(i)}$ in the task sequence. When "in order", $T_{finish_comp_t(i)}$ follows the sequence starting order, and when "reversed ", $T_{finish_comp_t(i)}$ follows the reversed sequence starting order. "Random" simply makes the case when $T_{finish_comp_t(i)}$ does not follow the sequence starting order. The reason we choose to use varied orders is to verify the correctness of Algorithm 1 in conducting the sorting-based calculation. By comparing the task sequence execution time on the GPU and the model prediction, we derive the model deviations for all cases, as shown in Table II. For data-intensive task sequence, the model matches the GPU results very well, which demonstrates that, our I/O assumption in the model agrees with reality. For compute-intensive task sequence, the model deviation is around 15%, which still matches the GPU result well. Meanwhile, we further conduct a concurrency analysis of running the same compute-intensive vector multiplication kernels but with zero I/O. We use an increasing number of streams to show the concurrent kernel execution occurred. As shown in Figure 8, as the stream number increases while each stream carries one kernel, the total execution time increases slightly until a sharp increase from 14 to 15 streams. This is due to the fact that there are 14 SMs in

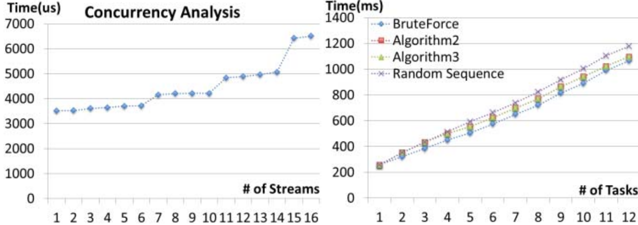

Figure 8. Kernel execution concurrency analysis

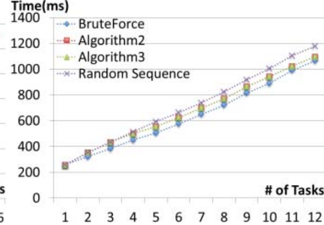

Figure 9. Sequencing efficiency: Intermediate(IM) Tasks

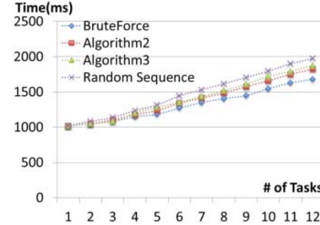

Figure 10. Sequencing efficiency: Compute-intensive(C-I) Tasks

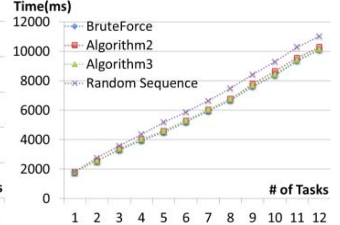

Figure 11. Sequencing efficiency: I/O-intensive(IO-I) Tasks

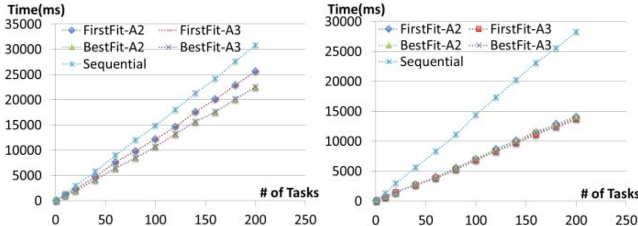

Figure 12. Scheduling efficiency: Tasks utilizing moderate resource

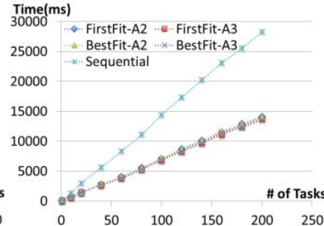

Figure 13. Scheduling efficiency: Tasks utilizing low resource

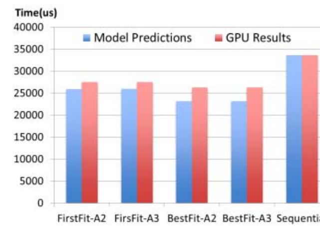

Figure 14. Comparisons of the GPU and model (Application Benchmark)

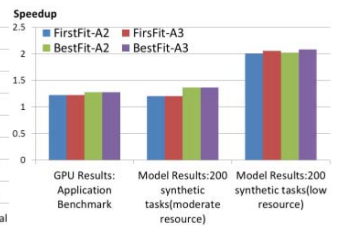

Figure 15. Speedups comparisons

TABLE II. EXECUTION MODEL VERIFICATION RESULTS

Sequence	Compute-intensive			I/O-intensive		
	GPU (us)	Model (us)	Deviation	GPU (us)	Model (us)	Deviation
In order	4325	3653	15.54%	322	323	2.83%
Reversed	4035	3456	14.35%	269	257	4.29%
Random	4139	3485	15.81%	277	266	3.99%

TABLE III. SEQUENCING HEURISTICS COMPARISONS

	Performance gain over random for 12 tasks (IM / C-I / IO-I)	Running Time (12 tasks)
Algorithm 2	6.96% / 7.54% / 6.64%	158us
Algorithm 3	6.96% / 4.93% / 7.86	159us
BruteForce	9.67% / 14.87% / 8.45%	225.48s

C2070 and each kernel is using one block that occupies one SM. It also shows that inter-kernel overlapping involves slight overheads with the increasing number of concurrent kernels to be executed, which has not been considered in the proposed model and explains the slightly higher model deviation for compute-intensive task sequence.

With the verified model accuracy, we analyze the algorithm heuristics by using synthetic micro-benchmarks. We first analyze the efficiency of the proposed sequencing heuristics and use three types of workloads which have the randomly generated task profiles(in certain intervals (ms)): intermediate(IM) ($50<T_{comp}<100, 50<T_{I/O}<100$), compute-intensive(C-I) ($500<T_{comp}<1000, 50<T_{I/O}<100$), and I/O intensive(IO-I) ($50<T_{comp}<100, 500<T_{I/O}<1000$). We compare the model results from the two proposed sequencing heuristics with model results from both a random sequence and the optimal sequence (by the brute-force method), as shown in Figure 9, 10, 11 for all three types of workloads. While both heuristics perform close to the optimal results, Algorithm 3 performs slighter better than Algorithm 2 in IM and IO-I applications due to its priority of starting the "compute" stage early for more concurrency. Algorithm 2 only performs better in C-I applications due to giving priority to tasks with the longest T_{comp}, which is especially helpful in providing more concurrency when tasks have longer T_{comp} and shorter T_{data_in}. Table III describes the performance gain of the sequencing algorithms over a random sequence as well as the algorithm running time, which shows that both heuristics can provide near optimal results with much less running time.

We further analyze the scheduling efficiency of Algorithm 4 by increasing the task count and comparing the model results from four proposed heuristics (*FirstFit-A2, FirstFit-A3, BestFit-A2, and BestFit-A3*) with the sequential results using the synthetic task workload (Intermediate Tasks). Figure 12 shows the scenario when tasks utilizing moderate resources (R_{task} is randomly generated from 25% to 75%) while Figure 13 shows when tasks utilizing lower resources (R_{task} is randomly generated from 5% to 25%). As the results show, *BestFit* performs slightly better than *FirstFit* with better grouping results and Algorithm 3 performs slightly better than Algorithm 2 in both cases. In general, the lower resource tasks utilize, the higher inter-task concurrency can be achieved using our scheduling approach.

As a step further, we utilize the application benchmark composed of 3 applications. 5 different tasks have been created from each application with varied problem sizes, which makes the total of 15 tasks to be scheduled. Table IV shows the profiles of the applications. MM64 refers to 64x64 matrix multiplication, with 1 to 40 matrices to be computed among the 5 tasks (evenly distributed). Electrostatics refers to the fast molecular electrostatics algorithm as a part of the molecular visualization program VMD [17] and we evenly vary the atom sizes among 5 tasks. BlackScholes [18],

TABLE IV. PROFILES OF APPLICATIONS USED IN THE BENCHMARK

	MM64	Electrostatics	BlackScholes
# of Tasks	5	5	5
Problem Size Ranges	64x64 matrix (1-40 calculations)	4,000-20,000 atoms	100K-500K calls
# of Blks	4	8	2
# of Blks/SM	1	1	1
Class	IM	C-I	IM

obtained from the NVIDIA CUDA SDK, is a European option pricing benchmark used in financial field with evenly varied number of calls among 5 tasks. All 15 tasks have been profiled using CUDA profiler with resource results such as number of blocks launched per SM and timing results. We utilize the results as inputs to the Algorithm 4 and derive the 4 schedules with the proposed 4 heuristics and the modeled total execution time. The 15 tasks are executed under the proposed GPU execution framework with derived 4 schedules. When considering the GPU time, we ignore the inter-group resource (memory and stream) allocation overheads since they are not considered in the model. The comparisons are between model and GPU results for all scenarios as shown in Figure 14. The results demostrate an agreement between the model and GPU execution with deviations less than 12%. Figure 15 shows the speedups achieved from our scheduling appoach. The left bars demonstrate a 28% performance gain (GPU results) of the application benchmark (15 tasks) with our scheduling algorithm over the sequential execution without scheduling running on the GPU; the middle and right bars demostrate 37% (moderate resource) and 108% (low resource) performance gains (model results) from a theoretical scenario which has 200 synthetic tasks to be scheduled. Therefore, while the proposed scheduling approach improves the GPU multi-tasking performance and device utilization, the performance improvements also depend on the task profiles. In general, our experimental results demonstrate the efficiency of the proposed scheduling approach and the accuracy of our execution model analysis.

VI. Conclusion and Future Work

In this paper, we proposed a scheduling framework which enables efficient GPU resource sharing among multiple GPU tasks. Our approach investigated the concurrency and overlapping potential and scenarios that can be achieved on modern GPU devices. We analyzed the execution overlapping scenarios by proposing a multi-tasking GPU execution model. The analytical model provides a theoretical performance prediction, which is utilized by the proposed scheduling framework. We presented several algorithm heuristics as part of our initial efforts in addressing the scheduling problem. We further implemented the presented algorithm heuristics and conducted a series of experimental benchmarks on our NVIDIA Fermi GPU computing node, ranging from evaluating the execution model accuracy and scheduling efficiency, to evaluation of real-life benchmark performance gain using our approach. The experimental results demonstrate that the use of our scheduling approach can provide significant performance gains. Furthermore, the results also show an agreement between our execution model analysis and the actual experiments carried out on the GPU.

Future work should consider improvements to the performance model to further reduce the deviations observed from the experiment. This will involve accounting for the various overheads in GPU execution. Furthermore, speeding up the scheduling algorithms will provide opportunities for using the developed framework for run-time task scheduling, thereby extending the applicability of the proposed approach for scenarios when multiple CPU processes share a GPU.

Acknowledgement

This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. IIP-0706352.

References

[1] Top 500 Supercomputer Sites Webpage, http://www.top500.org, Last Accessed: 28th June 2011.

[2] "NVIDIA CUDA C-Programming Guide," v3.2, Sep. 8th 2010.

[3] NVIDIA Corporation, "NVIDIA's next generation CUDA compute architecture: Fermi," White paper V1.1, Jun.2009, available online on http://www.nvidia.com.

[4] GPGPU Webpage, http://www.gpgpu.org, Last Accessed: Oct. 1st 10.

[5] M. Guevara, C. Gregg, K. Hazelwood, K. Skadron, "Enabling Task Parallelism in the CUDA Scheduler," in Proc. of the Workshop on Programming Models for Emerging Architectures (PMEA), pp. 69-76, Sep. 2009.

[6] A. A. Saba and R. Mangharam,"Anytime Algorithms for GPU Architectures," in Proc. of the Analytic Virtual Integration of Cyber-Physical Systems Workshop, Co-located with RTSS, 2010.

[7] L. Chen, O. Villa, S. Krishnamoorthy and G. R. Gao, "Dynamic Load Balancing on Single- and Multi-GPU Systems," in Proc. of the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2010), Apr. 19-23, 2010.

[8] C. Augonnet, S. Thibault, R. Namyst and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," Concurrency and Computation: Practice and Experience, vol. 23, pp. 187–198, 2011. doi: 10.1002/cpe.1631.

[9] D. Grewe and M. F. P. O'Boyle, "A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL," in Proc. of the 20th International Conference on Compiler Construction, pp. 286-305, 2011.

[10] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt, "Dynamic scheduling of tasks on partially reconfigurable fpgas," IEEE Proc. Computers and Digital Techniques, Special Issue on Reconfigurable Systems, vol. 147, no. 3, pp. 181–188, May 2000.

[11] M. Huang, V. K. Narayana, H. Simmler, O. Serres, and T. El-Ghazawi, "Reconfiguration and Communication-Aware Task Scheduling for High-Performance Reconfigurable Computing," ACM Transactions on Reconfigurable Technology and Systems (TRETS), vol. 3, no. 4, pp. 20:1-20:25, Nov. 2010.

[12] P. Saha, "Automatic software hardware co-design for reconfigurable computing systems," in Proc. of International Conference on Field Programmable Logic and Applications, 2007 (FPL 2007), pp. 507-508, Aug. 2007.

[13] J. Angermeier, S. P. Fekete, T. Kamphans, N. Schweer, J. Teich, "Maintaining Virtual Areas on FPGAs using Strip Packing with Delays," in Proc. of the 17th Reconfigurable Architecture Workshop (RAW 2010), May 2010.

[14] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," IEEE Micro, pp. 39-55, 2008.

[15] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, 1996, "Approximation algorithms for bin packing: a survey," In Approximation algorithms for NP-hard problems, Dorit S. Hochbaum (Ed.), pp. 46-93, PWS Publishing Co., Boston, MA, USA.

[16] D. S. Johnson, "Fast Algorithms for Bin Packing," Journal of Computer and System Sciences, vol. 8, pp. 272-314, 1974.

[17] Visual Molecular Dynamics Program Webpage, http://www.ks.uiuc.edu/Research/vmd/, Last Accessed: Mar. 14th 11.

[18] F. Black and M. Scholes, "The pricing of options and corporate liabilities," Journal of Political Economy, vol. 81, no. 3, pp. 637–654, May-June 1973.