CuMAS: Data Transfer Aware Multi-Application Scheduling for Shared GPUs

Mehmet E. Belviranli Farzad Khorasani Laxmi N. Bhuyan Rajiv Gupta

Computer Science Department
University of California, Riverside
{belviram, fkhor001, bhuyan, gupta}@cs.ucr.edu

ABSTRACT

Recent generations of GPUs and their corresponding APIs provide means for sharing compute resources among multiple applications with greater efficiency than ever. This advance has enabled the GPUs to act as shared computation resources in multi-user environments, like supercomputers and cloud computing. Recent research has focused on maximizing the utilization of GPU computing resources by simultaneously executing multiple GPU applications (i.e., concurrent kernels) via temporal or spatial partitioning. However, they have not considered maximizing the utilization of the PCI-e bus which is equally important as applications spend a considerable amount of time on data transfers.

In this paper, we present a complete execution framework, CuMAS, to enable 'data-transfer aware' sharing of GPUs across multiple CUDA applications. We develop a novel host-side CUDA task scheduler and a corresponding runtime, to capture multiple CUDA calls and re-order them for improved overall system utilization. Different from the preceding studies, CuMAS scheduler treats PCI-e up-link & down-link buses and the GPU itself as separate resources. It schedules corresponding phases of CUDA applications so that the total resource utilization is maximized. We demonstrate that the data-transfer aware nature of CuMAS framework improves the throughput of simultaneously executed CUDA applications by up to 44% when run on NVIDIA K40c GPU using applications from CUDA SDK and Rodinia benchmark suite.

1. INTRODUCTION

General purpose GPU (GP-GPU) computing has had a remarkable impact on the evolution of scientific applications over the past decade. Various SW and HW improvements like unified memory access in CUDA 6 and dynamic parallelism in Kepler architecture [17] have enabled developers to utilize GPUs for a wide range of application classes with moderate programming effort. The introduction of concurrent kernel execution [27] and simultaneous work queues (i.e., hyper-Q) has enabled GPUs to be shared across multiple applications; hence acting more like general purpose processors rather than dedicated accelerators. Therefore, researchers have started focusing more on efficient sharing of GPUs across different applications.

The research on executing multiple kernels simultaneously on GPUs spans both SW and HW based approaches. Early works [21, 3] have focused on efficient execution of kernels by identifying best matches based on their compute and memory intensiveness to prevent any resource conflict that may occur during runtime. Some spatial multitasking solutions [1, 19] have been proposed to improve SM utilization by executing multiple small kernels. Sajjapongse et al. [23] developed a SW runtime that employs a multi-GPU kernel scheduler to decrease the device idle times in the presence of unsatisfied inter-kernel dependencies. More recent works [20, 26] have improved the GPU Thread Block scheduler by supporting various pre-emption policies in HW to enable a CPU-like multi process execution.

While the attempts to improve concurrent kernel performance in various scenarios has been beneficial, they do not address execution phases other than kernel call(s); in particular, CPU↔GPU data transfer(s) phases also account for considerable portion of the execution time. Figure 1 shows the execution time breakdown of applications among three phases: device-to-host (D2H) memory transfers, host-to-device(H2D) memory transfers, and kernel execution (KE). This data corresponds to stand alone executions of 12 Rodinia benchmark suite [6] applications on NVIDIA K40C GPU. The breakdown shows that data transfers occupy a considerable portion of a CUDA application's lifetime.

Data transfer over the PCI-e bus is a major performance bottleneck for GP-GPU computing. Although general I/O improvement techniques like zero copy DMA transfer and WC buffers have been adopted by some GPUs [4], architecture and application specific software and hardware techniques are also necessary to use them. Overlapping data copy operations with kernel execution has been one of the most popular software solutions to hide the transfer latency. There have been some application-specific studies [25, 9]; however, only a few researchers were able to generalize their techniques beyond a limited class of applications [15, 10]. These approaches either require significant programming effort involving adaptation of libraries and changes to the programs [3] or are capable of handling a very limited set of applications with accompanying HW modifications (e.g., [15] allows only regular and linear data accesses).

On shared GPU systems, data transfers introduce an ad-

ICS '16, June 01-03, 2016, Istanbul, Turkey

DOI: http://dx.doi.org/10.1145/2925426.2926271

ditional challenge beyond the problem of computational resource sharing. Kernel executions and two-way memory copy operations of the applications sharing the GPU should be overlapped to maximize the utilization of both the PCI-e bus and GPU. However, the high disparity between transfer and execution time ratios of different GP-GPU applications, and the dynamically changing characteristics of scientific workloads, makes the overlapping on shared GPU systems highly complex. The aforementioned concurrent kernel execution related works simply disregard memory transfers. They either assume that the data is already in the memory [1] or it is fetched on demand via device mapped/pinned host memory [21].

The open problem addressed in this work is '*how to schedule multiple applications sharing the same GPU to improve overall system throughput*' via data-transfer/kernel-execution overlapping while supporting a wide range of applications, without requiring any code modifications. A notable study in recognizing the importance of data-transfer in GPU sharing is carried by Sengupta et al. [24]. The authors propose a runtime framework, *Strings*, to auto-balance the load between multiple servers with GPUs and exploit automated transfer/execution overlapping for each server/GPU. Whenever applications request different type of GPU operations (i.e., D2H, KE, or H2D) around the same time, *Strings* scheduler tries to execute these calls concurrently on the corresponding resources, if they are idle. *Strings* was shown to improve the overall system throughput when compared to the case where no overlapping exists. However, the success of exploiting overlapping between three types of calls depends on:

- Arrival order of same type of operations belonging to different applications
- Idleness of the corresponding resource (PCI-e uplink & download and the GPU itself)

Currently, to the best of our knowledge, neither *Strings* [24] nor the latest CUDA runtime API take the above two factors into account while scheduling different type of GPU operations on a single shared device. To better understand the impact of these limitations, let us consider an experiment involving the shared execution of the subset of Rodinia benchmarks introduced in Figure 1. Figure 2(a) shows the execution where concurrent D2H, KE, and H2D calls are overlapped based upon a random call issue order of CUDA applications while Figure 2(b) shows how total execution

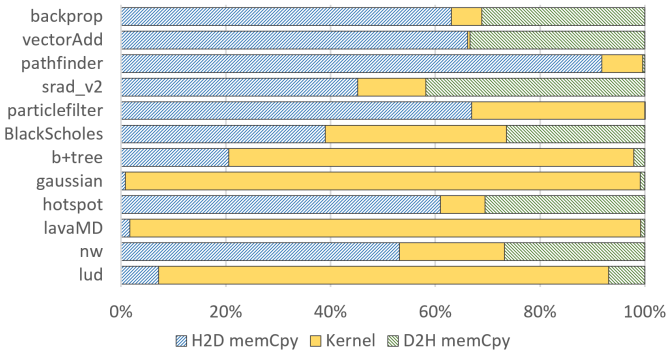

Figure 1: Execution time breakdown of 12 NVIDIA SDK and Rodinia applications.

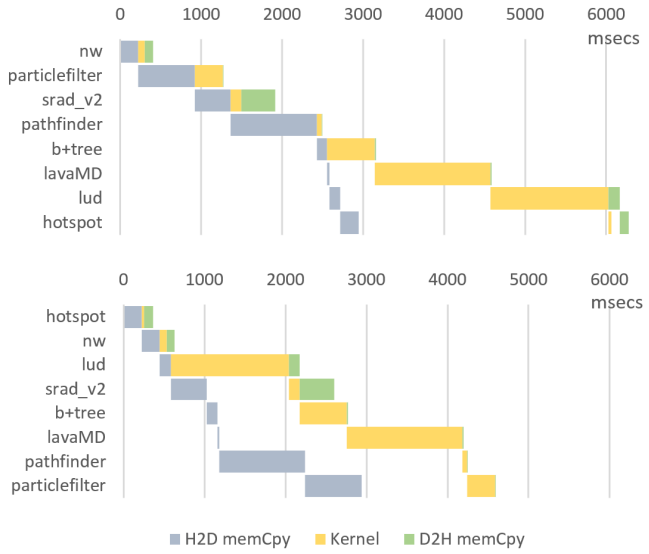

Figure 2: (a) On the top, an overlapping example exploited on a subset of Rodinia applications using a random arrival ordering; and (b) On the bottom, a data transfer-aware re-ordering of the same subset.

time can be minimized by simply re-arranging the call execution order of the CUDA applications. Thus, we can see that by changing the execution order of different applications, much higher resource utilization and thus performance can be achieved. If the call scheduler is made aware of the utilization of each underlying resource as well as the execution time of the issued device calls, the re-ordering can be performed to increase overlapping for higher resource utilization.

In this paper, we present CuMAS, a data-transfer aware CUDA call scheduling framework to efficiently overlap transfer/execution phases of multiple applications on a shared GPU. CuMAS captures CUDA API calls and re-schedules them to minimize the total execution time. The theoretical roots of finding such an optimal execution sequence is studied under '*open-shop problems*' [5] and 'multi-operation sequencing models'[13]. In this work, we represent the GPU multi-application scheduling as a '*flow-shop problem*' by treating CUDA calls as *jobs* while uplink/downlink PCI-e channels and the GPU as *machines* that process the *job*s. The processing time of each job is dependent on the size of the job (i.e., data to be transferred, thread blocks in a kernel) and the processing speed of the resources. Multi-machine flow-shop scheduling has been proven as NP-hard by Leung et al.[14]; however, approximate solutions based on dynamic programming [7] exist for more generalized versions of the problem.

CuMAS run-time uses profiling and timing models to estimate durations for data transfers and kernel executions. Our run-time dynamically attaches to existing programs and it does not require any changes to the programs' source code. CuMAS operates on a queue of CUDA requests issued by multiple applications each having arbitrary number of data transfer (H2D and D2H) and kernel launch (KE) calls. CuMAS ensures computation accuracy by preserving the execution order of calls belonging to the same application while improving the performance by re-ordering requests across

applications. CuMAS employs a novel CUDA call scheduler to find the optimal execution sequence of queued GPU operations. The scheduling decision involves estimated kernel execution and data transfer times for each application and delays occurring across applications due to resource waiting. The CuMAS scheduler utilizes a novel technique called *Re-Ordering Windows (ROW)* to control the granularity of transfer-execution overlapping while limiting the elapsed time between application's initial request and the actual execution of the CUDA call.

The key contributions of this paper are as follows:

- We propose CuMAS, a complete scheduling framework to exploit automatic data transfer and kernel execution overlapping for multiple applications sharing the same GPU.
- We design a novel CUDA call scheduler to find an execution ordering of a given list of pending calls to minimize the total execution time.
- We implement a run-time library that intercepts CUDA calls from existing applications, estimates duration of calls and re-issues them to improve overall system utilization.
- We support a wide range of applications including the ones with irregular data structures and non-linear access pattern, without requiring any additional programming effort.
- We show that CuMAS framework improves the throughput of multiple kernel executions by up to 44% when run on NVIDIA K40c GPU using applications from CUDA SDK and Rodinia benchmark suite.

The remainder of this paper is organized as follows. We first give a background on CUDA and shared application execution on GPUs in Section 2. Then we introduce the main idea (request re-ordering) of our proposed framework in Section 3. We focus on CuMAS scheduling algorithm in Section 4 and give details about our runtime in Section 5. We present evaluation of CuMAS in Section 6 and finish the paper with related work and conclusion sections.

2. GPU SHARING IN CUDA

In this section we first provide an overview of mechanisms used for sharing of NVIDIA GPUs across multiple CUDA applications including time and space sharing for concurrent kernel execution, CUDA streams, and Hyper-Q. We then elaborate on the considerations to solve transfer/execution overlapping problem.

Concurrent CUDA Kernel Execution: NVIDIA's Fermi architecture has enabled simultaneous execution of kernels to better utilize SMs (Streaming Multiprocessors) in a GPU. NVIDIA thread block (TB) scheduler allows concurrent kernel execution if not enough TBs remain from the prior kernel to utilize all idle SMs. Further advanced time and space sharing of SMs are possible via non-conventional CUDA programming techniques [23].

CUDA Streams and Hyper-Q: CUDA API provides streams to enable concurrent execution of multiple GPU applications. *Stream*s are analogous to threads in conventional CPU programming and they may represent different applications or different phases of the same CUDA application. A *command* is created for each CUDA call and queued into the *stream* specified in the call (the default 0 stream is used if none is specified). Commands queued in the same stream are serially executed whereas commands in different streams may be executed in parallel if corresponding resources (e.g., PCI-e bus, SMs on the GPU) are available. CUDA Kernel calls are asynchronous and memory transfers (i.e., *cudaMemcpy()* and *cudaMemcpyAsync()*) can be either synchronous or asynchronous. Synchronization between asynchronous CUDA calls is achieved via *events*.

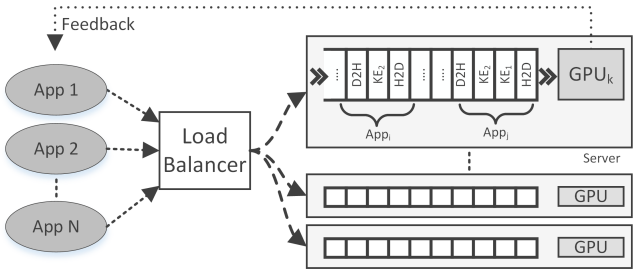

Figure 3: Multi server GPU Sharing and per-server CUDA call queues in a typical cloud computing environment.

Overlapping Kernel Execution and Data Transfers: Tesla and Quadro family of NVIDIA GPUs employ dual DMA copy engines [18] to enable simultaneous H2D and D2H data transfers. An ideal multi-kernel execution maximizes the use of both channels while keeping the GPU always busy. In a shared environment, with pending *transfer* and *execution* jobs, the scheduling order of jobs impacts system utilization. Currently, CUDA API schedules transfers in the order the calls are received from the applications (FIFO). In a shared environment, the applications may issue transfers at any time, hence appearing in a random order in the CUDA call queue. However, this may lead to performance degradation since the runtime is unaware of transfer times and execution times. If these times can be estimated prior to execution, a smarter scheduling decision can be made leading to improvements in utilization well beyond the default FIFO scheme.

Efficient sharing of GPUs across multiple applications requires each of the following issues to be handled carefully:

- *Maximization of overall resource utilization:* Individual and cumulative idleness of the resources –including the 2-way PCI-e links and the GPU itself– should be minimized.
- *User transparent execution:* The shared execution should seamlessly take place without requiring any code modifications or interventions from the users of the environment.
- *Support for various application classes:* Data access, data transfer, and execution patterns of an application should not limit the efficiency of sharing.

To address the issues listed above, we propose CuMAS (CUDA Multi-Application Scheduling), a host-side CUDA call scheduling framework, to enable efficient sharing of GPUs by multiple applications. CuMAS employs a scheduler and an accompanying run-time to enable on-the-fly integration to existing applications with no source modifications.

3. CUMAS: TASKS AND CALL ORDERING

CuMAS is based on the idea of *re-ordering CUDA calls* to improve the overall system throughput. In this section we will explain this novel idea in detail.

In a typical shared execution environment (e.g., cloud computing), a high level load balancer will distribute incoming user requests across servers, as depicted in Figure 3. Each server queues incoming requests (e.g., CUDA calls)

and executes them on their own GPU(s). In this paper we assume that the user applications are already assigned to a GPU and we focus on how requests from multiple applications are retrieved from the server queue and executed on the GPU.

3.1 CuMAS Tasks

We logically group the CUDA calls in the server queue into *CuMAS tasks*, which will be inputs to our scheduling algorithm in the later phase. *CuMAS tasks* are comprised of a series of H2D, KL and D2H calls all of which should belong to the same application. A typical CUDA application typically consists of multiple data transfer and kernel launch phases and thus we end up with multiple tasks being created for each application. A CuMAS task spans the CUDA calls which can be continuously executed without requiring any control on the execution flow.

From the application's perspective, CuMAS forces all CUDA API calls (except the last one of each CuMAS task) to act like *asynchronous call*s (i.e. they are immediately returned to the calling function). Whenever the last CUDA call is reached by the user program, all calls captured and grouped under the same task are issued on the real hardware. Since tasks are created dynamically as the application issues requests, at most one task per application can exist at any given time. Once all CUDA calls belonging to a task are processed, the caller application is unblocked and allowed to resume issuing CUDA calls.

To build tasks from a series of CUDA calls , we use the state diagram given in Figure 4(a). We group the calls into CuMAS tasks such that KE and D2H call in a task come after all H2D calls and similarly, D2H calls come after all KE calls. In our state diagram, any sequence of CUDA calls that breaks the ordering of H2D $\rightarrow$ KE $\rightarrow$ D2H causes current task creation to be finalized and a new task to be formed. Also, any occurrence of D2H at any time finalizes the creation of current task, due to possible control flow changes on the host side based upon the data received from the GPU. In addition to data transfer and kernel execution calls, *cudaDeviceSync()* calls also cause a new task to be formed.

Existence of multiple tasks in the same stream implies a dependency between different phases of an application and such tasks must be executed in the order of the operations. Figure 4(b) depicts the break down of a sample CUDA application into multiple tasks (T_1 to T_3) based on the order and type of the operations that it contains. The dependencies between tasks are shown with arrows and they imply a serial execution for the CUDA calls from the same application. For most CUDA applications, the dependent tasks are not created until the previous task is finished and new CUDA calls are captured. On the other hand, since there are no dependencies between the tasks of different applications, CuMAS exploits overlapping across the calls belonging to such tasks.

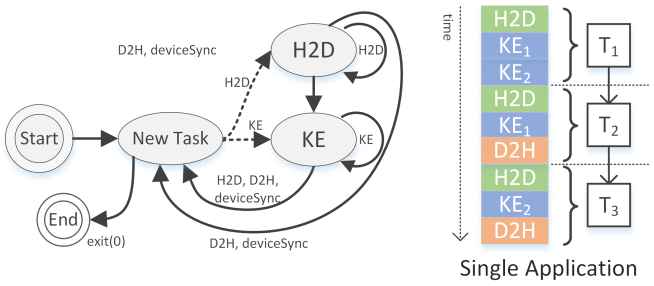

Figure 4: (a) On the left: The state diagram to decide whether create a new task based on the operation flow (b) On the right: An example stream showing the grouping of CUDA calls into tasks according to the state diagram given on the left.

3.2 Re-Ordering Window (ROW)

As CUDA calls are captured, tasks corresponding to the calls in each application's stream are created on-the-fly using the state diagram described in the previous sub-section. Similar to the queue of CUDA calls shown in Figure 3, tasks from all streams that are yet to be processed form a *task queue*.

We introduce *Re-Ordering Window* (ROW) to retrieve CuMAS tasks from the queue in batches and re-order them using our scheduling algorithm. Re-Ordering Window allows CuMAS to exploit better overlapping by allowing the tasks to be executed in different order than the arrival order while limiting the maximum distance that a task can be moved ahead or back in the queue.

Figure 5 shows a scenario where five artificial CUDA applications (A to E) are scheduled by CuMAS using ROW. Figure 5(a) depicts the initial state of the queue at time t_0, where applications have already issued CUDA calls around same time and corresponding CuMAS tasks (e.g. A_1 corresponds to the first task of application A) are created and placed in the queue in the original arrival order. In this example the size of ROW is set to three, which forces the last two ready to execute tasks, D_1 and E_1, to wait in the queue.

The tasks in a ROW are reordered using our scheduler (which is explained in the next section) and the CUDA calls in these tasks are issued in the order given by the scheduler. Figure 5(d) shows the HW pipeline for the execution of H2D, KE and D2H calls of the ordered tasks on the corresponding resources. *The execution order of the tasks are different than the initial placement of tasks in the queue due to the re-ordering applied by the CuMAS scheduler.* However, re-ordering remains only within the ROW, and the execution order of tasks across different ROWs is maintained. As CUDA calls grouped by the tasks in ROW are executed, corresponding applications are sent responses to continue their execution, which in turn generates new tasks, if any.

As soon as all the H2D calls of the tasks in a ROW are issued on the PCI-e device, ROW is shifted to cover the next available set of tasks, without waiting for the KE and D2H calls of the tasks in the prior window to finish. Figure 5(b) shows the time step t_1, where H2D calls of A_1, B_1 and C_1 are completed and up-link PCI-e bus has just become idle. At this moment, CuMAS shifts the ROW to next set of tasks in the queue. Since all CUDA calls in task A_1 are finished before t_1 and the application A has given chance to issue new calls, the ROW now covers a new task A_2 for these new calls in addition to the already existing tasks D_1 and E_1. On the other hand, the second task B_2 of application B is yet to be created due to unfinished calls remaining in B_1 at time t_1. Similarly, at time t_2 shown in Figure 5(c), only two

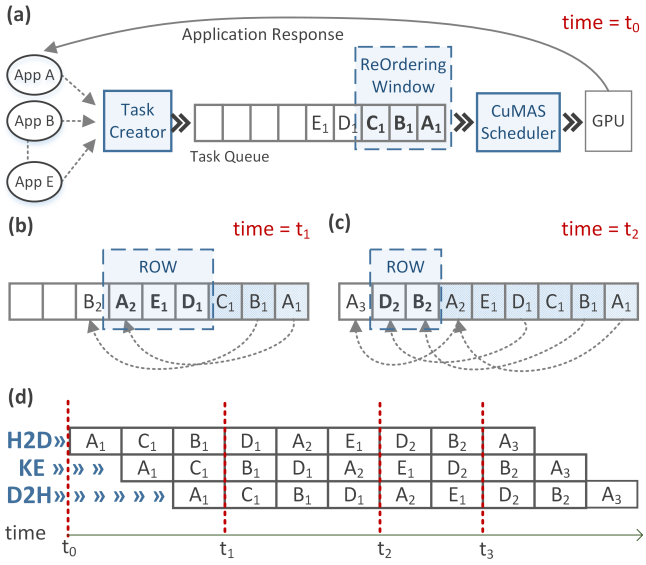

Figure 5: (a) CuMAS task creation, queuing and re-ordering for different applications at initial time t_0, (b) t_1, and (c) t_2. (d) On the bottom: View of the HW pipeline of tasks after re-ordering.[2]

tasks, B_2 and D_2 are included in ROW and A_3 is yet to be created after calls in A_2 finish execution.

3.3 Scheduler Invocation

CuMAS is designed for shared GPU environment; therefore we assume there is a continuous flow of CUDA API call requests from existing or new applications sharing the system. The scheduler is invoked in two situations, whichever arises earlier:

- PCI-e uplink occupation of the H2D calls in a previously scheduled window is about to finish; and
- Total number of tasks ready to execute reaches the ROW size.

The first case is necessary to keep the resources busy as long as there are more tasks to process. In the second case, the window size w is to control the trade-off between the room to exploit better overlapping and per-task re-ordering distance. If w is kept large, the scheduler will have more opportunities for re-ordering, hence allowing a higher transfer-execution overlap. On the other hand, if w is smaller, then the wait time between initial call issue and actual execution will vary less due to smaller distance of task re-ordering. Also, larger w values will increase runtime overhead due to the complexity of the scheduling algorithm. We evaluate the effects of ROW size w in detail in Section 6.

4. CUMAS: SCHEDULER

Scheduler is the core component of CuMAS and it takes a set of ready-to-execute CUDA applications as its input. We assume that the underlying architecture has three concurrent resource channels: up-link PCI-e bus, GPU computation, and down-link PCI-e bus. The goal of the scheduler is to find an ordering that minimizes total execution time by maximizing the overlap between kernel executions and two-way data transfers of successive applications.

CuMAS scheduler takes the tasks in the re-ordering window (ROW) as input and produces a sequence between the tasks as output. The sequencing decision is based on the estimated duration of the tasks in current window and the tasks in the previous window which are still executing. The search space of the optimal ordering is limited to the possible permutation of the tasks in current ROW and such problems are proven to be NP-Hard [7]. Next, we discuss how CuMAS scheduler models and solves the problem of finding an optimal sequence for the tasks in a given ROW.

Our scheduler follows the '*flow shop*'[5] sequencing model, which is well studied in applied mathematics literature. In this model, there are n jobs and m machines with different processing times p_{ij}. Jobs consist of a sequence of operations o_k with precedence constraints and each operation must be processed on a single machine. The goal is minimize the maximum job completion time (C_{max}). The sequencing triple to identify the problem in the literature is represented with $F|prec|C_{max}$.

We establish the correspondence between our proposed scheduler and the *flow shop* problem as follows. Each CUDA application correspond to a series of *task*s (i.e., *job*s) where each CUDA call represents an *operation* of a task. We map machines m to resources S, P and R, which represents host-to-device (H2D) PCI-e bus (i.e., send), GPU itself (i.e., process), and device-to-host (D2H) PCI-e bus (i.e., receive), respectively. Each *operation* in a *task* is one-to-one mapped to the machines (i.e., S, P and R) and these operations have a precedence constraint $S \rightarrow P \rightarrow R$.

We define an *operation* as an atomic GPU call, a *stream* as an ordered series of *operations*, and *task* as a consecutive subset of operations belonging to the same stream. In CUDA terminology, *operation*s correspond to CUDA API calls such as *memcpy()* and *kernel* launches. A *stream* represents the entire CUDA application [3] and it is analogous to *CUDA streams* which guarantees the serial execution of the CUDA API calls issued by the owner application. A *task* represents a consecutive subset of the operations from a stream. A *stream* is composed of one or more tasks. Tasks are the basic unit of scheduling in CuMAS, and only one task from a given stream can be scheduled and run at a given time. Each *operation* has an associated estimated duration of completion and each *task* maintains the cumulative sums of execution time for each operation type (i.e., S, P, and R) that it contains.

4.1 Scheduling Considerations

In a real-life scenario, the scheduling algorithm will be repeatedly run on a queue as new tasks arrive. We refer to each of these iterations as a '*a scheduling round*' and the goal of the scheduler is to minimize the total time for an execution round. Our scheduler treats the three resource channels (S, P and R) as pipeline stages for a task, but tasks may run in parallel as long as there is only one task executing on a channel at a given time.

[2]In the figure, for the sake of simplicity, calls are shown with rectangles of same size, although their execution times may actually vary.

[3]In this work, we assume that the CUDA application does not use multiple streams, which is the default case for the majority of the applications in the two popular benchmark suites, Rodinia and CUDA SDK, that we use in our evaluation.

The scheduler finds an ordering between ready to execute tasks so that overall system utilization is maximized. To achieve this, the scheduling decision must be based on: the durations of S, P, and R operations in each task; the delays occurring in S, P, and R channels due to resources being busy with operations from previous tasks; and the total memory usage requirement of the set of tasks that will be occupying at least one of the resource channels at any given time.

4.2 Formulating the Total Execution Time

The total execution time T_{total} for a given order of N tasks can be represented as a function of the time to send the data for all N tasks; the time to wait for each resource S, P and R to become available (i.e., delays); and the time required to process and receive data for the last task.

Let a task i be defined as a triple of the execution times of send, process and receive operations: $\tau_i = [s_i, p_i, r_i]$. Send channel (S) can be assumed to have no delays, since S is always the first operation in each task and is not dependent on completion of prior operations in the same task. However, P and R channels may include some delays due to stalls in previous operations belonging to the same or previous tasks. The delays that need to be inserted right before executing p and r operations of i^{th} task are represented by *positive definite* functions $\delta^p_{\tau_i}$ and $\delta^r_{\tau_i}$, respectively, and they can be expressed with equations given in 1 and 2.

$$\delta^p_{\tau_i} = \delta^p_{\tau_{i-1}} + p_{i-1} - s_i \tag{1}$$

$$\delta^r_{\tau_i} = \delta^r_{\tau_{i-1}} + r_{i-1} - p_i - \delta^p_{\tau_{i-1}} \tag{2}$$

The processing of a task i cannot start until either of the following finishes: sending operation s_i of current task or processing operation p_{i-1} of previous task. However, if there are any delays $\delta^p_{\tau_{i-1}}$ before p_{i-1}, then P resource will not be available until the larger of the following two finishes: s_i or $p_{i-1} + \delta^p_{\tau_{i-1}}$. If s_i takes longer than $p_{i-1} + \delta^p_{\tau_{i-1}}$, then we should not insert any delays for p, therefore $\delta^p_{\tau_i}$ is a positive definite function.

The receive delay $\delta^r_{\tau_i}$ is similar to processing delay, but it also takes into account of the processing delay $\delta^p_{\tau_{i-1}}$ of the previous task due to propagated P resource idleness to the R channel.

Using the delay equations, total time can be expressed as the summation of total send time, $\delta^P_{\tau_i}$ and $\delta^R_{\tau_i}$ for the last task as well as the the duration for processing P and R operations of the last task, p_N and r_N respectively:

$$T_{total} = \sum_{i=1}^{N} s_i + \delta^p_{\tau_N} + p_N + \delta^r_{\tau_N} + r_N \tag{3}$$

4.3 Finding a Solution

Due to recursive conditionals in equations 1 and 2, a closed form solution could not be obtained for the total time given in Equation 3. Moreover, since our scheduling decision is indeed an ordering problem and we are not looking for values of variables, a closed form solution will not help in finding the optimal ordering. Therefore, the complexity of calculating total time of a given ordering is $O(N)$, due to a single iteration over N tasks.

A brute force search on the total execution times across all possible task permutations would give us the optimal execution time, hence the schedule. However, complexity of such a search is $O(NN!)$ which becomes impractical for large values of N.

An approximate solution to the problem can be obtained via *dynamic programming* (DP) in exponential time, which is much faster than the factorial time for larger values of N. The DP approach relies on the incremental representation of total execution time based on a subset of the solution space and can be described as follows.

Let $T = \{\tau_1, \tau_2, ..., \tau_N\}$ be the set of tasks to be scheduled and let U be a subset of T, $U \in T$. We define the 'near-optimal' completion time $C(U)$ of a given subset U as follows.

$$\begin{cases} C(\{\tau_i\}) = c(\tau_i, \{\emptyset\}), & \text{if } |U| = 1 \\ C(U) = min_{\tau_i \in U}[C(U - \{\tau_i\}) + c(\tau_i, U - \{\tau_i\})], & \text{if } |U| > 1 \end{cases} \tag{4}$$

where the incremental cost function $c(t_i, U)$ is defined as:

$$\begin{cases} c(\tau_i, \{\emptyset\}) = s_i + p_i + r_i, & \text{if} |U| = 0 \\ c(\tau_i, U) = \delta^r_{\tau_i} + r_i, & \text{if } |U| > 0 \end{cases} \tag{5}$$

Equation 4 finds the task $\tau_i \in U$, which minimizes the total execution time when appended to the end of the near-optimal ordering of subset $U - \{\tau_i\}$. The completion time $C(U)$ of the set U relies on the minimum completion time of subset $U - \{\tau_i\}$ plus the cost $c(\tau_i, U - \{\tau_i\})$ of appending τ_i to the end of a the set $U - \{\tau_i\}$.

Equation 5 defines the cost function $c(\tau_i, U)$, which is basically the receive operation r_i plus the receive delay δ_{τ_i} required before the operation. As described previously, the calculation of δ_{τ_i} relies on the ordering of the elements in the set U. The base case for $c(\tau_i, U)$ is when U is empty and it equals to the summation of all operations $(s_i + p_i + r_i)$. Since overlapping is not possible for the first element, s_1 and p_1 operations of the first task directly contributes to the total execution time, whereas the execution time is incremented by only $r_i + \delta_{\tau_i}$ when there are other tasks in U which overlap with s_i and p_i.

The dynamic programming approach starts with the minimum possible subsets where $|U| = 1$ and grows these subsets by iteratively appending tasks τ_i. The ordering of the subsets giving the minimum cost is saved and used in future iterations. We employ this DP approach in our scheduling algorithm.

4.4 Scheduling Algorithm

The CuMAS scheduling algorithm given in Algorithm 1 finds an ordering of tasks using the method described above. The algorithm is invoked by the run-time as the conditions given in the previous Section are satisfied. [Lines 1-2] The scheduling algorithm takes the set T of all tasks as input and outputs a near-optimal ordering $\Omega_{min}(T)$ of T. [Line 3] The algorithm iterates through the smallest to largest subset size i. [Line 5] For every subset size i, we look all subsets $U \in T$ that has size i. [Line 5-6] We want to find the minimum completion time $C_{min}(U)$ and an ordering $\Omega_{min}(U)$ which gives $C_{min}(U)$. [Line 7-10] If $|U| = 1$ then $C_{min}(U)$ is simply the cost $c(\tau_k, \{\emptyset\})$ of the only task τ_k. [Line 12] Otherwise, we iterate through all tasks $\tau_k \in U$. [Line 13] If any τ_k is exceeding the memory requirements when appended to the end of the minimum ordering of Ω_{min}(U-$\{\tau_i\}$), then we do not consider this ordering as minimum. [Line 16] We

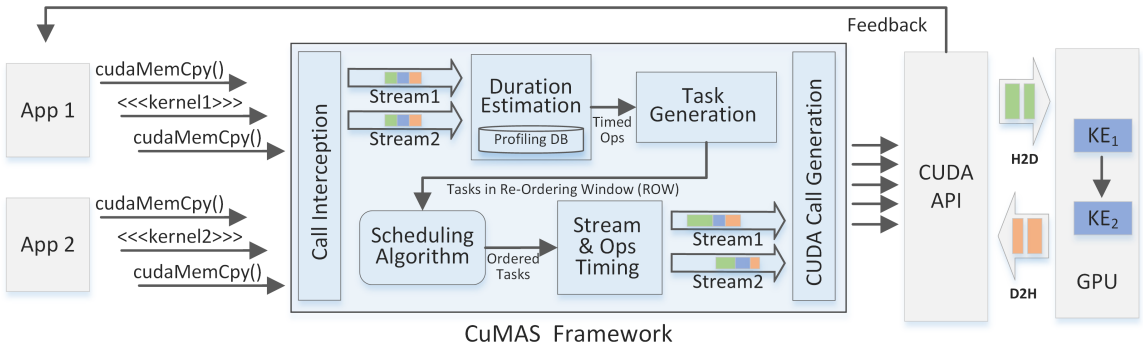

Figure 6: CuMAS framework, highlighted in the middle with shaded box.

calculate the completion time $C(U)$ of subset U where τ_k is the last element. [Line 17-19] If the new $C(U)$ is less then the $C_{min}(U)$ found so far, then we save both the minimum completion time and ordering for the case where τ_k is the last element.

The last iteration of the outer loop has only one subset where $U = T$. In this iteration, the $\Omega(U_{min})$ found after the most inner loop will be the output $\Omega_{min}(T)$.

4.5 Complexity

The outer loop in Line 3 is iterated N times and the selection of subsets U with size i results in the loop at Line 5 and the innermost loop to be iterated $\binom{N}{i}$ and $\sum_{i=1}^{N} i\binom{N}{i}$ times, respectively, resulting in a complexity of $O(N^2 2^{N-1})$. Although this complexity is still exponential, it grows much slower than $O(NN!)$.

DP solution is faster than the brute force solution for any large N, however, it may not always yield to the optimal schedule. The recursive equation in (4) relies on the assumption that the minimum cost solution for a set U will always include the subset $U - \{\tau_j\}$. However, this may not always be true. We will evaluate the effectiveness of our proposed algorithm in the evaluation section.

Algorithm 1 CuMAS Scheduling Algorithm

```
1: Input: Task set T = {τ1, τ2...τN}
2: Output: Minimal ordering Ωmin(T) = (τω1, τω2...τωN)
3: for i = 1 to N do
4:     for each U ∈ T where |U| = i do
5:         Cmin(U) =FLOAT_MAX
6:         Ωmin(U) = {∅}
7:         if |U| = 1 then
8:             C(U) = c(τk, {∅}) where U^i = {τk}
9:             Set Cmin(U) = C(U)
10:            Set Ω(Umin) = {τk}
11:        else
12:            for each τk ∈ U do
13:                if MaxMem(Ωmin(U-{τi})∪{τi}) then
14:                    continue
15:                end if
16:                C(U) = Cmin(U − {τk}) + c(τk, U − {τk})
17:                if C(U) < Cmin(U) then
18:                    Set Cmin(U) = C(U)
19:                    Set Ωmin(U) = Ωmin(U − {τk}) ∪ {τk}
20:                end if
21:            end for
22:        end if
23:    end for
24: end for
```

5. CUMAS: FRAMEWORK AND RUNTIME

CuMAS framework is composed of several components as shown in Figure 6. *Call Interception* library dynamically captures CUDA calls issued by the applications assigned to the server. *Duration Estimation* component uses offline profiling results to estimate kernel execution times and a data-transfer model for estimating durations for D2H and H2D calls. *Task Generator* creates tasks from the captured calls and queues them. *The scheduler*, which is the core framework, re-orders the tasks in the ready-queue to improve overall utilization via transfer-execution overlapping. Once proper events and timers are inserted by *Stream&OPs Timer*, *CUDA Call Generator* re-issues CUDA calls in the given scheduling order. CuMAS also employs an offline profiler to measure standalone kernel execution times that are required by the scheduler. We elaborate on the details of these components in the rest of this section.

5.1 Call Interception

The entry point of CuMAS framework is *Call Interception* which employs wrapper functions for a subset of the original CUDA API. CuMAS is attached to any existing CUDA C++ binary via the interception library through LD_ PRELOAD flag, hence requiring no modifications to the user code. Each application is allocated a dedicated CUDA stream and as the interceptor captures new calls a new operation is created for each call, along with all configuration parameters, and added to the application's stream.

Our library intercepts *kernel* launches, *memcpy()* and *cudaDeviceSync()* calls. H2D *cudaMemcpy* and *kernel* launches (KE) are immediately returned to the caller, regardless of whether they are synchronous or asynchronous. This is necessary to keep the applications continue issuing their CUDA API calls so that larger tasks are created with as many operations as possible to maximize overlapping in the generated schedule. On the other hand, synchronous D2H *cudaMemcpy* calls are not immediately returned to user and the calling procedure is blocked until all queued operations *plus* the last D2H call for the stream have been converted to a *task*, scheduled, and executed. D2H *cudaMemcpy* calls are always treated as blocking (i.e., synchronous) to prevent any miscalculation of conditional statements that depend on the retrieved GPU results.

5.2 CuMAS Runtime - HW Interaction

Using the task order returned by the scheduling algorithm, we issue CUDA API calls for each operation in the task list. Figure 7 illustrates the how CUDA calls are captured by CuMAS and scheduled for execution when the PCI-e bus and GPU becomes available.

CuMAS *runtime* is a daemon process which communicates with applications via CuMAS *interceptor*. Whenever the interceptor captures a CUDA call, it lets the *runtime* using a public POSIX message queue (*m_queue*). Upon the receipt of first message from each application, the *runtime* creates a private *m_queue* with the process id of the caller and tells the interceptor either *block* the call or *continue* collecting calls until a task is created for the caller. In the given example, both applications are instructed to *continue* for the H2D and kernel calls.

When the *runtime* receives D2H call for an application, it instructs *the interceptor* to *block* the caller and *wait* for the scheduling decision for that specific application. The *runtime* daemon constantly monitors the events corresponding to previously called CUDA calls so that it can call the scheduler whenever resources are idle. Once a schedule is determined, the *runtime* messages the caller applications to issue their collected calls. Here, it is important to note that, collected CUDA calls are called by the same process, to prevent any inaccessible host memory pointers due to inter-process security restrictions.

If the CUDA call does not specify any streams, the interceptor creates a CUDA stream and modifies *cudaMemcpy()* and kernel call arguments accordingly. Also, to enable overlapped execution, the host memory locations pointed by *cudaMemcpy()* calls are converted into pinned memory using *cudaHostRegister()* call. CuMAS assumes that the kernels do not use any mapped pointers to host memory.

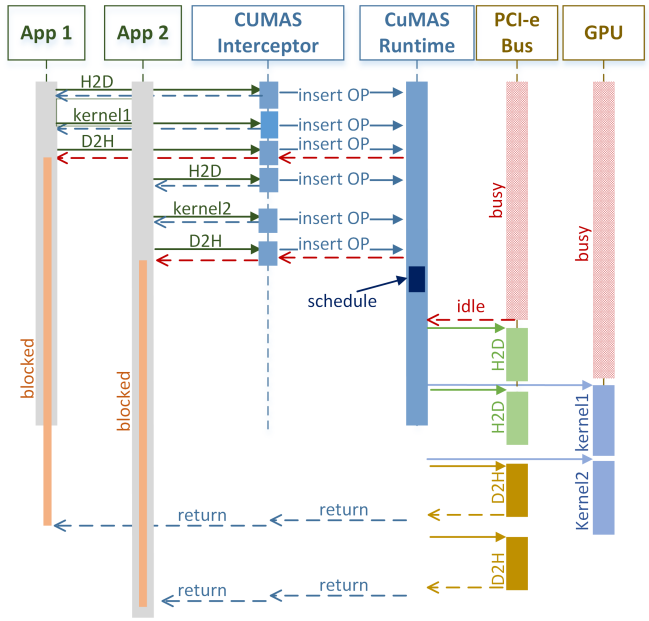

Figure 7: Activity diagram showing the interaction between CUDA application, CuMAS framework and the GPU. (The spacing and the length of rectangles do not represent actual timings.)

5.3 Duration Estimation & Profiling

Accurate estimation of transfer and kernel execution times is crucial for our scheduler to produce an optimal schedule. CuMAS uses both profiling data and mathematical model to estimate durations for S, P and R type of operations of the tasks in the ROW.

Data transfer time (s and r) is dependent on the size and direction of the data to be transferred. Data transfer time is a linear function of the total data size plus a constant, and the data transfer rate can be modeled with the following equation where x is the data size while a and b are device constants:

$$y = x/(ax + b) \tag{6}$$

The constants a and b are measured through experiments and linear curve fitting. The details of the measurement are given in the evaluation section.

Kernel execution time (p), on the other hand, is dependent on the kernel and the size of the input data, hence total thread block (TB) count, on which the kernel operates. Estimating p is harder since it varies across applications therefore CuMAS requires an offline run on a fraction of input data and obtain *msecs/TB* metric for each profiled kernel. We maintain a database of profiled kernels associated with the corresponding *msecs/TB* metric for future executions.

Later, during runtime, we use the dimension of the grid (i.e., TB count) specified during kernel launch and we linearly scale the metric up to estimate the execution time of the intercepted kernel. The accuracy of this simple estimation technique relies on the following factors:

- Same number of threads per TB is used for the profiling kernel and the actual execution.
- Total number of TBs used for the profiling kernel are large enough to utilize all SMs at a given time.
- There is a linear correlation between the grid size and kernel execution time.

If no profiling information is provided on the first execution of a specific kernel, CuMAS does not re-order the tasks containing such kernels and immediately issues them to the CUDA API call queue. If that kernel is encountered again in the future, initial execution time is pulled up from DB.

6. EVALUATION

In this section we first describe the details of our experimental setup and then present our results.

Architecture: We evaluate CuMAS on NVIDIA's Tesla K40c series GPU attached to an AMD Opteron 6100 based system. The GPU supports PCI-e v3.0 and it has 15 SMX units each having 192 CUDA cores accessing 12GB of global DDR3 memory. K40c has a shared L2 cache size of 1.5MB. Host has 64 cores organized as 8 NUMA nodes connected with AMD's HyperTransport interconnect. For optimal PCI bandwidth we only use the cores in the NUMA node 5, which is directly connected to the GPU via the south-bridge.

Applications: We use a total of 12 applications from Rodinia Benchmark suite and CUDA SDK samples. Table 1 lists total number of H2D and D2D calls along with the transfer size, number of kernel calls and tasks created for each application. To evaluate a mixed load of long and short running CUDA applications, we have adjusted the input sizes so that total runtimes vary between 500 msecs and 2 seconds.

Although some of the applications (BlackScholes, lud, nw,

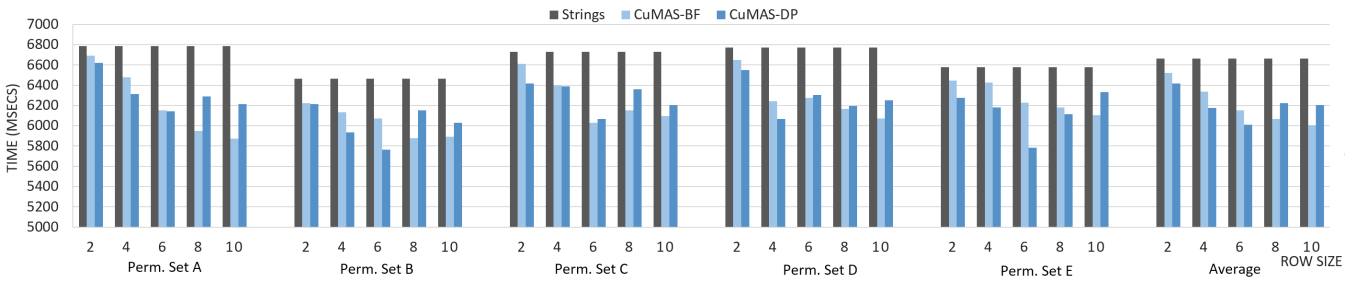

Figure 8: Total execution time with varying number of ROW sizes.

gaussian and particlefilter) in our test-bed issues many number of kernel executions, they do not end up in creation of multiple CuMAS tasks. Our task scheduler merges consecutive kernel calls into fewer CuMAS tasks based on the state diagram given in Figure 4. This is mainly due to either redundant computations or statically scheduled kernel launches by the application programmer and the merge does not affect the accuracy of the computation.

Methodology: In our evaluation, we have assumed that first tasks of each application arrive the CuMAS task queue around same time. To achieve this behavior, we initially force our runtime to keep collecting CUDA calls until it captures at least one task from each CUDA application. In our experiments, we have measured data transfers and kernel executions only and excluded the time spent on host-side data and device initialization, cudaMalloc() and cudaFree() calls from our analysis.

We have made all 12 applications run to finish and used a round robin policy to insert the new tasks that are created dynamically as applications progress. Other common policies like fair scheduling were not used because current GPU architecture does not support preemptive kernel execution. Due to many possible permutations of 12 applications, the total executions times are largely affected by the order in which the initial tasks of each application are issued. Therefore, we have performed 25 runs and in each run we started with a randomly selected permutation of the given applications. For each run, we have varied Re-Ordering Window (ROW) size from 2 to 10.

6.1 Execution Time

In our experiments we compare CuMAS with the only (to the best of our knowledge) multi-application automatic transfer/execution overlapping technique, *Strings* [24]. As described in the introduction, *Strings* exploits overlapping only if two different type of CUDA calls are issued around same time. On the other hand, for CuMAS, we have used two scheduling algorithms; *CuMAS-BF*, which uses brute force search over all possible permutations of the tasks in the ROW and *CuMAS-DP*, which uses the faster dynamic programming based heuristic as described in subsection 4.

Application	H2D#	D2H#	KE#	Task#
b+tree	15 (402 MB)	1 (40 MB)	2	2
backprop	5 (1568 MB)	1 (830 MB)	2	2
BlackScholes	3 (1200 MB)	2 (800 MB)	256	1
gaussian	3 (19 MB)	3 (19 MB)	3070	1
hotspot	2 (1212 MB)	2 (606 MB)	1	1
lavaMD	4 (43 MB)	3 (18 MB)	1	1
lud	1 (205 MB)	1 (205 MB)	670	1
nw	2 (1072 MB)	1 (536 MB)	511	1
particlefilter	6 (1043 MB)	1 (1 MB)	77	2
pathfinder	2 (2139 MB)	3 (1 MB)	3	1
srad_v2	2 (1073 MB)	1 (1073 MB)	4	2
vectorAdd	2 (1608 MB)	3 (804 MB)	1	1

Table 1: Total data transfer sizes and operation/task counts for each application.

We have grouped 25 initial permutations of 12 application into 5 permutation sets, A to E, to better understand the execution discrepancies across different sets. The results given in Figure 8 show the average total execution time for each permutation set as well as the global average. x axis corresponds to different sizes of ROW and y axis denotes the total execution time in milliseconds. ROW size differences only affect CuMAS and *Strings* results remain same, since *Strings* relies solely on the incoming call order and does not involve re-ordering.

The results show that the benefits of re-sequencing CUDA calls is significant even with a window size of 2. In most of the permutation sets, increasing ROW sizes reduce the total execution time up to 14% on average and up to 44% percent when compared to the execution of *Strings* with the worst case initial permutation.

For small ROW sizes (2-6) CuMAS-DP performs better than CuMAS-BF due to the way we implement the two techniques. For DP, while calculating the schedule for the current ROW, cost calculation recursively depends on the tasks from previous ROWs, therefore the finish times of the CUDA commands in those tasks are taken into account as well. On the other hand, in BF, we find the optimal execution time of a given ROW by only looking at the tasks inside the current ROW regardless of the task execution times in the previous ROW. This causes possible resource idleness between the bordering tasks in consecutive ROWs.

For larger ROW sizes (8-10), we observe that CuMAS-BF performs better than CuMAS-DP since the target permutation set grows significantly larger than the heuristic algorithm can efficiently address. Despite the high overhead of BF technique for these ROW size, the resulting schedule is fast enough so that the total execution time is still less than the heuristic (DP). Moreover, since the scheduler is invoked right after the send operation of the last task in the ROW, algorithm execution overlaps with the remaining P and R operations of the last task, hence the overhead is partially hidden. However, for ROW sizes larger than 10, BF scheduling overhead is enormously high (as explained in sub-section 6.3) and is not compensated neither by the overlapping between pipeline stages nor the performance gain.

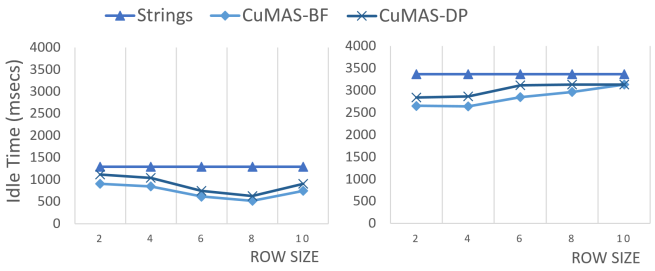

Figure 9: Idle time observed two resource channels: P[GPU] on the left (a) and R[D2H] on the right (b)

ROW size of 10 is evaluated for the sake of analysis only and it is not a practical value for real systems. Although CUDA supports up to 32 concurrent SW streams, latest generation of NVIDIA GPUs employ only 8 HW stream controllers. Any ROW size value, hence total stream count, above 8 is subject to serialization in HW.

It is also important to note that the execution times shown in Figure 8 are the combined outcome of the speedup provided by the scheduling techniques and their overhead. Since there is no straightforward way to clearly isolate the effects of possible overheads on the total time in pipelined execution schemes, we will evaluate overheads separately in the rest of this section.

6.2 Resource Idleness

To better understand how CuMAS improves the total execution time, we have measured the idle time spent by P(GPU) and R(D2H) resources. These times correspond to the sum of non-busy periods between the first and last KE call and D2H call, respectively, across the scheduled tasks in a ROW. We have excluded the idle time for S(H2D) channel, because we send the data for the next task as soon as the H2D operations in the previous window finish. There is no accumulated delay or idleness for S resource channel, provided that there is always a ready to execute task in the task queue.

The results given in Figure 9 show that both BF and DP approaches manage to keep the idleness lower than *Strings*. Another observation is that the idle times are more significant in R channel, which are also affected by the accumulated delays in the P channel. Also, the idleness difference between the *Strings* and CuMAS scheduling becomes less significant as the ROW size increases. However, although resource idleness gives an overall idea on the reasons for speedup, call overlapping and cumulative resource utilization are hard to quantify and the overall speedup cannot be directly related with the idleness values given here.

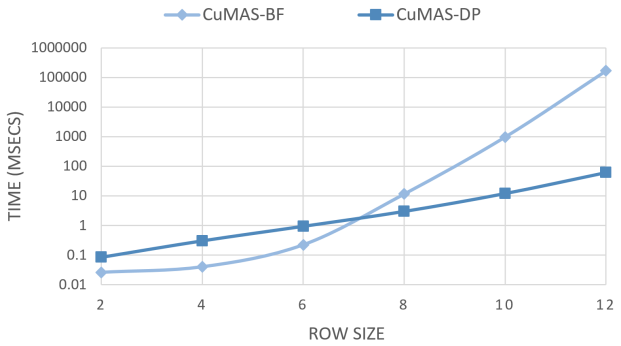

Figure 10: CuMAS scheduling overhead.

6.3 Scheduling Overhead

The cost of scheduling for dynamic frameworks like CuMAS is crucial for higher system utilization and it corresponds to the majority of runtime overhead. We measure how much serial CPU time is taken to perform the scheduling for varying ROW sizes – the results are shown in Figure 10.

As stated previously in Section 3, there is a tradeoff between the increased overlapping benefits and the scheduling overhead as the ROW size is incremented. Although the brute force search approach (CuMAS-BF) provides the best speedup as shown in Figure 8, the overhead of enumerating all task permutations in the window becomes considerably larger as ROW size exceeds 6. It takes 12 and 972 milliseconds in total for CuMAS-BF to search the best ordering for the tasks in a window of size 8 and 10, respectively. On the other hand, CuMAS-DP manages to keep the overhead at 2 and 12 milliseconds, respectively, for the same window sizes. For a ROW size of 12 tasks, CUMAS-BF takes considerably longer (168 seconds), which is not acceptable for a runtime solution. On the other hand CuMAS-DP overhead (61 msecs) still remains under practical limits.

A ROW size of 8 ends up in negligible overhead, which is under 0.5%, for both CuMAS scheduling approaches. Considering the HW stream controller limit, 8, we may conclude that even with the exponential and factorial scheduling complexities, both CuMAS call re-ordering policies (DP and BF) are practical enough to be deployed in real systems.

6.4 Data Transfer and Kernel Execution Time Estimation

Estimating transfer and kernel execution times accurately is essential for CuMAS to exploit maximum resource utilization. We build a linear model for data transfer times whereas we use a profiling based approach to estimate kernel times.

For data transfer time estimation we have executed H2D calls with data sizes varying from 4KB to 100KB and plotted the execution time (solid line) in Figure 11. We have fitted (dashed line) our measurement to the linear model given in Equation 6 and obtained architecture specific parameters $a = 0.0002$ and $b = 0.0072$.

To estimate kernel execution times, we have profiled each application using kernels with thread block (TB) counts

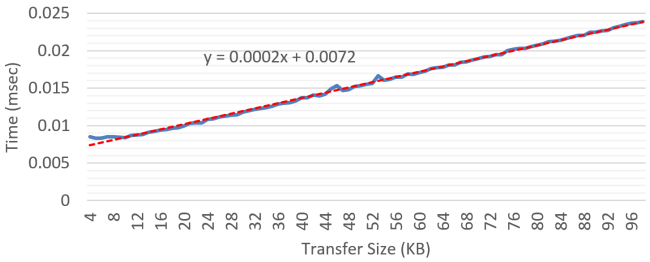

Figure 11: Data transfer times (solid) and corresponding curve fitting (dashed).

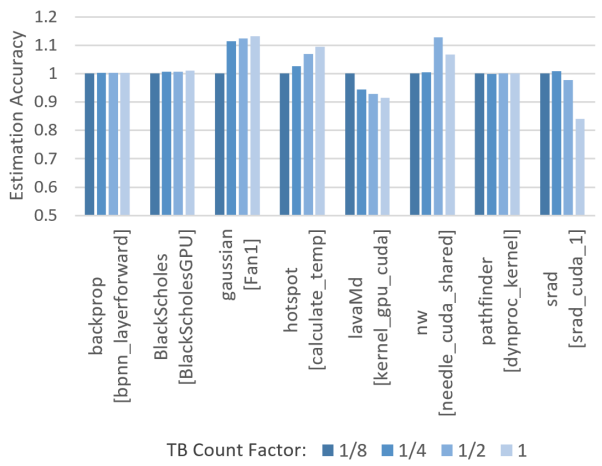

Figure 12: Kernel duration estimation accuracy for varying TB count factors.

equaling to $1/8^{th}$ of the kernels, which are to be used in the experiment. We have calculated a *msecs/TB* value for each profiled kernel and we have compared these estimated values with the real kernel runs. We have doubled the TB count at each run until we reach the TB count of the kernel in the actual experiment. Figure 12 plots estimation accuracies for 8 different kernels. The kernels used in the initial profiling with $1/8^{th}$ TB count are taken as baseline with estimation accuracy of 1.0.

The verification results showed that the *msecs/TB* metric we acquired by running the kernels with a fraction of the actual TB counts, estimated the execution times for larger kernels with a maximum error rate of 15% and an average rate of 4%. Such a high rate of estimation accuracy enabled CuMAS to exploit maximum overlapping between CUDA calls.

It is also important to note that, although simple interpolation works for most of the applications in Rodinia benchmark suite and CUDA SDK, the estimation will be incorrect if the correlation with the TB count is not linear (e.g. quadratic equations, kernel bodies with arbitrary loop iterations). In the future, CuMAS profiler can be extended to have more accurate estimation for such applications via compiler analysis or runtime performance profiling. Such techniques would require derivation of non-linear estimation curves by using multiple execution time-TB count data points obtained during training phase. More information about the literature on GPU performance estimation are given in the next section.

7. RELATED WORK

Concurrent kernel execution: Ravi et al.[21] identified and proposed SW based concurrent kernel execution mechanisms using spatial and temporal sharing. They characterized several kernels and paired them using the two sharing methodologies. Adriaens et al.[1] have implemented HW based spatial sharing solutions to improve utilization in the existence of smaller kernels with less thread blocks. Elastic kernels proposed in [19] use different concurrency policies for various classes of kernels. The study in [26] developed two HW based preemptive execution policies, context switch and drain, to replace a running thread block with another. Chimera [20] has improved the pre-emption with an additional policy, flush, allowed dynamic selection of policies depending on the load characteristics. Jog et al.[11] have studied the on-GPU memory system to improve throughput in the existence of concurrent kernels.

Multi application scheduling: The researches in [10] and [3] have proposed SW runtime environments to handle data allocation and transfers on-the-fly by keeping track of dependencies across kernel executions. Using a similar technique, Sajjapongse et al.[23] distributed kernels to multiple GPUs, to reduce the wait times on kernel dependencies. TimeGraph [12] is a driver-level GPU scheduler for graphics APIs and it supports various priority-aware scheduling policies.

Data transfer / kernel execution overlapping: Huynh et al.[9] proposed a transfer/execution overlapping enabled framework for streaming applications written using StreamIt language. Similarly, GStream[25] provides a graph processing methodology to handle overlapping. Among the few general purpose automatic overlapping work, Lustig et al.[15] proposed HW extensions to enable the detection of the transferred bits so that the kernel can start execution as soon as the required data arrives. PTask [22] is an OS task scheduler extension and uses existing OS scheduling policies and an API to exploit transfer/execution overlapping. However, this work does not employ task re-ordering. Helium [16] focuses on the task graph of a single application and performs a compiler-based DAG analysis to exploit any overlapping if possible. However the optimizations in this work are limited to a single application.

Kernel performance estimation: Hong et al.[8] have built an analytical model to estimate the overall execution time of a given kernel based on the number of parallel memory requests issued within a warp. A later study by Baghsorkhi et al.[2] has proposed an adaptive performance model to identify the bottlenecks in execution flow by taking bank conflicts, control divergence and uncoalesced memory accesses into account, in warp level. Both studies have used average warp execution times to find thread block execution times and scale them to the total number of TBs. A more recent study[28] have proposed machine learning techniques for scalable performance estimation on varying architectures and applications. They employ a runtime that collects performance counters and feeds them to a neural network that decides which scaling curve should be applied to properly estimate the execution time based on the application characteristics and number of SMs employed by the GPU.

8. CONCLUSION

In this paper, we have proposed CuMAS, a scheduling framework, to enable data-transfer aware execution of multiple CUDA applications in shared GPUs. CuMAS improves overall system utilization by capturing and re-organizing CUDA memory transfer and kernel execution calls, without requiring any changes to the application source code. CuMAS is evaluated on an NVIDIA K40c GPU, using a suite of 12 CUDA applications and it is shown to improve the total execution time by up to 44% when compared to best known automatic transfer/execution overlapping technique.

9. ACKNOWLEDGMENTS

This work is supported by NSF Grants CCF-1423108, CCF-1513201, and CCF-1318103 to UC Riverside.

10. REFERENCES

[1] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for gpgpu spatial multitasking. In *HPCA'12*, pages 1–12. IEEE, 2012.

[2] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu. An adaptive performance modeling tool for gpu architectures. In *PPoPP '10*, pages 105–114. ACM, 2010.

[3] M. Becchi, K. Sajjapongse, I. Graves, A. Procter, V. Ravi, and S. Chakradhar. A virtual memory based runtime to support multi-tenancy in clusters with gpus. In *HPDC'12*, pages 97–108. ACM, 2012.

[4] P. Boudier and G. Sellers. Memory system on fusion apus - the benefits of zero copy. In *AMD fusion developer summit*. AMD, 2011.

[5] P. Brucker. *Shop Scheduling Problems*, volume 3. Springer, 2007.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC'09*, pages 44–54. IEEE.

[7] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial & Applied Mathematics*, 10(1):196–210, 1962.

[8] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ISCA'09*, pages 152–163. ACM, 2009.

[9] H. P. Huynh, A. Hagiescu, W.-F. Wong, and R. S. M. Goh. Scalable framework for mapping streaming applications onto multi-gpu systems. In *PPoPP'12*, volume 47, pages 1–10. ACM, 2012.

[10] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August. Dynamically managed data for cpu-gpu architectures. In *CGO'12*, pages 165–174. ACM, 2012.

[11] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das. Application-aware memory system for fair and efficient execution of concurrent gpgpu applications. In *GP-GPU'14*. ACM, 2014.

[12] S. Kato, K. Lakshmanan, R. R. Rajkumar, and Y. Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *USENIX ATC'11*, page 17, 2011.

[13] E. L. Lawler, J. K. Lenstra, A. H. R. Kan, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. *Handbooks in operations research and management science*, 4:445–522, 1993.

[14] J. Y. Leung, O. Vornberger, and J. D. Witthoff. On some variants of the bandwidth minimization problem. *SIAM Journal on Computing*, 13(3):650–667, 1984.

[15] D. Lustig and M. Martonosi. Reducing gpu offload latency via fine-grained cpu-gpu synchronization. In *HPCA'13*, pages 354–365. IEEE, 2013.

[16] T. Lutz, C. Fensch, and M. Cole. Helium: a transparent inter-kernel optimizer for opencl. In *GP-GPU'15*, pages 70–80. ACM, 2015.

[17] NVIDIA. Nvidia kepler gk110 architecture whitepaper. http://www.nvidia.com/content/PDF/kepler/NVIDIA Kepler-GK110-Architecture-Whitepaper.pdf.

[18] NVIDIA. Nvidia quadro dual copy engines. https://www.nvidia.com/docs/IO/40049/Dual_copy_engines.pdf.

[19] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *ASPLOS'13*, pages 407–418. ACM, 2013.

[20] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. In *ASPLOS'15*, pages 593–606. ACM, 2015.

[21] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar. Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework. In *HPDC'11*, pages 217–228. ACM, 2011.

[22] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, pages 233–248. ACM, 2011.

[23] K. Sajjapongse, X. Wang, and M. Becchi. A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with gpus. In *HPDC'13*, pages 179–190. ACM, 2013.

[24] D. Sengupta, A. Goswami, K. Schwan, and K. Pallavi. Scheduling multi-tenant cloud workloads on accelerator-based systems. In *SC'14*, pages 513–524. IEEE, 2014.

[25] H. Seo, J. Kim, and M.-S. Kim. Gstream: a graph streaming processing method for large-scale graphs on gpus. In *PPoPP'15*, pages 253–254. ACM, 2015.

[26] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on gpus. In *ISCA'14*, pages 193–204. IEEE, 2014.

[27] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi gf100 gpu architecture. *IEEE Micro*, (2):50–59, 2011.

[28] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. Gpgpu performance and power estimation using machine learning. In *HPCA'15*, pages 564–576. IEEE, 2015.