A Paradigm Shift in GP-GPU Computing: Task Based Execution of Applications with Dynamic Data Dependencies

Mehmet E Belviranli, Chih-Hsun Chou, Laxmi N Bhuyan and Rajiv Gupta
University of California, Riverside
Riverside, CA
{belviram, cchou011, bhuyan, gupta}@cs.ucr.edu

ABSTRACT

General purpose computing on GPUs have became increasingly popular over the last decade. Scientific applications with SIMD computation characteristics show considerable performance improvements when run on these massively parallel architectures. However, data dependencies across thread blocks significantly impact the degree of achievable parallelism by requiring global synchronization across multi-processors (SMs) inside the GPU.

In order to efficiently run applications with inter-block data dependencies, we need fine-granular 'task-based execution models' that will treat SMs inside GPU as stand-alone parallel processing units. Such a scheme will enable efficient execution by utilizing all internal computation elements inside GPU and eliminating unnecessary waits during global barriers.

In this paper, we propose a new, dynamic and 'all-in-GPU' task execution framework for executing both regular and irregular data-dependent applications on GPUs. Our runtime eliminates the need for global synchronization and minimize inter-SM communication through distributed queues. In our preliminary experiments run on a Tesla c2050 GPU, we have obtained up to 62% more speedup when compared to centralized queue approach. The overhead of system has been measured as low as 5%.

Categories and Subject Descriptors

D.1.3 [**Software**]: Concurrent Programming—*Parallel programming*

Keywords

GP-GPU Computing; Task Scheduling; Data Dependency

1. INTRODUCTION

Using Graphical Processing Units (GPUs) for general purpose computation have been increasingly popular over the last decade. Massive parallelism provided by hundreds of GPU cores offers considerable speedups for SPMD (single process multiple data) type of computation. Most of the top supercomputers today employ GPUs as primary co-processors in their configuration. Many scientific applications have been ported and optimized to run on GPUs to efficiently process massive amounts of data.

As all good things come with a price, designing algorithms for GPUs requires additional considerations due to the SPMD nature of GPU operations. Lack of efficient hardware communication mechanisms between SMs forces parallel threads running on different SMs either being data independent from each other or synchronize via software barriers [8]. Irregular dependencies will result in statically unpredictable workloads, which are possibly generated during run-time. Due to this dynamically changing behavior of such workloads, algorithms with complex dependencies have been classified as 'unsuitable' for '*data-parallel*' programming approach of GP-GPU computing[6].

We believe GPUs, with its hundreds of cores, can be used to solve problems with '*data-dependent*' characteristics as well. This needs development of a task-based execution model and efficient synchronization techniques somewhat similar to the MIMD operations. The conventional GP-GPU programming paradigms (e.g. OpenCL, CUDA) executes a TB on any SM in no specific order. We propose to treat each thread block as a 'generic task' and smartly assign them into SMs as and when their dependencies are resolved. This will greatly expand the pool of the applications which can be accelerated on GPU, however, implementing the support for this approach presents several challenges.

The key challenge in implementing such a task-based approach is employing an efficient 'synchronization' mechanism between SMs. Checking dependencies between tasks require communication and synchronization between SMs. With the existing CUDA execution model, inter-thread block (TB) communication is possible either via global memory using atomic and memory fence functions which tend to be a very slow with increasing number of thread blocks and data[8]. Although, NVIDIA's latest Kepler GK110 architecture enables device-side finer granular synchronization using *cudaThreadSynchronize()*, using this mechanism as an inter-SM communication method presents several hardware limitations and programmability issues. Overall, minimizing inter-SM communication while resolving dependencies is a priority consideration in building a task-based execution environment.

DIDC 2014, June 23, 2014, Vancouver, BC, Canada.

http://dx.doi.org/10.1145/2608020.2608024.

Another challenge is to decide *'where to execute the scheduler'* (e.g. CPUs or GPUs). The 'serial' and 'iterative' nature of CPU oriented schedulers will result in under-utilization of SIMD-based SP cores when they are placed on GPU. Host controlled scheduling[7, 2] comes with the high overhead of continuous queue transfer between CPU and GPU, hence a huge performance penalty due to slow host to GPU interconnect (i.e. PCI-e) [5].

In this paper, we propose a new, dynamic task execution framework for executing both regular and irregular data-dependent applications on GPUs. We have implement the proposed task-based execution model on a Tesla C2050 GPU and perform preliminary measurements.

Our study makes the following contributions:

- We present a new, CUDA based dynamic task-based execution framework for applications having regular and irregular workload dependencies.
- We implement a novel GPU based light-weight parallel task scheduler, co-existentially running along with the workers, via concurrent kernel execution.
- We compare our results with a centralized queue, all-worker approach based solution, and present the results.

The rest of this paper is organized as follows: In Section 2 we give background and motivation. We present our framework with full details in Section 3. Then, we show the results and evaluation of our framework in Section 4 and conclude the paper with the last section.

2. A PARADIGM SHIFT: TASK BASED EXECUTION ON GP-GPUS

General purpose usage of GPU architectures have enabled considerably faster execution for applications which exploits data parallelism in an SIMD fashion. Hierarchical grouping of hundreds of scalar cores(SPs) into larger symmetric multiprocessors (SMs) provides a fair trade-off between scalability and inter-SM communication. Threads belonging to same thread block (TB) run on the same SM with access to a fast, shared memory and can synchronize with each other very efficiently. On the other hand, they are totally isolated from the threads in other TBs so that applications can easily scale up to thousands of threads without any degradation in performance. Redundant number of thread blocks, which are provided by the programmer, are executed in no specific order by the thread block scheduler, during a specific kernel launch.

2.1 Dependency: An inherent problem of traditional GP-GPU programming

Sacrificing efficient inter-SM synchronization capability in exchange for scalability was a necessary decision made by GP-GPU designers [3]. However, this sacrifice has resulted in severe restrictions on the domain of applications that can be accelerated by GPUs. An ideal GP-GPU application would require a data-parallel region in the program that requires no global synchronization points whereas data dependencies across different threads might require inter-TB communication for computational accuracy.

Except for a limited number of simple kernels, many applications in popular benchmark suites like Rodinia[1] and CUDA SDK Samples implement synchronization mechanisms by dividing parallel regions further into smaller data-independent sub-routines which are then executed on the GPU by consequent kernel launches. However, this approach is hard to design and implement, and also is inefficient for applications having dynamically generated task graphs due to redundant resource allocation required for unpredictable dependencies.

2.2 Task-based Execution Model

Treating SMs like standalone processors, which are capable of running tasks independent of each other, will provide a more generalized approach to address the issues mentioned above. Once applications are represented as task graphs, similar to multi-CPU systems, SMs will be able to consume ready tasks as they become idle.

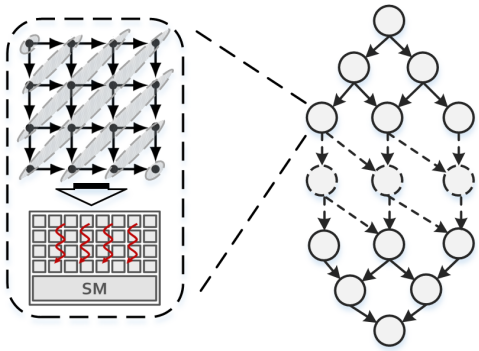

Figure 1: Sample task graph for Heat2D application.

Figure 1 shows a sample task graph for Heat2D application, where each task corresponds to an SIMD parallel region of the application. Entire 2D surface is divided into chunks where each of them is represented by a task. Each task is processed by symmetrical threads in an SM using the standard synchronization techniques. Dependencies between the tasks are represented by the task graph edges and tasks can be executed on different SMs in parallel, subject to the dependency resolution.

In this study, we introduce a new task-based dynamic task execution model. Based on our preliminary experiments, we propose a distributed queue based worker-scheduler mechanism running completely on GPU without relying the host side.

3. PROPOSED FRAMEWORK

In this section we first give an overview of our framework at a high level. Later we will focus on details of our runtime where we explain the scheduler and worker threads' operations.

3.1 Overview of Task-based Execution

An overview of our framework is illustrated in Figure 2. The run-time is composed of N Worker thread blocks(TB) and a scheduler TB, where all TBs continuously run until the application ends. N is equal to the number of SMs. Each SM (interchangeably used with 'worker TB') is associated with a private task queue located in global memory. Queue is decentralized (one per SM) so that each worker TB can read from/write to its own queue without synchronizing with other TBs.

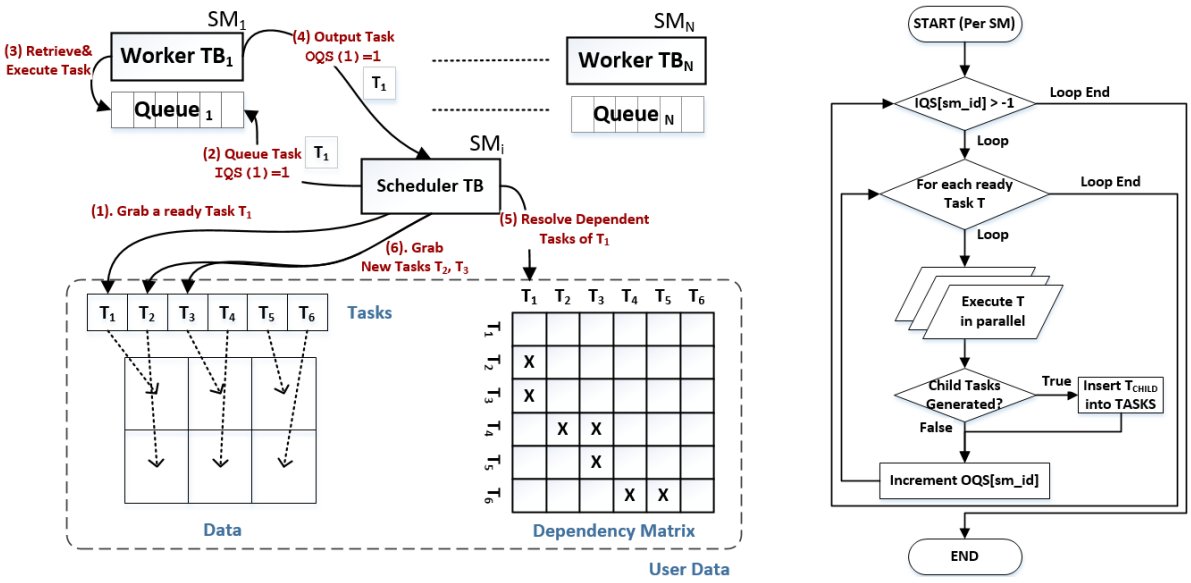

Figure 2: (a) An overview of our proposed in-GPU task scheduling framework[left] (b) Control-flow diagram for worker TB.[right]

Scheduler TB operates in a parallel manner where each thread accesses to a specific element in the task queues associated with each SM. Dependency checks are also done in parallel.

Initial tasks list, dependency graph and the actual data which tasks point to are to be provided by the user, as shown in the bottom part of Figure 2.a. Both static and dynamically populated task lists are supported. In a typical scenario, each task structure contains pointers to the actual data on the GPU global memory.

The proposed task execution follows a sequence of six operations as marked clearly in Figure 2.a.

1. Scheduler initially searches the tasks list and identifies the starting tasks whose *dependencyCounter* is zero (which means that all prior dependencies of the task is resolved and it is *ready* to execute).
2. Scheduler TB inserts the ready task into a proper worker queue based on the scheduling policy. Corresponding worker TB is notified by an increment of the Input Queue Size (IQS) counter, which is implemented as an atomic variable.
3. Worker TB grabs tasks, forwards it to the user application. The application can fully utilize the SM, since worker TB transfer the full control to the supplied user kernel.
4. When the user kernel finishes processing the task, control is transferred back to worker TB. To let the scheduler know that the task is finished, worker TB, increments Output Queue Size (OQS) counter for the corresponding SM. Please note that this counter is for signaling purposes only and there is no separate output queue.
5. Scheduler TB continuously checks each worker TB for outputted tasks in parallel. Once a task is processed by an SM (i.e. OQS is greater than zero), the scheduler TB goes through its dependents via user supplied dependency graph and discovers whether new tasks become ready to be processed (i.e. independent).
6. The process repeats with the new 'ready' tasks. This step replaces step (1) for the rest of the execution.

Both scheduler and worker TBs terminate when the scheduler decides that there are no tasks to process.

3.2 Run-time Components

In this sub-section, we explain several components of our task-based execution model.

(a) Concurrent worker&scheduler kernels: Our runtime is composed of two kernels: worker and scheduler. Worker kernels are wrappers for actual user functions. They also carry consumer operations like grabbing tasks from the distributed queues and writing processed tasks back to the queue. Scheduler thread, on the other hand, is solely transparent from the application code and runs as an on-GPU daemon.

In order to have both scheduler and worker TBs to run simultaneously, our run-time uses concurrent kernel execution. We asynchronously launch two kernels via two different streams with scheduler and worker grid with dimensions of 1 and N, respectively. During kernel launch, GPU hardware scheduler assigns corresponding thread blocks to available SMs according to their resource availability.

(b) Inter-SM Signaling *IQS* & *OQS*: As an efficient way for workers and scheduler to check the entire task queue for any completed or newly inserted tasks, we use two special arrays (i.e. counters), *input_queue_size[]* (*IQS*) and *output_queue_size[]* (*OQS*) to provide an atomic signaling between them. Each worker and scheduler TB continuously checks for *IQS* and *OQS*, respectively. When a worker TB, as illustrated in 2, encounters a greater-than-zero value in *IQS*,this implies that the scheduler has placed a task into that specific worker's queue. Similarly, a positive value encountered by the scheduler in the *OQS* of a worker implies that the worker has finished processing a task. To further decrease the contention, we have forced worker threads to briefly stay idle when *IQS* is zero.

(c) Lock-free Queues: Our run-time employs a distributed queue scheme in order to eliminate synchronization between workers on queue retrieval. Moreover, writes and reads between scheduler and worker are also lock-free. As also illustrated in figure 2.b, both worker and scheduler TBs hold separate pointers to the index of the task which has been inserted and processed, respectively.

(d) Parallel Queue Access and Task Scheduling: Another important part of our queue access optimization is exploiting parallel and aligned access. Worker and scheduler threads belonging to same wrap access consecutive queue locations in order to avoid 'costly' unaligned accesses. Once the scheduler detects processed tasks, resolves dependencies and identifies new tasks, the only remaining operation is to decide the queue in which the new task should be placed.

(e) Queue insertion policies: As in all other multiprocessor decentralized task queue based systems, queue insertion policies play an important role on achieving load balance as while preserving data locality as much as possible. We have embedded two different policies and evaluated their effects through experiments. *Round robin* is the most primitive policy which simply places a given task to next available SM provided that its task queue is not full. *Tail-submit*[4] executes the first generated on the local processor immediately. Later tasks are inserted to the queue that has minimum number of tasks. Performance implications of these policies will be investigated in detail in the Evaluation section.

4. EVALUATION

In this section we will present a detailed evaluation of our proposed framework.

4.1 Platform

We have carried our experiments on a 64-core AMD based system with a nVidia Tesla C2050 GPU running on 64GB of total system memory.The GPU is based on Fermi Architecture and it is attached to the system via PCI-Express slot. It has 3 GB of on-board memory along with 448 CUDA cores delivering up to 515 Gigaflops of peak performance. There are 14 SMs in total and each SM contains 32 SPs (CUDA cores).

In order to demonstrate the efficiency of our distributed queue (DQ) scheme, we have also implemented the centralized queue (CQ) approach described in [9]. For a fair comparison, we have done our best to embed all implementation-related optimizations while re-implementing CQ scheme.

To evaluate our task-scheduling scheme, we have selected two applications that represent the extreme ends of applications with dependencies. Heat Equation (Heat2D) has tasks with regular dependency patterns and breadth first search algorithm (BFS) has dynamically created tasks with irregular dependency patterns.

4.2 Initial Results

In order to evaluate our proposed run-time we have performed a series of experiments. We have first measured execution time of our proposed distributed queue (DQ) with two different insertion policies (Round Robin and Tail Submit) as well as the already existing Centralized Queue approach (CQ). Then we have investigated load balancing abilities of these policies. Finally we have evaluated scalability of our distributed queue scheme (DQ) versus CQ approach

As described in the previous section, we have implemented two different distributed queue insertion policies: Round-Robin (RR) and Tail-Submit (TS).

We have run task based Heat2D and BFS applications using 14 SMs (Thread Blocks). The input size for Heat2D was a 2D array of 8K by 8K. Tile size is taken as 256x256 and total number of tasks was 4K. For BFS, We have used a graph containing 2K nodes, resulting in a task graph with the same number of nodes. The generated graph has 4 neighbors at max and the maximum breadth size was fixed at 128.

The results in Figure 3 depicts the breakdown of the average total execution time. The following timings are shown:

- **Computation:** The time spent for executing a task already fetched from queues
- **Idle Time:** The total idle time spent by SM while waiting for a task to be assigned.
- **Queue Retrieval/Insertion:** (Central queue only) The time spent while acquiring central queue lock and retrieving/inserting a task.
- **DRT (Dependency Resolution Time):** (Central queue only) The time spent by workers while going through dependents of an executed task.
- ***IQS* & *OQS*:** (Distributed queue only) The time spent during atomic read of input and output queue signals between scheduler and workers.

Our newly introduced distributed queue based scheduling with with Tail Submit has provided speedups up to 15% for Heat2d and 62% for BFS when compared to centralized queue approach.

For both Heat2D and BFS, it can clearly be observed that centralized queue scheme shows poor performance in all cases. For Heat2D, computation time is significantly higher due to increased memory contention. The Heat2D algorithm is memory intensive, therefore memory accesses issued while accessing central queue are adversely affecting the execution. For BFS, quick execution time of the algorithm dramatically increases the retrieval/insertion rates of tasks, hence causing a significant jump on central queue access latencies.

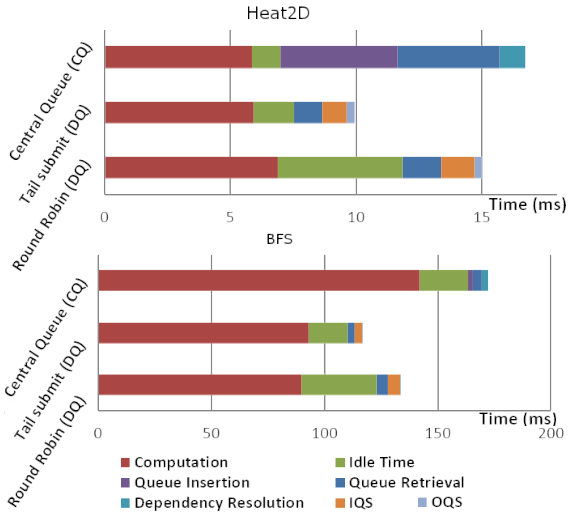

Figure 3: Average execution time breakdown for various scheduling and queuing policies for Heat2D (a) and BFS (b)

Distributed queue schemes, on the other hand, all result in faster execution times against central queue. Signaling timings, IQS and OQS, replaces the delays for queue insertion/retrieval times by resulting in less overhead in total. Especially for BFS, the difference between these two set of timings shows the clear advantage of distributed queue approach over the centralized queue scheme.

When we compare distributed queue schemes with each other, it can easily be observed that round-robin (RR) causes SMs to stay idle more than others. This is primarily due to destroyed locality between tasks. When a task is processed, all of its children tasks whose dependencies are resolved are placed into other queues due to increasing round-robin counter. On the other hand, Tail-Submit (TS) preserves locality on the insertion of first task, hence decreasing memory access times considerably due to better cache locality.

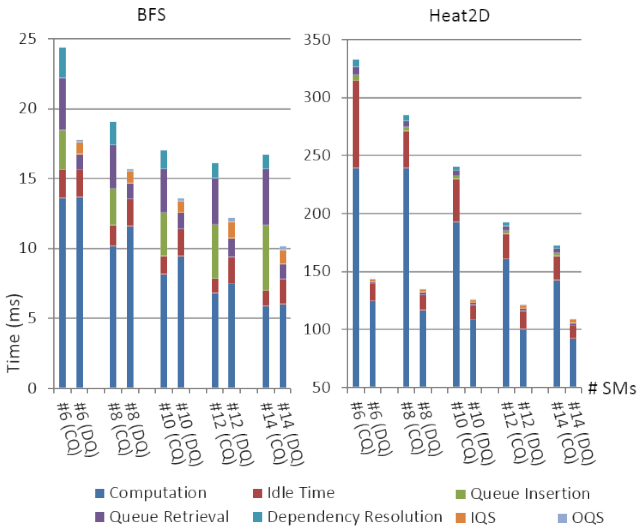

Figure 4: Comparison of Distributed Queue (DQ) and Centralized Queue (CQ) for varying number of SMs.

We have also observed the effects of varying number of SMs on performance for the Distributed Approach (DQ) and centralized queue approach (CQ). In this experiment, while keeping other parameters same, we have changed the total number of SMs from 6 to 14 in order to observe the changes in idle time and queue access time. Time breakdown for each SM is shown in pairs in Figure 4.

An immediate observation is that the average idle time for CQ increases significantly for increasing number of SMs.This is mainly due to adversely affected task locality as more SMs insert un-related tasks simultaneously to the central queue. In other words, effects of locality become more prominent as more SMs are involved in the computation.

5. CONCLUSION & FUTURE WORK

In this study we have proposed a new, dynamic task-based execution scheme for GP-GPU applications with regular and irregular data dependencies. Different from previous studies, our framework places both scheduler and worker threads inside the GPU in order to prevent the data transfer overhead between CPU and GPU and the centralized queue contention. Our framework further employs optimizations like concurrent kernel execution, input&output queue signaling and parallel scheduling.

Our initial experiments showed that the proposed dynamic scheme clearly overcomes the bottleneck of kernel launches, host-based and GPU-based centralized queue approaches. We have achieved speedups up to 15% for Heat2d and 62% for BFS when compared to centralized queue approach.

Considering the fact that today's GP-GPUs are getting more cores, we believe our framework is an important attempt to bring task-based execution paradigm on GP-GPU into a higher level. Although our software approach proves the feasibility of the paradigm, a complete solution needs to address following issues:

- Task queues and dependency handling mechanisms should be handled by a dedicated hardware and signaling mechanism to decrease the contention on global memory, hence enabling faster inter-SM communication.
- To efficiently exploit locality aware scheduling, we need to have native support for assigning a given task (i.e. thread block) to a given SM.
- Existing CUDA API should be extended to let programmer to access the hardware mechanisms proposed above.

To conclude, we believe, the scope of GP-GPU applications have tendency to migrate from SIMD based approach with regular data-dependencies towards MIMD style of computation with irregular data-dependencies. The literature as well as the manufacturers should provide faster solutions to address this inevitable inclination.

6. ACKNOWLEDGEMENTS

The paper was partly supported by NSF grants CCF 0905509 and CNS 1157377.

7. REFERENCES

[1] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC 2009*, pages 44–54. IEEE, 2009.

[2] L. Chen et al. Dynamic load balancing on single-and multi-gpu systems. In *IPDPS 2010*, pages 1–12. IEEE, 2010.

[3] M. Garland et al. Parallel computing experiences with cuda. *Micro, IEEE*, 28(4):13–27, 2008.

[4] J. Hoogerbrugge et al. A multithreaded multicore system for embedded media processing. In *Transactions on high-performance embedded architectures and compilers III*, pages 154–173. Springer, 2011.

[5] D. Lustig et al. Reducing gpu offload latency via fine-grained cpu-gpu synchronization. In *HPCA 2013*. IEEE, 2013.

[6] J. Nickolls et al. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[7] T. Okuyama et al. A task parallel algorithm for computing the costs of all-pairs shortest paths on the cuda-compatible gpu. In *ISPA'08*, pages 284–291. IEEE, 2008.

[8] J. A. Stuart et al. Efficient synchronization primitives for gpus. *arXiv preprint arXiv:1110.4623*, 2011.

[9] S. Tzeng et al. A gpu task-parallel model with dependency resolution. *Computer*, 45(8):34–41, 2012.