GPUScheduler : User Level Preemptive Scheduling for NVIDIA GPUs

Shaleen Garg*, Kishore Kothapalli* and Suresh Purini†
†*Centre For VLSI & Embedded Systems Research(CVEST)*
**Centre For Security Theory & Algorithmic Research(CSTAR)*
International Institute of Information Technology, Hyderabad, India
Email: {shaleen.garg@research., kkishore@, suresh.purini@}iiit.ac.in

***Abstract*—In this paper, we present a novel approach to extend the concepts of time sharing and preemption to GPGPU computational space. Our technique is applicable to any batch GPGPU program written in Compute Unified Device Architecture(CUDA) API provided for C/C++ programming languages. It also gives the user freedom to use different scheduling algorithms to schedule the GPGPU programs satisfying specific system level agreements. Our easy-to-use and minimal API makes it very easy for users to run their programs using the GPUScheduler.**

***Keywords*-Scheduler, preemption, GPGPU, NVIDIA, CUDA, time sharing, SIMD**

I. Introduction

GPUs are being increasingly deployed in computer systems research at a multitude of levels ranging from single node installations to supercomputers [3]. The popularity of GPUs can be attributed to their high FLOPS to Watt ratio, and their low cost. This has led to a massive amount of research on GPU computing where one sees libraries, tools, applications, and algorithms, e.g., [2], [1], [6].

One drawback in all of the above is the way the GPU is exposed to the user. The GPU does not expose an operating system of its own and acts like an external device that is to be attached to a host. The GPU can be invoked only by the host, usually the CPU, along with passing the data and the program to be run. Moreover, once the GPU is invoked by starting a kernel to be run on the GPU, the host has to wait for the completion of the kernel. During this time, the GPU resources cannot be timeshared for executing other kernels.

Since CUDA Version 3.1, limited resource sharing is enabled on the GPU by supporting execution of multiple kernels. However, kernels that run simultaneously split the resources of the GPU provided the resource requirement of the kernels executing simultaneously do not exceed the resources available on the GPU. These limitations mean that effective resource sharing on GPUs is not available.

In the absence of resource sharing, users sharing a GPU resource are left to use the GPU in a first-come-first-served (FCFS) basis. In such an FCFS order, it is clear that users trying to run short jobs will have to wait for a potential long job submitted earlier to finish. As GPUs are used for varied computations, the behavior of applications and their time taken can vary significantly. For instance, a matrix multiplication program running on a Tesla K40c GPU on a matrix of size $2^{15}x2^{15}$ runs in 2.7 seconds where as a coAuthersDBLP program (betweeness-centrality problem) on 300k $\times$ 1 million takes 1.25 hours [6].

Traditional CPUs on the other hand have supported time-sharing from a long time and use Round-Robin (RR) scheduling or its variants. In the RR model, the entire CPU resources are given to one program for a fixed time quantum. At the end of the time quantum, the program currently using the CPU is *context-switched* out by storing the entire state of the program including the contents of the hardware registers. Another program is loaded for execution. This process continues till all programs are finished.

Developing such a seemingly simple RR scheduling algorithm for GPUs is fraught with several challenges. Implementing the RR scheduling algorithm requires one to be able to stop a program under execution arbitrarily, capture and store the state of a program that is currently under execution, (re)start a program by restoring the state of the program. In the absence of access to the internal hardware and other aspects of the GPU, these are difficult to support directly.

Past attempts at such round-robin GPU scheduling reported by Calhoun and Jiang [4] are not efficient. When only one program is running in their framework, the slowdown experienced is an order of magnitude compared to running the same program outside of the proposed framework.

In this paper, we present a simple and efficient framework that supports a round-robin scheduling mechanism on GPUs. Our framework is to divide the kernel into multiple micro kernels so as to simulate preemption on GPU. This prevents starvation of programs with smaller run time.

II. The NVIDIA GPU Architecture

The NVidia Tesla K40c GPU, that we use for our experiments, has 2880 compute cores arranged as 192 cores each in 15 *Streaming Multiprocessors* SMXs. It has 12 GB on board memory and 64 KB of on chip memory per each SMX. An L2 cache of 1.5 MB is shared among all SMXs. Each SMX has a hardware scheduler which schedules 32 threads at a time. This group is called a warp and a half-warp is a group of 16 threads that execute in a SIMD fashion. For programming the K40c GPU, we use CUDA Version 7.5.17 as described in [5].

GPU programs called *kernels* are structured as a logical *grid* of up to three dimensional *blocks* of up to three dimensional *threads*. A thread is the smallest grain of the computational hierarchy. Each thread and block is identified by its three dimensional id. Physically, thread blocks are scheduled on *Streaming Multiprocessors* (SMxs) which is a group of very small computational units known as *Streaming Processors* (SPs). SMs contain special functional units and Load/Store units common to the SPs in that SMX. We refer

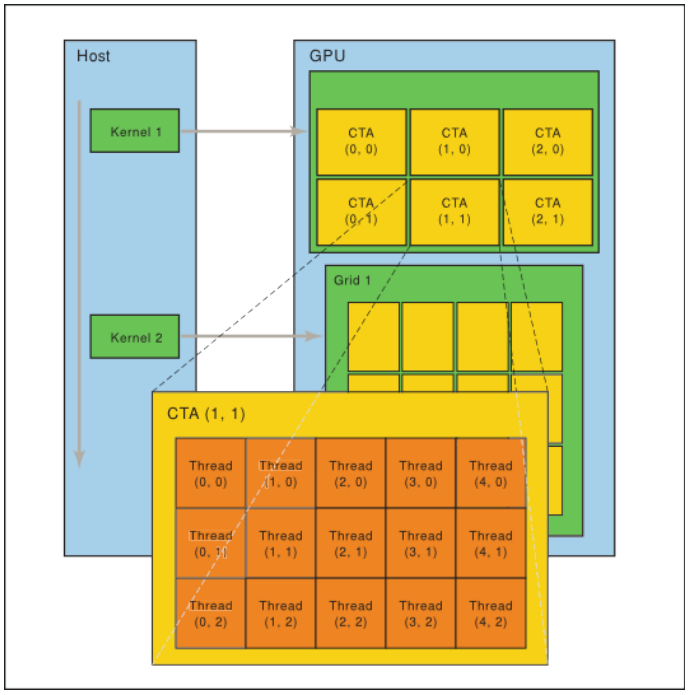

Figure 1. NVIDIA Execution Model

the reader to Figure 1 and the CUDA refernence manual [5] for more details.

III. Our Approach

Our focus is on enabling multiple kernels to use the GPU resource in a time sharing manner. We assume that, alike most operating system scheduling algorithms, there is a pool of runnable programs that are contending for the resource (GPU) and that the programs are independent of each other.

In our scheduler framework, a user program can have a number of states as shown in the Figure 2. The `Launch` state is the first state acquired when the user launches the program for execution. The program then moves to the state `preprocessing` which involves all the work before the kernel launch and device data load instructions like `cudaMalloc` and `cudaMemcpy`. Once, the preprocessing is done, the GPUScheduler deems the program to be ready for GPU execution. This is characterized by the program entering the state `Ready`. The size of the execution unit is defined by the attribute #blocks_left. This attribute is program dependent and will be determined apriori. The program cycles in the states `Ready` and `OnGPU` till #blocks_left is non-zero. Once all the blocks are run on the GPU, i.e., the program finishes execution, the program state changes to `postprocessing` where all the cpu computations and memory copies from the device take place and thereafter it ends. The GPUScheduler does not take control over the pre and post kernel computations as they concern the CPU and device memory copies. NVIDIA devices natively support multiple memory copy requests concurrently.

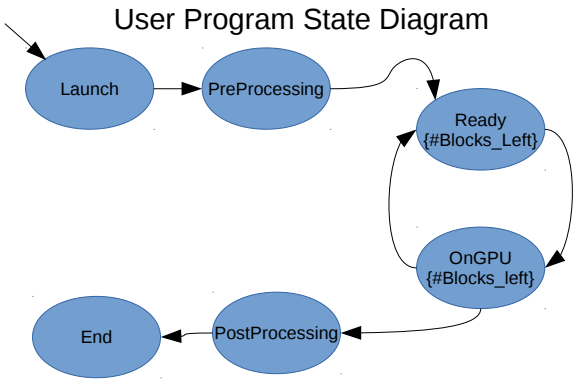

Figure 2. User Program State Diagram while running

The user kernel that runs on a GPU can be viewed as a grid of CUDA block computations. Threads inside a block can choose to communicate with each other using shared memory but threads across different blocks cannot communicate by design. Also, blocks are run concurrently natively, and there is no guarantee to the order of running of the blocks in the kernel. The final results are not affected by the order of execution of the blocks. This gives us the ability to divide big kernels into smaller kernels, referred to as *micro kernels*. The attribute `#Blocks_Left` indicates the numer of blocks in each micro kernel. This allows us to hide the complexity involved in dividing kernels from the user by providing an easy to use API to call user kernels. The preparation of micro kernels and their execution is done by our scheduling framework and the user can use the framework API to access this functionality. The API can be understood by considering the CUDA code snippets as shown in Figure 3. Unlike the usual CUDA kernel as shown in the top box of Figure 3, program using our scheduler use the variable `Sc_Blocks` instead of the `Block` CUDA defined variable. The variable `Sc_Blocks` is defined by the scheduler. The user also has to use the function `KernelCall` instead of issuing the kernel call as mentioned in the CUDA manual. The user also

includes two function calss `WantToRunKernel()` and `FinishedKernel()` to indicate the end of preprocessing and the start of the postprocessing phases.

Native vectorAdd Kernel Call

```
//############ WITHOUT USING THE SCHEDULER ##############
vecAdd <<<Block, ThreadSize>>>(d_a, d_b, d_c, numElements);
```

GPUScheduler compliant vectorAdd Kernel Call

```
//############ USING THE SCHEDULER ###############
//Finished Preprocessing
WantToRunKernel();
//Tells the Scheduler that preprocessing is finished.(Enqueue)

//Block is a dim3 variable defined and populated by the user
//It is the grid the user wants to run
KernelCall(Block,
vecAdd<<<Sc_Blocks, ThreadSize>>>(d_a, d_b, d_c, numElements));
//Sc_Blocks is a Scheduler defined dim3 variable
//Scheduler controls the block dimension to run per slice

FinishedKernel();
//Tells the Scheduler that Kernel process is finished.(Dequeue)
//Start Post Processing
```

Figure 3. Code snippets of using the scheduler framework.

Within the kernel program, there are a few changes that the user has to notice. Our framework provides variables such as `BlockIdx` that can be used to compute the necessary thread-specific offsets and also a mapping from a microkernel to the user kernel block. Refer to the following code snippets to observe the changes made to a rudimentary vectorAdd kernel function.

Native Kernel Code for vectorAdd

```
//########## Kernel Code WITHOUT USING THE SCHEDULER ##########
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;

    //Makes sure we dont go out of bounds
    if(id <= n)
        c[id] = a[id] + b[id];
}
```

GPUScheduler compliant Kernel Code for vectorAdd

```
//############## Kernel Code USING THE SCHEDULER ###########
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    int id = BlockIdx * blockDim.x + threadIdx.x;
    //BlockIdx is a scheduler provided API for the user

    //Makes sure we dont go out of bounds
    if(id <= n)
        c[id] = a[id] + b[id];
}
```

These offsets and variables serve multiple purposes in our framework. These changes make sure that the kernel program uses variables provided by the scheduler framework to correctly compute the required indices and offsets. Using these framework specific variables also ensures that in the user program no computations are repeated and hence allows us to argue about the correctness of the framework.

We provide these offsets as a set of APIs which can be used in the kernel. This helps in minimizing the changes made in the user program to make it easy to schedule and also makes it hassle free and usable.

A. Implementation Details

The scheduler is constructed over a message passing model. We use FIFO queues provided by the Linux kernel for message passing between the user programs and the GPUScheduler. Each user program is responsible for sending its current state(Figure 2) to the GPUScheduler. We provide multiple predefined macros for the users to use in their programs. These macros do the needful after each state is achieved like updating the GPUScheduler and waiting for a message from GPUScheduler.

We note that other Linux kernel functionalities such as POSIX signals are not sufficient to support our framework. In particular, POSIX signals may not count the number of signals and rather indicate the presence or absence of a signal. In such a case, the multiplicity of particular events such as multiple programs sending the same signal is lost.

B. Discussion

Currently, our framework supports the round-robin scheduling algorithm to schedule multiple kernels. Alternatively, one can choose to use different scheduling algorithms in a plug-and-play fashion to suit their needs. For instance, one can imagine a round-robin plus priority based scheduling algorithm. The framework, the API, and the mechanisms that a user has to follow to access our framework can largely remain unchanged.

As noticed in in Figure 4, each user kernel program is divided into multiple micro kernels. The size of micro kernel for each program depends on the number of blocks the program can finish in the specified time slice; for example, program P2, has a bigger micro kernel as compared to program P1. Program P2 has a smaller micro kernel in the end because those are the computations left after the first micro kernel execution and are kept for the next time slice. Currently, for each program, we pick block size using empirical study. Other methods like ones mentioned in Pai et al. [] can be incorporated to our GPUScheduler in a later stage. However, new programs can join the queue concurrently while execution. This makes our model dynamic and can result in a shorter turnaround time for user programs.

Since our scheduling framework runs as a user level program, and not a system level program, there are certain limitations of our framework. For instance, our framework currently cannot catch signals such as SIGKILL intended for the user program. However, in future, we hope that our framework can be written as a system (kernel) program on the CPU so that the signals intended for user GPU programs can be processed registered by the framework also.

From the user point of view, when using our scheduler, each user program should necessarily use the APIs provided by the framework. Otherwise, the framework cannot do the necessary scheduling. We note that the above restrictions are natural and do not restrict any class of programs.

C. Sources of Overheads

Preemptive scheduling algorithms are useful as they improve the resource usage and reduce the turnaround time for short jobs. Preemptive scheduling algorithms by nature however introduce overheads in many ways. In our framework, we note the following overheads.

For each kernel, the required offset variables are to be loaded on the device memory using cudaMemcpyToSymbol. While the kernel is executing, the new block numbers have to be calculated using those offsets provided. Further, each program kernel is typically broken into multiple kernel calls. Each kernel invocation is an expensive operation. Hence, we try to increase the time slice in order to run more blocks which decrease the number of total kernel calls per program.

IV. Experimental Results

In this section, we study our framework on two example instances. In the experiments reported in this section, we take the time slice to be 1 ms. The time slice can be changed according to the system's requirements. All the experiments are performed on an NVIDIA Tesla K40c GPU with driver version 375.66 attached to an Intel Xeon E5-2650 CPU. To proram the GPU, we use CUDA version 7.5.17.

We conduct two experiments. In one experiment, we study the overhead of our scheduler on a sample program of vector addition. The second experiment studies how two kernels can share the GPU resource.

A. Overhead

VectorAdd is one of the basic problems in HPC. We take double precision VectorAdd of different size vectors as examples in our results. We run the VectorAdd program fom the CUDA samples in two different modes: one without using the scheduler, and the other using the scheduler framework with no other program also using the scheduler framework.

Figure 5 shows the difference between runtimes of vectorAdd programs when run with the preemptive scheduler and natively. All the times are in milliseconds. As can be observed from Figure 5, the two version of the vectorAdd program differ in their runtime by no more than a factor of two. This difference can be explained by the overheads of the framework.

However, this overhead can be minimized by several means such as increasing the time slice. This increase can be achieved by increasing the number of blocks in each micro kernel. Figure 6 shows the runtime of the vectorAdd program with vector length 2^{15} when run using the scheduler framework with different time slices. It can be seen that the runtime decreases significantly as the time slice is increased. The same trend was observed for vector lengths 2^{20} and 2^{25} respectively. This suggests that the time slice should be chosen so as to minimize the overhead of scheduling.

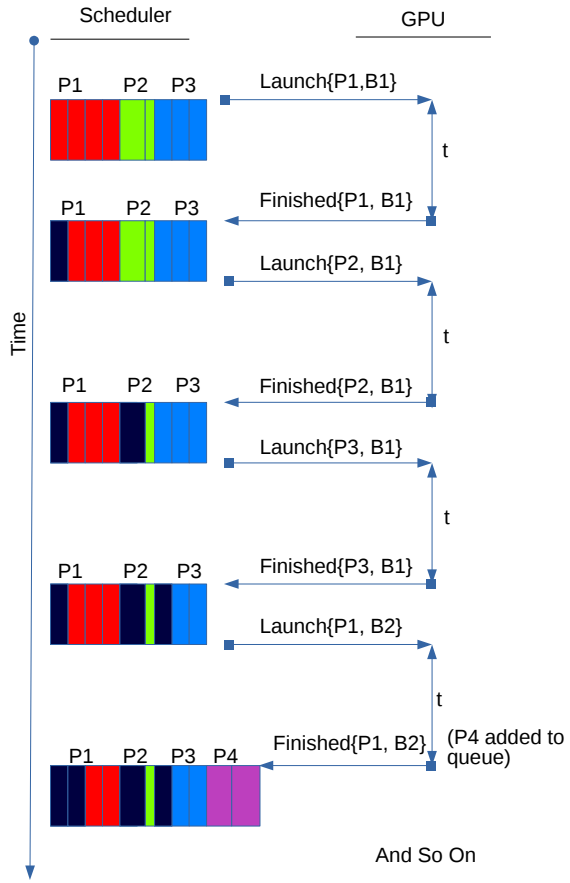

Figure 4. RoundRobin Scheduling Algorithm

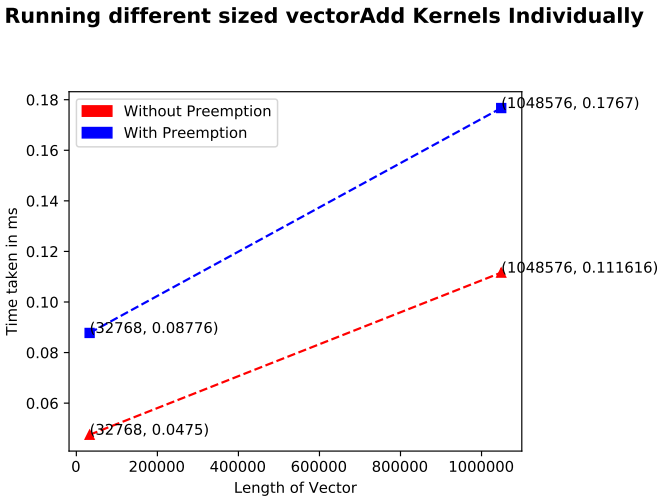

Figure 5. Running VectorAdd of different vector lengths with and without preemption

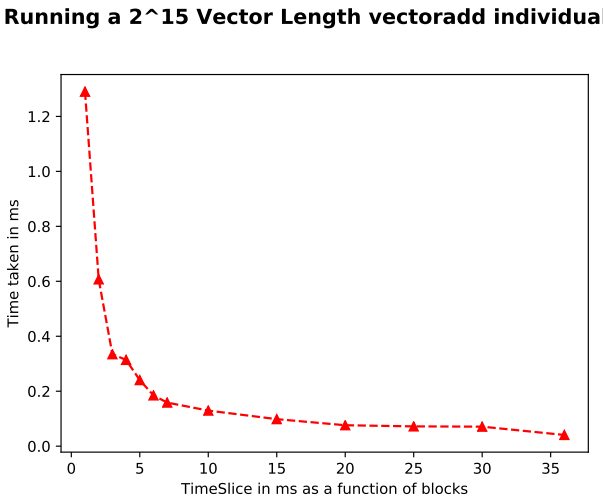

Figure 6. Running 2^{15} VectorAdd with different time slices

B. Multiple Programs

To demonstrate how multiple programs can timeshare the GPU using our framework, we pick the matrix transpose and the (dense) matrix multiplication programs. These programs exist as sample programs in the CUDA installation. We consider two scenarios in which these programs are run: without using the scheduler framework, and using the scheduler framework.

Figure 7 shows how the scheduler switches across the two programs at the end of every time slice. Note that while the time slice is fixed, the number of blocks of the transpose program that run in each time slice is more than the number of blocks of the multiplication program. At the end of six time slices, of which the transpose program uses three time slices, the transpose program ends. In the absence of other programs using the scheduler framework, all the next time slices are used by the multiplication program.

Figure 8 shows the overhead of using hte scheduling framework as the ratio of time when run together using the framework to the time taken when run alone. This ratio is computed for input square matrices of various sizes ranging from 1024 rows to 8192 rows. As can be seen, we note that the overheads ratio is near 1.5 for the multiplication program and near 2.5 for the transpose program.

V. Related Work

Preemptive thread block scheduling, Pai et al. [7] uses linear profiling technique to predict kernel runtime. Using these runtime, they use non-preemptive Shortest Job First scheduling algorithm to schedule the kernels. This is a static model i.e., all the kernels should arrive at the same time in order for the SJF scheduling algorithm to schedule smallest kernels first. In a scenario where a small kernel arrives after the execution of a large kernel has already started, the waiting time shall be very high. The profiling techniques proposed by them could be used to predict time-slices for each kernel in GPUScheduler queue for improved performance.

Kernlets by Zhong et al. [8] uses a similar model of dividing kernels into micro-kernels so as to run multiple kernels at once using the concurrent kernel function provided by the NVIDIA architecture. They claim that this increases the throughput of the device. They assume that each kernel they are provided, does not use all of the computational units in the GPU, hence, multiple kernels can be scheduled on the device concurrently provided the size of the kernel in decreased. Also, any kernel has to be submitted as PTX of SASS code to their kernel slicer in order to run. Our model provides the users to run their programs transparently, just like they would do natively.

Preemption of CUDA kernel [4] by Calhoun et al. 2012, is trying to achieve the same objective as we are trying to do. They are taking a user level snapshot of each kernel variable in order to capture state before preempting the kernel. This makes their scheduling algorithm significantly slow. Their results show a slowdown of at least 40x. We capture state just by taking care of the number of blocks finished in the computational grid.

VI. Conclusion

In this paper we have presented a scheduling framework that can allow multiple programs to share time on a GPU in a roundobin manner. The usage and effectives of the proposed framework is studied with example GPU kernels. In future, we would like to see how to extend the framework to multiple GPUs and also provide more compiler assisted mechanisms that make the framework more usable.

References

[1] Cuda applications. geforce.com/hardware/technology/cuda/applications.

[2] The cusparse library project. http://docs.nvidia.com/cuda/cusparse/.

[3] Top500 supercomputer sites. www.top500.org.

[4] Jon Calhoun and Hai Jiang. Preemption of a CUDA kernel function. In *13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2012, Kyoto, Japan, August 8-10, 2012*, pages 247–252, 2012.

[5] NVidia Corporation. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[6] C. Pachorkar, M. Chaitanya, K. Kothapalli, and D. Bera. Efficient parallel ear decomposition of graphs with application to betweenness-centrality. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 301–310, Dec 2016.

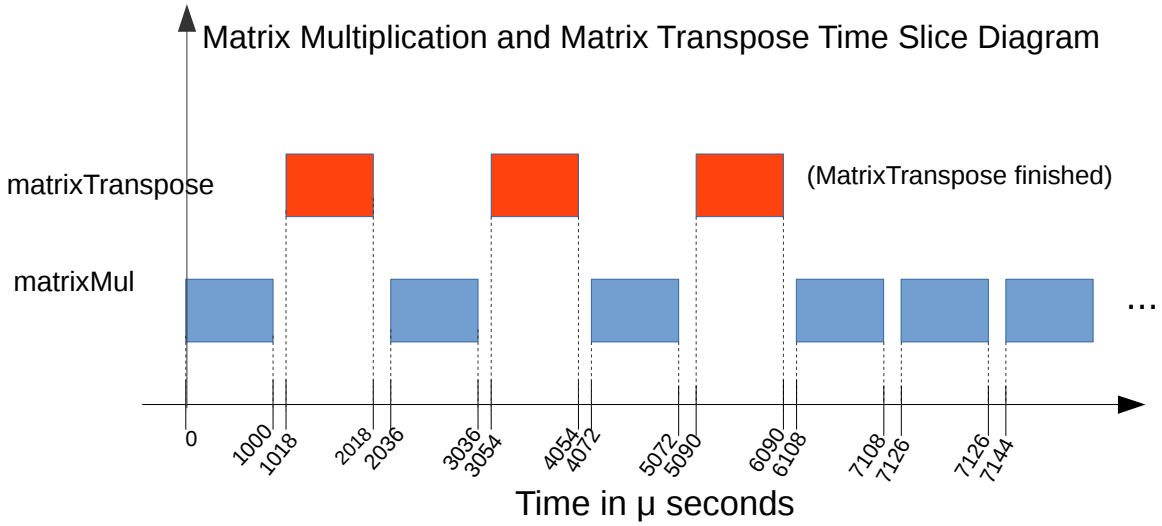

Figure 7. MatrixMul & MatrixTranspose time slice diagram

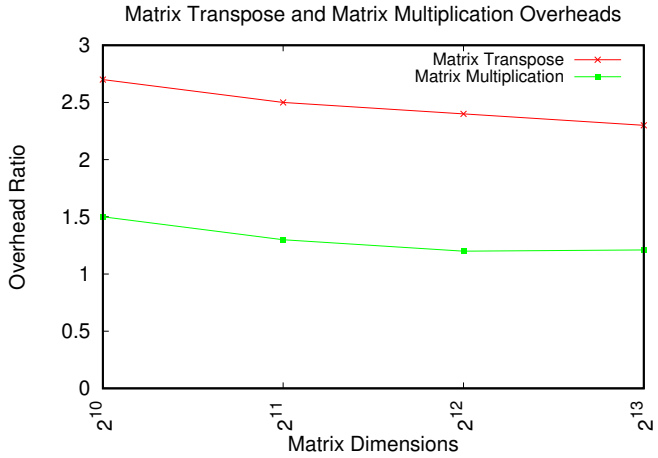

Figure 8. Overheads while Running matrixMul and matrixTranspose concurrently

[7] Sreepathi Pai, R. Govindarajan, and Matthew J. Thazhuthaveetil. Preemptive thread block scheduling with online structural runtime prediction for concurrent GPGPU kernels. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 483–484, 2014.

[8] Jianlong Zhong and Bingsheng He. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1522–1532, June 2014.