GPUScheduler

User-Level Preemptive Scheduling for NVIDIA GPUs

Shaleen Garg, Kishore Kothapalli, Suresh Purini
CSTAR, International Institute of Information Technology, Hyderabad

Current GPU Model

Following is the current GPU model. When Multiple Programs come into picture, the model looks like so.

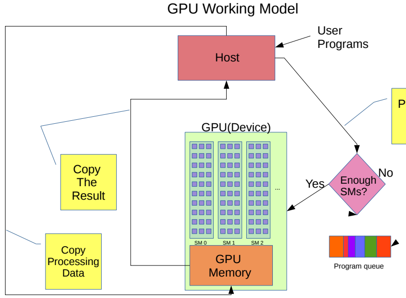

Figure 1: GPU Working Model

Limitations of the Current Model

Assume arrival two programs in the following order:

- P0(large kernel): Matrix multiplication program on 2^{13} x 2^{13} sized matrices ($\sim$ 3 seconds).
- P1(small kernel): Matrix transpose program on 2^{13} x 2^{13} sized matrices ($\sim$ 3 Milliseconds).

The following is what happens when P0 arrives before P1.

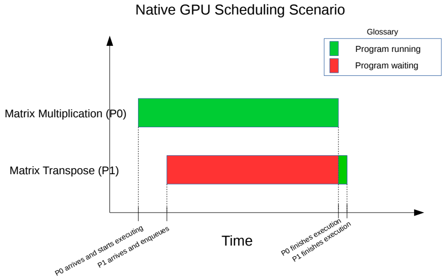

Figure 2: Program P0 arrives before Program P1

Traits of a Good Scheduler

- **Preemptive**: To reduce wait time of a program waiting in the queue.
- **Low Overheads**: To reduce scheduling overheads so as to reduce the response time.
- **Flexibility**: Ability to support different scheduling policies to cater to different scheduling needs and Service Level Agreements(SLAs).

Our Approach

We fullfill the above traits of a good scheduler by using the following technique.

- We break the kernel into smaller micro-kernels to facilitate preemption.
- Our State save policy involves saving one dim3 variable, hence very low overheads.
- The scheduling framework can employ different scheduling policies in a plug and play fashion.

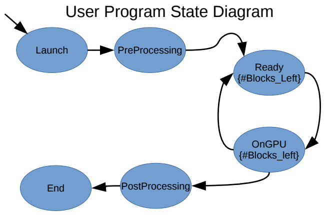

Figure 3: User Program State Diagram

Saving the State

Consider a GPUScheduler compliant program running. It's state needs to be saved in order to resume computations when it is context switched back at a later stage.

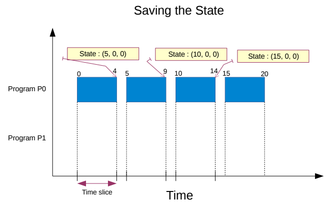

Figure 4: Example Round Robin

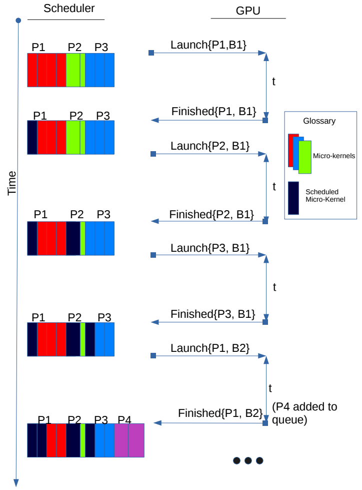

Figure 5: Example Round Robin

Example

Here is an example to show conversion of a native GPU program to a GPUScheduler compliant GPU program.

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
```

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];
}
```

Experimental Results

Overheads ratio when Matrix Multiplication program is run with and without using GPUScheduler.

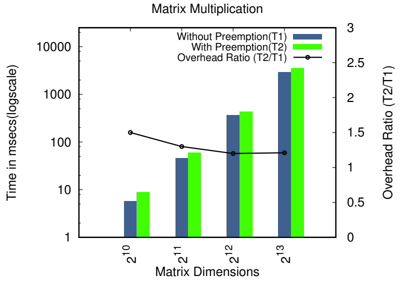

Figure 6: Overheads for Matrix Multiplication

Scheduling Scenario when GPUScheduler is used for two programs. Overheads Ratio when Matrix Transpose and Matrix Multiplication program are run together.

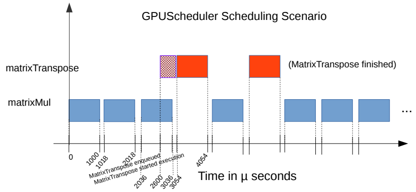

Figure 7: Slice Diagram for Matrix Multiplication and Matrix Transpose

1

---

[1] For further queries, contact shaleen.garg@research.iiit.ac.in