**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:

![Current GPU Model Diagram](image)

**Limitations of the Current Model**

Assume arrival two programs in the following order:

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

The following is what happens when P0 arrives before P1:

1. **Program P0 arrives before Program P1**

   - **Time slice**
   - **State**: $(5, 0, 0)$

   **Saving the State**
   - Consider a GPUScheduler compliant program running. Its’s state needs to be saved in order to resume computations when it is context switched back at a later stage.

   ![User Program State Diagram](image)

**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 fulfill 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.

**Example**

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

**Glossary**

- **Micro-kernels**: Smallest feasible unit of computation.
- **Scheduled Micro-kernel**: A micro-kernel that has been scheduled by the scheduler.

**Native Kernel Code for vectorAdd**

```c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if(id <= n)
        c[id] = a[id] + b[id];
}
```

**GPUScheduler compliant Kernel Code for vectorAdd**

```c
void vecAdd<<<Block, ThreadSize>>>(d_a, d_b, d_c, numElements);
```

**Experimental Results**

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

**Matrix Dimensions**

- **Without Preemption (T1)**
- **With Preemption (T2)**

**Overheads Ratio (T2/T1)**

<table>
<thead>
<tr>
<th>Matrix Dimensions</th>
<th>Overheads Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_x$</td>
<td>1.5</td>
</tr>
<tr>
<td>$C_y$</td>
<td>2</td>
</tr>
<tr>
<td>$C_z$</td>
<td>2.5</td>
</tr>
</tbody>
</table>

**Figure 1:** Example Round Robin

**Figure 2:** Program P0 arrives before Program P1

**Figure 3:** User Program State Diagram

**Figure 4:** Example Round Robin

**Figure 5:** Example Round Robin

**Figure 6:** Overheads for Matrix Multiplication

**Figure 7:** State Diagram for Matrix Multiplication and Matrix Transpose