Bringing Low-Latency and Fault-Tolerant Computing to Tegra SoCs with Persistent Threading

Andrew Milluzzi
Doctoral Candidate
University of Florida
Outline

- What is Persistent Threading?
- Why Persistent Threading on Tegra?
- How to Improve Fault-Tolerance with Persistent Threading
- A Different Approach to Sub-Frame, Low-Latency Image Processing on Tegra Using Persistent Threading
- Conclusions on using Persistent Threading on Tegra
- Questions
What is Persistent Threading?

Typical Threading Model

1. Copy Memory to GPU
2. Launch Kernel
3. Kernel Execution on GPU
   - GPU executes on data
   - CPU stalls or executes other tasks
4. Kernel Finishes
5. Copy Memory to CPU
What is Persistent Threading?

Persistent Threading Model

1. Launch Kernel
2. Kernel Execution on GPU
   • CPU Copies memory to/from GPU as needed
   • CPU and GPU communicate via memory locations
   • GPU kernel never finishes
   • CPU and GPU can work together on task
Why Persistent Threading on Tegra?

Advantages of PT
- No need to launch kernel multiple times
- CPU and GPU can work together on task
- Potential for reduced load on schedulers
- Enables new applications

Disadvantages of PT
- No global synchronization when kernel finishes
- CPU has to interact with GPU to control execution
- Limited block and thread size for kernel
- No direct support
Why Persistent Threading on Tegra?

Advantages of Tegra
- Unified CPU-GPU memory space
- Very low power
- 100% CUDA support
- GPU offers high degree of parallelism

Disadvantages of Tegra
- Less memory GPU bandwidth
- Limited device resources
- Slower device driver
- CPUs cannot saturate memory controllers
Why Persistent Threading on Tegra?

- Fault Tolerance
  - Prior work\(^1\),\(^2\) indicates scheduler vulnerabilities to radiation upsets
  - Persistent Threading can reduce load on block and warp schedulers

- Sub-Frame, Low-Latency Image Processing
  - For simple, sub-frame kernels, driver launch latency can be greater than kernel execution time
  - Persistent threading can be faster than CUDA streams


Fault Tolerance

- Goal: Reduce warp and block scheduler workload
  - Persistent Thread
  - 1 Block
  - ≤ 192 (TK1) or 256 (TX1) Threads

```c
__global__ void foo(volatile int flag, int* data){
    int id = threadIdx.x;
    while(1){
        if(flag){
            data[id] = data[id] + 1;
            // do computation here
            flag = 0;
        }
        else{
            // sleep
            cudaSleep();
        }
    }
}
```
Sub-Frame, Low-Latency Processing

`__global__ void shader(int* imageIn, int* imageOut, volatile int lineCounter, volatile int start, volatile int stop){
    int pixelsToProcess = 1920/THREADS;
    int offset = threadIdx.x*pixelsToProcess;
    int processedLine=0;
    while(stop == 0){
        if(start == 1){
            if(lineCounter >= 0 &&
               processedLine <= lineCounter){
                newImage = 0;
                for(i=0; i < pixelsToProcess; i++){
                    imageOut[offset+i] = \imageIn[offset+i] + 1;
                }
            }
            syncthreads();
            if(processedLine >= 1080 && threadIdx == 0){
                newImage = 1;
            }
        }
    }
}
`

- **Goal:** Line-based 1080P HD image processing at 60 FPS
  - 2 Persistent Threads
    - Camera Interface
    - Image Processor
  - 2 total blocks
  - 1200 total threads
Sub-Frame, Low-Latency Processing

Line-Based Image Processing with Persistent Threading GPU Kernels

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shader PT</td>
<td>10.05</td>
</tr>
<tr>
<td>Shader PT with 200 Additional Adds or Mults</td>
<td>16.125</td>
</tr>
<tr>
<td>Shader PT with Shared Memory</td>
<td>15.242</td>
</tr>
<tr>
<td>Shader PT with Shared Memory and 10 Additional Adds or Mults</td>
<td>15.985</td>
</tr>
<tr>
<td>Shader PT with Shared Memory and 20 Additional Adds or Mults</td>
<td>16.999</td>
</tr>
</tbody>
</table>

60 fps = 16.67 ms

90 fps = 11.11 ms
PT on Tegra Conclusions

- Persistent Threads never stop running on GPU
- Reduce warp and block scheduler workload to improve fault tolerance
- Enable FPGA-like, sub-frame, low-latency image processing applications for HD images
Questions?

Andrew Milluzzi
milluzaj@ufl.edu