PORTABLE PERFORMANCE FOR MONTE CARLO SIMULATION OF PHOTON MIGRATION IN 3D TURBID MEDIA FOR SINGLE AND MULTIPLE GPUs

Fanny Nina-Paravecino
Leiming Yu  Qianqian Fang*  David Kaeli

Department of Electrical and Computer Engineering
Department of Bioengineering*
Northeastern University
Boston, MA
Outline

- Portable Performance Monte Carlo Extreme (MCX)
  - MCX in CUDA
  - Persistent Threads in MCX
  - Portable Performance MCX
- MCX on multiple GPUs
  - Linear Performance
  - Linear Programming Model
  - Performance Results
PORTABLE PERFORMANCE MCX

Photons initialization

3D voxelated media

KEPLER
135%
Performance Core

2x
Performance/Watt

MAXWELL
1st Generation

CONTROL LOGIC

CONTROL LOGIC

CONTROL LOGIC

CONTROL LOGIC
Monte Carlo Extreme (MCX) in CUDA

• Estimates the 3D light (fluence) distribution by simulating a large number of independent photons
• Most accurate algorithm for a wide ranges of optical properties, including low-scattering/voids, high absorption and short source-detector separation
• Computationally intensive, so a great target for GPU acceleration
• Widely adopted for bio-optical imaging applications:
  • Optical brain functional imaging
  • Fluorescence imaging of small animals for drug development
  • Gold stand for validating new optical imaging instrumentation designs and algorithms
MCX in CUDA

Simulation of photon transport inside human brain

Imaging of bone marrow in the tibia

Imaging of a complex mouse model using Monte Carlo simulations
MCX in CUDA [1]

Loop of repetitions

Start

Seed GPU RNG with CPU RNG

Repetition complete?

Retrieve solution

End of simulation

CPU

Thread i

Launch a photon

Compute the scattering length

Move photo one voxel

Compute attenuation based on absorption

Accumula. Probability to the volume

Seed GPU RNG

Global Memory

Compute a scattering direction vector

Scattering ends?

Total move or photon # reached?

Terminate thread

Exceeds time gate?

Thread i+1

Persistent Threads (PT) in MCX

• PT kernels alter the notion of a virtual thread lifetime, treating those threads as physical hardware threads
• PT kernels provide a view that threads are active for the entire duration of the kernel
  • We schedule only as many threads as the GPU SMs can concurrently run
  • The threads remain active until end of kernel execution

CUDA Grid Structure
Persistent Threads (PT) in MCX

- A PT kernel bypasses the hardware scheduler, relying on a work queue to schedule blocks.
- A PT kernel checks the queue for more work and continues doing so until no work is left.
- PT MCX works on a FIFO blocking queue.

![Diagram of Persistent Threads (PT) in MCX]

Blocks → Enqueue → Back → Queue → Front → Shared Multiprocessor
## Portable Performance for MCX

<table>
<thead>
<tr>
<th></th>
<th>Fermi</th>
<th>Kepler</th>
<th>Maxwell</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxThreadBlocks/Multiprocessor</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>MaxThreads/Multiprocessor</td>
<td>1536</td>
<td>2048</td>
<td>2058</td>
</tr>
<tr>
<td>Multiprocessors (MP)</td>
<td>16</td>
<td>14</td>
<td>22</td>
</tr>
<tr>
<td>CUDA cores / MP</td>
<td>32</td>
<td>192</td>
<td>128</td>
</tr>
</tbody>
</table>

\[
\text{# threadsPerBlock} = \frac{\text{MaxThread}/\text{MP}}{\text{MaxThreadBlocks}/\text{MP}}
\]

\[
\text{# blocks} = \text{# threadsPerBlock} \times \frac{\text{MaxThreadBlocks}/\text{MP}}{\text{MP}}
\]
### Portable Performance MCX - Results

<table>
<thead>
<tr>
<th></th>
<th>Kepler GK110</th>
<th></th>
<th>Maxwell 980Ti</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Baseline</td>
<td>Improved Code</td>
<td>Baseline</td>
<td>Improved Code</td>
</tr>
<tr>
<td>ThreadsPerBlock</td>
<td>32</td>
<td>128</td>
<td>32</td>
<td>128</td>
</tr>
<tr>
<td># Total Threads</td>
<td>86,016</td>
<td>28,672</td>
<td>90,112</td>
<td>45,056</td>
</tr>
<tr>
<td># Blocks</td>
<td>2688</td>
<td>224</td>
<td>2816</td>
<td>352</td>
</tr>
<tr>
<td>Performance (Photons/ms)</td>
<td>2383</td>
<td>2887</td>
<td>13,369</td>
<td>15,015</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.0</td>
<td>1.21</td>
<td>1.0</td>
<td>1.12</td>
</tr>
</tbody>
</table>

#### Graph
- **X-axis**: GPU Type
  - Kepler GK110
  - Maxwell 980Ti
- **Y-axis**: Speedup
- **Baseline**
- **Code Improved**
MCX ON MULTIPLE GPUs
Linear Programming Model

• Given $n$ devices: $D_1, D_2, \ldots D_n$
• Given linear performance for each device
• Given the performance for 10 Million photons and 100 Millions for each device
• We can obtain the linear equation for each device as follow:

Device 1 \hspace{1cm} f_1: \hspace{1cm} y_1 = b_1 + (x_1 - 1)a_1 + C_1 \\
Device 2 \hspace{1cm} f_2: \hspace{1cm} y_2 = b_2 + (x_2 - 1)a_2 + C_2 \\
\vdots \\
Device n \hspace{1cm} f_3: \hspace{1cm} y_n = b_n + (x_n - 1)a_n + C_n
Performance Results

- We evaluated our Linear Programming on Linear Model (LPLM) scheme for two different configurations of NVIDIA devices.
- The resulting partition of the workload achieves an average 8% speedup over the baseline.
Summary

• We have improved the performance of MCX across a range of NVIDIA GPU architectures
• We have showed how to exploit Persistent Thread kernel to automatically tune MCX kernel
• We developed a linear programming model to find the best partition to run MCX on multiple GPUs
• We improved performance of MCX run on multiple NVIDIA GPUs, including Kepler and Maxwell
• We obtained an 8% speedup when using automatic partitioning
Future Work

- **PT MCX**
  - The queue of blocks can either can be static (know at compile time) or dynamic (generated at runtime), and can be used to control the order, location, and the timing of each block

- **Instrumentation of MCX**
  - Leverage SASSI to instrument MCX and better characterize the behavior of a kernel to guide auto-tuning

- **MCX on Multiple GPUs**
  - Evaluate our partitioning optimization for multiple devices
THANK YOU!

Questions?