Porting Monte Carlo Algorithms to the GPU

Ryan Bergmann
UC Berkeley
Serpent Users Group Meeting
9/20/2012
Madrid, Spain
Outline

• Introduction to GPUs
  – Why they are interesting
  – How they operate
  – Pros and cons

• Problems with MC algorithms
  – Thread divergence

• Solutions to these problems
  – Reference remapping
  – Changing task parallel to data parallel
  – MC applications where GPUs excel

• Preliminary 2D mono-energetic results

• Future Plans
Introduction To GPUs
Why Are GPUs Interesting?

• [General Purpose] Graphics Processing Cards = [GP]GPUs
• GPUs are being used across all scientific fields
• Major company endorsements (NVIDIA, Adobe...)
  – Guaranteed future development
• Top supercomputers use GPUs to gain efficiency
  – Issue blocking exascale computing is POWER
• All architectures are getting *wider*, not faster
  – Porting codes to heterogeneous programming structures will make them much more future-proof
• GPU-like programming models are the future!
Differences

• GPUs are “manycore”
  – Optimized for total throughput
  – Individual core performance de-emphasized
  – “flock of chickens” or assembly line workers

• CPUs are “multicore”
  – Optimized for executing a small number of threads
  – Geared toward individual performance
  – “yolk of oxen” or master craftsmen

Taken from Bryan Cantanzaro’s CS-267 slides, Feb 2011.
GPU Architecture

Graphic taken from Wen-mei Hwu’s UCB CITRIS presentation, Jan 24-25, 2011.
### CPU/GPU Comparison

<table>
<thead>
<tr>
<th>Specifications</th>
<th>Westmere-EP</th>
<th>Fermi (GF110)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processing Elements</strong></td>
<td>6 cores, 2 issue, 4 way SIMD @3.46 GHz</td>
<td>16 cores, 2 issue, 16 way SIMD @1.54 GHz</td>
</tr>
<tr>
<td><strong>Resident Strands/Threads (max)</strong></td>
<td>6 cores, 2 threads, 4 way SIMD: 48 strands</td>
<td>16 cores, 48 SIMD vectors, 32 way SIMD: 24,576 threads</td>
</tr>
<tr>
<td><strong>SP GFLOP/s</strong></td>
<td>166</td>
<td>1577</td>
</tr>
<tr>
<td><strong>Memory Bandwidth</strong></td>
<td>32 GB/s</td>
<td>192 GB/s</td>
</tr>
<tr>
<td><strong>Register File</strong></td>
<td>6 kB (?)</td>
<td>2 MB</td>
</tr>
<tr>
<td><strong>Local Store/L1 Cache</strong></td>
<td>192 kB</td>
<td>1024 kB</td>
</tr>
<tr>
<td><strong>L2 Cache</strong></td>
<td>1536 kB</td>
<td>0.75 MB</td>
</tr>
<tr>
<td><strong>L3 Cache</strong></td>
<td>12 MB</td>
<td>-</td>
</tr>
</tbody>
</table>

Taken from Bryan Cantanzaro’s CS-267 slides, Feb 2011.

<table>
<thead>
<tr>
<th>Spec</th>
<th>CPU node</th>
<th>GPU node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor specs</td>
<td>4x Opteron 12-core 2.1 Ghz</td>
<td>3x NVIDIA TESLA C2070</td>
</tr>
<tr>
<td>Price</td>
<td>$10,000</td>
<td>$13,000</td>
</tr>
<tr>
<td>Max. TeraFLOPs</td>
<td>0.4</td>
<td>3.1 (N.I. 16 CPU cores)</td>
</tr>
<tr>
<td>Min. Price/GigaFLOP</td>
<td>$25</td>
<td>$4.19</td>
</tr>
</tbody>
</table>
SIMT

- SIMT = Single Instruction Multiple Thread
- “Data Parallelism” – the same operations are conducted on different pieces of data
- Takes advantage of regularity in instruction sets
- Easy to see how this would be advantageous in array operations
  - Matrix operations
  - Iterative methods
Threads

- SIMT execution is abstracted by threads
  - Each thread in a “warp” executes the same instruction set on different data
  - Warps are 32 threads wide
  - Strict SIMT *not* enforced, divergence causes serialization

- Thread block execution is scheduled by hardware
  - Each thread must be completely independent of any others
  - Threads must be allowed to execute in any order

- Each thread and thread block has a unique ID which can be used to access different pieces of data.
  - tid = ThreadID + BlockID * BlockDim
  - data[tid]

Graphic taken from Wen-mei Hwu’s UCB CITRIS presentation, Jan 24-25, 2011.
GPU Strengths & Weaknesses

• Strengths
  – Very high computational capacity
  – High memory bandwidth (compared to CPU)
  – Much cheaper and energy efficient per FLOP (1/10 & 1/20 respectively)

• Weaknesses
  – Reliance on SIMT means control divergence is a problem
  – Only very high performance on data-parallel tasks
  – Kernels must be lightweight for high performance (small caches)
  – Limited DRAM (currently max 6GB per card)
    • New drivers allow peer-to-peer transfers between cards over PCIe bus, eliminating CPU overhead (basically GPU RDMA), but this is slower than local memory (of course)
CUDA

• Extended C – additional data types, function definitions, and operators

• Heterogeneous
  – CPU and GPU parts
  – CPU/GPU parts can execute concurrently (pipelining)

• GPU kernels can be callable from host & device, or from device only.
  – Kernels are pieces of code that run on many different threads
  – Calling them “kernels” reflect the lightweight and repeated nature of the code
CUDA

Serial Code (host)

Parallel Kernel (device)
KernelA<< nBlk, nTid >>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<< nBlk, nTid >>>(args);

©Wen-mei W. Hwu and David Kirk/NVIDIA,
Berkeley, January 24-25, 2011
PROBLEMS WITH MC ALGORITHMS
Thread Divergence

- MC method involves lots of ‘if’ statements based on random numbers
- Creates many areas where thread control flow can diverge
- Highly divergent problems can cause severe under utilization of GPU resources
  - serialization of divergent threads
  - warps to remain idle while waiting for longest thread to complete

Graphic taken from Wen-mei Hwu’s UCB CITRIS presentation, Jan 24-25, 2011.
Hardware Limitations

• Current cards have a maximum of 6GB of global memory
  – Fitting necessary cross section data in GPU memory could be a problem

• Kernels can only be launched from CPU
  – No “master thread” on GPU, so jobs cannot “make work” for themselves depending on problem progress
SOLUTIONS
Data Remapping

• Must keep in-warp threads in same control branch
  – remap data
    • Expensive
    • Might not be necessary
  – remap references to the data (pointers)
    • Uses the form: new thread id = remap[thread id]
    • Allows threads to access only “active” particle data
    • Can be extended from “active” to “performing reaction X”
Reference Remapping

DATA (Usually reference by DATA[tid])

1 done
2 active
3 active
4 done
5 active
6 done

... to N

REMAP

2 3 5 7 11 8

... to N

THREAD

1 2 3 4 5 6

... to $N_{\text{active}}$
Task to Data Parallelism

- Break history loop into individual task sections
- Transport kernel does one step of transport
- Purge kernel updates a remapping vector
  - First $n$ entries are all “active” particles
  - Last $N_{\text{tot}} - n$ entries are absorbed particles
- Next iteration in history loop only transports “active” $n$ particles, effectively purging all completed histories from all thread blocks
- Turns “a particle per thread” into “$N$ particles shared by $N$ threads”, ie **DATA PARALLEL**
PRELIMINARY
2D MONO-ENERGETIC RESULTS
Program Details

- Written in CUDA C
- Data layout AOS (array of structures) for better coherent data access
- Break history loop into “transport” and “purge” sections
  - Transport kernel does one step of transport
  - Purge kernel updates a remapping vector
    - First $n$ entries are all “active” particles
    - Last $N_{tot} - n$ entries are absorbed particles
- Next iteration in history loop only transports “active” $n$ particles, effectively purging all completed histories from all thread blocks
- Turns “a particle per thread” into “N particles shared by N threads”
- Uses libraries when available
  - More flexible, better performance than handwritten routines
  - CuRand for random number generation
  - CUDPP for sorts and scans
  - OptiX for ray tracing and geometry representation (future work)
Problem Geometry

• 2D fixed geometry
• Mono-energetic
• Only isotropic scatter and capture
• All neutrons uniformly born in cell 1
• Cell 0 extends to infinity
Serial CPU / Naïve GPU Algorithm

1. Initialize particle
2. Transport
3. Intersect boundary? If yes, place particle there, set resample bit
4. Determine location, update parameters
5. Resample?
   - Yes: Resample
   - No: Absorbed?
      - Yes: Absorbed
      - No: Determine reaction, update tally
Purging GPU algorithm

- Initialize dataset

CPU

GPU

- Seed/advance random number databank
- Transport N particles
- Determine reaction, update tally, set ‘done’ bit if absorbed
- Update remap vector, update N to number of active particles
- Intersect boundary? If yes, place particle there, set ‘resample’ bit
- Determine location, update parameters

**Dashed lines indicate an independent kernel launch**
Visual Results
Speedup over single CPU
Profiler Stats 1
Profiler Stats 2

![Graph showing % Control Flow Divergence over iterations with lines for Purging Iteration value, Average, and Naive.](image-url)
Profiler Stats

- x-axis is time
- Pink sections are for methods
- Blue sections are API processes

- Active warps/active cycle
  - Naïve code: 9.93 for the single kernel call
  - Purging: 26 for reaction kernel
    15 for transport kernel
    30 for purge/remap
Conclusions

• Remapping a SUCCESS for reducing divergence
• GPU code is faster than CPU
  – Speedup factors increase with histories, but have decreasing marginal gain
  – Serpent 20x faster than MCNP, GPU ~25x faster than CPU, so core calculations on the order of 500x faster than MCNP possible if use serpent methodology on GPU???
• Purging code slower than naïve
  – Three independent kernel launches per iteration, API overhead gets expensive with many iterations, API calls take 60% of total time
  – Have to copy active particle number back to host every iteration
  – These overheads will be removed in next generation hardware
    • Purging approach will most likely faster than naive
    • Shows promise for accelerating reactor simulations, up to 25x speedup over CPU
• Initial testing with ENDFB-VII libraries show that data can fit on-card
  – GPUs use shared memory, only need one copy of cross sections for each material
  – Entire library is 1.13GB at a single temperature
  – Threads may have to do on-the-fly cross section arithmetic to keep memory utilization down (i.e. no cross section pre-processing)
Further Work

• Use NVIDIA OpitIX instead of handwritten intersection finder
  – Low level parallel 3D ray tracing framework for CUDA
  – Develop routines to translate combinatorial geometry to OpitIX representations
  – Can import CAD drawings
  – Can compare to real world problems and produce much more relevant speedup comparisons to MCNP, Serpent, etc.

• Parallel cross section processing/access routines
  – On-the-fly arithmetic and access routines
  – Unified grid for fast lookup if memory is available
  – Otherwise fast lookup algorithm
    • Binary/ternary search
    • Interpolation search
    • Compute inversion function since energy data is monotonic
      – lookup will be done in (small) constant time!

• Serpent Routine????
Thank You!

Training the Next Generation

This material is based upon work supported by the Department of Energy National Nuclear Security Administration under Award Number(s) DE-NA0000979
Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or limited, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.