GPU programming

Gil Speyer
Mayo Clinic
November 14, 2012
Outline

• The GPU story (briefly)
• A digression on CPUs
• Parallel programming on the GPU
• Some specs
• Programming the GPU
• Performance considerations
• Prefix scan
• Conclusion
The GPU story (briefly)

- GPU: A processor optimized for 2D/3D graphics, video, visual computing & display
- Implementation: A highly parallel, highly multithreaded, multiprocessor, optimized to provide real-time visual interaction with computed objects via graphics images
- More and more programmable and versatile to handle various graphics elements (vertices, fragments, etc.) and geometric operations
- GPGPU: A scalable parallel computing platform
- Heterogeneous programming (via PCIe)
A digression on CPUs

• Moore’s Law continues apace... except for clock frequency
  – Power scales with frequency squared
Frequency wall
A digression on CPUs

• Moore’s Law continues apace... except for clock frequency
  – Power scales with frequency squared
• But the number of transistors can still be scaled
  – Is there a viable solution within the Van Neumann model?
  – Increase cache size?
Cache size keeps increasing
...but has the benefit plateaued?
A digression on CPUs

• Moore’s Law continues apace... except for clock frequency
  – Power scales with frequency squared
• But the number of transistors can still be scaled
  – Is there a viable solution within the Van Neumann model?
  – Increase cache size? Performance benefit plateau
  – Other enhancements: Pipelines, OoO, SSE, etc. ??
A digression on CPUs

• Moore’s Law continues apace... except for clock frequency
  – Power scales with frequency squared
• But the number of transistors can still be scaled
  – Is there a viable solution within the Van Neumann model?
    – Increase cache size? Performance benefit plateau
    – Other enhancements: Pipelines, OoO, SSE, etc. ??
      • Plateau + power consumption (schedule:instruction 50:1)
• Result: We can no longer count on technology or design to accelerate single threaded code
Microprocessor Transistor Counts 1971-2011 & Moore's Law

The graph shows the trend of transistor count over the years, with the curve demonstrating that the transistor count doubles every two years. The data points represent various microprocessors introduced from 1971 to 2011, illustrating the exponential growth in transistor count as technology advances.
Parallel programming on a GPU

• Ubiquity
• Ready visualization
• A better option for higher granularity applications
• And then there’s the performance...
Some specs

- 3.0 GHz quad-core Sandybridge: 400 GFLOPS, 50 GB/s
- Nvidia GTX 680: 3 TFLOPS, 190 GB/s
- CPUs: 1.4X annual growth
- GPUs: 1.7X (pixels) to 2.3X (vertices) annual growth
- 4 years ago, no GPU supercomputers in top 500
- This year, 3 of top 5 are GPU supercomputers
Programming the GPU

- OpenGL/Cg
- CUDA (Nvidia)
- Thrust – high level API
- OpenCL
- OpenACC
CPU/GPU Program Flow

1. CPU builds data structures in CPU memory
2. CPU transfers data to GPU memory over PCIe
3. CPU calls GPU program (set of kernels)
4. GPU runs
5. CPU copies data back into CPU memory

Computation must be ample
Kernels and Streams

• Kernels (computation)
  – Subroutines called by host that execute on GPU
  – Correspond to “inner loops” in CPU programs

• Streams (data)
  – Large amount of data of same type
  – Independent elements – map
  – Write once/ read once – optimized for throughput
  – Hide latency

Computation must be regular
Thread hierarchy

- 2 level hierarchy: blockid & threadid
- gridDim and blockDim to get size
- Threads in different blocks cannot synchronize with each other
- Local high speed memories
- Governed by thread execution manager (rasterizer)
Memory hierarchy

- Texture/surface memory - low latency, high BW – optimized for 2D spatial locality
- Registers for each thread, shared memory for each block – high speed. Lifetime = kernel
- “Tiling” into shared memory to improve performance
Performance considerations

• Threads bundled into warps which execute all at once – avoid divergence
  – Conditional statement may change control flow of all threads, making warp repeatedly execute. 16 branches = 16X slower
  – Fix by moving conditional statements outside of nested for loops, loop fission
  – Implicit branching through filter method (explained later)
Performance considerations

- Optimize global memory accesses by having warp access contiguous memory – **coalescing**
- Shared memory stored in 16 (now 32) memory banks. **Bank conflict:** When threads in same warp accessing same memory bank serialize (unless broadcast).
- Specialized math functions
Scatter and **gather**

**Atom-centric (scatter):**
Multiple threads write grid point at the same time

**Grid-centric (gather):**
Threads accumulate data in private registers

---

for each spring
  f = computed force  
  mass_force[left] += f;  
  mass_force[right] -= f;

for each spring
  f = computed force  
  for each mass  
    mass_force = f[left] - f[right];

---

developer.nvidia.com
Address Sorting

- Scatter to gather method
- Example: Position grid of moving particles

### Scatter Data with Write Addresses

<table>
<thead>
<tr>
<th>Data</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

### Sorted Scatter Data

<table>
<thead>
<tr>
<th>Data</th>
<th>h</th>
<th>c</th>
<th>b</th>
<th>e</th>
<th>f</th>
<th>a</th>
<th>d</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

```plaintext
array[i] = binarysearch(data, i)
```

<table>
<thead>
<tr>
<th>Array</th>
<th>h</th>
<th>c</th>
<th>b+e</th>
<th>f</th>
<th>f</th>
<th>a+d</th>
<th>g</th>
</tr>
</thead>
</table>
(Exclusive) Scan

- We are given a series, \([a_0, a_1, \ldots, a_n]\) and we must return a series:
  \[
  [l, a_0, (a_0 \star a_1), \ldots, (a_0 \star a_1 \star \ldots \star a_{n-2})]
  \]
  where \(\star\) can be any binary associative operator with identity \(l\).
- For example, if \(\star\) is addition and \(l = 0\),
  \([3, 1, 7, 0, 4, 1, 6, 3] \rightarrow [0, 3, 4, 11, 11, 15, 16, 22]\)
- Inclusive scan can be generated by moving terms to the left and adding last element of input array to last element of exclusive scan.
Scan applications

- Line-of-sight
- Split-radix sort
- Recurrence eqns. $x_i = a_i x_{i-1} + b_i x_{i-2}$
- Adding multi-precision numbers
- Lexical comparison (dictionary, grep, parsing)
- Image element labeling, histograms
- Tree operations, e.g. vertex depth
## Split-radix Sort

| A         | [5 7 3 1 4 2 7 2] |
| A<0> (odd/even) | [1 1 1 1 0 0 1 0] |
| A←split(A,A<0>) | [4 2 2 5 7 3 1 7] |
| A<1> (middle bit) | [0 1 1 0 1 1 0 1] |
| A←split(A,A<1>) | [4 5 1 2 2 7 3 7] |
| A<2> (>3) | [1 1 0 0 0 1 0 1] |
| A←split(A,A<2>) | [1 2 2 3 4 5 7 7] |
### Sort

- Implementation of split-radix sort using scan operations (first step)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>[5 7 3 1 4 2 7 2]</td>
</tr>
<tr>
<td>A⟨0⟩</td>
<td>[1 1 1 1 0 0 1 0]</td>
</tr>
<tr>
<td>Ā⟨0⟩</td>
<td>[0 0 0 0 1 1 0 1]</td>
</tr>
<tr>
<td>Scan Ā⟨0⟩ ➔</td>
<td>[0 0 0 0 0 1 2 2]</td>
</tr>
<tr>
<td>Reverse Scan A⟨0⟩ ⇐</td>
<td>[4 3 2 1 1 1 0 0]</td>
</tr>
<tr>
<td>Subtract from n (=7)</td>
<td>[3 4 5 6 6 7 7]</td>
</tr>
<tr>
<td>Re-index</td>
<td>[4 2 2 5 7 3 1 7]</td>
</tr>
</tbody>
</table>
## Quicksort

<table>
<thead>
<tr>
<th>A</th>
<th>[6 9 3 1 8 4 9 3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment flag</td>
<td>[1 0 0 0 0 0 0 0]</td>
</tr>
<tr>
<td>Pivots</td>
<td>[6 6 6 6 6 6 6 6]</td>
</tr>
<tr>
<td>F</td>
<td>[=&gt;&lt;&lt;&lt;&gt;&lt;&gt;&lt;&lt;]</td>
</tr>
<tr>
<td>A←split(A,F)</td>
<td>[3 1 4 3 6 9 8 9]</td>
</tr>
<tr>
<td>Segment flag</td>
<td>[1 0 0 0 1 1 0 0]</td>
</tr>
<tr>
<td>Pivots</td>
<td>[3 3 3 3 6 9 9 9]</td>
</tr>
<tr>
<td>F</td>
<td>[=&lt;&gt;&lt;===&lt;==]</td>
</tr>
<tr>
<td>A←split(A,F)</td>
<td>[1 3 3 4 6 8 9 9]</td>
</tr>
<tr>
<td>Segment flag</td>
<td>[1 1 0 1 1 1 1 0]</td>
</tr>
</tbody>
</table>
Serial Scan

• Simple implementation

\[
\text{out}[0] := 0 \\
\text{for } k := 1 \text{ to } n \text{ do} \\
\quad \text{out}[k] := \text{in}[k-1] + \text{out}[k-1]
\]

• \(O(n)\) steps, \(O(n)\) work
Parallel Scan (Naïve version)

- \(O(\log n)\) steps, \(O(n \log n)\) work

1: for \(d = 1\) to \(\log_2 n\) do
   2: for all \(k\) in parallel do
      3: if \(k \geq 2^d\) then
      4: \(x[k] = x[k - 2^{d-1}] + x[k]\)
Parallel Scan (Naïve version)

- $O(\log n)$ steps, $O(n \log n)$ work

- Step efficient but not work efficient
Parallel Scan (Improved)

- Upsweep (reduce) phase: $O(\log n)$ steps

1: for $d = 0$ to $\log_2 n - 1$ do
2: for all $k = 0$ to $n - 1$ by $2^{d+1}$ in parallel do
3: $x[k + 2^{d+1} - 1] = x[k + 2^d - 1] + x[k + 2^{d+1} - 1]$
• Downsweep phase: $O(n \log n)$ steps

1: $x[n - 1] \rightarrow 0$
2: for $d = \log_2 n - 1$ down to 0 do
3: for all $k = 0$ to $n - 1$ by $2^d + 1$ in parallel do
4: $t = x[k + 2^d - 1]$
5: $x[k + 2^d - 1] = x[k + 2^{d+1} - 1]$
6: $x[k + 2^{d+1} - 1] = t + x[k + 2^{d+1} - 1]$
Implements a filter

• which we can follow with a **reduce** (sum, product, max, min...) locally and then globally
Resolving bank conflicts

Addressing Without Padding

Offset = 1: Address (ai) stride is 2, resulting in 2-way bank conflicts

Bank

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

ai

0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30

thid

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Offset = 2: Address (ai) stride is 4, resulting in 4-way bank conflicts

Bank

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

ai

1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61

thid

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

int ai = offset*(2*thid+1)-1;
int bi = offset*(2*thid+2)-1;
temp[bi] += temp[ai]
Resolving bank conflicts

Addressing With Padding

Offset = 1: Padding addresses every 16 elements removes bank conflicts

Bank

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

ai

0 2 4 6 8 10 12 14 17 19 21 23 25 27 29 31

thid

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Offset = 2: Padding addresses every 16 elements removes bank conflicts

Bank

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

ai

1 5 9 13 18 22 26 30 35 39 43 47 52 56 60 64

thid

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Padding Increment: 0 1 2 3
RUSSIAN PEASANT

METHOD OF MULTIPLICATION

- Set up two columns: small # first, larger # second
- Simultaneously halve the first column and double the second column (if the first column has an odd number, subtract one from it before halving)
- Add up all the values in the second column that have corresponding odd numbers in the first column

For example: $37 \times 56$

\[
\begin{align*}
18 & \times 112 \\
9 & \times 224 \\
4 & \times 448 \\
2 & \times 896 \\
1 & \times 1792 \\
\end{align*}
\]

Therefore

\[
1792 + 224 + 56 = 2072
\]
Russian Peasant Algorithm - Exponentiation

\( n \geq 1, \text{ calculate } P = x^n \)

While \( n \) is even do

\[ x \leftarrow x \cdot x \]
\[ n \leftarrow n/2 \]
\[ P \leftarrow x \]
\[ n \leftarrow n/2 \]

While \( n > 0 \) do

\[ x \leftarrow x \cdot x \]
\[ n \leftarrow n/2 \]

if \( n \) is odd \( P \leftarrow P \cdot x \)
Fibonacci series

• Second order recurrence formula:

\[ f_0 = 0, \; f_1 = 1, \; f_i = f_{i-1} + f_{i-2} \]

\[
F_j = \begin{bmatrix} f_j \\ f_{j-1} \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}
\]

\[
F_i = A_i F_{i-1} = A_i A_{i-1} \cdots A_2 F_1
\]

\[
F_N = A_N \cdots A_2 F_1
\]
Conclusion

• The GPU moves programming (even more) toward parallel thinking

• Effective algorithms will exploit GPU features
  – Gather
  – Map
  – Filter
  – Reduce

• Not a fad
  – Intel Xeon Phi
  – AMD Fusion combines CPU and GPU on one IC
References

• Guy Blelloch, “Prefix Sums and Their Applications”
• Ian Buck, GPU Gems 2, Chapter 32, “Taking the Plunge into GPU Computing”
• Cliff Woolley, GPU Gems 2, Chapter 35, “GPU Program Optimization”
• Peter Kipfer and Rudiger Westermann, GPU Gems 2, Chapter 46, “Improved GPU Sorting”
• Mark Harris, Sengupta Shubhabrata and John Owens, GPU Gems 3, Chapter 39, “Parallel Prefix Sum (Scan) with CUDA”
• David Kirk and Wen-mei Hwu, Programming Massively Parallel Processors: A Hands-on Approach