Lecture 20
Computing with GPUs
Supercomputing
Final Exam Review
Announcements

• The Final is on
  Tue March 15\textsuperscript{th} from 3pm to 6pm
  ‣ Bring photo ID
  ‣ You may bring a single sheet of notebook sized paper “8x10 inches” with notes on both sides (A4 OK)
  ‣ You may not bring a magnifying glass or other reading aid unless authorized by me

• Review session in section Friday

• Don’t forget to do the Peer Review Survey, which is worth 1.5\% of your final exam grade
  \url{https://www.surveymonkey.com/r/Baden_CSE160_Wi16}
Today’s Lecture

• Computing with GPUs
• Logarithmic barrier strategy
• Supercomputers
• Review
Experiments - increment benchmark

- Total time: timing taken from the host, includes copying data to the device
- Device only: time taken on device only
- Loop repeats the computation inside the kernel – 1 kernel launch and 1 set of data transfers in and out of device

\[ N = 8388480 \text{ (8M ints), block size = 128, times in } \textit{milliseconds}, \]

<table>
<thead>
<tr>
<th>Repetitions</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>(10^4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.88</td>
<td>14.7</td>
<td>144</td>
<td>1.44s</td>
</tr>
<tr>
<td>Device time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>19.4</td>
<td>32.3</td>
<td>162</td>
<td>1.46s</td>
</tr>
<tr>
<td>Kernel launch + data xfer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What is the cost of moving the data and launching the kernel?

A. About 1.75 ms ((19.4-1.88)/10)
B. About 0.176 ms (32.3-14.7)/100
C. About 0.018 ms ((162-144)/1000)
D. About 17.5 ms (19.4-1.88)

N = 8 M block size = 128, times in milliseconds

<table>
<thead>
<tr>
<th>Repetitions</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10^4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.88</td>
<td>14.7</td>
<td>144</td>
<td>1.44s</td>
</tr>
<tr>
<td></td>
<td>19.4</td>
<td>32.3</td>
<td>162</td>
<td>1.46s</td>
</tr>
</tbody>
</table>

Device time

Kernel launch + data xfer
Matrix Multiply on the GPU

- Naïve algorithm
  - Each thread loads all the data it needs, independently loads a row and column of input
  - Each matrix element loaded multiple times

- Tiled algorithm with shared memory
  - Divide the matrices into tiles, similar to blocking for cache
  - Threads cooperate to load a tile of A&B into on-chip shared memory
  - Each tile in the result matrix C corresponds to a thread block
  - Each thread performs: 
    \( b \) mpy-adds + 1 load + 1 store
What is the floating point intensity of the tiled algorithm?

A. 1  
B. 2  
C. N  
D. b  
E. $b^2$

The analysis is the same as for blocked matrix multiplication
Results with shared memory

• N=512, double precision
• Last fall, CSE 260 students got up to 468 Gflops (Naïve implementation: 116 GF)
• Compare with about 7.6 Gflops/core on Bang
  19.1 GF per core on Intel Sandy Bridge
  [2.7GHz, 256 bit SSE4, peak speed 21.6 GF]
• What happened?
  ‣ Reduced global memory accesses, and accessed in contiguous regions (coalesced memory accesses)
  ‣ Blocking involves both shared memory and registers
GPU performance highlights

- Simplified processor design, but more user control over the hardware resources
- Use or lose the available parallelism
- Avoid algorithms that present intrinsic barriers to utilizing the hardware
- **Rethink the problem solving technique primarily to cut data motion costs**
  - Minimize serial sections
  - Avoid host ↔ device memory transfers
  - Global memory accesses → fast on-chip accesses
  - Hide device memory transfers behind computation
  - Coalesced memory transfers
  - Avoid costly branches, or render them harmless
Today’s Lecture

• Computing with GPUs
• Logarithmic barrier strategy
• Supercomputers
• Review
An improved barrier

• Replacing mutexes with atomics improves performance dramatically, but still doesn’t scale
• Use a log time barrier, relying on a combining tree
• Each tree node on a separate cache line
• Each processor begins at its leaf node
  ‣ Arrival signal(s) sent to parent
  ‣ Last arriving core continues the process, other(s) drop out
• 1 processor left at root, starts the continue process, signals move in the opposite direction
• See code in $PUB/Examples/Threads/CTBarrier.h
• More efficient variations based sense reversal, see Mellor-Crummey’s lecture
Today’s Lecture

• Computing with GPUs
• Logarithmic barrier strategy
• Supercomputers
• Review
What does a supercomputer look like?

- Hierarchically organized parallelism
- Hybrid communication
  - Threads within each server
  - Pass messages between servers (or among groups of cores) “shared nothing architectures”

conferences.computer.org/sc/2012/papers/1000a079.pdf
What is the world's fastest supercomputer?

- Top500 #1, Tianhe-2 @ NUDT (China)
  - 3.12 Million cores
  - 54.9 Tflop/sec peak
  - 17.8 MW power (+6MW for cooling)
  - 1 PB memory ($2^{50}$ Bytes)
State-of-the-art applications

Blood Simulation on Jaguar
Gatech team

<table>
<thead>
<tr>
<th>p</th>
<th>48</th>
<th>384</th>
<th>3072</th>
<th>24576</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec)</td>
<td>899.8</td>
<td>116.7</td>
<td>16.7</td>
<td>4.9</td>
</tr>
<tr>
<td>Efficiency</td>
<td>1.00</td>
<td>0.96</td>
<td>0.84</td>
<td>0.35</td>
</tr>
</tbody>
</table>

Strong scaling

<table>
<thead>
<tr>
<th>p</th>
<th>24576</th>
<th>98304</th>
<th>196608</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec)</td>
<td>228.3</td>
<td>258</td>
<td>304.9</td>
</tr>
<tr>
<td>Efficiency</td>
<td>1.00</td>
<td>0.88</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Weak scaling

Ab Initio Molecular Dynamics (AIMD) using Plane Waves Density Functional Theory
Eric Bylaska (PNNL)

Exchange time on HOPPER

Slide courtesy Tan Nguyen, UCSD
Have you ever seen a supercomputer in real life?

A. Yes
B. No
C. Not sure
Up and beyond to Exascale

• In 1961, President Kennedy mandated a Moon landing by decade’s end
• July 20, 1969 at tranquility base “The Eagle has landed”
• The US Govt set an ambitious schedule to reach $10^{18}$ flops by 2023, x100 performance increase.
• DOE is taking the lead in the US, China and the EU also engaged
• Massive technical challenges esp software, resilience and power consumption
Why numerically intensive applications?

• Highly repetitive computations are prime candidates for parallel implementation
• Improve quality of life, economically and technologically important
  ‣ Data Mining
  ‣ Image processing
  ‣ Simulations – financial modeling, weather, biomedical
Classifying the application domains

- Patterns of *communication* and computation that persist over time and across implementations
  - Structured grids
    - Panfilov method
  - Dense linear algebra
    - Matrix multiply, Vector-Mtx Mpy
    - Gaussian elimination
  - N-body methods
  - Sparse linear algebra
    - In a sparse matrix, we take advantage of knowledge about the locations of non-zeros, improving some aspect of performance
  - Unstructured Grids
  - Spectral methods (FFT)
  - Monte Carlo

Courtesy of Randy Bank
I increased performance – so what’s the catch?

• Currently there exists no tool that can convert a serial program into an efficient parallel program
  
  … for all applications … all of the time… on all hardware

• The more we know about the application…
  … specific problem … math/physics … initial data …
  … context for analyzing the output…
  … the more we can improve performance

• We can classify applications according to Patterns of communication and computation that persist over time and across implementations - Phillip Colella’s 7 Dwarfs

• Performance Programming Issues
  √ Data motion and locality
  √ Load balancing
  √ Serial sections
What you learned in this class

• How to solve computationally intensive problems on parallel computers effectively
  ‣ Theory and practice
  ‣ Software techniques
  ‣ Performance tradeoffs

• Emphasized multi-core implementations, threads programming, but also the memory hierarchy and SSE vector instructions

• Developed technique customized to different application classes

• We built on what you learned earlier in your career about programming, algorithm design & analysis and generalize them
Do you have an application in mind for multithreading?

A. Yes
B. No
C. Maybe
How about SSE?

A. Yes
B. No
C. Maybe
Today’s Lecture

• Computing with GPUs
• Logarithmic barrier strategy
• Supercomputers
• Review
What are the main issues in implementing multitheaded applications?

• Conserve locality: cache, registers, minimize use of shared memory
• Maximize concurrency: avoid serial sections, take advantage of ILP, SSE
• Ensure correctness
• Avoid overheads: serial sections, load imbalance, excessive thread spawning, false sharing, contention on shared resources, including synchronization variables
Why we need a memory model

- When one thread changes memory, then there needs to be a definite order to those changes, as seen by other threads
- Ensure that multithreaded programs are *portable*: they will run correctly on different hardware
- Clarify which optimizations will or will not break our code
  - Compiler optimizations can move code and may need to obey extra constraints and to generate special code to prevent potential hardware optimizations that could re-order the time to access the variables in memory (e.g. cache)
  - Hardware scheduler executes instructions out of order
- The memory model makes certain guarantees that a particular update to a particular variable made by one thread will *eventually* be visible to another
- The programmer uses synchronization variables
  - To ensure that the variable is accessed indivisibly, that changes become visible and that there are strict orderings in how writes are seen by other threads
  - Prevent both the compiler and the hardware from reordering memory accesses in ways that are visible to the program and could break it
- The model does not require visibility failures across threads, they merely allow these failures to occur
- Not using synchronization in multithreaded code doesn't guarantee safety violations, it just allows them
Consistency vs coherence

• Cache coherence is a *mechanism*, a hardware protocol to ensure that memory updates propagate to other cores.

• Which will then be able to agree on the values of information stored in memory, *as if* there were no cache at all

• *Cache consistency* defines a *programming model*: when do memory writes become visible to other cores?
  ‣ Defines the ordering of memory updates
  ‣ A contract between the hardware and the programmer: if we follow the rules, the results of memory operations are guaranteed be predictable
The central role of synchronization variables

- The C++ \textit{atomic} variable provides a special mechanism to guarantee that communication happens between threads
  - Which writes get seen by other threads
  - The order in which they will be seen
- The \textit{happens-before} relationship provides the guarantee that memory writes by one specific statement are visible to another specific statement
- Different ways of accomplishing this: atomics, variables, thread creation and completion
- When one thread writes to a \textit{synchronization variable} (e.g. an \textit{atomic} or \textit{mutex}) and another thread sees that write, the first thread is telling the second about \textbf{all} of the contents of memory up until it performed the write to that variable

Ready is a synchronization variable

In C++ we use load and store member functions

\begin{itemize}
  \item All the memory contents seen by T1, before it wrote to \textit{ready}, must be visible to T2,
  \item after it reads the value true for \textit{ready}.
\end{itemize}


Scott B. Baden / CSE 160 / Wi '16
Where are there happens-before relationships?

A. (T1:1) -> (T2:1)
B. (T1:3) -> (T2:1)
C. (T1:2) -> (T2:6)
D. A and B
E. B and C

Thread 1
(1) lock();
(2) NT++;  
(3) unlock();
(4) BARRIER( );
(5) if(TID==0)
     cout<<NT<<endl;

Thread 2
(1) lock();
(2) NT++;  
(3) unlock();
(4) BARRIER( );
(5) if(TID==0)
(6) cout<<NT<<endl;

Inter-thread happens-before relations are defined between pairs of synchronization operations. Atomics are one example another is the beginning and end of a critical section, as in this example
**Synchronization: What is minimal?**

```c
(1) void sweep(int TID, int myMin, int myMax, double ε, atomic<double>& err){
(2)     for (int s = 0; s < 100; s++) {
(3)         double localErr = 0;
(4)         for (int i = myMin; i < myMax; i++){
(5)             unew[i] = (u[i-1] + u[i+1])/2.0;
(6)             double δ = fabs(u[i] - unew[i]);
(7)             localErr += δ * δ ;
(8)         }
(9)         err += localErr;
(10)        if ((s > 0) && ( err < ε ))
(11)             break;
(12)        if (!TID){double *t = u; u = unew; unew = t; } // Swap u ↔ unew
(13)        err = 0;
(14)     } // End of s loop
(15) }
```
Which barriers can we remove?

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>void sweep(int TID, int myMin, int myMax, double ε, atomic&lt;double&gt;&amp; err){</td>
</tr>
<tr>
<td>(2-8)</td>
<td>for (int s = 0; s &lt; 100; s++) {</td>
</tr>
<tr>
<td></td>
<td>... unew[i] = (u[i-1] + u[i+1])/2.0</td>
</tr>
<tr>
<td>(9)</td>
<td>err += localErr;</td>
</tr>
<tr>
<td>(9a)</td>
<td>BARRIER()</td>
</tr>
<tr>
<td>(10)</td>
<td>if ((s &gt; 0) &amp;&amp; (err &lt; ε))</td>
</tr>
<tr>
<td>(11)</td>
<td>break;</td>
</tr>
<tr>
<td>(11a)</td>
<td>BARRIER()</td>
</tr>
<tr>
<td>(12)</td>
<td>if (!TID){ Swap u ↔ unew }</td>
</tr>
<tr>
<td>(12a)</td>
<td>BARRIER()</td>
</tr>
<tr>
<td>(13)</td>
<td>If (!TID) err = 0;</td>
</tr>
<tr>
<td>(13a)</td>
<td>BARRIER()</td>
</tr>
<tr>
<td>(14)</td>
<td>} // End of s loop</td>
</tr>
<tr>
<td>(15)</td>
<td>} // End of sweep</td>
</tr>
</tbody>
</table>

A. 9a & 11a
B. 11a & 12a
C. 11a or 12a
D. 3 of them
E. Something else