Lecture 5

Parallel Performance
Introduction with Graphical Processing Units
Announcements
Assignment #1

• Blocking for cache will boost performance but a lot more is needed to approach ATLAS’ performance

8.14 GFlops

\[ R_\infty = 4 \times 2.33 = 9.32 \text{ Gflops} \]

\( \sim 87\% \text{ of peak} \)
Today’s Lecture

- Parallel Performance Metrics
- Introduction to Programming with Graphical Processing Units (GPUs)
Measures of Performance

• Why do we measure performance?

• Measures of performance
  ◆ Completion time
  ◆ Processor time product
    Completion time \times \# \text{ processors}
  ◆ Throughput: amount of work that can be accomplished in a given amount of time
  ◆ Relative performance: given a reference architecture or implementation
    AKA \textit{Speedup}
Parallel Speedup and Efficiency

• How much of an improvement did our parallel algorithm obtain over the serial algorithm?

• Define the parallel speedup, $S_P$

\[
S_P = \frac{\text{Running time of the best serial program on 1 processor}}{\text{Running time of the parallel program on } P \text{ processors}}
\]

• $T_1$ is defined as the running time of the “best serial algorithm”

• In general: not the running time of the parallel algorithm on 1 processor

• **Definition:** Parallel efficiency $E_P = S_P/P$
Performance questions

- You observe the following running times for a parallel program running a fixed workload N
- Assume that the only losses are due to serial sections
- What is the speedup and efficiency on 8 processors?
- What will the running time be on 4 processors?
- What is the maximum possible speedup on an infinite number of processors?
- What fraction of the total running time on 1 processor corresponds to the serial section?
- What fraction of the total running time on 2 processors corresponds to the serial section?

<table>
<thead>
<tr>
<th>NT</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10000</td>
</tr>
<tr>
<td>2</td>
<td>6000</td>
</tr>
<tr>
<td>8</td>
<td>3000</td>
</tr>
</tbody>
</table>
What can go wrong with speedup?

- Not always an accurate way to compare different algorithms….
- .. or the same algorithm running on different machines
- We might be able to obtain a better running time even if we lower the speedup
- If our goal is performance, the bottom line is running time $T_p$
Superlinear speedup

• We have a *super-linear* speedup when
\[ S_p > P \Rightarrow E_P > 1 \]

• Super-linear speedups are often an artifact of inappropriate measurement technique

• Where there is a super-linear speedup, a better serial algorithm may be lurking
Scalability

- A computation is **scalable** if performance increases as a “nice function” of the number of processors, e.g. linearly.
- In practice scalability can be hard to achieve:
  - Serial sections: code that runs on only one processor
  - “Non-productive” work associated with parallel execution, e.g. communication
  - Load imbalance: uneven work assignments over the processors
- Some algorithms present intrinsic barriers to scalability leading to alternatives
  \[
  \text{for } i=0:n-1 \quad \text{sum} = \text{sum} + x[i]
  \]
Serial Section

• Limits scalability

• Let $f =$ the fraction of $T_1$ that runs serially

• $T_1 = f \times T_1 + (1-f) \times T_1$

• $T_P = f \times T_1 + (1-f) \times T_1 / P$

  Thus $S_P = 1 /[f + (1-f)/p]$

• As $P \rightarrow \infty$, $S_P \rightarrow 1/f$

• This is known as Amdahl’s Law (1967)
Amdahl’s law (1967)

- A serial section limits scalability
- Let $f = \text{fraction of } T_1 \text{ that runs serially}$
- *Amdahl's Law (1967)*: As $P \to \infty$, $S_P \to 1/f$
Weak scaling

- Is Amdahl’s law pessimistic?
- Observation: Amdahl’s law assumes that the workload \((W)\) remains fixed
- But parallel computers are used to tackle more ambitious workloads
- If we increase \(W\) with \(P\) we have **weak scaling**
  \[ f \text{ often decreases with } W \]
- We can continue to enjoy speedups
  - Gustafson’s law [1992]
    - [www.scl.ameslab.gov/Publications/Gus/FixedTime/FixedTime.pdf](www.scl.ameslab.gov/Publications/Gus/FixedTime/FixedTime.pdf)
Computing scaled speedup

• Instead of asking what the speedup is, we ask: “how long a parallel program would run on a single processor?”
• Let $T_P = 1$
• $f' = \text{fraction of serial time spent on the parallel program}$
• $T_1 = f' + (1-f') \times P = S'_P = \text{scaled speedup}$
• Scaled speedup is linear in $P$
Isoefficiency

- Consequence of Gustafson’s observation is that we increase $N$ with $P$
- Kumar: We can maintain constant efficiency so long as we increase $N$ appropriately
- The *isoefficiency* function specifies the growth of $N$ in terms of $P$
- If $N$ is linear in $P$, we have a scalable computation
- Problem: the amount of memory per core is shrinking
Today’s lecture

• Performance

• Computing with Graphical Processing Units (GPUs)
Recall processor design trends

• No longer possible to use growing population of transistors to boost single processor performance
  ‣ Can no longer increase the clock speed
  ‣ Instead, we replicate the cores

• An opportunity: Specialize the processing core
  ‣ Simplified design, pack more onto the chip
  ‣ Boost performance
  ‣ Reduce power

• Simplified core
  ‣ Remove architectural enhancements like branch caches
  ‣ Constrain memory access and control flow
  ‣ Partially expose the memory hierarchy
Graphical Processing Units

- Specialized many-core processor (1000s)
  - NVIDIA, AMD
- SIMT vector processing: long vectors
- Reduced on-chip memory per core
- Explicitly manage the memory hierarchy
Heterogeneous processing with Graphical Processing Units

- Specialized many-core processor
- Explicit data motion
  - between host and device
  - inside the device
NVIDIA GeForce GTX 280

- Hierarchically organized clusters of streaming multiprocessors
  - 240 cores @ 1.296 GHz
  - Peak performance 933.12 Gflops/s
- SIMT parallelism
- 1 GB “device” memory (frame buffer)
- 512 bit memory interface @ 132 GB/s

GTX 280: 1.4B transistors
Intel Penryn: 410M (110mm²) (dual core)
Nehalem: 731M (263mm²)
Streaming processor cluster

- GTX-280 GPU
  10 clusters @ 3 streaming multiprocessors or vector cores
- Each vector core
  - 8 scalar cores: fused multiply adder + multiplier (32 bits), truncate intermediate rslt
  - Shared memory (16KB) and registers (16K × 32 bits = 64KB)
  - 1 64-bit fused multiply-adder + 2 super function units (2 fused multiply-adders)
  - 1 FMA + 1 multiply per cycle = 3 flops / cycle / core * 240 cores = 720 flops/cycl @1.296 Ghz: 933 GFLOPS

David Kirk/NVIDIA and Wen-mei Hwu/UIUC

Scott B. Baden /CSE 260/ Winter 2014
Streaming Multiprocessor
## Memory Hierarchy

### (Device) Grid

<table>
<thead>
<tr>
<th>Name</th>
<th>Latency (cycles)</th>
<th>Cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>DRAM – 100s</td>
<td>No</td>
</tr>
<tr>
<td>Local</td>
<td>DRAM – 100s</td>
<td>No</td>
</tr>
<tr>
<td>Constant</td>
<td>1s – 10s – 100s</td>
<td>Yes</td>
</tr>
<tr>
<td>Texture</td>
<td>1s – 10s – 100s</td>
<td>Yes</td>
</tr>
<tr>
<td>Shared</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>Register</td>
<td>1</td>
<td>--</td>
</tr>
</tbody>
</table>

### Host

- **Global Memory**
- **Constant Memory**
- **Texture Memory**

### Shared Memory

- **Instruction Fetch/Dispatch**
- **Instruction L1**
- **Data L1**
- **Shared Memory**

### Courtesy

David Kirk/NVIDIA and Wen-mei Hwu/UIUC
CUDA

- Programming environment with extensions to C
- Under control of the host, invoke sequences of multithreaded kernels on the device (GPU)
- Many lightweight threads
- CUDA: programming environment + C extensions
Thread execution model

- Kernel call spawns virtualized, hierarchically organized threads
- Hardware handles dispatching, 0 overhead
- Compiler re-arranges loads to hide latencies
- Global synchronization: kernel invocation
Hierarchical Thread Organization

- Thread organization
  - Grid $\supset$ Block $\supset$ Thread
  - Specify number and geometry of threads in a block and similarly for blocks
- A block may have a different number of dimensions (1d, 2d or 3d) than a grid (1d/2d, +3d in Cuda 5)
- Each thread uniquely specified by block & thread ID
- Programmer determines the mapping of virtual thread IDs to global memory locations
  - $\Pi: \mathbb{Z}^n \rightarrow \mathbb{Z}^2 \times \mathbb{Z}^3$
  - $\Theta(\Pi_t), \forall \Pi_t \in \Pi$

David Kirk/NVIDIA & Wen-mei Hwu/UIUC
Thread execution

• Thread Blocks
  ♦ Unit of workload assignment
  ♦ Each thread has its own set of registers
  ♦ All have access to a fast on-chip *shared memory*
  ♦ Synchronization only among all threads in a block
  ♦ Threads in different blocks communicate via slow global memory
  ♦ Processor groups threads into *warps* of 32 threads

• SIMT parallelism: all threads in a warp execute the same instruction
  ♦ All branches followed
  ♦ Instructions disabled
  ♦ Divergence, serialization

**KernelA<<<2,3>,<3,5>>>()**

**Grid Block**

Scott B. Baden /CSE 260/ Winter 2014
Constraints

• SM
  ♦ Up to 8 resident blocks
  ♦ Not more than 1024 threads
  ♦ Up to 32 warps
• All threads in a warp execute the same instruction
  ♦ All branches followed
  ♦ Instructions disabled
  ♦ Divergence, serialization
• Grids: 1d, 2d [+3d/CUDA 5] (64k-1)
• Blocks – 1, 2, or 3-dimensional
  ♦ ≤ 512 threads
  ♦ Max dimensions: 512, 512, 64
  ♦ Registers subdivided over threads
  ♦ Synchronization only among all threads in a block
Parallel Speedup

• How much did our GPU implementation improve over the traditional processor?
• \textit{Speedup, }\(S\)

Running time of the fastest program on conventional processors

Running time of the accelerated program

• Baseline: a multithreaded program
How to Maximize Performance

- Avoid algorithms that present intrinsic barriers to utilizing the hardware
  - Avoid costly branches, or render harmless
  - Minimize serial sections
- Cut data motion costs
  - Hide latency of host $\leftrightarrow$ device memory transfers
  - Reduce global memory accesses $\rightarrow$ fast on-chip accesses
  - Coalesced memory transfers
Performance issues

- If we don’t use the parallelism, we lose it
  - Amdahl’s law - serial sections
  - Von Neumann bottleneck – data transfer costs
  - Workload Imbalances
- Simplified processor design, but more user control over the hardware resources
- Rethink the problem solving technique
Fermi

- Larger vector units, more total cores
- Higher peak double precision performance
- L1 Cache, configurable as $\frac{3}{4}$ L1 or SM, $\frac{1}{4}$ SM or L1
- 64 KB/vector unit
- Shared L2 Cache (768 KB)
- Dual thread schedulers
- Concurrent kernel execution (some models)
- Reduced kernel launch overhead (25 $\mu$s)
- Improved predication
Vector units

• Each vector unit
  • 32 CUDA cores for integer and floating-point arithmetic
  • 4 special function units for Single Precision transcendental
  • FMA without truncation (32 or 64 bits)
• For devices of compute capability 2.1
  • 48 CUDA cores for arithmetic operations
  • 8 special function units for single-precision
• CUDA C Programming Guide, §G.4
Fermi platforms in the class

CSEClass 01, 02: GeForce GTX 580 [2.0, GF100]
  15 Vector units @ 32 cores/unit (480 cores), 4 SFUs
  1.25 GB device memory [01 not working currently]

CSEClass 03-07: GeForce GTX 460 [2.1, GF104]
  7 Vector units @ 48 cores (384 total cores), 8 SFUs
  1.0 GB device memory

Dirac: Tesla C2050 [2.0, GF100]
  1 device per node
  14 Vector units @ 32 cores (448 total cores), 4 SFUs
  3 GB device memory + ECC (2.625GB usable)
  SP MAD: 1030.4 Gflops, DP FMA: 515.2

www.anandtech.com/show/3809/nvidias-geforce-gtx-460-the-200-king/2

Scott B. Baden /CSE 260/ Winter 2014 34