Motivation

Most applications run at < 10% of the “peak” performance of a system

• Peak is the maximum the hardware can physically execute
• Much of this performance is lost on a single processor, i.e., the code running on one processor often runs at only 10-20% of the processor peak
• Most of the single processor performance loss is in the memory system
  • Moving data takes much longer than arithmetic and logic

• To understand this, we need to look under the hood of modern processors
  • For today, we will look at only a single “core” processor
  • These issues will exist on processors within any parallel computer

Possible conclusions to draw from today’s lecture

• “Computer architectures are fascinating, and I really want to understand why apparently simple programs can behave in such complex ways!”
• “I want to learn how to design algorithms that run really fast no matter how complicated the underlying computer architecture.”
• “I hope that most of the time I can use fast software that someone else has written and hidden all these details from me so I don’t have to worry about them!”
• All of the above, at different points in time
Outline

• Idealized and actual costs in modern processors
• Memory hierarchies
  • Use of microbenchmarks to characterized performance
• Parallelism within single processors
• Case study: Matrix Multiplication
  • Use of performance models to understand performance
  • Attainable lower bounds on communication

Idealized Uniprocessor Model

• Processor names bytes, words, etc. in its address space
  • These represent integers, floats, pointers, arrays, etc.
• Operations include
  • Read and write into very fast memory called registers
  • Arithmetic and other logical operations on registers
• Order specified by program
  • Read returns the most recently written data
  • Compiler and architecture translate high level expressions into "obvious" lower level instructions
  \[
  A = B + C \Rightarrow \begin{align*}
  & \text{Read address}(B) \text{ to } R1 \\
  & \text{Read address}(C) \text{ to } R2 \\
  & R3 = R1 + R2 \\
  & \text{Write } R3 \text{ to Address}(A)
  \end{align*}
\]
• Hardware executes instructions in order specified by compiler
• Idealized Cost
  • Each operation has roughly the same cost
    (read, write, add, multiply, etc.)

Uniprocessors in the Real World

• Real processors have
  • registers and caches
    • small amounts of fast memory
    • store values of recently used or nearby data
    • different memory ops can have very different costs
  • parallelism
    • multiple "functional units" that can run in parallel
    • different orders, instruction mixes have different costs
  • pipelining
    • a form of parallelism, like an assembly line in a factory
• Why is this your problem?
  • In theory, compilers and hardware "understand" all this
  • and can optimize your program; in practice they don’t.
  • They won’t know about a different algorithm that might be a much better "match" to the processor

In theory there is no difference between theory and practice. But in practice there is. - Yogi Berra
Outline

- Idealized and actual costs in modern processors
- Memory hierarchies
  - Temporal and spatial locality
  - Basics of caches
  - Use of microbenchmarks to characterize performance
- Parallelism within single processors
- Case study: Matrix Multiplication
  - Use of performance models to understand performance
  - Attainable lower bounds on communication

Memory Hierarchy

- Most programs have a high degree of locality in their accesses
  - Spatial locality: accessing things nearby previous accesses
  - Temporal locality: reusing an item that was previously accessed
- Memory hierarchy tries to exploit locality to improve average performance

Approaches to Handling Memory Latency

- Eliminate memory operations by saving values in small, fast memory (cache) and reusing them
  - Need temporal locality in program
- Take advantage of better bandwidth by getting a chunk of memory and saving it in small fast memory (cache) and using whole chunk
  - Bandwidth improving faster than latency: 23% vs 7% per year
  - Need spatial locality in program
- Take advantage of better bandwidth by allowing processor to issue multiple reads to the memory system at once
  - Concurrency in the instruction stream, e.g., load whole array, as in vector processors; or prefetching
- Overlap computation & memory operations
  - Prefetching

Processor-DRAM Gap (latency)

- Memory hierarchies are getting deeper
- Processors get faster more quickly than memory

Memory Hierarchy Table:

<table>
<thead>
<tr>
<th>Speed</th>
<th>1ns</th>
<th>10ns</th>
<th>100ns</th>
<th>10ms</th>
<th>10sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>KB</td>
<td>MB</td>
<td>GB</td>
<td>TB</td>
<td>PB</td>
</tr>
</tbody>
</table>

Memory Hierarchy Diagram:

- Processor
- Control
- Datapath
- Registers
- On-chip cache
- Second level cache (DRAM)
- Main memory (DRAM)
- Secondary storage (Disk)
- Tertiary storage (Disk/Tape) (“Cloud”)
**Cache Basics**

- **Cache** is fast (expensive) memory which keeps copy of data in main memory; it is hidden from software
  - Simplest example: data at memory address xxxxx1101 is stored at cache location 1101
- **Cache hit**: in-cache memory access—cheap
- **Cache miss**: non-cached memory access—expensive
  - Need to access next, slower level of cache
- **Cache line length**: # of bytes loaded together in one entry
  - Ex: If either xxxxx1100 or xxxxx1101 is loaded, both are
- **Associativity**
  - direct-mapped: only 1 address (line) in a given range in cache
    - Data stored at address xxxxx1101 stored at cache location 1101, in 16 word cache
  - n-way: \( n \geq 2 \) lines with different addresses can be stored
    - Up to \( n \approx 16 \) words with addresses xxxxx1101 can be stored at cache location 1101 (so cache can store 16n words)

**Why Have Multiple Levels of Cache?**

- On-chip vs. off-chip
  - On-chip caches are faster, but limited in size
- A large cache has delays
  - Hardware to check longer addresses in cache takes more time
  - Associativity, which gives a more general set of data in cache, also takes more time
- Some examples:
  - Cray T3E eliminated one cache to speed up misses
  - IBM uses a level of cache as a "victim cache" which is cheaper
- There are other levels of the memory hierarchy
  - Register, pages (TLB, virtual memory), ...
  - And it isn't always a hierarchy

**Experimental Study of Memory (Membench)**

- Microbenchmark for memory system performance

  ```
  for i from 0 to L-1
  load A[i] from memory (4 Bytes)
  ```

- for array A of length L from 4KB to 8MB by 2x
  - for stride s from 4 Bytes (1 word) to L/2 by 2x
  - time the following loop
    (repeat many times and average)
  - for i from 0 to L-1 by s
    - load A[i] from memory (4 Bytes)

**Membench: What to Expect**

- Consider the average cost per load
  - Plot one line for each array length, time vs. stride
  - Small stride is best: if cache line holds 4 words, at most \( \frac{1}{4} \) miss
  - If array is smaller than a given cache, all those accesses will hit (after the first run, which is negligible for large enough runs)
  - Picture assumes only one level of cache
  - Values have gotten more difficult to measure on modern procs
**Memory Hierarchy on a Sun Ultra-2i**

- **L1:** 16 KB, 2 cycles (6ns)
- **L2:** 64 byte line
  - See [www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps](http://www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps) for details
- **L2:** 2 MB, 12 cycles (36 ns)

**Memory Hierarchy on a Power3 (Seaborg)**

- **Power3:** 375 MHz
- **L1:** 32 KB, .5-2 cycles
- **L2:** 8 MB, 128 B line
  - 9 cycles

**Memory Hierarchy on an Intel Core 2 Duo**

- **Intel Core 2 Duo (Merom, P6, 2.39 GHz, 128 KB L2, 1.39 GHz):**

**Stanza Triad**

- Even smaller benchmark for prefetching
- Derived from STREAM Triad
- Stanza (L) is the length of a unit stride run

\[
\text{while } i < \text{arraylength} \\
\quad \\
\text{for each } L \text{ element stanza} \\
A[i] = \text{scalar} \times X[i] + Y[i] \\
\text{skip } k \text{ elements}
\]

1) do $L$ triads stanza
2) skip $k$ elements
3) do $L$ triads stanza

Source: Kamil et al, MSP05
Stanza Triad Results

- This graph (x-axis) starts at a cache line size (>=16 Bytes)
- If cache locality was the only thing that mattered, we would expect flat lines equal to measured memory peak bandwidth (STREAM) as on Pentium3
- Prefetching gets the next cache line (pipelining) while using the current one
- This does not “kick in” immediately, so performance depends on L

Lessons

- Actual performance of a simple program can be a complicated function of the architecture
  - Slight changes in the architecture or program change the performance significantly
  - To write fast programs, need to consider architecture
    - True on sequential or parallel processor
  - We would like simple models to help us design efficient algorithms
- We will illustrate with a common technique for improving cache performance, called blocking or tiling
  - Idea: used divide-and-conquer to define a problem that fits in register/L1-cache/L2-cache

Outline

- Idealized and actual costs in modern processors
- Memory hierarchies
  - Use of microbenchmarks to characterized performance
- Parallelism within single processors
  - Hidden from software (sort of)
  - Pipelining
  - SIMD units
- Case study: Matrix Multiplication
  - Use of performance models to understand performance
  - Attainable lower bounds on communication

What is Pipelining?

Dave Patterson’s Laundry example: 4 people doing laundry
wash (30 min) + dry (40 min) + fold (20 min) = 90 min

- In this example:
  - Sequential execution takes 4 * 90min = 6 hours
  - Pipelined execution takes 30+4+4+40+20 = 3.5 hours
  - Bandwidth = loads/hour
    - BW = 4/6 l/h w/o pipelining
    - BW = 4/3.5 l/h w pipelining
    - BW = 1.5 l/h w pipelining, more total loads
  - Pipelining helps bandwidth but not latency (90 min)
  - Bandwidth limited by slowest pipeline stage
  - Potential speedup = Number of pipe stages
**Example: 5 Steps of MIPS Datapath**

Figure 3.4, Page 134, CA:AQA 2e by Patterson and Hennessy

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Next PC</td>
<td>Next SEQ PC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Pipelining is also used within arithmetic units
  - A fp multiply may have latency 10 cycles, but throughput of 1/cycle

**SIMD: Single Instruction, Multiple Data**

- **Scalar processing**
  - Traditional mode
  - One operation produces one result
- **SIMD processing**
  - With SSE / SSE2
  - SSE = streaming SIMD extensions
  - One operation produces multiple results

![SIMD Operations](image)

**SSE / SSE2 SIMD on Intel**

- SSE2 data types: Anything that fits into 16 bytes, e.g.,
  - 4x floats
  - 2x doubles
  - 16x bytes
- Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel
- Challenges:
  - Need to be contiguous in memory and aligned
  - Some instructions to move data around from one part of register to another
  - Similar on GPUs, vector processors (but many more simultaneous operations)

**What does this mean to you?**

- In addition to SIMD extensions, the processor may have other special instructions
  - Fused Multiply-Add (FMA) instructions:
    \[ x = y + c \times z \]
    - Is so common some processor execute the multiply/add as a single instruction, at the same rate (bandwidth) as + or * alone
- In theory, the compiler understands all of this
  - When compiling, it will rearrange instructions to get a good “schedule” that maximizes pipelining, uses FMA's and SIMD
  - It works with the mix of instructions inside an inner loop or other block of code
- But in practice the compiler may need your help
  - Choose a different compiler, optimization flags, etc.
  - Rearrange your code to make things more obvious
  - Using special functions (“intrinsics”) or write in assembly

Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

01/21/2016 CS267 - Lecture 2
Outline

- Idealized and actual costs in modern processors
- Memory hierarchies
  - Use of microbenchmarks to characterized performance
- Parallelism within single processors
- Case study: Matrix Multiplication
  - Use of performance models to understand performance
  - Attainable lower bounds on communication
  - Simple cache model
  - Warm-up: Matrix-vector multiplication
  - Naive vs optimized Matrix-Matrix Multiply
    - Minimizing data movement
    - Beating $O(n^3)$ operations
    - Practical optimizations (continued next time)

Why Matrix Multiplication?

- An important kernel in many problems
  - Appears in many linear algebra algorithms
  - Bottleneck for dense linear algebra, including Top500
  - One of the 7 dwarfs / 13 motifs of parallel computing
  - Closely related to other algorithms, e.g., transitive closure on a graph using Floyd-Warshall
- Optimization ideas can be used in other problems
- The best case for optimization payoffs
- The most-studied algorithm in high performance computing

Matrix-multiply, optimized several ways

What do commercial and CSE applications have in common?

Motif/Dwarf: Common Computational Methods
(Red Hot → Blue Cool)

1 Finite State Mach.
2 Combinational
3 Graph Traversal
4 Structured Grid
5 Dense Matrix
6 Sparse Matrix
7 Spectral (FFT)
8 Dynamic Prog
9 N-Body
10 MapReduce
11 Backtrack/B&B
12 Graphical Models
13 Unstructured Grid

Speed of n-by-n matrix multiply on Sun Ultra-1/170, peak = 330 MFlops
**Note on Matrix Storage**

- A matrix is a 2-D array of elements, but memory addresses are “1-D”

- Conventions for matrix layout
  - by column, or “column major” (Fortran default) \( A(i,j) \) at \( A+i+j*n \)
  - by row, or “row major” (C default) \( A(i,j) \) at \( A+i*n+j \)
  - recursive (later)

- Column major (for now)

- Column major matrix in memory

- Blue row of matrix is stored in red cachelines

- Column major in memory

**Using a Simple Model of Memory to Optimize**

- Assume just 2 levels in the hierarchy, fast and slow
- All data initially in slow memory
  - \( m \) = number of memory elements (words) moved between fast and slow memory
  - \( t_m \) = time per slow memory operation
  - \( f \) = number of arithmetic operations
  - \( t_f \) = time per arithmetic operation \( << t_m \)
  - \( q \) = \( f / m \) average number of flops per slow memory access

- Minimum possible time = \( f * t_f \) when all data in fast memory
- Actual time
  - \( f * t_f + m * t_m - f * t_f * (1 + t_m / t_f * 1/q) \)

- Larger \( q \) means time closer to minimum \( f * t_f \)
  - \( q \approx t_f / t_m \) needed to get at least half of peak speed

**Warm up: Matrix-vector multiplication**

```
\{ \text{implements } y = y + A*x \}
\text{for } i = 1:n
\text{for } j = 1:n
\quad y(i) = y(i) + A(i,j)*x(j)
```

- \( m \) = number of slow memory refs = \( 3n + n^2 \)
- \( f \) = number of arithmetic operations = \( 2n^2 \)
- \( q \approx f / m = 2 \)

- Matrix-vector multiplication limited by slow memory speed
Modeling Matrix-Vector Multiplication

- Compute time for \(nxn = 1000 \times 1000\) matrix
- Time
  \[t_x + t_y + t_m = f \cdot t_x + m \cdot t_y = f \cdot t_x + m \cdot t_y + \frac{1}{3} \frac{2n^2 \cdot t_y}{t_x} + \frac{1}{3} \frac{1}{q} \]
- For \(t_x\) and \(t_y\) using data from R. Vuduc’s PhD (pp 351-3)
  - For \(t_m\) use minimum-memory-latency / words-per-cache-line

<table>
<thead>
<tr>
<th>Machine</th>
<th>Clock (MHz)</th>
<th>Peak MFlops/s</th>
<th>Mem Lat (Min,Max)</th>
<th>Linesize (Bytes)</th>
<th>(t_m/t_x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra 2i</td>
<td>333</td>
<td>667</td>
<td>38</td>
<td>66</td>
<td>16</td>
</tr>
<tr>
<td>Ultra 3</td>
<td>900</td>
<td>1800</td>
<td>28</td>
<td>200</td>
<td>32</td>
</tr>
<tr>
<td>Pentium 3</td>
<td>500</td>
<td>500</td>
<td>25</td>
<td>60</td>
<td>32</td>
</tr>
<tr>
<td>Pentium3M</td>
<td>800</td>
<td>800</td>
<td>40</td>
<td>60</td>
<td>32</td>
</tr>
<tr>
<td>Power3</td>
<td>375</td>
<td>1500</td>
<td>35</td>
<td>139</td>
<td>128</td>
</tr>
<tr>
<td>Power4</td>
<td>1300</td>
<td>5200</td>
<td>60</td>
<td>10000</td>
<td>128</td>
</tr>
<tr>
<td>Itanium1</td>
<td>900</td>
<td>3200</td>
<td>36</td>
<td>85</td>
<td>32</td>
</tr>
<tr>
<td>Itanium2</td>
<td>900</td>
<td>3600</td>
<td>11</td>
<td>60</td>
<td>64</td>
</tr>
</tbody>
</table>

Simplifying Assumptions

- What simplifying assumptions did we make in this analysis?
  - Ignored parallelism in processor between memory and arithmetic within the processor
  - Sometimes drop arithmetic term in this type of analysis
  - Assumed fast memory was large enough to hold three vectors
    - Reasonable if we are talking about any level of cache
    - Not if we are talking about registers (~32 words)
  - Assumed the cost of a fast memory access is 0
    - Reasonable if we are talking about registers
    - Not necessarily if we are talking about cache (1-2 cycles for L1)
  - Memory latency is constant
  - Could simplify even further by ignoring memory operations in X and Y vectors
    - Mflop rate/element = \(2 / (2 \cdot t_x + t_y)\)

Validating the Model

- How well does the model predict actual performance?
  - Actual DGEMV: Most highly optimized code for the platform
  - Model sufficient to compare across machines
  - But under-predicting on most recent ones due to latency estimate

Naïve Matrix Multiply

\begin{align*}
\text{(Implements} \ C &= C + A^B) \\
\text{for } i &= 1 \text{ to } n \\
\text{for } j &= 1 \text{ to } n \\
\text{for } k &= 1 \text{ to } n \\
C(i,j) &= C(i,j) + A(i,k) \cdot B(k,j)
\end{align*}

Algorithm has \(2n^3 = O(n^3)\) Flops and operates on \(3n^2\) words of memory

\[q \text{ potentially as large as } 2n^2 / 3n^2 = O(n)\]
**Naïve Matrix Multiply**

\( \{ \text{implies } C = C + A \times B \} \)

for \( i = 1 \) to \( n \)

\( \{ \text{read row } i \text{ of } A \text{ into fast memory} \} \)

for \( j = 1 \) to \( n \)

\( \{ \text{read } C(i,j) \text{ into fast memory} \} \)

for \( k = 1 \) to \( n \)

\( C(i,j) = C(i,j) + A(i,k) \times B(k,j) \)

\( \{ \text{write } C(i,j) \text{ back to slow memory} \} \)

---

**Naïve Matrix Multiply**

Number of slow memory references on unblocked matrix multiply:

\( m = n^3 \) to read each column of \( B \)

\( n^2 \) to read each row of \( A \) once

\( 2n^2 \) to read and write each element of \( C \) once

So \( q = \frac{f}{m} = \frac{2n^3}{n^3 + 3n^2} \)

\( \approx 2 \) for large \( n \), no improvement over matrix-vector multiply

Inner two loops are just matrix-vector multiply, of row \( i \) of \( A \) times \( B \)
Similar for any other order of 3 loops

---

**Matrix-multiply, optimized several ways**

Speed of \( n \)-by-\( n \) matrix multiply on Sun Ultra-1/170, peak = 330 MFlops

**Naïve Matrix Multiply on RS/6000**

- \( T = N^{4.7} \)
- \( 12000 \) would take \( 1095 \) years
- Size 2000 took 5 days

\( O(N^3) \) performance would have constant cycles/flop
Performance looks like \( O(N^{4.7}) \)

Slide source: Larry Carter, UCSD
Naïve Matrix Multiply on RS/6000

Consider A, B, C to be N-by-N matrices of b-by-b subblocks where $b = n / N$ is called the block size.

For $i = 1$ to $N$
- Read block $C(i,j)$ into fast memory

For $j = 1$ to $N$
- Read block $A(i,k)$ into fast memory
- Read block $B(k,j)$ into fast memory

$C(i,j) = C(i,j) + A(i,k) * B(k,j)$ (do a matrix multiply on blocks)

Write block $C(i,j)$ back to slow memory

Blocked (Tiled) Matrix Multiply

Recall:
- $m$ is amount memory traffic between slow and fast memory
- Matrix has $n^2$ elements, and $N \times N$ blocks each of size $b \times b$
- $f$ is number of floating point operations, $2n^3$ for this problem
- $q = f / m$ is our measure of algorithm efficiency in the memory system

So:
- $m = N^2 n^2$: read each block of B $N^2$ times ($N^3 * b^3 = N^3 * (n/N)^2 = N^3 n^2$)
- $N^2 n^2$: read each block of A $N^3$ times
- $2n^2$: read and write each block of C once

So computational intensity $q = f / m = 2n^3 / ((2N + 2) * n^2)$

$= n / N = b$ for large $n$

So we can improve performance by increasing the blocksize $b$

Can be much faster than matrix-vector multiply ($q=2$)

Using Analysis to Understand Machines

The blocked algorithm has computational intensity $q = b$

- The larger the block size, the more efficient our algorithm will be
- Limit: All three blocks from A, B, C must fit in fast memory (cache), so we cannot make these blocks arbitrarily large
- Assume your fast memory has size $M_{fast}$

$3b^2 = M_{fast}$, so $q = b = (M_{fast}/3)^{1/2}$

- To build a machine to run matrix multiply at 1/2 peak arithmetic speed of the machine, we need a fast memory of size $M_{fast} = 3b^2 = 3q^2 = 3t_m/8$.

- This size is reasonable for L1 cache, but not for register sets

- Note: analysis assumes it is possible to schedule the instructions perfectly

<table>
<thead>
<tr>
<th>Machine</th>
<th>$t_{m/f}$</th>
<th>8 KB</th>
<th>16 KB</th>
<th>32 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra 2i</td>
<td>24.8</td>
<td>14.8</td>
<td>7.4</td>
<td>3.7</td>
</tr>
<tr>
<td>Ultra 3</td>
<td>14</td>
<td>4.7</td>
<td>2.4</td>
<td>1.2</td>
</tr>
<tr>
<td>Pentium 3</td>
<td>6.25</td>
<td>0.9</td>
<td>0.45</td>
<td>0.22</td>
</tr>
<tr>
<td>Pentium3M</td>
<td>6.25</td>
<td>0.9</td>
<td>0.45</td>
<td>0.22</td>
</tr>
<tr>
<td>Power3</td>
<td>8.75</td>
<td>1.8</td>
<td>0.9</td>
<td>0.45</td>
</tr>
<tr>
<td>Power4</td>
<td>15</td>
<td>5.4</td>
<td>2.7</td>
<td>1.35</td>
</tr>
<tr>
<td>Itanium1</td>
<td>36</td>
<td>31.1</td>
<td>15.6</td>
<td>7.8</td>
</tr>
<tr>
<td>Itanium2</td>
<td>5.5</td>
<td>0.7</td>
<td>0.35</td>
<td>0.18</td>
</tr>
</tbody>
</table>
Limits to Optimizing Matrix Multiply

- The blocked algorithm changes the order in which values are accumulated into each $C[i,j]$ by applying commutativity and associativity. Get slightly different answers from naïve code, because of roundoff - OK.
- The previous analysis showed that the blocked algorithm has computational intensity:
  $$q = b = (M_{\text{fast}}/3)^{1/2}$$
- There is a lower bound result that says we cannot do any better than this (using only associativity, so still doing $n^3$ multiplications).
- Theorem (Hong & Kung, 1981): Any reorganization of this algorithm (that uses only associativity) is limited to $q = O((M_{\text{fast}})^{1/2})$.
  - #words moved between fast and slow memory = $\Omega(n^3/M_{\text{fast}}^{3/2})$.

Communication lower bounds for Matmul

- Hong/Kung theorem is a lower bound on amount of data communicated by matmul.
- Number of words moved between fast and slow memory (cache and DRAM, or DRAM and disk, or ...) = $\Omega(n^3/M_{\text{fast}}^{3/2})$.
- Cost of moving data may also depend on the number of “messages” into which data is packed.
  - Eg: number of cache lines, disk accesses, ...
  - #messages = $O(n^3/M_{\text{fast}}^{3/2})$.
- Lower bounds extend to anything “similar enough” to nested loops.
  - Rest of linear algebra (solving linear systems, least squares...)
  - Dense and sparse matrices
  - Sequential and parallel algorithms, ...
- More recent: extends to any nested loops accessing arrays
  - Need (more) new algorithms to attain these lower bounds...

Review of lecture 2 so far (and a look ahead)

- Applications
  - How to decompose into well-understood algorithms (and their implementations)
- Algorithms (matmul as example)
  - Need simple model of hardware to guide design, analysis: minimize accesses to slow memory
  - If lucky, theory describing “best algorithm”
  - For $O(n^3)$ sequential matmul, must move $\Omega(n^3/M_{\text{fast}}^{1/2})$ words
- Software tools
  - How do I implement my applications and algorithms in most efficient and productive way?
- Hardware
  - Even simple programs have complicated behaviors
  - “Small” changes make execution time vary by orders of magnitude

Basic Linear Algebra Subroutines (BLAS)

- Industry standard interface (evolving)
- Vendors, others supply optimized implementations
- History
  - BLAS1 (1970s):
    - vector operations: dot product, saxpy ($y=\alpha x+y$), etc
    - $m=2n$, $f=2n$, $q = f/m = \text{computational intensity} \leq 1$ or less
  - BLAS2 (mid 1980s)
    - matrix-vector operations: matrix vector multiply, etc
    - $m=n^2$, $f=2n^2$, $q=-2$, less overhead
    - somewhat faster than BLAS1
  - BLAS3 (late 1980s)
    - matrix-matrix operations: matrix matrix multiply, etc
    - $m=n$, $f=n^3$, $q=n$, so $q/f$ can possibly be as large as $n$, so BLAS3 is potentially much faster than BLAS2
- Good algorithms use BLAS3 when possible (LAPACK & ScaLAPACK)
  - See www.netlib.org/lapack,scalapack
  - More later in course
BLAS speeds on an IBM RS6000/590

Peak speed = 266 Mflops

Dense Linear Algebra: BLAS2 vs. BLAS3

- BLAS2 and BLAS3 have very different computational intensity, and therefore different performance

What if there are more than 2 levels of memory?

- Need to minimize communication between all levels
  - Between L1 and L2 cache, cache and DRAM, DRAM and disk...
- The tiled algorithm requires finding a good block size
  - Machine dependent
  - Need to “block” b x b matrix multiply in innermost loop
    - 1 level of memory => 3 nested loops (naive algorithm)
    - 2 levels of memory => 6 nested loops
    - 3 levels of memory => 9 nested loops ...
- Cache Oblivious Algorithms offer an alternative
  - Treat nxn matrix multiply as a set of smaller problems
  - Eventually, these will fit in cache
  - Will minimize # words moved between every level of memory hierarchy – at least asymptotically
  - “Oblivious” to number and sizes of levels

Recursive Matrix Multiplication (RMM) (1/2)

- True when each A_{ij} etc. 1x1 or n/2 x n/2
- For simplicity: square matrices with n = 2^m
- Extends to general rectangular case

```python
func C = RMM (A, B, n)
if n = 1, C = A * B, else
{  C_{11} = RMM (A_{11}, B_{11}, n/2) + RMM (A_{12}, B_{21}, n/2)
   C_{12} = RMM (A_{11}, B_{12}, n/2) + RMM (A_{12}, B_{22}, n/2)
   C_{21} = RMM (A_{21}, B_{11}, n/2) + RMM (A_{22}, B_{21}, n/2)
   C_{22} = RMM (A_{21}, B_{12}, n/2) + RMM (A_{22}, B_{22}, n/2)  
} return
```
Recursive Matrix Multiplication (2/2)

```plaintext
func C = RMM (A, B, n)
if n=1, C = A * B, else
{  C_{11} = RMM (A_{11}, B_{11}, n/2) + RMM (A_{12}, B_{21}, n/2)
  C_{12} = RMM (A_{11}, B_{12}, n/2) + RMM (A_{12}, B_{22}, n/2)
  C_{21} = RMM (A_{21}, B_{11}, n/2) + RMM (A_{22}, B_{21}, n/2)
  C_{22} = RMM (A_{21}, B_{12}, n/2) + RMM (A_{22}, B_{22}, n/2) }
return
```

A(n) = # arithmetic operations in RMM( . , . , n)
= 8 \cdot A(n/2) + 4(n/2)^2 if n > 1, else 1
= 2n^3 if same operations as usual, in different order

W(n) = # words moved between fast, slow memory by RMM( . , . , n)
= 8 \cdot W(n/2) + 4 \cdot 3(n/2)^2 if 3n^2 > M_{fast}, else 3n^2
= O( n^3 / (M_{fast})^{1/2} + n^2 ) same as blocked matmul
Don’t need to know M_{fast} for this to work!

Experience with Cache-Oblivious Algorithms

- In practice, need to cut off recursion well before 1x1 blocks
  - Call “micro-kernel” on small blocks
- Implementing a high-performance Cache-Oblivious code is not easy
  - Careful attention to micro-kernel is needed
- Using fully recursive approach with highly optimized recursive micro-kernel, Pingali et al report that they never got more than 2/3 of peak. (unpublished, presented at LACSI’06)
- Issues with Cache Oblivious (recursive) approach
  - Recursive Micro-Kernels yield less performance than iterative ones using same scheduling techniques
  - Pre-fetching is needed to compete with best code: not well-understood in the context of Cache-Oblivious codes
- More recent work on CARMA (UCB) uses recursion for parallelism, but aware of available memory, very fast (later)

Recursion: Cache Oblivious Algorithms

- The tiled algorithm requires finding a good block size
- Cache Oblivious Algorithms offer an alternative
  - Treat nxn matrix multiply set of smaller problems
  - Eventually, these will fit in cache
- Cases for A (nxm) * B (mxp)
  - Case 1: m >= max{n,p}: split A horizontally
  - Case 2: n >= max{m,p}: split A vertically and B horizontally
  - Case 3: p >= max{m,n}: split B vertically

Recursive Data Layouts

- A related idea is to use a recursive structure for the matrix
  - Improve locality with machine-independent data structure
  - Can minimize latency with multiple levels of memory hierarchy
- There are several possible recursive decompositions depending on the order of the sub-blocks
- This figure shows Z-Morton Ordering (“space filling curve”)
- See papers on “cache oblivious algorithms” and “recursive layouts”

Advantages:
  - the recursive layout works well for any cache size
Disadvantages:
  - The index calculations to find A[i,j] are expensive
  - Implementations switch to column-major for small sizes
Strassen’s Matrix Multiply

- The traditional algorithm (with or without tiling) has \(O(n^3)\) flops
- Strassen discovered an algorithm with asymptotically lower flops
  - \(O(n^{2.81})\)
- Consider a 2x2 matrix multiply, normally takes 8 multiplies, 4 adds
  - Strassen does it with 7 multiplies and 18 adds

\[
\begin{pmatrix}
m_{11} & m_{12} \\
m_{21} & m_{22}
\end{pmatrix}
= \begin{pmatrix}
a_{11} & a_{12} \\
a_{21} & a_{22}
\end{pmatrix}
\begin{pmatrix}
b_{11} & b_{12} \\
b_{21} & b_{22}
\end{pmatrix}
\]

Let \(p_1 = (a_{12} - a_{22}) \times (b_{21} + b_{22})\)
\(p_5 = a_{11} \times (b_{12} - b_{22})\)
\(p_2 = (a_{11} + a_{22}) \times (b_{11} + b_{22})\)
\(p_6 = a_{22} \times (b_{21} - b_{11})\)
\(p_3 = (a_{11} - a_{21}) \times (b_{11} + b_{12})\)
\(p_7 = (a_{21} + a_{22}) \times b_{11}\)

\[
\begin{align*}
m_{11} &= p_1 + p_2 - p_4 + p_6 \\
m_{12} &= p_4 + p_5 \\
m_{21} &= p_6 + p_7 \\
m_{22} &= p_2 - p_3 + p_5 - p_7
\end{align*}
\]

Extends to nxn by divide & conquer

Strassen (continued)

\[
T(n) = Cost \ of \ multiplying \ nxn \ matrices
\]
\[
= 7^r(T(n/2)) + 18^r(n/2)^2
\]
\[
= O(n \log_2 7) \approx O(n^{2.81})
\]
- Asymptotically faster
- Several times faster for large \(n\) in practice
- Cross-over depends on machine
- “Tuning Strassen’s Matrix Multiplication for Memory Efficiency”, M. S. Thottethodi, S. Chatterjee, and A. Lebeck, in Proceedings of Supercomputing ’98
- Possible to extend communication lower bound to Strassen
- Words moved between fast and slow memory
  \(= \Omega(n \log^2 7 / M^{0.4})\)
- (Ballard, D., Holtz, Schwartz, 2011, SPAA Best Paper Prize)
- Attainable too, more on parallel version later

Other Fast Matrix Multiplication Algorithms

- World’s record was \(O(n^{2.37548\ldots})\)
  - Coppersmith & Winograd, 1987
- New Record! 2.37548 reduced to 2.37293
  - Virginia Vassilevska Williams, UC Berkeley & Stanford, 2011
- Newer Record! 2.37293 reduced to 2.37286
  - François Le Gall, 2014
- Lower bound on #words moved can be extended to (some) of these algorithms
- Possibility of \(O(n^{2.37})\) algorithm!
- Cohn, Umans, Kleinberg, 2003
- Can show they all can be made numerically stable
  - D., Dumitriu, Holtz, Kleinberg, 2007
- Can do rest of linear algebra (solve \(Ax=b\), \(Ax=\lambda x\), etc) as fast, and numerically stably
  - D., Dumitriu, Holtz, 2008
- Fast methods (besides Strassen) may need unrealistically large \(n\)

Tuning Code in Practice

- Tuning code can be tedious
- Lots of code variations to try besides blocking
- Machine hardware performance hard to predict
- Compiler behavior hard to predict
- Response: “Autotuning”
  - Let computer generate large set of possible code variations, and search them for the fastest ones
  - Used with CS267 homework assignment in mid 1990s
    - PHiPAC, leading to ATLAS, incorporated in Matlab
    - We still use the same assignment
  - We (and others) are extending autotuning to other dwarfs / motifs
- Still need to understand how to do it by hand
  - Not every code will have an autotuner
  - Need to know if you want to build autotuners
Search Over Block Sizes

- Performance models are useful for high level algorithms
  - Helps in developing a blocked algorithm
  - Models have not proven very useful for block size selection
    - too complicated to be useful
      - See work by Sid Chatterjee for detailed model
    - too simple to be accurate
      - Multiple multidimensional arrays, virtual memory, etc.
  - Speed depends on matrix dimensions, details of code, compiler, processor

What the Search Space Looks Like

A 2-D slice of a 3-D register-tile search space. The dark blue region was pruned.
(Platform: Sun Ultra-IIi, 333 MHz, 667 Mflop/s peak, Sun cc v5.0 compiler)

ATLAS (DGEMM n = 500)

- ATLAS is faster than all other portable BLAS implementations and it is comparable with machine-specific libraries provided by the vendor.

Optimizing in Practice

- Tiling for registers
  - loop unrolling, use of named "register" variables
- Tiling for multiple levels of cache and TLB
- Exploiting fine-grained parallelism in processor
  - superscalar; pipelining
- Complicated compiler interactions (flags)
- Hard to do by hand (but you’ll try)
- Automatic optimization an active research area
  - ASPIRE: aspire.eecs.berkeley.edu
  - BeBOP: bebop.cs.berkeley.edu
    - Weekly group meeting Mondays 1pm
  - PHiPAC: www.icsi.berkeley.edu/~bilmes/phipac
    in particular tr-98-035.ps.gz
  - ATLAS: www.netlib.org/atlas
Removing False Dependencies

• Using local variables, reorder operations to remove false dependencies

```
a[i] = b[i] + c;  \quad \text{false read-after-write hazard}
a[i+1] = b[i+1] + d;  \quad \text{between} \ a[i] \ \text{and} \ b[i+1]
```

```
float f1 = b[i];
float f2 = b[i+1];
a[i] = f1 + c;
a[i+1] = f2 + d;
```

With some compilers, you can declare `a` and `b` unaliased.

• Done via “restrict pointers,” compiler flag, or pragma

Exploit Multiple Registers

• Reduce demands on memory bandwidth by pre-loading into local variables

```
while( … ) {
  \*res++ = filter[0]*signal[0] + filter[1]*signal[1] + filter[2]*signal[2];
  signal++;
}
```

```
float f0 = filter[0];
float f1 = filter[1];
float f2 = filter[2];
while( … ) {
  \*res++ = f0*signal[0] + f1*signal[1] + f2*signal[2];
  signal++;
}
```

Example is a convolution

Exploit Independent Operations

• Expose instruction-level parallelism

```
float f0 = filter[0], f1 = filter[1], f2 = filter[2];
float s0 = signal[0], s1 = signal[1], s2 = signal[2];
*res++ = f0*s0 + f1*s1 + f2*s2;
do {
  signal += 3;
  s0 = signal[0];
  res[0] = f0*s1 + f1*s2 + f2*s0;
  s1 = signal[1];
  res[1] = f0*s2 + f1*s0 + f2*s1;
  s2 = signal[2];
  res[2] = f0*s0 + f1*s1 + f2*s2;
  res += 3;
} while( … );
```

• Expose independent operations that can execute in parallel or in a pipelined fashion

```
f1 = f5 * f9;
f2 = f6 + f10;
f3 = f7 * f11;
f4 = f8 + f12;
```

Exposé Independent Operations
Copy optimization

- Copy input operands or blocks
  - Reduce cache conflicts
  - Constant array offsets for fixed size blocks
  - Expose page-level locality
  - Alternative: use different data structures from start (if users willing)
    - Recall recursive data layouts

Locality in Other Algorithms

- The performance of any algorithm is limited by $q$
  - $q = \text{"computational intensity"} = \frac{\text{#arithmetic ops}}{\text{#words moved}}$
- In matrix multiply, we increase $q$ by changing computation order
  - increased temporal locality
- For other algorithms and data structures, even hand-transformations are still an open problem
  - Lots of open problems, class projects

Summary of Lecture 2

- Details of machine are important for performance
  - Processor and memory system (not just parallelism)
  - Before you parallelize, make sure you’re getting good serial performance
  - What to expect? Use understanding of hardware limits
- There is parallelism hidden within processors
  - Pipelining, SIMD, etc
- Machines have memory hierarchies
  - 100s of cycles to read from DRAM (main memory)
  - Caches are fast (small) memory that optimize average case
- Locality is at least as important as computation
  - Temporal: re-use of data recently used
  - Spatial: using data nearby to recently used data
- Can rearrange code/data to improve locality
  - Goal: minimize communication = data movement

Class Logistics

- Homework 0 posted on web site
  - Find and describe interesting application of parallelism
  - Due Friday Jan 30
  - Could even be your intended class project
- Please fill in on-line class survey
  - We need this to assign teams for Homework 1
- Please fill out on-line request for Stampede account
  - Needed for GPU part of assignment 2
Some reading for today (see website)

- Sourcebook Chapter 3. (note that chapters 2 and 3 cover the material of lecture 2 and lecture 3, but not in the same order).
- Web pages for reference:
  - BeBOP Homepage
  - ATLAS Homepage
  - BLAS (Basic Linear Algebra Subroutines). Reference for (unoptimized) implementations of the BLAS, with documentation.
  - LAPACK (Linear Algebra PACKAGE), a standard linear algebra library optimized to use the BLAS effectively on uniprocessors and shared memory machines (software, documentation and reports)
  - ScalAPACK (Scalable LAPACK), a parallel version of LAPACK for distributed memory machines (software, documentation and reports).
  - Many related papers at bebop.cs.berkeley.edu