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
    
    \[
    \begin{align*}
    A &= B + C \\
    \text{Read address}(B) &\to R1 \\
    \text{Read address}(C) &\to R2 \\
    R3 &= R1 + R2 \\
    \text{Write R3 to Address}(A)
    \end{align*}
    \]

- 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 characterized 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

Memory Hierarchy Diagram

- Speed: 1ns, 10ns, 100ns, 10ms, 10sec
- Size: KB, MB, GB, TB, PB

Processor-DRAM Gap (latency)

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

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
**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 ≥ 2 lines with different addresses can be stored
    - Up to n = 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 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 by s
      load A[i] from memory (4 Bytes)
      1 experiment

**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 ¼ 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 (6 ns)
- L2: 64 byte line
- L2: 2 MB, 12 cycles (36 ns)

See www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps for details

Memory Hierarchy on a Power3 (Seaborg)

- L1: 32 KB, 128B line, 9 cycles
- L2: 8 MB, 128B line, 12 cycles (36 ns)

Memory Hierarchy on an Intel Core 2 Duo

Stanza Triad

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

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

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

Latency

- In this example:
  - Sequential execution takes 4 * 90min = 6 hours
  - Pipelined execution takes 30 * 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

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

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)

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

What does this mean to you?

- In addition to SimD extensions, the processor may have other special instructions
  - Fused Multiply-Add (FMA) instructions:
    \[ x = y + c \cdot z \]
    - Is so common some processor execute the multiply/add as a single instruction, at the same rate (bandwidth) as + or \( \cdot \) alone
  - In theory, the compiler understands all of this
    - When compiling, it will rearrange instructions to get a good “schedule” that maximizes pipelining, uses FMAs 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
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
• Naïve 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

What do commercial and CSE applications have in common?

Motif/Dwarf: Common Computational Methods
(\text{Red Hot} \rightarrow \text{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

Matrix-multiply, optimized several ways

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)**

![Matrix Storage Diagram](image)

01/23/2014

**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
- \( q = f / m \) average number of flops per slow memory access
- Minimum possible time = \( f * t_f \) when all data in fast memory
- Actual time
  - \( f * t_f + m * t_m = f * t_f + (1 + t_m / t_f) * t_m / q \)
- Larger \( q \) means time closer to minimum \( f * t_f \)
  - \( q > t_m / t_f \) needed to get at least half of peak speed

**Computational Intensity:** Key to algorithm efficiency

**Machine Balance:** Key to machine efficiency

01/23/2014

**Warm up: Matrix-vector multiplication**

(implies \( y = y + A*x \))

for \( i = 1:n \)

for \( j = 1:n \)

\[ y(i) = y(i) + A(i,j) * x(j) \]

![Matrix-vector multiplication](image)

01/23/2014
Warm up: Matrix-vector multiplication

{read x(1:n) into fast memory}
{read y(1:n) into fast memory}
for i = 1:n
  {read row i of A into fast memory}
  for j = 1:n
    y(i) = y(i) + A(i,j)*x(j)
{write y(1:n) back to slow memory}

• \( m = \text{number of slow memory refs} = 3n + n^2 \)
• \( f = \text{number of arithmetic operations} = 2n^2 \)
• \( q = \frac{f}{m} = 2 \)

Matrix-vector multiplication limited by slow memory speed

Modeling Matrix-Vector Multiplication

• Compute time for \( nxn = 1000\times1000 \) matrix

  \[
  t = t_f \cdot f + t_m \cdot m = t_f \cdot f \left(1 + \frac{t_m}{t_f} \cdot \frac{1}{q} \right)
  \]

• For \( t_f \) and \( t_m \), using data from R. Vuduc’s PhD (pp 351-3)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Clock</th>
<th>Peak Mflop/s</th>
<th>Mem Lat (Min,Max)</th>
<th>Linesize</th>
<th>( t_m/f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra 2i</td>
<td>333</td>
<td>667</td>
<td>(38, 66)</td>
<td>16</td>
<td>24.8</td>
</tr>
<tr>
<td>Ultra 3</td>
<td>900</td>
<td>1800</td>
<td>(28, 200)</td>
<td>32</td>
<td>14.0</td>
</tr>
<tr>
<td>Pentium 3</td>
<td>500</td>
<td>600</td>
<td>(25, 60)</td>
<td>32</td>
<td>6.3</td>
</tr>
<tr>
<td>Pentium3M</td>
<td>800</td>
<td>800</td>
<td>(40, 60)</td>
<td>32</td>
<td>10.0</td>
</tr>
<tr>
<td>Power3</td>
<td>375</td>
<td>1500</td>
<td>(35, 139)</td>
<td>128</td>
<td>8.8</td>
</tr>
<tr>
<td>Power4</td>
<td>1300</td>
<td>5200</td>
<td>(40, 10000)</td>
<td>128</td>
<td>15.0</td>
</tr>
<tr>
<td>Itanium1</td>
<td>800</td>
<td>3200</td>
<td>(36, 85)</td>
<td>32</td>
<td>36.0</td>
</tr>
<tr>
<td>Itanium2</td>
<td>900</td>
<td>3600</td>
<td>(11, 60)</td>
<td>64</td>
<td>5.5</td>
</tr>
</tbody>
</table>

Itanium1
- \( 800 \) MHz
- \( 3200 \) Mflop/s
- Mem Lat (Min, Max) (cycles)
- Linesize (Bytes)
- \( t_m/f = 36.0 \)

Itanium2
- \( 900 \) MHz
- \( 3600 \) Mflop/s
- Mem Lat (Min, Max) (cycles)
- Linesize (Bytes)
- \( t_m/f = 5.5 \)

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

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 = \( \frac{2}{(2/t_f + t_m)} \)
Naïve Matrix Multiply

\[
\text{implement } C = C + A \cdot B
\]

\[
\text{for } i = 1 \text{ to } n
\]
\[
\text{for } j = 1 \text{ to } n
\]
\[
\text{for } k = 1 \text{ to } n
\]
\[
C_{ij} = C_{ij} + A_{ik} \cdot B_{kj}
\]

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

\(q\) potentially as large as \(2n^3 / 3n^2 = O(n)\)
Naïve Matrix Multiply on RS/6000

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

Size 2000 took 5 days
12000 would take 1095 years

Blocked (Tiled) Matrix Multiply

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
for j = 1 to N
{read block C(i,j) into fast memory}
for k = 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}

So computational intensity q = f / m = 2n^3 / (2N + 2) * n^2
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 \( q \approx b \)
- 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 \( M_{\text{fast}} \)
  \[
  3b^2 = M_{\text{fast}}, \quad \text{so} \quad q = b \leq (M_{\text{fast}}/3)^{1/2}
  \]
- To build a machine to run matrix multiply at 1/2 peak arithmetic speed of the machine, we need a fast memory of size
  \[
  M_{\text{fast}} = 3b^2 = 3q^2 = 3(t_m/t_f)^2
  \]
- 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></th>
<th>( t_m/t_f )</th>
<th>required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra 2i</td>
<td>24.8</td>
<td>14.8</td>
</tr>
<tr>
<td>Ultra 3</td>
<td>14</td>
<td>4.7</td>
</tr>
<tr>
<td>Pentium 3</td>
<td>8.25</td>
<td>0.9</td>
</tr>
<tr>
<td>Pentium3M</td>
<td>10</td>
<td>2.4</td>
</tr>
<tr>
<td>Power3</td>
<td>8.75</td>
<td>1.8</td>
</tr>
<tr>
<td>Power4</td>
<td>15</td>
<td>5.4</td>
</tr>
<tr>
<td>Itanium1</td>
<td>36</td>
<td>31.1</td>
</tr>
<tr>
<td>Itanium2</td>
<td>5.5</td>
<td>0.7</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:
  \[
  q = b \leq (M_{\text{fast}}/3)^{1/2}
  \]
- 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 = \( \Omega(n^3/(M_{\text{fast}})^{1/2}) \)

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 …) = \( \Omega(n^3/M_{\text{fast}}^{1/2}) \)
- 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 = \( \Omega(n^3/M_{\text{fast}}^{1/2}) \)
- Lower bounds extend to anything “similar enough” to 3 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)

- Hardware
  - Even simple programs have complicated behaviors
  - “Small” changes make execution time vary by orders of magnitude
- 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 \( O(n^2) \) sequential matmul, must move \( O(n^3/M_{\text{fast}}) \) 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=αx+y), etc
    - m=2n, f=2n², q = f/m = computational intensity ~1 or less
  - BLAS2 (mid 1980s)
    - matrix-vector operations: matrix vector multiply, etc
    - m=n², f=2n², q=2, less overhead
    - somewhat faster than BLAS1
  - BLAS3 (late 1980s)
    - matrix-matrix operations: matrix matrix multiply, etc
    - m => 3n², f=O(n³), so q=f/m 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

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 inner most loop
    - 1 level of memory ⇒ 3 nested loops (naïve algorithm)
    - 2 levels of memory ⇒ 6 nested loops
    - 3 levels of memory ⇒ 9 nested loops …
- Cache Oblivious Algorithms offer an alternative
  - Treat non 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)

\[
\begin{align*}
\text{func } C &= \text{RMM} (A, B, n) \\
\text{if } n=1, C &= A \cdot B, \text{ else } \\
C &= \begin{cases} 
C_{11} = RMM (A_{11}, B_{11}, n/2) + RMM (A_{12}, B_{21}, n/2) \\
C_{12} = RMM (A_{11}, B_{12}, n/2) + RMM (A_{12}, B_{22}, n/2) \\
C_{21} = RMM (A_{21}, B_{11}, n/2) + RMM (A_{22}, B_{21}, n/2) \\
C_{22} = RMM (A_{21}, B_{12}, n/2) + RMM (A_{22}, B_{22}, n/2) 
\end{cases}
\end{align*}
\]

- True when each \(A_i\) etc 1x1 or \(n/2 \times n/2\)
- For simplicity: square matrices with \(n = 2^m\)
  - Extends to general rectangular case

\[
\begin{align*}
C &= A \cdot B \\
C &= A_{11}B_{11} + A_{12}B_{21} + A_{21}B_{12} + A_{22}B_{22} \\
C &= \begin{cases} 
C_{11} = RMM (A_{11}, B_{11}, n/2) + RMM (A_{12}, B_{21}, n/2) \\
C_{12} = RMM (A_{11}, B_{12}, n/2) + RMM (A_{12}, B_{22}, n/2) \\
C_{21} = RMM (A_{21}, B_{11}, n/2) + RMM (A_{22}, B_{21}, n/2) \\
C_{22} = RMM (A_{21}, B_{12}, n/2) + RMM (A_{22}, B_{22}, n/2) 
\end{cases}
\end{align*}
\]

Recursion: Cache Oblivious Algorithms

- The tiled algorithm requires finding a good block size
- Cache Oblivious Algorithms offer an alternative
  - Treat \(m \times n\) 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

\[
\begin{align*}
\begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} B &= \begin{pmatrix} A_1B & A_2B \\ A_3B & A_4B \end{pmatrix} \\
&= \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} \\ A_{31}B_{11} + A_{32}B_{21} \end{pmatrix} + \begin{pmatrix} A_{11}B_{12} + A_{12}B_{22} \\ A_{31}B_{12} + A_{32}B_{22} \end{pmatrix}
\end{align*}
\]

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)
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"

![Recursive Data Layouts](image)

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

\[
\begin{pmatrix}
\text{Let } M &= m_{11} m_{12} m_{21} m_{22} = a_{11} a_{12} b_{11} b_{12} \\
\text{Here, } p_1 &= (a_{12} - a_{22}) \cdot (b_{21} + b_{22}) \quad p_5 = a_{11} \cdot (b_{12} - b_{22}) \\
\text{p2} &= (a_{11} + a_{22}) \cdot (b_{11} + b_{22}) \quad p6 = a_{22} \cdot (b_{21} - b_{11}) \\
p3 &= (a_{11} - a_{21}) \cdot (b_{11} + b_{12}) \quad p7 = (a_{21} + a_{22}) \cdot b_{11} \\
p4 &= (a_{11} + a_{12}) \cdot b_{22}\end{pmatrix}
\]

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

Extends to nxn by divide & conquer

Strassen (continued)

\[T(n) = \text{Cost of multiplying nxn matrices} = 7T(n/2) + 18*(n/2)^2 = O(n \log_2 7) = O(n^{2.81})\]

• 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
    \[
    \Omega\left(n^{\log_2 7} / M^{\log_2 7/2 - 1}\right) \approx \Omega(n^{2.81} / M^{0.4})
    \]
  
  (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 $O(n^{2.376...})$
  • Coppersmith & Winograd, 1987

• New Record! 2.376 reduced to 2.373
  • Virginia Vassilevska Williams, UC Berkeley & Stanford, 2011

• Lower bound on #words moved can be extended to (some) of these algorithms
• Possibility of $O(n^{2+\varepsilon})$ 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 stable
  • 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
  - Field started 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.

Source: Jack Dongarra
## 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
  
  ```c
  a[i] = b[i] + c;  // false read-after-write hazard
  a[i+1] = b[i+1] * d;
  ```

  
  ```c
  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

  ```c
  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++;
  }
  ```

## Loop Unrolling

- Expose instruction-level parallelism

  ```c
  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

• Hide instruction latency
  • Use local variables to expose independent operations that can execute in parallel or in a pipelined fashion
  • Balance the instruction mix (what functional units are available?)

\[
\begin{align*}
  f_1 &= f_5 \times f_9; \\
  f_2 &= f_6 + f_{10}; \\
  f_3 &= f_7 \times f_{11}; \\
  f_4 &= f_8 + f_{12};
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Original matrix (numbers are addresses)</th>
<th>Reorganized into 2x2 blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 4 8 12</td>
<td>0 2 8 10</td>
</tr>
<tr>
<td>1 5 9 13</td>
<td>1 3 9 11</td>
</tr>
<tr>
<td>2 6 10 14</td>
<td>4 6 12 13</td>
</tr>
<tr>
<td>3 7 11 15</td>
<td>5 7 14 15</td>
</tr>
</tbody>
</table>

Locality in Other Algorithms

• The performance of any algorithm is limited by \( q \)
  • \( q = \text{"computational intensity"} = \text{#arithmetic ops}/\text{#words moved} \)
  • 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 31
- Could even be your intended class project
- Please fill in on-line class survey
  - We need this to assign teams for Homework 1

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)
  - Tuning Strassen's Matrix Multiplication for Memory Efficiency Mithuna S. Thottethodi, Siddhartha Chatterjee, and Alvin R. Lebeck in Proceedings of Supercomputing '98, November 1998 postscript
  - Many related papers at bebop.cs.berkeley.edu