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
  - Hardware executes instructions in order specified by compiler
- Idealized Cost
  - Each operation has roughly the same cost
    (read, write, add, multiply, etc.)

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

Memory Hierarchy Diagram

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) 1 experiment
  - 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 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)
- Mem: 396 ns (132 cycles)
- 8 K pages, 32 TLB entries
- Array length
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, 5-2 cycles
- Mem: 396 ns (132 cycles)
array size

Memory Hierarchy on an Intel Core 2 Duo

- Array length

Stanza Triad

- Even smaller benchmark for prefetching
- Derived from STREAM Triad
- Stanza (L) is the length of a unit stride run
while i < arraylength
for each L element stanza
A[i] = scalar * X[i] + Y[i]
skip k 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

<table>
<thead>
<tr>
<th>6 PM</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>30</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>40</td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

- 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

- Instruction Fetch
- Instruction Decode
- Execute/Address Calculation
- Memory Access
- Write Back

- 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: \( x = y + c * z \)
  - 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 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
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

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

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)

```
0 5 10 15
1 6 11 16
2 7 12 17
3 8 13 16
4 9 14 19
```

Row major

```
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
16 17 18 19
```

- Column major matrix in memory

```
0 1 2 3
4 5 6 7
```

- Row major matrix in memory

```
0 1 2 3
4 5 6 7
```

- Blue row of matrix is stored in red cachelines

**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

*Computationally Intensive: Key to algorithm efficiency*

\[ q = \frac{f}{m} \text{ average number of flops per slow memory access} \]

- Minimum possible time = \( f \times t_f \) when all data in fast memory
- Actual time
  - \( f \times t_f + m \times t_m = f \times t_f + \left( 1 + \frac{t_m}{t_f} \times \frac{1}{q} \right) \)

*Machine Balance: Key to machine efficiency*

- Larger \( q \) means time closer to minimum \( f \times t_f \)
  - \( q = \frac{t_m}{t_f} \) needed to get at least half of peak speed

**Warm up: Matrix-vector multiplication**

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

for \( i = 1:n \)
  for \( j = 1:n \)
    \( y(i) = y(i) + A(i,j)*x(j) \)

```
y(1) = y(1) + A(1,1)*x(1)
y(2) = y(2) + A(2,2)*x(2)
```

```
y(1) = y(1) + A(1,1)*x(1)
y(2) = y(2) + A(2,2)*x(2)
```
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 = number of slow memory refs = 3n + n^2
- f = number of arithmetic operations = 2n^2
- q = f / m = 2

Matrix-vector multiplication limited by slow memory speed

Modeling Matrix-Vector Multiplication

- Compute time for nxn = 1000x1000 matrix
- Time
  - f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q)
  - 2n^2 * t_f * (1 + t_m/t_f * 1/2)
- For t_f and t_m, 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) cycles</th>
<th>Linesize Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra 2</td>
<td>333</td>
<td>667</td>
<td>38</td>
<td>66</td>
</tr>
<tr>
<td>Ultra 3</td>
<td>900</td>
<td>1800</td>
<td>28</td>
<td>200</td>
</tr>
<tr>
<td>Pentium 3</td>
<td>500</td>
<td>600</td>
<td>25</td>
<td>60</td>
</tr>
<tr>
<td>Pentium3M</td>
<td>800</td>
<td>800</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>Power3</td>
<td>375</td>
<td>1500</td>
<td>35</td>
<td>139</td>
</tr>
<tr>
<td>Power4</td>
<td>1300</td>
<td>5200</td>
<td>60</td>
<td>10000</td>
</tr>
<tr>
<td>Itanium1</td>
<td>800</td>
<td>3200</td>
<td>36</td>
<td>85</td>
</tr>
<tr>
<td>Itanium2</td>
<td>900</td>
<td>3600</td>
<td>11</td>
<td>60</td>
</tr>
</tbody>
</table>

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

**(implements \( C = C + A \cdot B \))**

for \( i = 1 \) to \( n \)
  - for \( j = 1 \) to \( n \)
    - for \( k = 1 \) to \( n \)
      \[ C(i,j) = C(i,j) + A(i,k) \cdot B(k,j) \]

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

\( q \) potentially as large as \( 2n^3 / 3n^2 = O(n) \)

### 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

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

Slide source: Larry Carter, UCSD

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
f is number of floating point operations, 2n^3 for this problem
m = amount memory traffic between slow and fast memory

m = N*n^2 * N^3 * (b^2) * (n/N)^3 * n^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 \( q \approx b \)
- 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 \( M_{\text{fast}} \)

\[
3b^2 \leq M_{\text{fast}}, \quad \text{so} \quad q = b \leq (M_{\text{fast}}/3)^{1/2}
\]

- 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>Itanium 1</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)

\[
\text{Theorem (Hong & Kung, 1981): Any reorganization of this algorithm (that uses only associativity) is limited to } q = \Theta((M_{\text{fast}})^{1/2})
\]

- \#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 ...) = \( \Omega(n^3 / M_{\text{fast}}^{1/2}) \)
  - 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 \( O(n^2/M_{\text{fast}}^{1/2}) \) words

- Software tools
  - How do I implement my applications and algorithms in most efficient and productive way?

- Applications
  - How to decompose into well-understood algorithms (and their implementations)

Layers

01/22/2015 CS267 - Lecture 2
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
    - \(m=2^n\), \(f=2^n\), \(q = f/m\) = computational intensity \(-1\) or less
  - BLAS2 (mid 1980s)
    - matrix-vector operations: matrix vector multiply, etc
    - \(m=n^2\), \(f=2n^2\), \(q\approx 2\), less overhead
    - somewhat faster than BLAS1
  - BLAS3 (late 1980s)
    - matrix-matrix operations: matrix multiply, etc
    - \(m \approx 3n^2\), \(f\propto n^3\), so \(q\) could 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

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 inner most loop
    - 1 level of memory \(\Rightarrow 3\) nested loops (naive algorithm)
    - 2 levels of memory \(\Rightarrow 6\) nested loops
    - 3 levels of memory \(\Rightarrow 9\) 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)

- C = \( C_{11} \), \( C_{12} \) = \( A \cdot B = A_{i,1} \cdot B_{11} + A_{i,2} \cdot B_{12} \)
- \( C_{21} \), \( C_{22} \) = \( A_{i,1} \cdot B_{11} + A_{i,2} \cdot B_{12} \)
- True when each \( A_{i,j} \) etc 1x1 or n/2 x n/2
- For simplicity: square matrices with n = 2^m
  - Extends to general rectangular case

```c
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 (RMM) (2/2)

```c
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
```

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"

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

Let M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}

Let p1 = (a_{12} - a_{22}) \times (b_{21} + b_{22})

p2 = (a_{11} + a_{22}) \times (b_{11} + b_{22})

p3 = (a_{11} - a_{21}) \times (b_{11} + b_{12})

p4 = (a_{11} + a_{12}) \times b_{22}

Then:
  m_{11} = p1 + p2 - p4 + p6
  m_{12} = p4 + p5
  m_{21} = p6 + p7
  m_{22} = p2 - p3 + p5 - p7

Extends to nxn by divide & conquer

Other Fast Matrix Multiplication Algorithms

• World’s record was O(n^{2.37548...})
  • 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
  • Francois Le Gall, 2014

• Lower bound on #words moved can be extended to (some) of these algorithms
• Possibility of O(n^{2+\epsilon}) algorithm!
  • Cohr, 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.

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
  
  ```
  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
  ```
  float f0 = filter[0];
  float f1 = filter[1];
  float f2 = filter[2];
  ```

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

  ```
  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?)

  ```
  \[ f_1 = f_5 \times f_9; \\
  f_2 = f_6 + f_{10}; \\
  f_3 = f_7 \times f_{11}; \\
  f_4 = f_8 + f_{12}; \]
  ```

**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 \( q \)
  - \( q = \text{"computational intensity"} = \#\text{arithmetic_ops} / \#\text{words\_moved} \)
- 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
  - 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