Single Processor Optimizations
Matrix Multiplication Case Study

Kathy Yelick
yelick@cs.berkeley.edu
www.cs.berkeley.edu/~yelick/cs267_spr07
Outline

• Recap from Lecture 2
  • Memory hierarchy is important to performance
  • Use of simple performance models to understand performance

• Case Study: Matrix Multiplication
  • Blocking algorithms
  • Other tuning techniques
  • Alternate algorithms

• Automatic Performance Tuning
Outline

• Recap from Lecture 2
  • Memory hierarchy is important to performance
  • Use of simple performance models to understand performance

• Case Study: Matrix Multiplication
  • Blocking algorithms
  • Other tuning techniques
  • Alternate algorithms

• Automatic Performance Tuning
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
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 = \( f \times t_f \) when all data in fast memory
- Actual time
  - \( f \times t_f + m \times t_m = f \times t_f \times (1 + t_m/t_f \times 1/q) \)
- Larger \( q \) means time closer to minimum \( f \times t_f \)
  - \( q \geq t_m/t_f \) needed to get at least half of peak speed
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}

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

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

- Compute time for nxn = 1000x1000 matrix
- Time
  - \( f \times t_f + m \times t_m = f \times t_f \times (1 + \frac{t_m}{t_f} \times \frac{1}{q}) \)
  - \( = 2 \times n^2 \times t_f \times (1 + \frac{t_m}{t_f} \times \frac{1}{2}) \)
- 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>Clock</th>
<th>Peak MHz</th>
<th>Peak Mflop/s</th>
<th>Mem Lat (Min,Max)</th>
<th>Linesize Bytes</th>
<th>( t_m/t_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>500</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>60, 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>

**machine balance**
(q must be at least this for \( \frac{1}{2} \) peak speed)
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)
  • Assumed memory latency is constant
• Could simplify even further by ignoring memory operations in X and Y vectors
  • Mflop rate = flops/sec = \( \frac{2}{2* t_f + t_m} \)
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
Outline

• Recap from Lecture 2
  • Memory hierarchy is important to performance
  • Use of simple performance models to understand performance

• Case Study: Matrix Multiplication
  • Blocking algorithms
  • Other tuning techniques
  • Alternate algorithms

• Automatic Performance Tuning
Phillip Colella’s “Seven dwarfs”

High-end simulation in the physical sciences = 7 numerical methods:

1. Structured Grids (including adaptive block structured)
2. Unstructured Grids
3. Spectral Methods
4. **Dense Linear Algebra**
5. Sparse Linear Algebra
6. Particle Methods
7. Monte Carlo

- A dwarf is a pattern of computation and communication
- Dwarfs are targets from algorithmic, software, and architecture standpoints

Dense linear algebra arises in some of the largest computations
- It achieves high machine efficiency
- There are important subcategories as we will see

*Slide from “Defining Software Requirements for Scientific Computing”, Phillip Colella, 2004 and Dave Patterson, 2006*
Naïve Matrix Multiply

\{\text{implements } C = C + A \cdot B\}

\text{for } i = 1 \text{ to } n \\
\hspace{1cm} \text{for } j = 1 \text{ to } n \\
\hspace{2cm} \text{for } k = 1 \text{ to } n \\
\hspace{3cm} C(i,j) = C(i,j) + A(i,k) \cdot B(k,j)

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

$q$ potentially as large as $2\cdot n^3 / 3\cdot n^2 = O(n)$
Naïve Matrix Multiply

{implements $C = C + A*B$}
for $i = 1$ to $n$
   {read row $i$ of $A$ into fast memory}
   for $j = 1$ to $n$
      {read $C(i,j)$ into fast memory}
      {read column $j$ of $B$ into fast memory}
      for $k = 1$ to $n$
         $C(i,j) = C(i,j) + A(i,k) * B(k,j)$
      {write $C(i,j)$ back to slow memory}

Naïve Matrix Multiply

Number of slow memory references on unblocked matrix multiply

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

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

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

\[ = n^3 + 3n^2 \]

So \( q = f / m = 2n^3 / (n^3 + 3n^2) \)

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

\[
\begin{array}{c}
\text{C}(i,j) \\
\text{red}
\end{array} = \begin{array}{c}
\text{C}(i,j) \\
\text{red}
\end{array} + \begin{array}{c}
\text{A}(i,:)
\end{array} \ast \begin{array}{c}
\text{B}(:,j)
\end{array}
\]

\[ 1/24/2007 \] CS267 Lecture 3
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

\[ T = N^{4.7} \]

- \( O(N^3) \) 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
Naïve Matrix Multiply on RS/6000

-log cycles/flop vs log Problem Size

- Page miss every iteration
- TLB miss every iteration
- Cache miss every 16 iterations
- Page miss every 512 iterations
Consider $A, B, C$ to be $n$-by-$n$ matrix viewed as $N$-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}
      $C(i,j) = C(i,j) + A(i,k) \times B(k,j)$ {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 $n \times n$ 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:
- $m = Nn^2$ read each block of $B$ $N^3$ times ($N^3 \cdot b^2 = N^3 \cdot (n/N)^2 = Nn^2$)
- $+ Nn^2$ read each block of $A$ $N^3$ times
- $+ 2n^2$ read and write each block of $C$ once
- $= (2N + 2) \cdot n^2$

So computational intensity $q = f / m = 2n^3 / ((2N + 2) \cdot n^2)$
- $\sim n / N = b$ 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 $M_{\text{fast}}$
  \[ 3b^2 \leq M_{\text{fast}}, \text{ so } q \approx b \leq \sqrt{M_{\text{fast}}/3} \]

- To build a machine to run matrix multiply at 1/2 peak arithmetic speed of the machine, we need a fast memory of size
  \[ M_{\text{fast}} \geq 3b^2 \approx 3q^2 = 3\left(\frac{t_m}{t_f}\right)^2 \]
- 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>Processor</th>
<th>$t_m/t_f$</th>
<th>KB</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>6.25</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 associativity
  • Get slightly different answers from naïve code, because of roundoff - OK
• The previous analysis showed that the blocked algorithm has computational intensity:
  \[ q \approx b \leq \sqrt{\frac{M_{\text{fast}}}{3}} \]

• There is a lower bound result that says we cannot do any better than this (using only associativity)

• Theorem (Hong & Kung, 1981): Any reorganization of this algorithm (that uses only associativity) is limited to $q = O(\sqrt{M_{\text{fast}}})$

• What if more levels of memory hierarchy?
  • Apply blocking recursively, once per level
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)
  • Case1: 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

\[
\begin{pmatrix}
  A_1 \\
  A_2 \\
\end{pmatrix}
\begin{pmatrix}
  B_1 \\
  B_2 \\
\end{pmatrix}
= \begin{pmatrix}
  A_1 B \\
  A_2 B \\
\end{pmatrix}
\]

Case 1

\[
(A_1, A_2)
\begin{pmatrix}
  B_1 \\
  B_2 \\
\end{pmatrix}
= (A_1 B + A_2 B)
\]

Case 2

\[
A(B_1, B_2)
= (A B_1, A B_2)
\]

Case 3
Experience

• In practice, need to cut off recursion
• Implementing a high-performance Cache-Oblivious code is not easy
  • Careful attention to micro-kernel and mini-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.
• 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 CO codes

Unpublished work, presented at LACSI 2006
Recursive Data Layouts

- Blocking seems to require knowing cache sizes – portable?
- A related idea is to use a recursive structure for the matrix
- 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 $M = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$

Let $p_1 = (a_{12} - a_{22}) \times (b_{21} + b_{22})$  
$p_2 = (a_{11} + a_{22}) \times (b_{11} + b_{22})$  
$p_3 = (a_{11} - a_{21}) \times (b_{11} + b_{12})$  
$p_4 = (a_{11} + a_{12}) \times b_{22}$  
$p_5 = a_{11} \times (b_{12} - b_{22})$  
$p_6 = a_{22} \times (b_{21} - b_{11})$  
$p_7 = (a_{21} + a_{22}) \times b_{11}$

Then $m_{11} = p_1 + p_2 - p_4 + p_6$  
$m_{12} = p_4 + p_5$  
$m_{21} = p_6 + p_7$  
$m_{22} = p_2 - p_3 + p_5 - p_7$

Extends to nxn by divide&conquer
Strassen (continued)

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

- Asymptotically faster
  - Several times faster for large \( n \) in practice
  - Cross-over depends on machine
  - Available in several libraries
  - “Tuning Strassen's Matrix Multiplication for Memory Efficiency”, M. S. Thottethodi, S. Chatterjee, and A. Lebeck, in Proceedings of Supercomputing '98

- Caveats
  - Needs more memory than standard algorithm
  - Can be less accurate because of roundoff error
Other Fast Matrix Multiplication Algorithms

• Current world’s record is $O(n^{2.376...})$ (Coppersmith & Winograd)
• Why does Hong/Kung theorem not apply?
• Possibility of $O(n^{2+\varepsilon})$ algorithm! (Cohn, Umans, Kleinberg, 2003)
• Fast methods (besides Strassen) may need unrealistically large $n$
Short Break

Questions about course?
Homework 1 coming soon (Friday)
Outline

• Recap from Lecture 2
  • Memory hierarchy is important to performance
  • Use of simple performance models to understand performance

• Case Study: Matrix Multiplication
  • Blocking algorithms
  • Other tuning techniques
  • Alternate algorithms

• Automatic Performance Tuning
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

• Some systems use search over “design space” of possible implementations
  • Atlas – incorporated into Matlab
  • BeBOP – http://bebop.cs.berkeley.edu/
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-Ill, 333 MHz, 667 Mflop/s peak, Sun cc v5.0 compiler)
Tiling Alone Might Not Be Enough

- Naïve and a “naïvely tiled” code on Itanium 2
  - Searched all block sizes to find best, $b=56$
  - Starting point for next homework
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
• Hard to do by hand (but you’ll try)
• Automatic optimization an active research area
  • BeBOP: bebop.cs.berkeley.edu/
  • 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

\[
\begin{align*}
a[i] &= b[i] + c; \\
a[i+1] &= b[i+1] \times d; \\
\end{align*}
\]
false read-after-write hazard between \(a[i]\) and \(b[i+1]\)

\[
\downarrow
\]

\[
\begin{align*}
float f1 &= b[i]; \\
float f2 &= b[i+1]; \\
a[i] &= f1 + c; \\
a[i+1] &= f2 \times d; \\
\end{align*}
\]

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++;
}
```

```c
float f0 = filter[0]; also: register float f0 = ...;
float f1 = filter[1];
float f2 = filter[2];
while( ... ) {
    *res++ = f0*signal[0]
    + f1*signal[1]
    + f2*signal[2];
    signal++;
}
```

Example is a convolution
Minimize Pointer Updates

• Replace pointer updates for strided memory addressing with constant array offsets

\[
\begin{align*}
f_0 &= \ast r_8; \quad r_8 += 4; \\
f_1 &= \ast r_8; \quad r_8 += 4; \\
f_2 &= \ast r_8; \quad r_8 += 4; \\
\vdots \\
f_0 &= r_8[0]; \\
f_1 &= r_8[4]; \\
f_2 &= r_8[8]; \\
r_8 &= r_8 += 12;
\end{align*}
\]

Pointer vs. array expression costs may differ.

• Some compilers do a better job at analyzing one than the other
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?)

\[
\begin{align*}
  f1 &= f5 \times f9; \\
  f2 &= f6 + f10; \\
  f3 &= f7 \times f11; \\
  f4 &= f8 + f12;
\end{align*}
\]
**Copy optimization**

- Copy input operands or blocks
  - Reduce cache conflicts
  - Constant array offsets for fixed size blocks
  - Expose page-level locality

<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>
ATLAS Matrix Multiply (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
Experiments on Search vs. Modeling

Study compares search (Atlas to optimization selection based on performance models)

• Ten modern architectures
• Model did well on most cases
  • Better on UltraSparc
  • Worse on Itanium
• Eliminating performance gaps: think globally, search locally
  - small performance gaps: local search
  - large performance gaps: refine model
• Substantial gap between ATLAS CGw/S and ATLAS Unleashed on some machines

Source: K. Pingali. Results from IEEE ’05 paper by K Yotov, X Li, G Ren, M Garzarán, D Padua, K Pingali, P Stodghill.
Locality in Other Algorithms

• The performance of any algorithm is limited by $q$
  • $q = \#\text{flops} / \#\text{memory refs} = \text{“Computational Intensity”}$

• In matrix multiply, we increase $q$ by changing computation order
  • Reuse data in cache (increased temporal locality)

• For other algorithms and data structures, tuning still an open problem
  • sparse matrices (blocking, reordering, splitting)
    • Weekly research meetings
    • Bebop.cs.berkeley.edu
    • OSKI – tuning sequential sparse-matrix-vector multiply and related operations
  • trees (B-Trees are for the disk level of the hierarchy)
  • linked lists (some work done here)
Dense Linear Algebra is not All Matrix Multiply

- Main dense matrix kernels are in an industry standard interface called the **BLAS**: Basic Linear Algebra Subroutines
  - [www.netlib.org/blas](http://www.netlib.org/blas), [www.netlib.org/blas/blast-forum](http://www.netlib.org/blas/blast-forum)
  - Vendors, others supply optimized implementations

- **History**
  - **BLAS1 (1970s):**
    - vector operations: dot product, saxpy ($y=\alpha x+y$), etc
    - $m=2n$, $f=2n$, $q \sim 1$ or less
  - **BLAS2 (mid 1980s):**
    - matrix-vector operations: matrix vector multiply, etc
    - $m=n^2$, $f=2n^2$, $q \sim 2$, less overhead
    - somewhat faster than BLAS1
  - **BLAS3 (late 1980s):**
    - matrix-matrix operations: matrix matrix multiply, etc
    - $m \leq 3n^2$, $f=O(n^3)$, so $q=f/m$ can possibly be as large as $n$, so BLAS3 is potentially much faster than BLAS2

- **Good algorithms used BLAS3 when possible (LAPACK & ScaLAPACK)**
  - See [www.netlib.org/lapack](http://www.netlib.org/lapack), [www.netlib.org/scalapack](http://www.netlib.org/scalapack)
  - Not all algorithms *can* use BLAS3
BLAS speeds on an IBM RS6000/590

Peak speed = 266 Mflops

RS2: Level 1, 2 and 3 BLAS

BLAS 3 (n-by-n matrix matrix multiply) vs BLAS 2 (n-by-n matrix vector multiply) vs BLAS 1 (saxpy of n vectors)
Dense Linear Algebra: BLAS2 vs. BLAS3

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

**BLAS3 (MatrixMatrix) vs. BLAS2 (MatrixVector)**

Data source: Jack Dongarra
Other Automatic Tuning Efforts

- FFTW (MIT): “Fastest Fourier Transform in the West”
  - Sequential (and parallel)
  - Many variants (real/complex, sine/cosine, multidimensional)
  - 1999 Wilkinson Prize
  - www.fftw.org
- Spiral (CMU)
  - Digital signal processing transforms
  - FFT and beyond
  - www.spiral.net
- BEBOP (UCB)
  - http://bebop.cs.berkeley.edu
  - OSKI - Sparse matrix kernels
    - Stencils – Structure grid kernels (with LBNL)
  - Interprocessor communication kernels
    - Bebop (UPC), Dongarra (UTK for MPI)
- Class projects available
Summary

- Performance programming on uniprocessors requires
  - understanding of memory system
  - understanding of fine-grained parallelism in processor

- Simple performance models can aid in understanding
  - Two ratios are key to efficiency (relative to peak)
    1. computational intensity of the algorithm:
      - \( q = \frac{f}{m} = \# \text{ floating point operations} / \# \text{ slow memory references} \)
    2. machine balance in the memory system:
      - \( \frac{t_m}{t_f} = \text{time for slow memory reference} / \text{time for floating point operation} \)

- Blocking (tiling) is a basic approach to increase \( q \)
  - Techniques apply generally, but the details (e.g., block size) are architecture dependent
  - Similar techniques are possible on other data structures and algorithms

- Now it’s your turn: Homework 1
  - Work in teams of 2 or 3 (assigned this time)
Reading for Today

• “Parallel Computing Sourcebook” Chapters 2 & 3
• 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
• Recursive Array Layouts and Fast Parallel Matrix Multiplication” by Chatterjee et al. IEEE TPDS November 2002.
Questions You Should Be Able to Answer

1. What is the key to understand algorithm efficiency in our simple memory model?
2. What is the key to understand machine efficiency in our simple memory model?
3. What is tiling?
4. Why does block matrix multiply reduce the number of memory references?
5. What are the BLAS?
6. Why does loop unrolling improve uniprocessor performance?