Uniprocessor Optimizations and Matrix Multiplication

BeBOP Summer 2002
http://www.cs.berkeley.edu/~richie/bebop
Applications ...

• Scientific simulation and modeling
  • Weather and earthquakes
  • Cars and buildings
  • The universe

• Signal processing
  • Audio and image compression
  • Machine vision
  • Speech recognition

• Information retrieval
  • Web searching
  • Human genome

• Computer graphics and computational geometry
  • Structural models
  • Films: Final Fantasy, Shrek
... and their Building Blocks (Kernels)

- Scientific simulation and modeling
  - Matrix-vector/matrix-matrix multiply
  - Solving linear systems

- Signal processing
  - Performing fast transforms: Fourier, trigonometric, wavelet
  - Filtering
  - Linear algebra on structured matrices

- Information retrieval
  - Sorting
  - Finding eigenvalues and eigenvectors

- Computer graphics and computational geometry
  - Matrix multiply
  - Computing matrix determinants
Outline

• Parallelism in Modern Processors
• Memory Hierarchies
• Matrix Multiply Cache Optimizations
• Bag of Tricks
Modern Processors: Theory & Practice

• Idealized Uniprocessor Model
  • Execution order specified by program
  • Operations (load/store, +/*, branch) have roughly the same cost

• Processors in the Real World
  • Registers and caches
    • Small amounts of fast memory
    • Memory ops have widely varying costs
  • Exploit Instruction-Level Parallelism (ILP)
    • Superscalar — multiple functional units
    • Pipelined — decompose units of execution into parallel stages
    • Different instruction mixes/orders have different costs

• Why is this your problem?
  • In theory, compilers understand all this mumbo-jumbo and optimize your programs; in practice, they don’t.
What is Pipelining?

Dave Patterson’s Laundry example: 4 people doing laundry

wash (30 min) + dry (40 min) + fold (20 min)

- In this example:
  - Sequential execution takes $4 \times 90 \text{min} = 6 \text{ hours}$
  - Pipelined execution takes $30 + 4 \times 40 + 20 = 3.3 \text{ hours}$

- Pipelining helps throughput, but not latency
- Pipeline rate limited by slowest pipeline stage
- Potential speedup = Number pipe stages
- Time to “fill” pipeline and time to “drain” it reduces speedup
Limits of ILP

Hazards prevent next instruction from executing in its designated clock cycle

- **Structural**: single person to fold and put clothes away
- **Data**: missing socks
- **Control**: dyed clothes need to be rewashed
- Compiler will try to reduce these, but careful coding helps!
Outline

• Parallelism in Modern Processors
• Memory Hierarchies
• Matrix Multiply Cache Optimizations
• Bag of Tricks
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

![Memory Hierarchy Diagram]

<table>
<thead>
<tr>
<th>Level</th>
<th>Speed (ns)</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tertiary storage</td>
<td>10 sec</td>
<td>Ps</td>
</tr>
<tr>
<td>Secondary storage</td>
<td>10 ms</td>
<td>Ts</td>
</tr>
<tr>
<td>Main memory (DRAM)</td>
<td>100</td>
<td>Gs</td>
</tr>
<tr>
<td>Second level cache (SRAM)</td>
<td>10</td>
<td>Ms</td>
</tr>
<tr>
<td>On-chip cache</td>
<td>1</td>
<td>100,1Ks</td>
</tr>
<tr>
<td>Processor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

control

datapath

registers

on-chip cache
Processor-DRAM Gap (latency)

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

"Moore’s Law"

Processor-Memory Performance Gap:
(grows 50% / year)

µProc 60%/yr.

DRAM 7%/yr.
Cache Basics

- **Cache hit**: in-cache memory access—cheap
- **Cache miss**: non-cached memory access—expensive
- Consider a tiny cache (for illustration only)

\[
\begin{array}{ll}
X000 & X001 \\
X010 & X011 \\
X100 & X101 \\
X110 & X111 \\
\end{array}
\]

- **Cache line length**: # of bytes loaded together in one entry
- **Associativity**
  - direct-mapped: only one address (line) in a given range in cache
  - \( n \)-way: 2 or more lines with different addresses exist
Experimental Study of Memory

- Microbenchmark for memory system performance (Saavedra '92)

- time the following program for each size(A) and stride s
  (repeat to obtain confidence and mitigate timer resolution)
  for array A of size from 4KB to 8MB by 2x
    for stride s from 8 Bytes (1 word) to size(A)/2 by 2x
      for i from 0 to size by s
        load A[i] from memory (8 Bytes)
Memory Hierarchy on a Sun Ultra-IIi

Sun Ultra-IIi, 333 MHz

Array size

Time (msec)

450
400
350
300
250
200
150
100
50
0

4 16 64 256 1K 4K 16K 64K 256K 1M 2M 4M 8M 16M 32M

L1: 16 byte line
L2: 64 byte line
Stride (bytes)
8 K pages

Mem: 396 ns (132 cycles)
L2: 2 MB, 36 ns (12 cycles)
L1: 16K, 6 ns (2 cycle)

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

7/10/2003
CS267 Lecture 2
Memory Hierarchy on a Pentium III

Katmai processor on Millennium, 550 MHz

Array size
- 4KB
- 8KB
- 16KB
- 32KB
- 64KB
- 128KB
- 256KB
- 512KB
- 1MB
- 2MB
- 4MB
- 8MB
- 16MB
- 32MB
- 64MB

Time (nsec)
- L1: 32 byte line?
- L2: 512 KB 60 ns
- L1: 64K 5 ns, 4-way?

Stride (bytes)

7/10/2003 CS267 Lecture 2
Lessons

• True performance can be a complicated function of the architecture
  • Slight changes in architecture or program change performance significantly
  • To write fast programs, need to consider architecture
  • We would like simple models to help us design efficient algorithms
  • Is this possible?

• Next: Example of improving cache performance: blocking or tiling
  • Idea: decompose problem workload into cache-sized pieces
Outline

• Parallelism in Modern Processors
• Memory Hierarchies
• Matrix Multiply Cache Optimizations
• Bag of Tricks
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)
  • by row, or “row major” (C default)

![Matrix Storage Diagram]
Note on “Performance”

• For linear algebra, measure performance as rate of execution:
  • Millions of floating point operations per second (Mflop/s)
  • Higher is better
  • Comparing Mflop/s is not the same as comparing time unless flops are constant!

• Speedup taken wrt time
  • Speedup of A over B = (Running time of B) / (Running time of A)
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 \( \ll t_m \)
  - \( q = f / m \) = average number of flops per slow element access

- Minimum possible time = \( f^* t_f \) when all data in fast memory
- Actual time
  \[ f \cdot t_f + m \cdot t_m = f \cdot t_f \left( 1 + \frac{t_m}{t_f} \cdot \frac{1}{q} \right) \]
- Larger \( q \) means time closer to minimum \( f^* t_f \)
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)
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 = \( 2n^2 \)
• \( q \) = \( f / m \sim 2 \)

• Matrix-vector multiplication limited by slow memory speed
"Naïve" Matrix Multiply

\[
\{\text{implements } C = C + A \times B\}
\]

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

\[
\quad \text{for } j = 1\text{ to } n
\]

\[
\quad \quad \text{for } k = 1\text{ to } n
\]

\[
C(i,j) = C(i,j) + A(i,k) \times B(k,j)
\]
“Naïve” Matrix Multiply

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

\text{for } i = 1 \text{ to } n

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

\text{for } j = 1 \text{ to } n

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

\{\text{read column } j \text{ of } B \text{ into fast memory}\}

\text{for } k = 1 \text{ to } n

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

\{\text{write } C(i,j) \text{ back to slow memory}\}
“Naïve” Matrix Multiply

Number of slow memory references on unblocked matrix multiply

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

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

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

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

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

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

\[
\begin{align*}
C(i,j) & = C(i,j) + A(i,k) \times B(k,j) \\
\end{align*}
\]
Blocked (Tiled) Matrix Multiply

Recall:
- $m$: # of moves from slow to fast memory
- Matrix is $n \times n$, and $N \times N$ blocks each of size $b \times b$
- $f = 2n^3$ for this problem
- $q = f/m$ is algorithmic memory efficiency

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

So $q = f/m = 2n^3 / ((2N + 2) \times n^2)$
- $\sim n/N = b$ for large $n$

So we can improve performance by increasing the block size $b$
Can be much faster than matrix-vector multiply ($q=2$)
Limits to Optimizing Matrix Multiply

Blocked algorithm has ratio $q \approx b$

- Larger block size $\Rightarrow$ faster implementation
- Limit: All three blocks from $A, B, C$ must fit in fast memory (cache):
  \[ 3b^2 \leq M \]
  So: $q \approx b \leq \sqrt{M/3}$

Lower bound:

Theorem (Hong & Kung, 1981): Any reorganization of this algorithm (using only algebraic associativity) is limited to: $q = O(\sqrt{M})$
Basic Linear Algebra Subroutines

- Industry standard interface (evolving)
- Hardware vendors, others supply optimized implementations

History
- BLAS1 (1970s):
  - vector operations: dot product, saxpy \( y = \alpha \cdot x + y \), etc
  - \( m = 2 \cdot n, f = 2 \cdot n, q \approx 1 \) or less
- BLAS2 (mid 1980s)
  - matrix-vector operations: matrix vector multiply, etc
  - \( m = n^2, f = 2 \cdot n^2, q \approx 2 \), less overhead
  - somewhat faster than BLAS1
- BLAS3 (late 1980s)
  - matrix-matrix operations: matrix matrix multiply, etc
  - \( m \geq 4n^2, f = O(n^3) \), so \( q \) can possibly be as large as \( n \), so BLAS3 is potentially much faster than BLAS2

- Good algorithms used BLAS3 when possible (LAPACK)

See [www.netlib.org/blas](http://www.netlib.org/blas), [www.netlib.org/lapack](http://www.netlib.org/lapack)
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)
Locality in Other Algorithms

- The performance of any algorithm is limited by $q$
- 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
  - sparse matrices (reordering, blocking)
  - trees (B-Trees are for the disk level of the hierarchy)
  - linked lists (some work done here)
Outline

• Parallelism in Modern Processors
• Memory Hierarchies
• Matrix Multiply Cache Optimizations
• Bag of Tricks
Tiling Alone Might Not Be Enough

- Naïve and a “naïvely tiled” code
Optimizing in Practice

• Tiling for registers
  • loop unrolling, use of named “register” variables

• Tiling for multiple levels of cache

• 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: www.cs.berkeley.edu/~richie/bebop
  • PHiPAC: www.icsi.berkeley.edu/~bilmes/phipac
    in particular tr-98-035.ps.gz
  • ATLAS: www.netlib.org/atlas
PHiPAC: Portable High Performance ANSI C

Speed of n-by-n matrix multiply on Sun Ultra-1/170, peak = 330 MFlops
ATLAS (DGEMM n = 500)

Source: Jack Dongarra

- ATLAS is faster than all other portable BLAS implementations and it is comparable with machine-specific libraries provided by the vendor.
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 \(b[i+1]\)

\[
\begin{align*}
\downarrow \\
float f1 &= b[i]; \\
float f2 &= b[i+1]; \\
\end{align*}
\]

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

• With some compilers, you can say explicitly (via flag or pragma) that \(a\) and \(b\) are not aliased.
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];
float f1 = filter[1];
float f2 = filter[2];
while( ... ) {
    *res++ = f0*signal[0]
        + f1*signal[1]
        + f2*signal[2];
    signal++;
}
```

also: register float f0 = ...;
Minimize Pointer Updates

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

```c
f0 = *r8; r8 += 4;
f1 = *r8; r8 += 4;
f2 = *r8; r8 += 4;
```

```c
↓

f0 = r8[0];
f1 = r8[4];
f2 = r8[8];
r8 += 12;
```
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

Original matrix (numbers are addresses)

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
</tbody>
</table>

Reorganized into 2x2 blocks

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
Summary

• Performance programming on uniprocessors requires
  • understanding of fine-grained parallelism in processor
    • produce good instruction mix
  • understanding of memory system
    • levels, costs, sizes
    • improve locality

• Blocking (tiling) is a basic approach
  • 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 0 (due 6/25/02)
  • http://www.cs.berkeley.edu/~richie/bebop/notes/matmul2002
End

(Extra slides follow)
Example: 5 Steps of MIPS Datapath

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

- Pipelining is also used within arithmetic units
  - a fp multiply may have latency 10 cycles, but throughput of 1/cycle
Dependences (Data Hazards) Limit Parallelism

• A **dependence** or **data hazard** is one of the following:
  • **true of flow dependence:**
    • a writes a location that b later reads
    • (read-after write or RAW hazard)
  • **anti-dependence**
    • a reads a location that b later writes
    • (write-after-read or WAR hazard)
  • **output dependence**
    • a writes a location that b later writes
    • (write-after-write or WAW hazard)

<table>
<thead>
<tr>
<th>true</th>
<th>anti</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a =</td>
<td>= a</td>
<td>a =</td>
</tr>
<tr>
<td>= a</td>
<td>a =</td>
<td>a =</td>
</tr>
</tbody>
</table>
Observing a Memory Hierarchy

Dec Alpha, 21064, 150 MHz clock

See [www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps](http://www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps) for details