Uniprocessor Optimizations

Arvind Krishnamurthy
Fall 2004

**Idealized Uniprocessor Model**
- Processor names bytes, words, etc. in its address space
  - These represent integers, floats, pointers, arrays, etc.
  - Exist in the program stack, static region, or heap
- Operations include
  - Read and write (given an address/pointer)
  - Arithmetic and other logical operations
- 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
- Cost
  - Each operation has 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, different 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 understand all of this and can optimize your program; in practice they don't.

**Pipelining Review**
- 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" reduces speedup (overhead)

**Example: 5 Steps of MIPS Datapath**

**Limits to Instruction Level Parallelism (ILP)**
- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions (single dryer)
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).
- The hardware and compiler will try to reduce hazards
  - Reordering instructions, multiple issue, dynamic branch prediction, speculative execution...
  - You can also enable parallelism by careful coding
Dependences Limit Parallelism

- A dependence or data hazard is one of the following:
  - true or flow dependence: X writes a location that Y later reads (read-after write or RAW hazard)
  - anti-dependence: X reads a location that Y later writes (write-after-read or WAR hazard)
  - output dependence: X writes a location that Y 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 = 3.9/12.2</td>
<td>a = 1</td>
<td>a = 3.4*b/12.2</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Speed (ns)</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100s</td>
</tr>
<tr>
<td>10</td>
<td>Ks</td>
</tr>
<tr>
<td>100</td>
<td>Ms</td>
</tr>
<tr>
<td>10 ms</td>
<td>Gs</td>
</tr>
<tr>
<td>10 sec</td>
<td>Ts</td>
</tr>
</tbody>
</table>

Processor-DRAM Gap (latency)

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

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

Moore’s Law: 60%/yr.

Cache Basics

- Cache contains tag (upper bits of address) and values
- Cache hit: if tag matches—cheap
- Cache miss: non-cached memory access—expensive
- Consider cache with 4 entries and 4 words per entry

---

Experimental Study of Memory

- Microbenchmark for memory system performance

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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 (4 Bytes)

See lambda.cs.yale.edu/~arvind/papers/t3d-isca95.ps for details
Some Architecture Lessons

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

- We will illustrate with a common technique for improving cache performance.
- Idea: use divide-and-conquer to define a problem that fits in register/L1-cache/L2-cache.

Simple Model of Memory

- Assume just 2 levels in the hierarchy, fast and slow.
- All data initially slow memory.
- \( m \) = number of memory elements (words) moved between fast and slow memory.
- \( t_s \) = time per slow memory operation.
- \( f \) = number of arithmetic operations.
- \( t_a \) = time per arithmetic operation.
- \( q = f / m \): average number of flops per slow memory access.
- Minimum possible time = \( f \cdot t_a \), when all data in fast memory.
- Actual time \( f \cdot t_a + m \cdot t_s = f \cdot t_a (1 + t_s / t_a) \).

- Larger \( q \) means time closer to minimum \( f \cdot t_a \).

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).

<table>
<thead>
<tr>
<th>Column major</th>
<th>Row major</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19</td>
<td></td>
</tr>
</tbody>
</table>

Minimum time for matrix-vector multiplication is \( f \cdot t_a (1 + t_s / t_a) \).

Warm up: Matrix-vector multiplication

\[
\text{for } i = 1:n \\
\quad \text{for } j = 1:n \\
\quad \quad y(i) = y(i) + A(i,j) \cdot x(j) \\
\]

\[
\text{Compile: } y = y + A \cdot x \\
\text{Naive: } y = y + \sum_{j=1}^{n} A(i,j) \cdot x(j) \\
\]

\[
\text{for } i = 1:n \\
\quad \text{for } j = 1:n \\
\quad \quad C(i,j) = C(i,j) + A(i,k) \cdot B(k,j) \\
\]

\[
\text{Compile: } C = C + A \cdot B \\
\text{Naive: } C(i,j) = \sum_{k=1}^{n} A(i,k) \cdot B(k,j) \\
\]
Naive Matrix Multiply

\[ (\text{implies } C = C + A \cdot B) \]
for \( i = 1 \) to \( n \)
\{read row \( i \) of \( A \) into fast memory\}
\{read \( C(i,j) \) into fast memory\}
for \( j = 1 \) to \( n \)
\{read column \( j \) of \( B \) into fast memory\}
\{read \( C(i,j) \) into fast memory\}
for \( k = 1 \) to \( n \)
\[ C(i,j) = C(i,j) + A(i,k) \cdot B(k,j) \]
\{write \( C(i,j) \) back to slow memory\}

Naive Matrix Multiply Analysis

Number of slow memory references on unblocked matrix multiply
\[ m = n^3 \text{ read (each column of } B \text{ } n \text{ times)} \]
\[ + n^2 \text{ read (each row of } A \text{ once)} \]
\[ + 2n^2 \text{ read and write (each element of } C \text{ once)} \]
\[ = n^3 + 3n^2 \]
So \( \frac{f}{m} = \frac{2n^3}{(n^3 + 3n^2)} \)
\[ \approx 2 \text{ for large } n, \text{ no improvement over matrix-vector multiply} \]

Blocked (tiled) matrix multiply

Consider \( A, B, C \) to be \( N \times N \) matrices of \( b \times b \) subblocks where \( b = \frac{n}{N} \) 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\}
\[ C(i,j) = C(i,j) + A(i,k) \cdot B(k,j) \] (do a matrix multiply on blocks)
\{write block \( C(i,j) \) back to slow memory\}

Blocked Matrix Multiple (analysis)

Recall:
\( m \) is amount memory traffic between slow and fast memory
\( f \) is number of floating point operations, \( 2n^3 \) for this problem
\( q = f/m \) is our measure of algorithm efficiency in the memory system

So:
\[ m = N \cdot n^2 \text{ read each block of } A \text{ } N^2 \text{ times (} N^2 \cdot \frac{n}{N} \cdot \frac{n}{N} \text{)} \]
\[ + N \cdot n^2 \text{ read each block of } B \text{ } N^2 \text{ times} \]
\[ + 2n^2 \text{ read and write each block of } C \text{ once} \]
\[ = (2N + 2) \cdot n^2 \]
So \( q = \frac{f}{m} = \frac{2n^3}{(2N + 2) \cdot n^2} \)
\[ \approx \frac{n}{N} = b \text{ for large } n \]
So we can improve performance by increasing the blocksize \( b \)
Can be much faster than matrix-vector multiply \((q=2)\)

Tiled Matrix Multiply

Consider \( A, B, C \) to be \( N \times N \) matrices of \( \frac{n}{N} \times \frac{n}{N} \) tiles
Bring in a tile from \( A \) and a tile from \( B \), and multiply them to update a \( C \) tile

\[ C = A \cdot B \]

Limits to optimizing matrix multiply

The blocked algorithm has ratio \( q = b \)
- The large 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:
  \[ 3b^2 \ll M, \text{ so } q \approx b = \mathfrak{O}(\sqrt[n]{M}) \]

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

Theorem (Hong & Kung, 1981): Any reorganization of this algorithm (that uses only algebraic associativity) is limited to \( q = \mathfrak{O}(\sqrt[n]{M}) \)
Basic Linear Algebra Routines

- Industry standard interface (evolving)
- Vendors, others supply optimized implementations
- History
  - BLAS1 (1970s):
    - Vector operations: dot product, saxpy ($y=\alpha x+y$), etc
    - $m=3n, f=2n, q \sim 2/3$ or less
  - BLAS2 (mid 1980s):
    - Matrix-vector operations: matrix vector multiply, etc
    - $m=n^2, f=2n^2, q=2$, less overhead
    - Somewhat faster than BLAS1
  - BLAS3 (late 1980s):
    - Matrix-matrix operations: matrix matrix multiply, etc
    - $m \gg 4n^2, f=O(n^3)$, 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, www.netlib.org/lapack

Optimizing in Practice

- Tiling for multiple levels of cache
- Tiling for registers
  - Loop unrolling, use of named "register" variables
- Exploiting fine-grained parallelism in processor
  - Superscalar; pipelining
- Complicated compiler interactions
- Hard to do by hand
- Automatic optimization an active research area
  - BeBOP: www.cs.berkeley.edu/~richie/bebop
  - PHiPAC: www.cs.berkeley.edu/~bilmes/phipac
  - ATLAS: www.netlib.org/atlas

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 $2 \times 2$ matrix multiply
  - Normally 8 multiplies, Strassen does it with 7 multiplies (but many more adds)

Strassen’s Multiplication (contd.)

\[
T(n) = \begin{cases} 
7T(n/2) + 18n^2 & \text{if } n \text{ is } \text{even} \\
O(n^3) & \text{if } n \text{ is } \text{odd} 
\end{cases}
\]

- Asymptotically faster
- Several times faster for large $n$ in practice
- Cross-over depends on machine
- Available in several libraries

- Caveats
  - Needs more memory than standard algorithm
  - Can be less accurate because of roundoff error
  - Current world’s record is $O(n^{2.376..})$

Outline

- Optimizations roadmap
- Optimizing for cache hierarchy
- Strassen’s matrix multiply algorithm (better algorithms)
- Bag of tricks for optimizing serial code
Removing False Dependencies

- Using local variables, reorder operations to remove unnecessary/false dependencies

\[ a = b \cdot c + d \cdot e \]
\[ d = b + 1 \]
\[ \text{<use } d \text{> } \]
\[ a = b \cdot c + d \cdot e \]
\[ d_1 = b + 1 \]
\[ \text{<use } d_1 \text{> } \]

Removing False Dependencies

- Consider the following code:

\[ a[i] = b[i] + c; \]
\[ a[i+1] = b[i+1] \times d; \]

Will this generate good code?

Exploit Multiple Registers

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

\[ \text{while( ... ) } \{ \\
\quad *res++ = filter[0] \times signal[0] \\
\quad \quad + filter[1] \times signal[1] \\
\quad \quad + filter[2] \times signal[2]; \\
\quad \text{signal++;} \\
\} \]

\[ \text{float } f0 = \text{filter}[0]; \]
\[ \text{float } f1 = \text{filter}[1]; \]
\[ \text{float } f2 = \text{filter}[2]; \]
\[ \text{while( ... ) } \{ \\
\quad *res++ = f0 \times signal[0] \\
\quad \quad + f1 \times signal[1] \\
\quad \quad + f2 \times signal[2]; \\
\quad \text{signal++;} \\
\} \]

Minimize Pointer Updates

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

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

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

\[ \text{float } f0 = \text{filter}[0], f1 = \text{filter}[1], f2 = \text{filter}[2]; \]
\[ \text{float } s0 = \text{signal}[0], s1 = \text{signal}[1], s2 = \text{signal}[2]; \]
\[ \text{*res++ = f0} \times s0 + f1 \times s1 + f2 \times s2; \]
\[ \text{do { } signal += 3; } \]
\[ \quad \text{s0 = signal[0]; } \]
\[ \quad \text{res[0] = f0} \times s0 + f1 \times s2 + f2 \times s0; \]
\[ \quad \text{s1 = signal[1]; } \]
\[ \quad \text{res[1] = f0} \times s2 + f1 \times s0 + f2 \times s1; \]
\[ \quad \text{s2 = signal[2]; } \]
\[ \quad \text{res[2] = f0} \times s0 + f1 \times s1 + f2 \times s2; \]
\[ \quad \text{res += 3;} \]
\[ \text{while( ... ); } \]
Expose Independent Operations

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

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

Summary

- 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)
- Performance programming on uniprocessors requires
  - Understanding of fine-grained parallelism in processor
    - Produce good instruction mix
    - Understanding of memory system
    - Blocking (tiling) is a basic approach
    - Techniques apply generally, but the details (e.g., block size) are architecture dependent.