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

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

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 = \frac{3.4 \times b}{12.2})</td>
<td>(a = a)</td>
<td>(a = a)</td>
</tr>
<tr>
<td>(a = a)</td>
<td>(a = 1)</td>
<td>(a = a)</td>
</tr>
</tbody>
</table>

**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
**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**
- CPU 60%/yr.
- DRAM 7%/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

**Tag**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>X</td>
</tr>
</tbody>
</table>

- **Address:** 10100110

- **Cache line length**: # of bytes loaded together in one entry
  - Provides spatial locality
- **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

- 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)

Memory Hierarchy of a Sun Ultra-Il(333MHz)

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 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

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)
Simple Model of Memory

- 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 element access
- Minimum possible time = \( f * t_f \) when all data in fast memory
- Actual time
  \[ f * t_f + m * t_m = f * t_f * (1 + tm/tf * 1/q) \]
- Larger \( q \) means Time closer to minimum \( f * t_f \)

Key to algorithm efficiency

Key to machine efficiency

Warm up: Matrix-vector multiplication

\{implements \( y = y + A*x \}\ }

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

\[ y(0) = y(0) + A( :) * x( :) \]
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

Naïve Matrix Multiply

{implements C = C + A*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) * B(k,j)
Naïve Matrix Multiply

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

\[
\begin{array}{c}
C(i,j) \\
\end{array}
= \begin{array}{c}
\text{read (each column of B n times)} \\
\text{+ n}^2 \text{ read (each row of A once)} \\
\text{+ 2n}^2 \text{ read and write (each element of C once)} \\
\end{array} + \begin{array}{c}
A(i,:) \\
\text{*} \\
B(:,j) \\
\end{array}
\]

Naïve Matrix Multiply Analysis

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 \( q = f / m = 2n^3 / (n^3 + 3n^2) \)
\( \sim 2 \) 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}
```

**Tiled Matrix Multiply**

- Consider A, B, C to be N*N matrices of n/N by n/N tiles
- Bring in a tile from A and a tile from B, and multiply them to update a C tile

```
b = n/N
for i = 0 to N-1
    for j = 0 to N-1
        for k = 0 to N-1
            for x = 0 to n/N
                for y = 0 to n/N
                    C[i*b + x][j*b + y] += A[i*b+x][k*b+z]*B[k*b+z][j*b+y]
```
**Blocked Matrix Multiple (analysis)**

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 = N^2 \cdot n^2 \text{ read each block of } B \text{ \(N^3\) times (} N^3 \cdot n/N \cdot n/N) \\
+ N^2 \cdot n^2 \text{ read each block of } A \text{ \(N^3\) times} \\
+ 2n^2 \text{ read and write each block of } C \text{ once} \\
= (2N + 2) \cdot n^2
\]

So \(q = f / m = 2n^3 / ((2N + 2) \cdot n^2)\)
\(~= n / N = b\) for large \(n\)

So we can improve performance by increasing the blocksize \(b\)
Can be much faster than matrix-vector multiply (\(q=2\))

---

**Limits to optimizing matrix multiply**

The blocked algorithm has ratio \(q \sim b\)
- 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:
  \(3b^2 \leq M\), so \(q \sim b \leq \sqrt{M/3}\)

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 \(q = O(\sqrt{M})\)
Basic Linear Algebra Routines

- Industry standard interface (evolving)
- Vendors, others supply optimized implementations
- History
  - BLAS1 (1970s):
    - vector operations: dot product, saxpy (y=α*x+y), etc
    - m=3*n, f=2*n, q ~2/3 or less
  - BLAS2 (mid 1980s)
    - matrix-vector operations: matrix vector multiply, etc
    - m=n^2, f=2*n^2, q~2, less overhead
    - somewhat faster than BLAS1
  - BLAS3 (late 1980s)
    - matrix-matrix operations: matrix matrix multiply, etc
    - m >= 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

BLAS Speeds on IBM RS6000

Peak speed = 266 Mflops
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.icsi.berkeley.edu/~bilmes/phipac
    in particular tr-98-035.ps.gz
  - 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 2x2 matrix multiply
  - normally 8 multiplies, Strassen does it with 7 multiplies (but many more adds)

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

Let $p_1 = (a_{12} - a_{22}) \cdot (b_{21} + b_{22})$  \hspace{1cm} p_5 = a_{11} \cdot (b_{12} - b_{22})$

$p_2 = (a_{11} + a_{22}) \cdot (b_{11} + b_{22})$  \hspace{1cm} p_6 = a_{22} \cdot (b_{21} - b_{11})$

$p_3 = (a_{11} - a_{21}) \cdot (b_{11} + b_{12})$  \hspace{1cm} p_7 = (a_{21} + a_{22}) \cdot b_{11}$

$p_4 = (a_{11} + a_{12}) \cdot b_{22}$

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’s multiplication (contd.)

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

- Asymptotically faster
  - Several times faster for large \( n \) in practice
  - Cross-over depends on machine
  - 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

\[
a = b \times c + d \times e \\
d = b + 1 \\
... \langle \text{use } d \rangle \\
\downarrow \\
a = b \times c + d \times e \\
d_1 = b + 1 \\
... \langle \text{use } d_1 \rangle ... \\
\]

Removing False Dependencies

- Consider the following code:

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

- Will this generate good code?
Removing False Dependencies

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

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

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++ = \text{filter[0]}*\text{signal[0]}
    + \text{filter[1]}*\text{signal[1]}
    + \text{filter[2]}*\text{signal[2]};
    signal++;
}
```

```c
float f0 = \text{filter[0]};
float f1 = \text{filter[1]};
float f2 = \text{filter[2]};
while( … ) {
    *res++ = f0*\text{signal[0]}
    + f1*\text{signal[1]}
    + f2*\text{signal[2]};
    signal++;
}
```

also: \text{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;
```

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( ... );
```
Exposé Independent Operations

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

```plaintext
f1 = f5 * f9;
f2 = f6 + f10;
f3 = f7 * f11;
f4 = f8 + f12;
```

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