Lecture 4

Instruction Level Parallelism
Vectorization, SSE
Optimizing for the memory hierarchy
Announcements

• Partners?
Today’s lecture

• Why multicore?
• Instruction Level Parallelism
• Vectorization, SSE
• Optimizing for the memory hierarchy
Why Multicore processors?

- We can’t keep increasing the clock speed
- If we simplify the processor, we can reduce power consumption and pack more processors in a given amount of space
- Some capabilities, like multi-issue, can be costly in terms of chip “real estate” and power consumption

Source: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)
What makes a processor run faster?

- Registers and cache
- Pipelining
- Instruction level parallelism
- Vectorization, SSE
Pipelining

- **Assembly line processing - an auto plant**

  Dave Patterson’s Laundry example: 4 people doing laundry

  wash (30 min) + dry (40 min) + fold (20 min) = 90 min

  
  
  
  
  
  
  6 PM  7  8  9

  
  
  
  
  
  
  Time

  | 30 | 40 | 40 | 40 | 40 | 20 |

  
  A  B  C  D

  - Sequential execution takes
    4 * 90min = 6 hours

  - Pipelined execution takes
    30+4*40+20 = 3.5 hours

  - Bandwidth = loads/hour

  - Pipelining helps bandwidth but not latency (90 min)

  - Bandwidth limited by slowest pipeline stage

  - Potential speedup = Number pipe stages
Instruction level parallelism

• Execute more than one instruction at a time
  \[ x = \frac{y}{z}; \]
  \[ a = b + c; \]

• Dynamic techniques
  ‣ Allow stalled instructions to proceed
  ‣ Reorder instructions

\[ x = \frac{y}{z}; \]
\[ a = x + c; \]
\[ t = c - q; \]
\[ x = \frac{y}{z}; \]
\[ t = c - q; \]
\[ a = x + c; \]
Multi-execution pipeline

- MIPS R4000

\[
\begin{align*}
x &= y / z; \\
a &= b + c; \\
s &= q \times r; \\
i &= i + 1;
\end{align*}
\]
Scoreboarding, Tomasulo’s algorithm

• Hardware data structures keep track of
  ‣ When instructions complete
  ‣ Which instructions depend on the results
  ‣ When it’s safe to write a reg.

• Deals with data hazards
  ‣ WAR (Write after read)
  ‣ RAW (Read after write)
  ‣ WAW (Write after write)
What makes a processor run faster?

• Registers and cache
• Pipelining
• Instruction level parallelism
• Vectorization, SSE
SIMD (Single Instruction Multiple Data)

- Operate on regular arrays of data
- Two landmark SIMD designs
  - ILIAC IV (1960s)
  - Connection Machine 1 and 2 (1980s)
- Vector computer: Cray-1 (1976)
- Intel and others support SIMD for multimedia and graphics
  - SSE
    - Streaming SIMD extensions, Altivec
  - Operations defined on vectors
- GPUs, Cell Broadband Engine
- Reduced performance on data dependent or irregular computations

```
forall i = 0:N-1
p[i] = a[i] * b[i]
```

```
forall i = 0 : n-1
if ( x[i] < 0) then
  y[i] = x[i]
else
  y[i] = √x[i]
end if
end forall
```
Streaming SIMD Extensions (SSE)

- SSE (SSE4 on Intel Nehalem), Altivec
- Short vectors: 128 bits (256 bits coming)

\[ r[0:3] = a[0:3]*b[0:3] \]

4 floats
2 doubles

Jim Demmel

Courtesy of Mercury Computer Systems, Inc.
How do we use the SSE instructions?

- Low level: assembly language or libraries
- Higher level: a vectorizing compiler
- Lilliput.ucsd.edu (Intel Gainstown, Nehalem)
  (Gcc v 4.4.3, won’t work on ieng6-203 until we get new compilers)

```c
gcc –O3
float a[N], b[N], c[N];
for (int i=0; i<N; i++)
    a[i] = b[i] / c[i];
```

- Performance
  - Single precision:
    - With vectorization: 0.52 sec.
    - Without vectorization: 1.9 sec.
  - Double precision:
    - With vectorization: 1.5 sec.
    - Without vectorization: 3.0 sec.
How does the vectorizer work?

• Transformed code
  
  for (i = 0; i < 1024; i+=4)
    a[i:i+3] = b[i:i+3] + c[i:i+3];

• Vector instructions
  
  for (i = 0; i < 1024; i+=4){
    vB = vec_ld( &b[i] );
    vC = vec_ld( &c[i] );
    vA = vec_add( vB, vC );
    vec_st( vA, &a[i] );
  }

What prevents vectorization

• Data dependencies
  for (int i = 1; i < N; i++)
    b[i] = b[i-1] + 2;

  note: not vectorized: unsupported use in stmt.

b[1] = b[0] + 2;
Blocking for Cache
Matrix Multiplication

• An important core operation in many numerical algorithms
• Given two *conforming* matrices $A$ and $B$, form the matrix product $A \times B$
  $A$ is $m \times n$
  $B$ is $n \times p$
• Operation count: $O(n^3)$ multiply-adds for an $n \times n$ square matrix
• Discussion follows from Demmel
  [www.cs.berkeley.edu/~demmel/cs267_Spr99/Lectures/Lect02.html](http://www.cs.berkeley.edu/~demmel/cs267_Spr99/Lectures/Lect02.html)
Unblocked Matrix Multiplication

for i := 0 to n-1
  for j := 0 to n-1
    for k := 0 to n-1
      C[i,j] += A[i,k] * B[k,j]
Analysis of performance

for i = 0 to n-1
  // for each iteration i, load all of B into cache
  for j = 0 to n-1
    // for each iteration (i,j), load A[i,:] into cache
    // for each iteration (i,j), load and store C[i,j]
    for k = 0 to n-1
      C[i,j] += A[i,k] * B[k,j]
Analysis of performance

for i = 0 to n-1
   // n x n^2 / L loads = n^3/L, L=cache line size   B[:,i]
for j = 0 to n-1
   // n^2 / L loads = n^2/L          A[i,:]
   // n^2 / L loads + n^2 / L stores = 2n^2 / L   C[i,j]
for k = 0 to n-1
   C[i,j] += A[i,k] * B[k,j]       Total:(n^3 + 3n^2) / L
Flops to memory ratio

Let $q = \# \text{flops} / \text{main memory reference}$

$$q = \frac{2n^3}{n^3 + 3n^2}$$

$\approx 2$ as $n \to \infty$
Blocked Matrix Multiply

- Divide A, B, C into $N \times N$ sub blocks
- Assume we have a good quality library to perform matrix multiplication on subblocks
- Each sub block is $b \times b$
  - $b = n/N$ is called the block size
  - How do we establish $b$?

\[
C[i,j] = C[i,j] + A[i,k] \times B[k,j]
\]
Blocked Matrix Multiplication

for $i = 0$ to $N-1$
  for $j = 0$ to $N-1$
    // load each block $C[i,j]$ into cache, once : $n^2$
    // $b = n/N =$ cache line size
    for $k = 0$ to $N-1$
      // load each block $A[i,k]$ and $B[k,j]$ $N^3$ times
      // $= 2N^3 \times (n/N)^2 =$ $2Nn^2$
      $C[i,j] += A[i,k] \times B[k,j]$ // do the matrix multiply
      // write each block $C[i,j]$ once : $n^2$
  Total: $(2N+2)n^2$
Flops to memory ratio

Let \( q = \# \text{ flops} / \text{ main memory reference} \)

\[
q = \frac{2n^3}{(2N + 2)n^2} = \frac{n}{N + 1}
\]

\( \approx \frac{n}{N} = b \)

as \( n \to \infty \)
The results

<table>
<thead>
<tr>
<th>N,B</th>
<th>Unblocked Time</th>
<th>Blocked Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>256, 64</td>
<td>0.6</td>
<td>0.002</td>
</tr>
<tr>
<td>512, 128</td>
<td>15</td>
<td>0.24</td>
</tr>
</tbody>
</table>

Amortize memory accesses by increasing memory reuse
More on blocked algorithms

• Data in the sub-blocks are contiguous within rows only
• We may incur conflict cache misses
• Idea: since re-use is so high… let’s copy the subblocks into contiguous memory before passing to our matrix multiply routine

“The Cache Performance and Optimizations of Blocked Algorithms,”
M. Lam et al., ASPLOS IV, 1991

http://www-suif.stanford.edu/papers/lam91.ps
Fin
Partitioning

- Splits up the data over processors
- Different partitionings according to the processor geometry
- For P processors geometries are of the form $p_0 \times p_1$, where $P = p_0 \times p_1$
- For P=4: 3 possible geometries

\[
\begin{array}{c|c|c|c}
0 & 1 & 2 & 3 \\
\hline
1 & 4 & 1 & 1 \\
\hline
2 & 2 & 0 & 3 \\
\end{array}
\]