Lecture 16
Optimizing for the memory hierarchy
Announcements

• A4 has been released
• Using SSE intrinsics, you can speed up your code by nearly a factor of 2
Today’s lecture

• Motivation for using SSE intrinsics
• Managing Memory Locality
Vectorization limitations

- If we have simple data dependence patterns, GCC can generate good quality vectorized code
  
  ```
  double a[N], b[N], c[N];
  for (i=0; i<N; i++) {
      a[i] = sqrt(b[i] / c[i]);
  }
  ```

- Speedup due to vectorization: x1.7

- But for some loops it cannot, and must use the “unpacked” versions of SSE, effectively no SSE improvement

- The vectorizer cannot improve the performance of the cardiac simulator
Complications

- There was no benefit from vectorization
- Instead of using the packed form of instructions (128 bits) the code uses just the lower 64 bit portion of the xmm registers and the ALU: MULSD, MOVSD instead of MULPD and MOVAPD
- GCC can’t vectorize this code
- How does it know that pointers Ep and Et don’t overlap in memory?

```c
double **E, **E_prev;
int jEnd = (((m+2)*(n+2) - 1) - n) - (n+2);
for (j=n+2+1; j<= jEnd; j+=(n+2)){
    double *Et = E + j, *Ep = E_prev + j;
    for (i=0; i < n ; i++)
        Et[i] = Ep[i]+α*(Ep[i+1]+Ep[i-1] - 4*Ep[i]+Ep[i+(n+2)]+Ep[i-(n+2)));
}
```
Today’s lecture

• Motivation for using SSE intrinsics
• Managing Memory Locality
Computational Intensity

- Performance is limited by the ratio of computation performed and the volume of data moved between processor and main memory.

- We call this ratio the computational intensity, or $q$; it is intrinsic to the application.

- For example, for a conventional implementation of matrix multiplication $q=2$, but we can do much better.
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
Memory access patterns

\[ C(i, j) = \sum_k A(i, k) B(k, j) \]

for \( i = 0 \) to \( n-1 \)
for \( j = 0 \) to \( n-1 \)
for \( k = 0 \) to \( n-1 \)

\[ C[i,j] += A[i,k] \times B[k,j] \]
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 × n² / L loads = n³/L, L=cache line size
  B[::,::]
for j = 0 to n-1
  // n² / L loads = n²/L
  A[i,:]
  // n² / L loads + n² / L stores = 2n² / L
  C[i,j]
for k = 0 to n-1
  C[i,j] += A[i,k] * B[k,j]
  Total: (n³ + 3n²) / L

\[ q = \frac{2n^3}{n^3 + 3n^2} \approx 2 \text{ as } n \to \infty \]
Performance

- Performance is good until A&B no longer fit in L2 cache
  \[8 + (N^2 + N^2) > 4 \text{ MB}\]
- Improvement: transpose B before we multiply A*B, to avoid accessing B by columns
- Performance drops when B no longer fits in L2: \(N \sim 725\)
- Bimodal L1 miss rate 33% and 4% for N=128, 129

\[
\text{for } i = 0 \text{ to } n-1 \\
\text{for } j = 0 \text{ to } n-1 \\
\text{for } k = 0 \text{ to } n-1 \\
C[i,j] += A[i,k] \ast B^T[j,k];
\]
How many cache lines are required to store a column of a $128^2$ matrix?

A. 128  
B. $128 \times 16$  
C. $128/64$

- Bang’s Clovertown processor
- L1 is 32KB, 8 way set associative
- 64 byte line size
- 512 bytes in a set, 64 sets total

Row major

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

- L1 is 32KB, 8 way set associative, 64 byte line size
- There are 512 bytes in a set, 64 sets total
- A $128^2$ matrix exceeds L1’s capacity
  - A row uses 16 cache lines ($(128*8/64)$), one after another one in each set
  - A column needs 128 lines, spaced @ 16 line intervals
  - There aren’t enough lines in all sets ($128*16 >> 64*8$) so we evict a line and cannot reuse it when we get to the next column

Row major

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

Patterson and Hennessy
What happens when $N=129$?

- Matrix still doesn’t fit into L1, but as you move down the column you’ll be able to reuse some cached values

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

Column of matrix is stored in red cache lines

Larry Carter
Blocking for cache: motivation

• The processor wastes a lot of time loading and re-loading B into cache

• Observation: if B fits in L2 cache, the code runs much more quickly

• Idea: what if we keep pieces of B in L2, and re-use them many times?
Blocked Matrix Multiply

- Divide A, B, C into $N \times N$ sub blocks
- All 3 blocks must fit into cache
- Assume we have a good quality library to perform matrix multiplication on subblocks
- Each sub block is $b \times b$
  - $b = \frac{n}{N}$ is called the block size
  - How do we establish $b$?

\[
\]
What is the appropriate constraint for blocking for cache?

A. All 3 blocks must fit in cache
B. Only A
C. Only B
D. Only A and B
Blocked Matrix Multiplication Constraints

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 = block 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:

\( (2N + 2) \times n^2 \)

- All 3 blocks must fit into cache
- If \( M_{\text{fast}} = \) size of fast memory (L1)
  \[ 3b^2 \leq M_{\text{fast}} \Rightarrow b \leq (M_{\text{fast}}/3)^{1/2} \]
Computational intensity

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

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

\[\approx \frac{n}{N} = b\]

as \( n \to \infty \)

Compare with \( q=2 \) for the unblocked algorithm!
Why will results vary according to the blocking factor?

A. Floating point arithmetic isn’t associative
B. Roundoff
C. Both
The results

- Where did the performance go?
  - Compiler is unable to generate multiply-add instructions, so the best we can do is 4.66 Gflops
  - Clovertown supports simultaneous multiplication and addition
- L1 hit cost 3 cycles (3/8 cycle for each access to A, B and C)
  \[ R_\infty = 4 \times 2.33 = 9.32 \text{ Gflops} \]
  \[ \sim 13\% \text{ of peak} \]
Performance of Matrix Multiply

\[ R_\infty = 4 \times 2.33 = 9.32 \text{ Gflops} \]

\( \sim 87\% \) of peak

8.14 GFlops

Scott B. Baden / CSE 160 / Wi '16
Recall the blocked algorithm

\[
\begin{align*}
\text{for } & (ii=0; ii<n; ii+=BSZ) \\
& \text{for } (jj=0; jj<n; jj+=BSZ) \\
& \quad \text{// Initialize } C_{ij} \text{ to } 0 \\
& \quad \text{for (int } kk=0; kk<n; kk+=BSZ) \\
& \quad \quad \text{for (i=0; i<BSZ; i++)} \\
& \quad \quad \quad \text{for (j=0; j<BSZ; j++)} \\
& \quad \quad \quad \quad \text{sum } = 0 \\
& \quad \quad \quad \quad \text{for (int } k=0; k<BSZ; k++) \\
& \quad \quad \quad \quad \quad \text{sum } += A[i+ii][k+kk] * B[k+kk][j+jj]; \\
& \quad \quad \quad \quad \quad C_{ij}[i][j] += \text{sum;}
\end{align*}
\]

\[
\begin{align*}
\text{for } & (i=0; i<BSZ; i++) \\
& \text{for (j=0; j<BSZ; j++)} \\
& \quad C[i+ii][j+jj] += C_{ij}[i][j];
\end{align*}
\]
The assembler

- Run make with \texttt{asm=1}

```
for (int k=0; k<BSZ; k++)
  sum += A[i+ii][k+kk] * B[k+kk][j+jj];
```

See: cs.nyu.edu/courses/fall11/CSCI-GA.2130-001/x64-intro.pdf
http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

\texttt{.L11:}
\begin{verbatim}
mov        rdx, QWORD PTR [rdi+rax*8]
movsd      xmm0, QWORD PTR [rsi+rax*8]
add         rax, 1  \# ivtmp.52,
****        for (int k=0; k<BSZ; k++)
cmp         r15d, eax \# BSZ, ivtmp.52
****        sum += A[i+ii][k+kk] * B[k+kk][j+jj];
mulsd      xmm0, QWORD PTR [rdx+rcx] \# \{Scalar multiply\}
addsd      xmm1, xmm0 \# sum, D.3420
for (int k=0; k<BSZ; k++)
jg         .L11 \#,
\end{verbatim}
Going the full distance

- 2 levels of cache blocking + Cache friendly layouts
- Use SSE
- Register blocking
- Autotuning
  - Tune block size to matrix size
  - Computer generated variants & blocking factors
- See www.cs.berkeley.edu/~demmel/cs267_Spr12/Lectures/lecture02_

Max: 7.87 Gflops
Avg: 6.4 Gflops

$R_\infty = 4 \times 2.33 = 9.32$ Gflops
84% of peak (max)
69% peak (avg)
Going the full distance

Max: 7.87 Gflops
Avg: 6.4 Gflops

\[ R_\infty = 4 \times 2.33 = 9.32 \text{ Gflops} \]
84% of peak (max)
69% peak (avg)

H. Rodrigues, K. Korgaonkar, Y. Kim (CSE 260, W14)