Address space organization
Control Mechanism
Vectorization and SSE
Announcements
Summary from last time
Today’s lecture

• Address space organization
• Control mechanisms
• Vectorization and SSE
• Programming Lab #1
Address Space Organization

• We classify the address space organization of a parallel computer according to whether or not it provides global memory

• When there is a global memory we have a “shared memory” or “shared address space” architecture
  - multiprocessor vs partitioned global address space

• Where there is no global memory, we have a “shared nothing” architecture, also known as a multicomputer
Multiprocessor organization

- The address space is global to all processors
- Hardware automatically performs the global to local mapping using address translation mechanisms
- Two types, according to the uniformity of memory access times (ignoring contention)
  - **UMA**: Uniform Memory Access time
    - All processors observe the same memory access time
    - Also called **Symmetric Multiprocessors (SMPs)**
    - Usually bus based
  - **NUMA**: Non-uniform memory access time

[Diagram showing CPU and Memory connections]

computing.llnl.gov/tutorials/parallel_comp
NUMA

- Processors see distant-dependent access times to memory
- Implies physically distributed memory
- We often call these *distributed shared memory architectures*
  - Commercial example: SGI Origin Altix, up to 512 cores
  - But also many server nodes
  - Elaborate interconnect and software fabric
Architectures without shared memory

- Each processor has direct access to local memory only
- Send and receive messages to obtain copies of data from other processors
- We call this a *shared nothing* architecture, or a *multicomputer*

computing.llnl.gov/tutorials/parallel_comp
Hybrid organizations

- Multi-tier organizations are hierarchically organized
- Each node is a multiprocessor that may include accelerators
- Nodes communicate by passing messages
- Processors within a node communicate via shared memory but devices of different types may need to communicate explicitly, too
- All clusters and high end systems today
Today’s lecture

• Address space organization
• Control mechanisms
• Vectorization and SSE
• Programming Lab #1
Control Mechanism

Flynn’s classification (1966)
How do the processors issue instructions?

**SIMD:** Single Instruction, Multiple Data
Execute a global instruction stream in lock-step

**MIMD:** Multiple Instruction, Multiple Data
Clusters and servers processors execute instruction streams independently
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

```plaintext
forall i = 0 : n-1
  if (x[i] < 0) then
    y[i] = x[i]
  else
    y[i] = \sqrt{x[i]}
  end if
end forall
```

```
\begin{array}{c}
2 \\
4 \\
8 \\
7 \\
\end{array}
= \begin{array}{c}
1 \\
2 \\
3 \\
5 \\
\end{array}
+ \begin{array}{c}
1 \\
2 \\
5 \\
2 \\
\end{array}
```

forall i = 0 : n-1
  x[K[i]] = y[i] + z [i]
end forall
Today’s lecture

• Address space organization
• Control mechanisms
• Vectorization and SSE
• Programming Lab #1
Parallelism

- In addition to multithreading, processors support other forms of parallelism.
- Instruction level parallelism (ILP) – execute more than 1 instruction at a time, provided there are no data dependencies.

<table>
<thead>
<tr>
<th>No data dependencies</th>
<th>Data dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can use ILP</td>
<td>Cannot use ILP</td>
</tr>
<tr>
<td>x = y / z</td>
<td>x = y / z</td>
</tr>
<tr>
<td>a = b + c</td>
<td>a = b + x</td>
</tr>
</tbody>
</table>

- SIMD processing via streaming SIMD extensions (SSE).
- Applying parallelism implies that we can order operations arbitrarily, without affecting correctness.
Streaming SIMD Extensions

- SIMD instruction set on short vectors
- Called SSE on earlier processors, such as Bang’s (SSE3), AVX on Stampede

![Diagram of SIMD vector addition]

\[
\begin{align*}
X &= x_3 \quad x_2 \quad x_1 \quad x_0 \\
Y &= y_3 \quad y_2 \quad y_1 \quad y_0 \\
X + Y &= x_3+y_3 \quad x_2+y_2 \quad x_1+y_1 \quad x_0+y_0
\end{align*}
\]
How do we use SSE & how does it perform?

• Low level: assembly language or libraries
• Higher level: a vectorizing compiler

    g++ -O3 -ftree-vectorizer-verbose=2
    float b[N], c[N];
    for (int i=0; i<N; i++)
        b[i] += b[i]*b[i] + c[i]*c[i];

  7: LOOP VECTORIZED.
 vec.cpp:6: note: vectorized 1 loops in function..

• Performance
  Single precision:     With vectorization :  1.9  sec.
                        Without vectorization : 3.2  sec.
  Double precision:    With vectorization:  3.6 sec.
                        Without vectorization : 3.3 sec.

How does the vectorizer work?

• Original code
  ```c
  float b[N], c[N];
  for (int i=0; i<N; i++)
    b[i] += b[i]*b[i] + c[i]*c[i];
  ```

• Transformed code
  ```c
  for (i = 0; i < N; i+=4)  // Assumes that 4 divides N evenly
    a[i:i+3] = b[i:i+3] + c[i:i+3];
  ```

• Vector instructions
  ```c
  for (i = 0; i < N; 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**
  
  ```
  b[1] = b[0] + 2;
  ```

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

  Loop not vectorized: data dependency

- **Inner loops only**
  
  ```
  a[i] = b[i] + c[i];
  ```

  for(int j=0; j< reps; j++)
  
  for (int i=0; i<N; i++)
Which loop(s) won’t vectorize?

#1 for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
    maxval = (a[i] > maxval ? a[i] : maxval);
    if (maxval > 1000.0) break;
}

#2 for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
    maxval = (a[i] > maxval ? a[i] : maxval);
}

A. #1
B. #2
C. Both
C++ intrinsics

• The compiler may not be able to handle all situations, such as short vectors (2 or 4 elts)
• All major compilers provide a library of C++ functions and datatypes that map directly onto 1 or more machine instructions
• The interface provides 128 bit data types and operations on those datatypes
  ◦ _m128 (float)
  ◦ _m128d (double)
• Data movement and initialization
SSE Pragmatics

- SSE 2+ : 8 XMM registers (128 bits)
- AVX: 16 YMM data registers (256 bit) (Don’t use the MMX 64 bit registers)
- These are in addition to the conventional registers and are treated specially
- Vector operations on short vectors: + - / * etc
- Data transfer (load/store)
- Shuffling (handles conditionals)
- See the Intel intrisics guide: software.intel.com/sites/landingpage/IntrinsicsGuide
- May need to invoke compiler options depending on level of optimization
**Programming example**

- Without SSE vectorization: 0.777201 sec.
- With SSE vectorization: 0.457972 sec.
- Speedup due to vectorization: x1.697
- $PUB/Examples/SSE/Vec$

```c
double *a, *b, *c
__m128d vec1, vec2, vec3;
for (i=0; i<N; i+=2) {
    vec1 = _mm_load_pd(&b[i]);
    vec2 = _mm_load_pd(&c[i]);
    vec3 = _mm_div_pd(vec1, vec2);
    vec3 = _mm_sqrt_pd(vec3);
    _mm_store_pd(&a[i], vec3);
}
```
# SSE2 Cheat sheet

<table>
<thead>
<tr>
<th>Data transfer</th>
<th>Arithmetic</th>
<th>Compare</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm</td>
<td>ADD{SS/PS/SD/PD} xmm, mem/xmm</td>
<td>CMP{SS/PS/SD/PD}</td>
</tr>
<tr>
<td></td>
<td>SUB{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td>MOV {H/L} {PS/PD} xmm, mem/xmm</td>
<td>MUL{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DIV{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td>(load and store)</td>
<td>SQRT{SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAX {SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIN{SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
</tbody>
</table>

xmm: one operand is a 128-bit SSE2 register
mem/xmm: other operand is in memory or an SSE2 register
{SS} Scalar Single precision FP: one 32-bit operand in a 128-bit register
{PS} Packed Single precision FP: four 32-bit operands in a 128-bit register
{SD} Scalar Double precision FP: one 64-bit operand in a 128-bit register
{PD} Packed Double precision FP, or two 64-bit operands in a 128-bit register
{A} 128-bit operand is aligned in memory
{U} the 128-bit operand is unaligned in memory
{H} move the high half of the 128-bit operand
{L} move the low half of the 128-bit operand

Krstie Asanovic & Randy H. Katz
Today’s lecture

• Address space organization
• Control mechanisms
• Vectorization and SSE
• Programming Lab #1
Performance

• Blocking for cache will boost performance but a lot more is needed to approach ATLAS’ performance

\[ R_\infty = 4 \times 2.33 = 9.32 \text{ Gflops} \]
\[ \sim 87\% \text{ of peak} \]

\[ 8.14 \text{ GFlops} \]
Optimizing Matrix Multiplication

• Assume that we already have 2 levels of cache blocking (and possibly for TLB)
• Additional optimizations
  - Loop unrolling
  - Cache friendly layouts
  - Register tiling (with unrolling)
  - SSE intrinsics (vectorization)
  - Autotuning

• Will cover only some of these in lecture; for the rest, see
  http://www.cs.berkeley.edu/~demmel/cs267_Spr15/Lectures/lecture03_machines_jwd15.ppt
Loop Unrolling

- Common loop optimization strategy
- Duplicate the body of the loop

\[
\text{for (int } i=0; i < n; i++) \\
\quad z[i] = x[i] + y[i];
\]

- Register utilization, instruction scheduling
- May be combined with “jamming:” unroll and jam
- Not always advantageous. \textbf{Why?}
Register tiling in matrix multiply

- We can apply blocking to the registers, too
- In SSE4: 2x2 matrix multiply
- Store array values on the stack

\[
\begin{align*}
C_{00} &= A_{00}B_{00} + A_{01}B_{10} \\
C_{10} &= A_{10}B_{00} + A_{11}B_{10} \\
C_{01} &= A_{00}B_{01} + A_{01}B_{11} \\
C_{11} &= A_{10}B_{01} + A_{11}B_{11}
\end{align*}
\]

Rewrite as SIMD algebra

\[
\begin{pmatrix}
A00 & A01 \\
A10 & A11
\end{pmatrix}
\begin{pmatrix}
B00 & B01 \\
B10 & B11
\end{pmatrix}
\]

\[
\begin{align*}
C_{00} &= A_{00}A_{00} \times B_{00}B_{01} \\
C_{10} &= A_{10}A_{10} \times B_{00}B_{01} \\
C_{01} &= A_{01}A_{01} \times B_{10}B_{11} \\
C_{11} &= A_{11}A_{11} \times B_{10}B_{11}
\end{align*}
\]
2x2 Matmul with SSE intrinsics

#include <emmintrin.h>

void square_dgemm (int N, double* A, double* B, double* C){
    __m128d c1 = _mm_loadu_pd( C+0*N); //load unaligned block in C
    __m128d c2 = _mm_loadu_pd( C+1*N);
    for( int i = 0; i < 2; i++ ){
        __m128d a1 = _mm_load1_pd( A+i+0*N); // load i-th column of A (A0x,A0x) [x=0/1] (A1x,A1x)
        __m128d a2 = _mm_load1_pd( A+i+1*N);
        __m128d b = _mm_load_pd( B+i*N); // load aligned i-th row of B
        c1 = _mm_add_pd( c1, _mm_mul_pd( a1, b ) ); // rank-1 update
        c2 = _mm_add_pd( c2, _mm_mul_pd( a2, b ) ) ;
    }
    _mm_storeu_pd( C+0*N, c1 ); //store unaligned block in C
    _mm_storeu_pd( C+1*N, c2 );
}

\[
\begin{pmatrix}
A00 & A01 \\
A10 & A11
\end{pmatrix}
\begin{pmatrix}
B00 & B01 \\
B10 & B11
\end{pmatrix}
\]

Scott B. Baden / CSE 260, UCSD / Fall '15

30
**Autotuning**

- Performance tuning is complicated and involves many code variants
- Performance models are not accurate, for example, in choosing blocking factors
  - Need to tune to matrix size
  - Conflict misses
- **Autotuning**
  - Let the computer do the heavy lifting: generate and measure program variants & parameters
  - We write a program to manage the search space
  - PHiPAC → ATLAS, in Matlab
A search space

A 2-D slice of a 3-D register-tile search space. The dark blue region was pruned.
(Platform: Sun Ultra-IIi, 333 MHz, 667 Mflop/s peak, Sun cc v5.0 compiler)

Jim Demmel