Lecture 14
The C++ Memory model
Implementing synchronization
SSE vector processing
(SIMD Multimedia Extensions)
Announcements

• No section this Friday
Today’s lecture

• C++ memory model—continued
• Synchronization variables
• Implementing Synchronization
• SSE (SIMD Multimedia Extensions)
Visualizing cache locality

- The stencil’s bottom point traces the cache miss pattern: \([i, j+1]\)
- There are 6 reads per innermost iteration
- One miss every 8th access (8 doubles=1 line)
- We predict a miss rate of \((1/6)/8 = 2.1\%\)

```c
for (j=1; j<=m+1; j++) {
    for (i=1; i<=n+1; i++) {
        E[j,i] = Eprev[j,i] +
        \(\alpha \times (E_{prev}[j,i+1] + E_{prev}[j,i-1] - 4\times E_{prev}[j,i] + E_{prev}[j+1,i] + E_{prev}[j-1,i])\);
    }
}
```

\(i\)
\(j\)

Cache line

Scott B. Baden / CSE 160 / Wi '16
Recapping from last time: communication & synchronization variables

- The C++ atomic variable provides a special mechanism to guarantee that communication happens between threads
  - Which writes get seen by other threads
  - The order in which they will be seen
- The happens-before relationship provides the guarantee that memory writes by one specific statement are visible to another specific statement
- Different ways of accomplishing this: atomics, variables, thread creation and completion
- When one thread writes to a synchronization variable (e.g. an atomic or mutex) and another thread sees that write, the first thread is telling the second about all of the contents of memory up until it performed the write to that variable

Ready is a synchronization variable
In C++ we use load and store member functions

All the memory contents seen by T1, before it wrote to ready, must be visible to T2, after it reads the value true for ready.

Establishing a happens-before relationship

- Sequential consistency is guaranteed so long as the only conflicting concurrent accesses are to synchronization variables.
- Any write to a synchronization variable establishes a *happens-before* relationship with subsequent reads of that same variable: `x_ready=true` happens-before the read of `x_ready` in Thread 2.
- A statement *happens-before* another statement sequenced immediately after it: `x=42` happens-before `x_ready=true`.
- *Happens-before* is transitive: everything sequenced before a write to synchronization variable also happens-before the read of that synchronization variable by another thread: `x=42 (T1)` is visible after the read of `x_ready` by T2, e.g. the assignment to `r1`.
- The program is free from data races:
  - Thread 2 is guaranteed not to progress to the second statement until the first thread has completed and set `x_ready`.
  - There cannot be an interleaving of the steps in which the actions `x = 42` and `r1 = x` are adjacent.
- Declaring a variable as a synchronization variable:
  - Ensures that the variable is accessed indivisibly.
  - Prevents both the compiler and the hardware from reordering memory accesses in ways that are visible to the program and could break it.

<table>
<thead>
<tr>
<th></th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global:</strong> int x;</td>
<td></td>
<td>atomic&lt;bool&gt; x_ready;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Thread 1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = 42;</td>
<td></td>
<td>while (!x_ready) {}</td>
</tr>
<tr>
<td>x_ready = true;</td>
<td></td>
<td>r1 = x;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using synchronization variables to ensure sequentially consistent execution

- Declaring a variable as a synchronization variable
  - Ensures that the variable is accessed indivisibly
  - Prevents both the compiler and the hardware from reordering memory accesses in ways that are visible to the program and could break it
  - In practice, this requires the compiler to obey extra constraints and to generate special code to prevent potential hardware optimizations that could re-order the time to access the variables in memory (e.g., cache)

- The program is free from data races
  - Thread 2 is guaranteed not to progress to the second statement until the first thread has completed and set \( x_{\text{ready}} \). There cannot be an interleaving of the steps in which the actions \( x = 42 \) and \( r1 = x \) are adjacent.

- This ensures a sequentially consistent execution, guarantees that \( r1 = 42 \) at program’s end

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 42; )</td>
<td>( \text{while (!x\textunderscore ready)} ) {}</td>
</tr>
<tr>
<td>( x_{\text{ready}} = \text{true}; )</td>
<td>( r1 = x; )</td>
</tr>
</tbody>
</table>
Visibility

• Changes to variables made by one thread are guaranteed to be visible to other threads under certain conditions only
  ‣ A writing thread *releases* a synchronization lock and a reading thread subsequently *acquires* that same lock
  ‣ If a variable is declared as *atomic*

atomic<bool> ready = false;
int answer = 0

All the memory contents seen by T1, before it wrote to ready, must be visible to T2, after it reads the value true for ready.

Sequential consistency in action

• Thread 2 can only print “42”
• The assignment to `ready` doesn’t return a reference, but rather, the return type (bool)

```cpp
atomic<bool> ready;
int answer; // not atomic

void thread1() {
    answer=42;
    ready=true;
}

void thread2() {
    if (ready)
        print answer;
}
```
How visibility works

• A writing thread releases a synchronization lock and a reading thread subsequently acquires that same lock
  ‣ Releasing a lock flushes all writes from the thread’s working memory, acquiring a lock forces a (re)load of the values of accessible variables
  ‣ While lock actions provide exclusion only for the operations performed within a synchronized block, these memory effects are defined to cover all variables used by the thread performing the action
• If a variable is declared as atomic
  ‣ Any value written to it is flushed and made visible by the writer thread before the writer thread performs any further memory operation.
  ‣ Readers must reload the values of volatile fields upon each access
• As a thread terminates, all written variables are flushed to main memory.
• If a thread uses join to synchronize on the termination of another thread, then it’s guaranteed to see the effects made by that thread
Sequently consistency in practice

• Too expensive to guarantee sequentially consistency all the time
  ‣ Code transformations made by the compiler
  ‣ Instruction reordering in modern processors
  ‣ Write buffers in processors

• In short, different threads perceive that memory references are reordered
Caveats

- The memory model guarantees that a particular update to a particular variable made by one thread will **eventually** be visible to another.
- **But eventually can be an arbitrarily long time**
  - Long stretches of code in threads that use no synchronization can be hopelessly out of sync with other threads with respect to values of fields.
  - Shall not write loops waiting for values written by other threads unless the fields are atomic or accessed via synchronization.
- **But: guarantees made by the memory model are weaker than most programmers intuitively expect, and are also weaker than those typically provided by any given C++ implementation.**
- Rules do not require visibility failures across threads, they merely allow these failures to occur.
- Not using synchronization in multithreaded code doesn't guarantee safety violations, it just allows them.
- Detectable visibility failures might not arise in practice.
- Testing for freedom from visibility-based errors impractical, since such errors might occur extremely rarely, or only on platforms you do not have access to, or only on those that have not even been built yet.
Summary: why do we need a memory model?

• When one thread changes memory then there needs to be a definite order to those changes, as seen by other threads
• Ensure that multithreaded programs are *portable*: they will run correctly on different hardware
• Clarify which optimizations will or will not break our code
  ‣ Compiler optimizations can move code
  ‣ Hardware scheduler executes instructions out of order
Acquire and release

• Why can the program tolerate non-atomic reads and writes? (Listing 5.2, Williams, p. 120)
• How are the happens-before relationships established?

1. std::vector<int> data;
2. std::atomic<bool> data_ready(false);
3. void reader_thread() {
4.     while(!data_ready.load())
5.         std::this_thread::sleep(std::millseconds(1));
6.     std::cout << “The answer=” << data[0] << std::endl;
7. }
8. void writer_thread() {
9.     data.push_back(42);
10.    data_ready=true;
11. }
Which *happens-before* relationships established?

1. `std::vector<int> data;`
2. `std::atomic<bool> data_ready(false);`
3. `void reader_thread() {`
4. `while(!data_ready.load())`
5. `std::this_thread::sleep(std::milliseconds(1));`
6. `std::cout << "The answer="<< data[0] << std::endl;`
7. `}
8. void writer_thread() {`
9. `data.push_back(42);`
10. `data_ready=true;`
11. `}

A. Wr @ (9) h-b Wr @ (10)
B. Rd@ (4) h-b Rd @ (6)
C. Wr @ (9) h-b Rd@ (6)
D. A & B only
E. A, B & C
Today’s lecture

• C++ memory model
• Synchronization variables
• Implementing Synchronization
• SSE vector processing
How do we implement a synchronization variable?

- We said that when one thread writes to a synchronization variable such as an atomic..
- .. and another thread sees that write, …
- …. the first thread is telling the second about all of the contents of memory up until it performed the write to that variable
- How does that thread tell those other threads to synchronize with those variables?

```
x = 42;
atomic <bool> ready=true;
while (!x_ready){}
cout << data << endl;
```

Thread 1

```
x = 42;
atomic <bool> ready=true;
```

Thread 2

```
while (!x_ready){}
cout << data << endl;
```

All the memory contents seen by T1, before it wrote to ready, must be visible to T2, after it see that the value of read is true
Memory fences

• The thread needs to flush all variables to memory, i.e. it synchronizes them

• We implement flushing operations with a special fence instruction, e.g. MFENCE

• “A serializing operation guaranteeing that every load and store instruction that precedes, in program order, the MFENCE instruction is globally visible before any load or store instruction that follows the MFENCE instruction is globally visible.”

Intel 64 & IA32 architectures software developer manual, vol 3a
http://goo.gl/SrdKS2

• Also see www.cl.cam.ac.uk/~pes20/cpp/cpp0x mappings.html

```cpp
atomic<bool> ready;
int answer = 32
ready=true
```
Implementing synchronization primitives

• But there is more to the story: we have only ensured visibility
• What about atomicity, used to build mutexes & other synch variables?
• We use a special machine instruction, e.g. CMPXCHG, that implements *Compare and Swap (CAS)*
  jfdube.wordpress.com/2011/11/30/understanding-atomic-operations
• Do atomically: compare contents of memory location loc to expected; if they are the same, modify the location with newval

Do atomically:
CAS (*loc, expected , newval ):  
  if (*loc== expected ) {
    *loc= newval;
    return 0;
  }
else
  return 1

A CAS implementation of a mutex

1 = UNLOCKED, 0 = LOCKED

`Lock( *mutex ) { while (CAS ( *mutex , 1, 0)) ; }`

`Unlock( *mutex ) { *mutex = 1; }`
Building an atomic counter

- Let’s implement an atomic integer counter that exports two operations, plus a constructor
  - getValue()
  - incr()

```cpp
class AtomicCounter{
  private:
    std::atomic<int> _ctr;
  public:
    int getCtr() { return _ctr; }
    AtomicCounter(){ _ctr = 0; }
    int incr()
    {
        int oldCtr = getCtr();
        while (CAS(&_ctr, &oldCtr, oldCtr + 1) )
            oldCtr = getCtr();
        return oldCtr + 1;
    }
}
```

CAS (*loc, expected, newval):
  if (*loc == expected ) {
    *loc = newval;
    return 0;
  }
  else
    return 1
Implementing self scheduling

- C++ atomic<T> not guaranteed to be lock free, but probably more efficient than mutexes

```c
#include <openmp.h>

boolean getChunk(int& mymin){
    #pragma omp critical // Inefficient
    { // Critical Sect
        k = _counter;
        _counter += _chunk;
    }
    if ( k > (_n – _chunk) // not past last chunk
        return false;
    mymin = k;
    return true;
}
```

```c
#include <atomic.h>

atomic<int> _counter=0;

boolean getChunk(int& mymin){
    mymin = _counter.fetch_add(_chunk)
    if (mymin > (_n – _chunk)) // not past last chunk
        return false;
    else return true;
}
```
Today’s lecture

• C++ memory model
• Synchronization variables
• Implementing Synchronization
• SSE vector processing
Improving performance with SSE

• We’ve seen how we can apply multithreading to speed up the cardiac simulator

• But there is another kind of parallelism available to us: SSE
Hardware Control Mechanisms

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 (Sony Playstation)
- Reduced performance on data dependent or irregular computations

```latex
\begin{align*}
\forall i = 0 : n-1 \\
\text{if } ( x[i] < 0 ) \text{ then } \\
y[i] &= x[i] \\
\text{else } \\
y[i] &= \sqrt{x[i]} \\
\end{align*}
```

```latex
\begin{align*}
\forall i = 0 : n-1 \\
p[i] &= a[i] \star b[i] \\
\text{forall } i = 0 : n-1 \\
x[i] &= y[i] + z[ K[i] ] \\
\end{align*}
```
Are SIMD processors general purpose?

A. Yes

B. No
What kind of parallelism does multithreading provide?

A. MIMD
B. SIMD
Streaming SIMD Extensions

- SIMD instruction set on short vectors
- SSE: SSE3 on Bang, but most will need only SSE2
- Bang : 8x128 bit vector registers (newer cpus have 16)

\[
\text{for } i = 0: N-1 \{ \ p[i] = a[i] \times b[i]; \}\]

4 doubles
8 floats, ints etc
SSE Architectural support

• SSE2, SSE3, SSE4, AVX
• Vector operations on short vectors: add, subtract, 128 bit load store
• SSE2+: 16 XMM registers (128 bits)
• These are in addition to the conventional registers and are treated specially
• Vector operations on short vectors: add, subtract, Shuffling (handles conditionals)
• Data transfer: load/store
• See the Intel intrinsics guide: software.intel.com/sites/landingpage/IntrinsicsGuide
• May need to invoke compiler options depending on level of optimization


C++ intrinsics

- C++ functions and datatypes that map directly onto 1 or more machine instructions
- Supported by all major compilers
- The interface provides 128 bit data types and operations on those datatypes
  - \_m128 (float)
  - \_m128d (double)
- Data movement and initialization
  - mm_load_pd (aligned load)
  - mm_store_pd
  - mm_loadu_pd (unaligned load)
- Data may need to be aligned

```cpp
__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);
}
```
How do we vectorize?

• Original code

```c
double a[N], b[N], c[N];
for (i=0; i<N; i++) {
    a[i] = sqrt(b[i] / c[i]);
}
```

• Identify vector operations, reduce loop bound

```c
for (i = 0; i < N; i+=2)
    a[i:i+1] = vec_sqrt(b[i:i+1] / c[i:i+1]);
```

• The vector instructions

```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);
}
```
Performance

• Without SSE vectorization : 0.777 sec.
• With SSE vectorization : 0.454 sec.
• Speedup due to vectorization: x1.7
• $PUB/Examples/SSE/Vec

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);
}
The assembler code

double *a, *b, *c
for (i=0; i<N; i++) {
    a[i] = sqrt(b[i] / c[i]);
}

.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);
}

.L12:
movsd xmm0, QWORD PTR [r12+rbx]
divsd xmm0, QWORD PTR [r13+0+rbx]
sqrtsd xmm1, xmm0
ucomisd xmm1, xmm1 // checks for illegal sqrt
jp .L30
movsd QWORD PTR [rbp+0+rbx], xmm1
add rbx, 8  # ivtmp.135
cmp rbx, 16384
jne .L12
What prevents vectorization

- **Interrupted flow out of the loop**
  ```c
  for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
    maxval = (a[i] > maxval ? a[i] : maxval);
    if (maxval > 1000.0) break;
  }
  
  Loop not vectorized/parallelized: multiple exits
  
- **This loop will vectorize**
  ```c
  for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
    maxval = (a[i] > maxval ? a[i] : maxval);
  }
  ```
# SSE2 Cheat sheet

## Data transfer
- **MOV** {A/U}{SS/PS/SD/PD} xmm, mem/xmm

## Arithmetic
- **ADD** {SS/PS/SD/PD} xmm, mem/xmm
- **SUB** {SS/PS/SD/PD} xmm, mem/xmm
- **MUL** {SS/PS/SD/PD} xmm, mem/xmm
- **DIV** {SS/PS/SD/PD} xmm, mem/xmm
- **SQRT** {SS/PS/SD/PD} mem/xmm
- **MAX** {SS/PS/SD/PD} mem/xmm
- **MIN** {SS/PS/SD/PD} mem/xmm

## Compare
- **CMP** {SS/PS/SD/PD}

### Notes
- 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

---

Krste Asanovic & Randy H. Katz

Scott B. Baden / CSE 160 / Wi '16