CS 61C: Great Ideas in Computer Architecture
Thread-Level Parallelism (TLP) and OpenMP

Instructor:
Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa13

Agenda

• Review: Intel SSE Intrinsics
• Multiprocessors
• Administrivia
• Threads
• Technology Break
• OpenMP
• And in Conclusion, ...
Agenda

• Review: Intel SSE Intrinsics
  • Multiprocessors
  • Administrivia
  • Threads
  • Technology Break
  • OpenMP
  • And in Conclusion, ...

Review

• SIMD Parallelism via Intel SSE Instructions
• Use of SSE intrinsics to get access to assembly instructions from C code
• Laying data out in memory to provide aligned access for SSE loads and stores
Example: 2 x 2 Matrix Multiply  
(Part 1 of 2)

```c
#include <stdio.h>
// header file for SSE compiler intrinsics
#include <emmintrin.h>

// NOTE: vector registers will be represented in comments as v1 = [ a | b ]
// where v1 is a variable of type __m128d and a, b are doubles

int main(void) {
    // allocate A,B,C aligned on 16-byte boundaries
    double B[4] __attribute__((aligned(16)));
    double C[4] __attribute__((aligned(16)));
    int lda = 2;
    int i = 0;
    // declare several 128-bit vector variables
    __m128d c1, c2, a, b1, b2;
    // Initialize A, B, C for example
    /* A = 
     * 1 0
     * 0 1
     */
    /* B = 
     * 1 3
     * 2 4
     */
    B[0] = 1.0; B[1] = 2.0; B[2] = 3.0; B[3] = 4.0;
    /* C = 
     * 0 0
     * 0 0
     */
    C[0] = 0.0; C[1] = 0.0; C[2] = 0.0; C[3] = 0.0;

    for (i = 0; i < 2; i++) {
        /* a = 
         * i = 0: [a_{11} | a_{21}]
         * i = 1: [a_{12} | a_{22}]
         */
        a = _mm_load_pd(A + i * lda);
        /* b1 = 
         * i = 0: [b_{11} | b_{12}]
         * i = 1: [b_{21} | b_{22}]
         */
        b1 = _mm_load1_pd(B + i * lda);
        /* b2 = 
         * i = 0: [b_{12} | b_{12}]
         * i = 1: [b_{22} | b_{22}]
         */
        b2 = _mm_load1_pd(B + i + 1 * lda);
        c1 = _mm_add_pd(c1, _mm_mul_pd(a, b1));
        c2 = _mm_add_pd(c2, _mm_mul_pd(a, b2));
    }

    // store c1,c2 back into C for completion
    _mm_store_pd(C + 0 * lda, c1);
    _mm_store_pd(C + 1 * lda, c2);

    // print C
    printf("%g,%g
%g,%g\n", C[0], C[2], C[1], C[3]);
    return 0;
}
```

Example: 2 x 2 Matrix Multiply  
(Part 2 of 2)

```c
// used aligned loads to set
// c1 = [c_{11} | c_{21}]
// c2 = [c_{12} | c_{22}]
// c1 = _mm_load_pd(C*0*lda);
// c2 = _mm_add_pd(c1, _mm_mul_pd(a,b1));
// c2 = _mm_add_pd(c2, _mm_mul_pd(a,b2));
```

```
// print C
printf("%g,%g\n%g,%g\n", C[0], C[2], C[1], C[3]);
// return 0;
```
Inner loop from gcc –O -S

L2: movapd (%rax,%rsi), %xmm1  //Load aligned A[i,i+1]->m1
movddup (%rdx), %xmm0  //Load B[j], duplicate->m0
mulpd %xmm1, %xmm0  //Multiply m0*m1->m0
addpd %xmm0, %xmm3  //Add m0+m3->m3
movddup 16(%rdx), %xmm0  //Load B[j+1], duplicate->m0
mulpd %xmm0, %xmm1  //Multiply m0*m1->m0
addpd %xmm1, %xmm2  //Add m1+m2->m1
addq $16, %rax  // rax+16 -> rax (i+=2)
addq $8, %rdx  // rdx+8 -> rdx (j+=1)
cmpq $32, %rax  // rax == 32?
jne L2  // jump to L2 if not equal
movapd %xmm3, (%rcx)  //store aligned m3 into C[k,k+1]
movapd %xmm2, (%rdi)  //store aligned m2 into C[l,l+1]

Agenda

• Review: Intel SSE Intrinsics
• Multiprocessors
• Administrivia
• Threads
• Technology Break
• OpenMP
• And in Conclusion, ...
New-School Machine Structures
(It’s a bit more complicated!)

- Parallel Requests
  Assigned to computer
e.g., Search “Katz”

- Parallel Threads
  Assigned to core
e.g., Lookup, Ads

- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions

- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words

- Hardware descriptions
  All gates @ one time

- Programming Languages
  Hardware descriptions

Simple Multiprocessor

Processor 0
Control
Datapath
PC
Registers
(AlU)

Processor 1
Control
Datapath
PC
Registers
(AlU)

Memory
Bytes

Input
Output
I/O-Memory Interfaces

Processor 0 Memory Accesses
Processor 1 Memory Accesses
Multiprocessor Execution Model

- Each processor has its own PC and executes an independent stream of instructions (MIMD)
- Different processors can access the same memory space
  - Processors can communicate via shared memory by storing/loading to/from common locations
- Two ways to use a multiprocessor:
  1. Deliver high throughput for independent jobs via job-level parallelism
  2. Improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel-processing program

Use term core for processor ("Multicore") because "Multiprocessor Microprocessor" too redundant

Transition to Multicore

Data partially collected by M. Horowitz, F. Labonte, G. Shacham, K. Olukotun, L. Hammond
Parallelism Only Path to Higher Performance

- Sequential processor performance not expected to increase much, and might go down
- If want apps with more capability, have to embrace parallel processing (SIMD and MIMD)
- In mobile systems, use multiple cores and GPUs
- In warehouse-scale computers, use multiple nodes, and all the MIMD/SIMD capability of each node

Multiprocessors and You

- Only path to performance is parallelism
  - Clock rates flat or declining
  - SIMD: 2X width every 3-4 years
    - 128b wide now, 256b 2011, 512b in 2014, 1024b in 2018?
  - MIMD: Add 2 cores every 2 years: 2, 4, 6, 8, 10, ...
- Key challenge is to craft parallel programs that have high performance on multiprocessors as the number of processors increase – i.e., that scale
  - Scheduling, load balancing, time for synchronization, overhead for communication
- Project 3: fastest code on 8-core computers
  - 2 chips/computer, 4 cores/chip
Potential Parallel Performance (assuming SW can use it)

<table>
<thead>
<tr>
<th>Year</th>
<th>Cores</th>
<th>SIMD bits /Core</th>
<th>Core * SIMD bits</th>
<th>Peak DP FLOPs/Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>2003</td>
<td>MIMD 2</td>
<td>SIMD 128</td>
<td>256 4</td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td>+2/ 4</td>
<td>128</td>
<td>512 8</td>
<td></td>
</tr>
<tr>
<td>2007</td>
<td>2yrs 6</td>
<td>128</td>
<td>768 12</td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td>8</td>
<td>128</td>
<td>1024 16</td>
<td></td>
</tr>
<tr>
<td>2011</td>
<td>10</td>
<td>256</td>
<td>2560 40</td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>12</td>
<td>256</td>
<td>3072 48</td>
<td></td>
</tr>
<tr>
<td>2015</td>
<td>2.5x14</td>
<td>8x 512</td>
<td>7168 112</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>16</td>
<td>512</td>
<td>8192 128</td>
<td></td>
</tr>
<tr>
<td>2019</td>
<td>18</td>
<td>1024</td>
<td>18432 288</td>
<td></td>
</tr>
<tr>
<td>2021</td>
<td>20</td>
<td>1024</td>
<td>20480 320</td>
<td></td>
</tr>
</tbody>
</table>

Multiprocessor Caches

- Memory is a performance bottleneck even with one processor
- Use caches to reduce bandwidth demands on main memory
- Each core has a local private cache holding data it has accessed recently
- Only cache misses have to access the shared common memory
Shared Memory and Caches

• What if?
  – Processors 1 and 2 read Memory[1000] (value 20)

Now:
  – Processor 0 writes Memory[1000] with 40
Keeping Multiple Caches Coherent

- Architect’s job: shared memory => keep cache values coherent
- Idea: When any processor has cache miss or writes, notify other processors via interconnection network
  - If only reading, many processors can have copies
  - If a processor writes, invalidate any other copies
- Write transactions from one processor “snoop” tags of other caches using common interconnect
  - Invalidate any “hits” to same address in other caches
  - If hit is to dirty line, other cache has to write back first!

Shared Memory and Caches

- Example, now with cache coherence
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40
Flashcard Quiz: Which statement is true?

• Using write-through caches removes the need for cache coherence
• Every processor store instruction must check contents of other caches
• Most processor load and store accesses only need to check in local private cache
• Only one processor can cache any memory location at one time

Agenda

• Review: Intel SSE Intrinsics
• Multiprocessors
• Administrivia
• Threads
• Technology Break
• OpenMP
• And in Conclusion, …
Midterm Results

CS61C Fa13 Midterm Distribution

Project 2 Part 1

Number of Students

0% 15% 30% 45% 60% 75% 90% 100%
Intel delays key post-PC processor

Pushes 14nm Broadwell chip out by a quarter, posts respectable Q3 results but disappoints with

**Intel Reports Third-Quarter Revenue of $13.5 Billion, Net Income of $3.0 Billion**

Posted by IntelPR in Intel Newsroom on Oct 15, 2013 1:02:49 PM

- Total revenue up 5 percent sequentially, flat year-over-year
- Record Data Center Group revenue of $2.9 billion, up 12 percent year-over-year
- Launched 4th Generation Intel® Core™ products enabling faster, innovative tablet and 2-in-1 designs
- More than forty 22nm products introduced for ultra-mobile device, networking, storage, and server market segments

SANTA CLARA, Calif. October 15, 2013 — Intel Corporation today reported third-quarter revenue of $13.5 billion, operating income of $3.5 billion, net income of $3.0 billion and EPS of $0.58. The company generated approximately $5.7 billion in cash from operations, paid dividends of $1.1 billion, and used $536 million to repurchase 24 million shares of stock.

"The third quarter came in as expected, with modest growth in a tough environment," said Intel CEO Brian Krzanich. "We're executing on our strategy to offer an increasingly broad and diverse product portfolio that spans key growth segments, operating systems and form factors. Since August we have introduced more than 40 new products for market segments from the Internet-of-Things to datacenters, with an increasing focus on ultra-mobile devices and 2 in 1 systems."

Broadwell is important to help Intel put clear water between its own
Cache Coherency Tracked by Block

- Suppose block size is 32 bytes
- Suppose Processor 0 reading and writing variable X, Processor 1 reading and writing variable Y
- Suppose in X location 4000, Y in 4012
- What will happen?
Coherency Tracked by Cache Line

- Block ping-pongs between two caches even though processors are accessing disjoint variables
- Effect called *false sharing*
- How can you prevent it?

Fourth “C” of Cache Misses: *Coherence Misses*

- Misses caused by coherence traffic with other processor
- Also known as *communication* misses because represents data moving between processors working together on a parallel program
- For some parallel programs, coherence misses can dominate total misses
Agenda

• Review: Intel SSE Intrinsics
• Multiprocessors
• Administrivia
• Threads
• Technology Break
• OpenMP
• And in Conclusion, ...

Threads

• *Thread*: unit of work described by a sequential flow of instructions
• Each thread has a PC + processor registers and accesses the shared memory
• Each processor provides one (or more) *hardware* threads that actively execute instructions
• Operating system multiplexes multiple *software* threads onto the available hardware threads
Operating System Threads

Give the illusion of many active threads by time-multiplexing hardware threads among software threads

• Remove a software thread from a hardware thread by interrupting its execution and saving its registers and PC into memory
  – Also if one thread is blocked waiting for network access or user input
• Can make a different software thread active by loading its registers into processor and jumping to its saved PC

Hardware Multithreading

• Basic idea: Processor resources are expensive and should not be left idle
• Long memory latency to memory on cache miss?
• Hardware switches threads to bring in other useful work while waiting for cache miss
• Cost of thread context switch must be much less than cache miss latency
• Put in redundant hardware so don’t have to save context on every thread switch:
  – PC, Registers
• Attractive for apps with abundant TLP
  – Commercial multi-user workloads
Hardware Multithreading

- Two copies of PC and Registers inside processor hardware
- Looks like two processors to software (hardware thread 0, hardware thread 1)
- Control logic decides which thread to execute an instruction from next

Multithreading vs. Multicore

- Multithreading => Better Utilization
  - ≈1% more hardware, 1.10X better performance?
  - Share integer adders, floating-point adders, caches (L1 I $, L1 D$, L2 cache, L3 cache), Memory Controller
- Multicore => Duplicate Processors
  - ≈50% more hardware, ≈2X better performance?
  - Share outer caches (L2 cache, L3 cache), Memory Controller
Randy’s Mac Air

• /usr/sbin/sysctl -a | grep hw\.
  hw.model = MacBookAir5,1
  hw.physicalcpu: 2
  hw.logicalcpu: 4
  hw.cpufrequency = 2,000,000,000
  hw.physmem = 2,147,483,648

Machines in (old) 61C Lab

• /usr/sbin/sysctl -a | grep hw\.
  hw.model = MacPro4,1
  hw.physicalcpu: 8
  hw.logicalcpu: 16
  hw.cpufrequency = 2,260,000,000
  hw.physmem = 2,147,483,648

Therefore, should try up to 16 threads to see if performance gain even though only 8 cores
Agenda

- Review: Intel SSE Intrinsics
- Multiprocessors
- Administrivia
- Threads
- Technology Break
- OpenMP
- And in Conclusion, ...

100s of (Mostly Dead) Parallel Programming Languages

<table>
<thead>
<tr>
<th>ActorScript</th>
<th>Concurrent Pascal</th>
<th>JoCaml</th>
<th>Orc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>Concurrent ML</td>
<td>Join</td>
<td>Oz</td>
</tr>
<tr>
<td>Afnix</td>
<td>Concurrent Haskell</td>
<td>Java</td>
<td>Pict</td>
</tr>
<tr>
<td>Alef</td>
<td>Curry</td>
<td>Joule</td>
<td>Reia</td>
</tr>
<tr>
<td>Alice</td>
<td>CUDA</td>
<td>Joyce</td>
<td>SALS</td>
</tr>
<tr>
<td>APL</td>
<td>E</td>
<td>LabVIEW</td>
<td>Scala</td>
</tr>
<tr>
<td>Axum</td>
<td>Eiffel</td>
<td>Limbo</td>
<td>SISAL</td>
</tr>
<tr>
<td>Chapel</td>
<td>Erlang</td>
<td>Linda</td>
<td>SR</td>
</tr>
<tr>
<td>Cilk</td>
<td>Fortan 90</td>
<td>MultiLisp</td>
<td>Stackless Python</td>
</tr>
<tr>
<td>Clean</td>
<td>Go</td>
<td>Modula-3</td>
<td>SuperPascal</td>
</tr>
<tr>
<td>Clojure</td>
<td>Io</td>
<td>Occam</td>
<td>VHDL</td>
</tr>
<tr>
<td>Concurrent C</td>
<td>Janus</td>
<td>occam-n</td>
<td>XC</td>
</tr>
</tbody>
</table>
OpenMP

• OpenMP is an API used for multi-threaded, shared memory parallelism
  – Compiler Directives (inserted into source code)
  – Runtime Library Routines (called from your code)
  – Environment Variables (set in your shell)

• Portable
• Standardized
• Easy to compile: `cc -fopenmp name.c`

Simple Parallelization

```
for (i=0; i<max; i++) zero[i] = 0;
```

– For loop must have canonical shape for OpenMP to parallelize it
  • Necessary for run-time system to determine loop iterations
– No premature exits from the loop allowed
  • i.e., No break, return, exit, goto statements
Fork/Join Parallelism

- Start out executing the program with one master thread
- Master thread *forks* worker threads as enter parallel code
- Worker threads *join* (die or suspend) at end of parallel code

OpenMP Extends C with Pragmas

- Pragmas are a mechanism C provides for non-standard language extensions
  - `#pragma description`
- Commonly implemented pragmas:
  - structure packing, symbol aliasing, floating-point exception modes
- Good mechanism for OpenMP because compilers that don't recognize a pragma are supposed to ignore them
  - Runs on sequential computer even with embedded pragmas
The Parallel `for` Pragma

```c
#pragma omp parallel for
for (i=0; i<max; i++) zero[i] = 0;
```

- Master thread creates additional threads, each with a separate execution context
- Master thread becomes part of team of parallel threads inside parallel block

Controlling Number of Threads

- How many threads will OpenMP create?
  - Can set via clause in parallel pragma:
    ```c
    #pragma omp parallel for num_threads(NUM_THREADS)
    ```
  - or can set via explicit call to runtime function:
    ```c
    #include <omp.h> /* OpenMP header file. */
    omp_set_num_threads(NUM_THREADS);
    ```
  - or via `NUM_THREADS` an environment variable, usually set in your shell to the number of processors in computer running program
  - `NUM_THREADS` includes the master thread
What Kind of Threads?

• OpenMP threads are operating system threads.
• OS will multiplex requested OpenMP threads onto available hardware threads.
• Hopefully each get a real hardware thread to run on, so no OS-level time-multiplexing.
• But other tasks on machine can also use hardware threads!
• Be careful when timing results for project 3!

Invoking Parallel Threads

```c
#include <omp.h>
#pragma omp parallel
{
  int ID = omp_get_thread_num();
  foo(ID);
}
```

• Each thread executes a copy of the code within the structured block
• OpenMP intrinsic to get Thread ID number: `omp_get_thread_num()`
Data Races and Synchronization

- Two memory accesses form a *data race* if from different threads to same location, and at least one is a write, and they occur one after another
- If there is a data race, result of program can vary depending on chance (which thread first?)
- Avoid data races by synchronizing writing and reading to get deterministic behavior
- Synchronization done by user-level routines that rely on hardware synchronization instructions
- (more later)

Controlling Sharing of Variables

- Variables declared outside parallel block are shared by default.
- `private(x)` statement makes new private version of variable `x` for each thread.

```c
int i, temp, A[], B[];
#pragma omp parallel for private(temp)
for (i=0; i<N; i++)
{ temp = A[i]; A[i] = B[i]; B[i] = temp; }
```
\[ \pi \]

3. 
141592653589793238462643383279502
884197169399375105820974944592307
816406286208998628034825342117067
982148086513282306647093844609550
582231725359408128481117450284102
...

Calculating \( \pi \)

**Numerical Integration**

Mathematically, we know that:

\[
\int_0^1 \frac{4.0}{1+x^2} \, dx = \pi
\]

We can approximate the integral as a sum of rectangles:

\[
\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi
\]

Where each rectangle has width \( \Delta x \) and height \( F(x_i) \) at the middle of interval \( i \).
Sequential Calculation of $\pi$ in C

```c
#include <stdio.h>/* Serial Code */
static long num_steps = 100000; double step;
void main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=1; i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = sum/num_steps;
    printf ("pi = %6.12f\n", pi);
}
```

OpenMP Version (with bug)

```c
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
    int i; double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    #pragma omp parallel private (x)
    {
        int id = omp_get_thread_num();
        for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREADS)
        {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0; i<NUM_THREADS; i++)
    pi += sum[i];
    printf ("pi = %6.12f\n", pi / num_steps);
}
Experiment

• Run with NUM_THREADS = 1 multiple times
• Run with NUM_THREADS = 2 multiple times
• What happens?

OpenMP Version (with bug)

```c
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
    int i; double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    #pragma omp parallel private (x)
    {
        int id = omp_get_thread_num();
        for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREADS)
        {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0; i<NUM_THREADS; i++)
        pi += sum[i];
    printf("pi = %6.12f\n", pi/num_steps);
}
```

Note: loop index variable \( i \) is shared between threads
OpenMP Reduction

- **Reduction**: specifies that 1 or more variables that are private to each thread are subject of reduction operation at end of parallel region: `reduction(operation:var)` where
  - **Operation**: operator to perform on the variables (var) at the end of the parallel region
  - **Var**: One or more variables on which to perform scalar reduction.

```c
#include <omp.h>
#include <stdio.h>

// static long num_steps = 100000;
double step;
void main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel for private(x) reduction(+:sum)
    for (i=1; i<= num_steps; i++)
    {
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = sum / num_steps;
    printf ("pi = %6.8f\n", pi);
}
```

Note: Don’t have to declare for loop index variable i private, since that is default
Agenda

- Review: Intel SSE Intrinsics
- Multiprocessors
- Administrivia
- Threads
- Technology Break
- OpenMP
- And in Conclusion, ...

And in Conclusion, ...

- Sequential software is slow software
  - SIMD and MIMD only path to higher performance
- Multiprocessor/Multicore uses Shared Memory
  - Cache coherency implements shared memory even with multiple copies in multiple caches
  - False sharing a concern; watch block size!
- Multithreading increases utilization, Multicore more processors (MIMD)
- OpenMP as simple parallel extension to C
  - Threads, Parallel for, private, critical sections, ...
  - ≈ C: small so easy to learn, but not very high level and its easy to get into trouble