

In the News

Facebook gets OK to build 'Burger Shack' and 'BBQ House' at Menlo Park campus

The commission expressed enthusiasm for the project, particularly its potential to keep employees from hopping into their cars at lunch time. The BBQ and burger eateries will supplement two cafes on the campus.

MercuryNews.com

By Jason Green
Daily News Staff Writer
Posted: 10/07/2011

Tsuruoaka said he doesn't expect workers to leave the campus if they have on-site food options.
"The food for employees, it's provided without charge," he said. "So we would expect employees to stay on campus to take advantage of that."

10/12/11

Fall 2011 -- Lecture #20

1

CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Lecture 20 – Thread Level Parallelism

Instructors:
Michael Franklin
Dan Garcia

<http://inst.eecs.Berkeley.edu/~cs61c/Fa11>

10/11/11

Fall 2011 -- Lecture #20

2

Review

- Flynn Taxonomy of Parallel Architectures
 - SIMD: Single Instruction Multiple Data
 - MIMD: Multiple Instruction Multiple Data
 - SISD: Single Instruction Single Data (unused)
 - MISD: Multiple Instruction Single Data
- Intel SSE SIMD Instructions
 - One instruction fetch that operates on multiple operands simultaneously
 - 64/128 bit XMM registers

10/12/11

Fall 2011 -- Lecture #20

3

Agenda

- SSE Instructions in C
- Multiprocessor
- Cache Coherence

10/12/11

Fall 2011 -- Lecture #20

4

Intel SSE Intrinsics

- Intrinsics are C functions and procedures for putting in assembly language, including SSE instructions
 - With intrinsics, can program using these instructions indirectly
 - One-to-one correspondence between SSE instructions and intrinsics

10/12/11

Fall 2011 -- Lecture #20

5

Example SSE Intrinsics

Intrinsics:	Corresponding SSE instructions:
• Vector data type: <code>_mm128d</code>	
• Load and store operations: <code>_mm_load_pd</code> <code>_mm_store_pd</code> <code>_mm_loadu_pd</code> <code>_mm_storeu_pd</code>	MOVAPD/aligned, packed double MOVAPD/aligned, packed double MOVUPD/unaligned, packed double MOVUPD/unaligned, packed double
• Load and broadcast across vector <code>_mm_load1_pd</code>	MOVSD + shuffling/duplicating
• Arithmetic: <code>_mm_add_pd</code> <code>_mm_mul_pd</code>	ADDPD/add, packed double MULPD/multiple, packed double

10/12/11

Fall 2011 -- Lecture #20

6

Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

$$C_{i,j} = (A \times B)_{i,j} = \sum_{k=1}^2 A_{i,k} \times B_{k,j}$$

$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$	x	$\begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$	=	$\begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$	$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$ $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$ $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$ $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$
$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	x	$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$	=	$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$	$C_{1,1} = 1*1 + 0*2 = 1$ $C_{1,2} = 1*3 + 0*4 = 3$ $C_{2,1} = 0*1 + 1*2 = 2$ $C_{2,2} = 0*3 + 1*4 = 4$

10/12/11 Fall 2011 - Lecture #20 7

Example: 2 x 2 Matrix Multiply

- Using the XMM registers
 - 64-bit/double precision/two doubles per XMM reg

C_1	$\begin{bmatrix} C_{1,1} & C_{2,1} \\ C_{1,2} & C_{2,2} \end{bmatrix}$	Stored in memory in Column order
A	$\begin{bmatrix} A_{1,1} & A_{2,1} \\ A_{1,2} & A_{2,2} \end{bmatrix}$	
B ₁	$\begin{bmatrix} B_{1,1} & B_{1,1} \\ B_{1,2} & B_{1,2} \end{bmatrix}$	
B ₂	$\begin{bmatrix} B_{2,1} & B_{2,1} \\ B_{2,2} & B_{2,2} \end{bmatrix}$	

10/11/11 Fall 2011 - Lecture #20 8

Example: 2 x 2 Matrix Multiply

- Initialization

C ₁	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$
C ₂	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$

10/11/11 Fall 2011 - Lecture #20 9

Example: 2 x 2 Matrix Multiply

- Initialization
- | = 1

C ₁	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$	
C ₂	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$	
A	$\begin{bmatrix} A_{1,1} & A_{2,1} \\ A_{1,2} & A_{2,2} \end{bmatrix}$	_mm_load_pd: Load 2 doubles into XMM reg, Stored in memory in Column order
B ₁	$\begin{bmatrix} B_{1,1} & B_{1,1} \\ B_{1,2} & B_{1,2} \end{bmatrix}$	_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)
B ₂	$\begin{bmatrix} B_{2,1} & B_{2,1} \\ B_{2,2} & B_{2,2} \end{bmatrix}$	

10/11/11 Fall 2011 - Lecture #20 10

Example: 2 x 2 Matrix Multiply

- First iteration intermediate result
- | = 1

C ₁	$\begin{bmatrix} 0+A_{1,1}B_{1,1} & 0+A_{2,1}B_{1,1} \\ 0+A_{1,1}B_{1,2} & 0+A_{2,1}B_{1,2} \end{bmatrix}$	<code>c1 = _mm_add_pd(c1, _mm_mul_pd(a,b1));</code> <code>c2 = _mm_add_pd(c2, _mm_mul_pd(a,b2));</code> SSE instructions first do parallel multiplies and then parallel adds in XMM registers
C ₂	$\begin{bmatrix} 0+A_{1,1}B_{1,1} & 0+A_{2,1}B_{1,1} \\ 0+A_{1,1}B_{1,2} & 0+A_{2,1}B_{1,2} \end{bmatrix}$	
A	$\begin{bmatrix} A_{1,1} & A_{2,1} \\ A_{1,2} & A_{2,2} \end{bmatrix}$	_mm_load_pd: Stored in memory in Column order
B ₁	$\begin{bmatrix} B_{1,1} & B_{1,1} \\ B_{1,2} & B_{1,2} \end{bmatrix}$	_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)
B ₂	$\begin{bmatrix} B_{2,1} & B_{2,1} \\ B_{2,2} & B_{2,2} \end{bmatrix}$	

10/11/11 Fall 2011 - Lecture #20 11

Example: 2 x 2 Matrix Multiply

- First iteration intermediate result
- | = 2

C ₁	$\begin{bmatrix} 0+A_{1,1}B_{1,1} & 0+A_{2,1}B_{1,1} \\ 0+A_{1,1}B_{1,2} & 0+A_{2,1}B_{1,2} \end{bmatrix}$	<code>c1 = _mm_add_pd(c1, _mm_mul_pd(a,b1));</code> <code>c2 = _mm_add_pd(c2, _mm_mul_pd(a,b2));</code> SSE instructions first do parallel multiplies and then parallel adds in XMM registers
C ₂	$\begin{bmatrix} 0+A_{1,1}B_{1,1} & 0+A_{2,1}B_{1,1} \\ 0+A_{1,1}B_{1,2} & 0+A_{2,1}B_{1,2} \end{bmatrix}$	
A	$\begin{bmatrix} A_{1,2} & A_{2,2} \\ A_{1,1} & A_{2,1} \end{bmatrix}$	_mm_load_pd: Stored in memory in Column order
B ₁	$\begin{bmatrix} B_{2,1} & B_{2,1} \\ B_{2,2} & B_{2,2} \end{bmatrix}$	_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)
B ₂	$\begin{bmatrix} B_{2,1} & B_{2,1} \\ B_{2,2} & B_{2,2} \end{bmatrix}$	

10/11/11 Fall 2011 - Lecture #20 12

Example: 2 x 2 Matrix Multiply

- Second iteration intermediate result

	$C_{1,1}$	$C_{2,1}$
C_1	$A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$	$A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
C_2	$A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$	$A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$
	$C_{1,2}$	$C_{2,2}$

```

c1 = _mm_add_pd(c1, _mm_mul_pd(a,b1));
c2 = _mm_add_pd(c2, _mm_mul_pd(a,b2));
            
```

SSE instructions first do parallel multiplies and then parallel adds in XMM registers
- I = 2

A	$A_{1,2}$	$A_{2,2}$
---	-----------	-----------

`_mm_load_pd`: Stored in memory in Column order

B ₁	$B_{2,1}$	$B_{2,1}$
B ₂	$B_{2,2}$	$B_{2,2}$

`_mm_load1_pd`: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)

10/11/11 Fall 2011 - Lecture #20 13

Example: 2 x 2 Matrix Multiply (Part 1 of 2)

```

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

// Initialize A, B, C for example
/* A =
  1 0
  0 1
  A[0] = 1.0; A[1] = 0.0; A[2] = 0.0; A[3] = 1.0;
  */
/* 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;
  */

int main(void) {
    // allocate A,B,C aligned on 16-byte boundaries
    double A[4] __attribute__((aligned(16)));
    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;
            
```

10/11/11 Fall 2011 - Lecture #20 14

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

```

// used aligned loads to set
// c1 = [c_11 | c_21]
c1 = _mm_load_pd(C+0*lda);
// c2 = [c_12 | c_22]
c2 = _mm_load_pd(C+1*lda);

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_11]
     i = 1: [b_21 | b_21]
     */
    b1 = _mm_load1_pd(B+i*0*lda);
    /* b2 =
     i = 0: [b_12 | b_12]
     i = 1: [b_22 | b_22]
     */
    b2 = _mm_load1_pd(B+i+1*lda);
            
```

```

/* c1 =
 i = 0: [c_11 + a_11*b_11 | c_21 + a_21*b_11]
 i = 1: [c_11 + a_21*b_21 | c_21 + a_22*b_21]
 */
c1 = _mm_add_pd(c1, _mm_mul_pd(a,b1));
/* c2 =
 i = 0: [c_12 + a_11*b_12 | c_22 + a_21*b_12]
 i = 1: [c_12 + a_21*b_22 | c_22 + a_22*b_22]
 */
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\n%%g,%%g\n",C[0],C[2],C[1],C[3]);
return 0;
            
```

10/11/11 Fall 2011 - Lecture #20 15

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->m1
addpd    %xmm1, %xmm2 //Add m1+m2->m2
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]
            
```


10/11/11 Fall 2011 - Lecture #20 16

You Are Here!


- 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 functioning in parallel at same time

Hardware

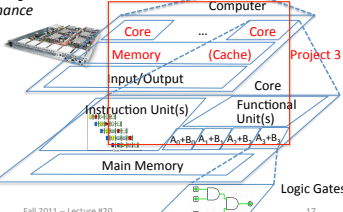
Warehouse Scale Computer



Smart Phone



Computer



Core ... Core

Memory (Cache)

Input/Output

Core

Instruction Unit(s)

Functional Unit(s)

Main Memory

Logic Gates

10/11/11 Fall 2011 - Lecture #20 17

Review

- Intel SSE SIMD Instructions
 - One instruction fetch that operates on multiple operands simultaneously
- SSE Instructions in C
 - Can embed the SSE machine instructions directly into C programs through the use of intrinsics

10/11/11 Fall 2011 - Lecture #20 18

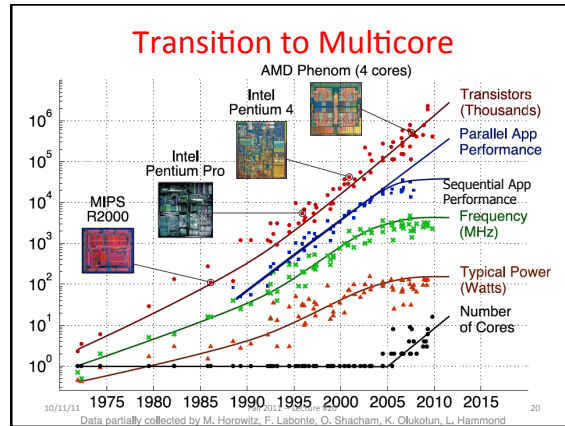
Parallel Processing: Multiprocessor Systems (MIMD)

- Multiprocessor (MIMD):** a computer system with at least 2 processors

- Deliver high throughput for independent jobs via job-level parallelism
- Improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel processing program

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

10/11/11 Fall 2011 - Lecture #20 19



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, ...
- A 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
- Will explore this further in labs and projects

10/11/11 Fall 2011 - Lecture #20 21

Parallel Performance Over Time

Year	Cores	SIMD bits /Core	Core * SIMD bits	Peak DP FLOPs
2003	2	128	256	4
2005	4	128	512	8
2007	6	128	768	12
2009	8	128	1024	16
2011	10	256	2560	40
2013	12	256	3072	48
2015	14	512	7168	112
2017	16	512	8192	128
2019	18	1024	18432	288
2021	20	1024	20480	320

10/11/11 Fall 2011 - Lecture #20 22

Multiprocessor Key Questions

- Q1 - How do they share data?
- Q2 - How do they coordinate?
- Q3 - How many processors can be supported?

10/11/11 Fall 2011 - Lecture #20 23

Shared Memory Multiprocessor (SMP)

- Q1 - Single address space shared by all processors/cores
- Q2 - Processors coordinate/communicate through shared variables in memory (via loads and stores)
 - Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one processor at a time
- All multicore computers today are SMP

10/11/11 Fall 2011 - Lecture #20 24

Example: Sum Reduction

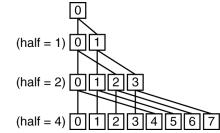
- Sum 100,000 numbers on 100 processor SMP
 - Each processor has ID: $0 \leq P_n \leq 99$
 - Partition 1000 numbers per processor
 - Initial summation on each processor


```
sum[Pn] = 0;
for (i = 1000*Pn;
     i < 1000*(Pn+1); i = i + 1)
    sum[Pn] = sum[Pn] + A[i];
```
- Now need to add these partial sums
 - Reduction: divide and conquer
 - Half the processors add pairs, then quarter, ...
 - Need to synchronize between reduction steps

10/11/11 Fall 2011 - Lecture #20 25

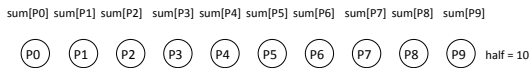
Example: Sum Reduction

```
half = 100;
repeat
    synch();
    if (half%2 != 0 && Pn == 0)
        sum[0] = sum[0] + sum[half-1];
        /* Conditional sum needed when half is odd;
        Processor0 gets missing element */
    half = half/2; /* dividing line on who sums */
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```



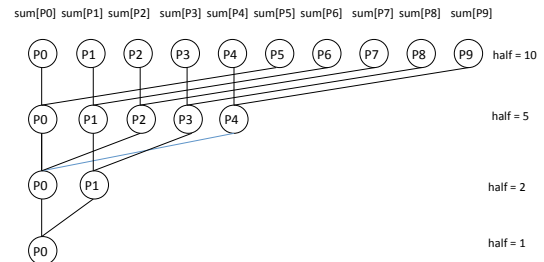
10/11/11 Fall 2011 - Lecture #20 26

An Example with 10 Processors



10/11/11 Fall 2011 - Lecture #20 27

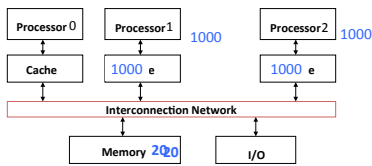
An Example with 10 Processors



10/11/11 Fall 2011 - Lecture #20 28

Shared Memory and Caches

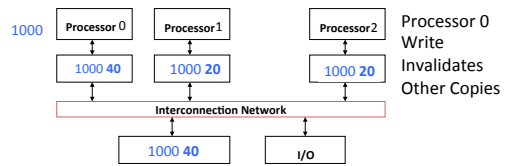
- What if?
 - Processors 1 and 2 read Memory[1000] (value 20)



10/11/11 Fall 2011 - Lecture #20 29

Shared Memory and Caches

- What if?
 - Processors 1 and 2 read Memory[1000]
 - Processor 0 writes Memory[1000] with 40



10/11/11 Fall 2011 - Lecture #20 30

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 all other copies
- Shared written result can "ping-pong" between caches

10/11/11

Fall 2011 – Lecture #20

31

How Does HW Keep \$ Coherent?

- Each cache tracks state of each *block* in cache:
 1. *Shared*: up-to-date data, other caches may have a copy
 2. *Modified*: up-to-date data, changed (dirty), no other cache has a copy, OK to write, memory out-of-date

10/11/11

Fall 2011 – Lecture #20

32

2 Optional Performance Optimizations of Cache Coherency via new States

- Each cache tracks state of each *block* in cache:
 3. *Exclusive*: up-to-date data, no other cache has a copy, OK to write, memory up-to-date
 - Avoids writing to memory if block replaced
 - Supplies data on read instead of going to memory
 4. *Owner*: up-to-date data, other caches may have a copy (they must be in Shared state)
 - Only cache that supplies data on read instead of going to memory

10/11/11

Fall 2011 – Lecture #20

33

Name of Common Cache Coherency Protocol: MOESI

- Memory access to cache is either
 - Modified (in cache)
 - Owned (in cache)
 - Exclusive (in cache)
 - Shared (in cache)
 - Invalid (not in cache)

10/11/11

Fall 2011 – Lecture #20

34

So, In Conclusion...

- Sequential software is slow software
 - SIMD and MIMD only path to higher performance
- SSE Intrinsics allow SIMD instructions to be invoked from C programs
- Multiprocessor (Multicore) uses Shared Memory (single address space)
- Cache coherency implements shared memory even with multiple copies in multiple caches
 - More on this next time

10/12/11

Fall 2011 – Lecture #20

35