

inst.eecs.berkeley.edu/~cs61c
CS61C : Machine Structures

Lecture 20
Thread Level Parallelism



Senior Lecturer SOE Dan Garcia
www.cs.berkeley.edu/~ddgarcia

Intel Xeon Phi




1997: THE FIRST INTEL™ TERAFLOP COMPUTER OPERATED OF 9,298 INTEL PROCESSORS 72 SERVER CABINETS

THE INTEL® XEON™ PHI™ COPROCESSOR will provide 1 TERAFLOP OF PERFORMANCE 1 PCIe SLOT

intel.com/xeonphi

Review


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



CS61C L20 Thread Level Parallelism I (2) Garcia, Spring 2014 © UCS

Intel SSE Intrinsic


- 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



CS61C L20 Thread Level Parallelism I (3) Garcia, Spring 2014 © UCS

Example SSE Intrinsics

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




CS61C L20 Thread Level Parallelism I (4) Garcia, Spring 2014 © UCS

Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

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

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




CS61C L20 Thread Level Parallelism I (5) Garcia, Spring 2014 © UCS

Example: 2 x 2 Matrix Multiply

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

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



CS61C L20 Thread Level Parallelism I (6) Garcia, Spring 2014 © UCS

Example: 2 x 2 Matrix Multiply

- Initialization

C ₁	0	0
C ₂	0	0

CS61C L20 Thread Level Parallelism (7) Garcia, Spring 2014 © UC Berkeley

Example: 2 x 2 Matrix Multiply

- Initialization
- l = 1

A	A _{1,1}	A _{2,1}	<small>_mm_load_pd: Load 2 doubles into XMM reg, Stored in memory in Column order</small>
B ₁	B _{1,1}	B _{1,1}	<small>_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)</small>
B ₂	B _{1,2}	B _{1,2}	

CS61C L20 Thread Level Parallelism (8) Garcia, Spring 2014 © UC Berkeley

Example: 2 x 2 Matrix Multiply

- First iteration intermediate result
- l = 1

C ₁	0+A _{1,1} B _{1,1}	0+A _{2,1} B _{1,1}	<small>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</small>
C ₂	0+A _{1,1} B _{1,2}	0+A _{2,1} B _{1,2}	

A	A _{1,1}	A _{2,1}	<small>_mm_load_pd: Stored in memory in Column order</small>
B ₁	B _{1,1}	B _{1,1}	<small>_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)</small>
B ₂	B _{1,2}	B _{1,2}	

CS61C L20 Thread Level Parallelism (9) Garcia, Spring 2014 © UC Berkeley

Example: 2 x 2 Matrix Multiply

- First iteration intermediate result
- l = 2

C ₁	0+A _{1,1} B _{1,1}	0+A _{2,1} B _{1,1}	<small>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</small>
C ₂	0+A _{1,1} B _{1,2}	0+A _{2,1} B _{1,2}	

A	A _{1,2}	A _{2,2}	<small>_mm_load_pd: Stored in memory in Column order</small>
B ₁	B _{2,1}	B _{2,1}	<small>_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)</small>
B ₂	B _{2,2}	B _{2,2}	

CS61C L20 Thread Level Parallelism (10) Garcia, Spring 2014 © UC Berkeley

Example: 2 x 2 Matrix Multiply

- Second iteration intermediate result
- l = 2

C ₁	0+A _{1,1} B _{1,1}	0+A _{2,1} B _{1,1}	<small>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</small>
C ₂	0+A _{1,1} B _{1,2}	0+A _{2,1} B _{1,2}	

A	A _{1,2}	A _{2,2}	<small>_mm_load_pd: Stored in memory in Column order</small>
B ₁	B _{2,1}	B _{2,1}	<small>_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)</small>
B ₂	B _{2,2}	B _{2,2}	

CS61C L20 Thread Level Parallelism (11) Garcia, Spring 2014 © UC Berkeley

Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

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

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

CS61C L20 Thread Level Parallelism (12) Garcia, Spring 2014 © UC Berkeley

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

```

#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 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;

    // Initialize A, B, C for example
    /* A = (note column order!)
       1 0
       0 1
    */
    A[0] = 1.0; A[1] = 0.0; A[2] = 0.0; A[3] = 1.0;

    /* B = (note column order!)
       1 3
       2 4
    */
    B[0] = 1.0; B[1] = 2.0; B[2] = 3.0; B[3] = 4.0;

    /* C = (note column order!)
       0 0
       0 0
    */
    C[0] = 0.0; C[1] = 0.0; C[2] = 0.0; C[3] = 0.0;
    
```

CS61C L20 Thread Level Parallelism I (13) Garcia, Spring 2014 © UC Berkeley

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 (j = 0; j < 2; j++) {
       /* a =
          i = 0: [a_11 | a_21]
          i = 1: [a_12 | a_22]
       */
       a = _mm_load_pd(A+j*lda);
       /* b1 =
          i = 0: [b_11 | b_12]
          i = 1: [b_21 | b_22]
       */
       b1 = _mm_load1_pd(B+(j+0)*lda);
       /* b2 =
          i = 0: [b_12 | b_12]
          i = 1: [b_22 | b_22]
       */
       b2 = _mm_load1_pd(B+(j+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;
    
```

CS61C L20 Thread Level Parallelism I (14) Garcia, Spring 2014 © UC Berkeley

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]
    
```

CS61C L20 Thread Level Parallelism I (15) Garcia, Spring 2014 © UC Berkeley

You Are Here!

Software

- 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

Project 3

CS61C L20 Thread Level Parallelism I (16) Garcia, Spring 2014 © UC Berkeley

Thoughts about Threads

“Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly non-deterministic, and the job of the programmer becomes one of pruning that nondeterminism.”
— *The Problem with Threads*, Edward A. Lee, UC Berkeley, 2006

CS61C L20 Thread Level Parallelism I (17) Garcia, Spring 2014 © UC Berkeley

Background: Threads

- A **Thread** stands for “thread of execution”, is a single stream of instructions
 - A program / process can **split**, or **fork** itself into separate threads, which can (in theory) execute simultaneously
 - An easy way to describe/think about parallelism
- A single CPU can execute many threads by **Time Division Multiplexing**
- Multithreading** is running multiple threads through the same hardware

CS61C L20 Thread Level Parallelism I (18) Garcia, Spring 2014 © UC Berkeley

Parallel Processing: Multiprocessor Systems (MIMD)

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

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

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

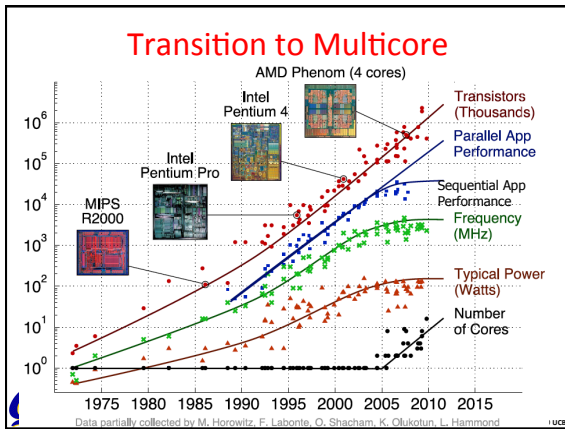
Cal CS61C L20 Thread Level Parallelism | 19 Garcia, Spring 2014 © UC Berkeley

Clicker Question

What significant thing happened in computer architecture around 2005?

- a) CPU heat densities approached nuclear reactors
- b) They started slowing the clock speeds down
- c) Power drain of CPUs hit a plateau
- d) CPU single-core performance hit a plateau
- e) CPU manufacturers started offering only multi-core CPUs for desktops and laptops

Cal 10/14 CS61C L20 Thread Level Parallelism | 20 Fall 2011 -- Lecture #20 Garcia, Spring 2014 © UC Berkeley



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?
 - **Advanced Vector Extensions** are 256-bits wide!
 - 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

Cal CS61C L20 Thread Level Parallelism | 22 Garcia, Spring 2014 © UC Berkeley

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

Cal CS61C L20 Thread Level Parallelism | 23 Garcia, Spring 2014 © UC Berkeley

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
- MIMD uses multithreading to achieve high parallelism

Cal CS61C L20 Thread Level Parallelism | 24 Garcia, Spring 2014 © UC Berkeley