CS 61C: Great Ideas in Computer Architecture *SIMD I*

Instructors:
Krste Asanovic, Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa12

Review

- Although caches are software-invisible, a "cache-aware" performance programmer can improve performance by large factors by changing order of memory accesses
- Three C's of cache misses
  - Compulsory
  - Capacity
  - Conflict

Sources of Cache Misses (3 C's)

- *Compulsory* (cold start, first reference):
  - 1st access to a block, "cold" fact of life, not a lot you can do about it.
    - If running billions of instructions, compulsory misses are insignificant
- *Capacity*:
  - Cache cannot contain all blocks accessed by the program
    - Misses that would not occur with infinite cache
- *Conflict* (collision):
  - Multiple memory locations mapped to the same cache location
    - Misses that would not occur with ideal fully associative cache

Flashcard Quiz: With a fixed cache capacity, what effect does a larger block size have on the 3Cs?

Decreases compulsory, increases conflicts
Increases conflicts
Increases compulsory, decreases conflicts
Decreases conflicts

Flashcard Quiz: With a fixed cache block size, what effect does a larger cache capacity have on the 3Cs?

Increases compulsory, decreases conflicts
Increases conflicts, decreases capacity misses
Decreases compulsory, decreases conflicts
Decreases conflicts, decreases capacity misses

Sources of Cache Misses (3 C's)

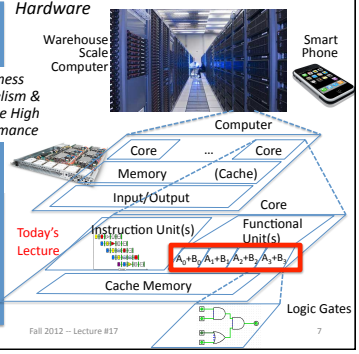
- *Compulsory* (cold start, first reference):
  - 1st access to a block, "cold" fact of life, not a lot you can do about it.
    - If running billions of instructions, compulsory misses are insignificant
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)
- *Capacity*:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size (may increase access time)
  - Or structure software so reuse data in cache before fetching new data
- *Conflict* (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size (may increase hit time)
  - Solution 2: (later in semester) increase associativity (may increase hit time)

New-School Machine Structures (It's a bit more complicated!)

Software | *Hardware*

- 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

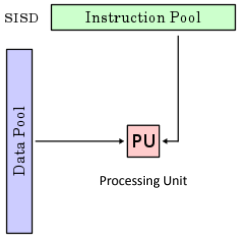
Harness Parallelism & Achieve High Performance



Alternative Kinds of Parallelism: The Programming Viewpoint

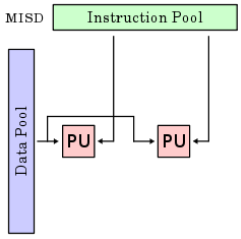
- Job-level parallelism/process-level parallelism
  - Running independent programs on multiple processors simultaneously
  - *Example?*
- Parallel-processing program
  - Single program that runs on multiple processors simultaneously
  - *Example?*

Alternative Kinds of Parallelism: Single-Instruction/Single-Data Stream



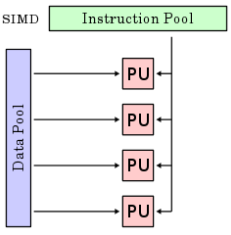
- Single Instruction, Single Data stream (SISD)
  - Sequential computer that exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are traditional uniprocessor machines

Alternative Kinds of Parallelism: Multiple-Instruction/Single-Data Stream



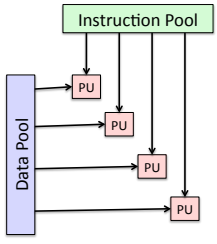
- Multiple-Instruction, Single-Data stream (MISD)
  - Computer that exploits multiple instruction streams against a single data stream for data operations that can be naturally parallelized. For example, certain kinds of array processors.
  - No longer commonly encountered, mainly of historical interest only

Alternative Kinds of Parallelism: Single-Instruction/Multiple-Data Stream



- Single-Instruction, Multiple-Data streams (SIMD or "sim-dee")
  - Computer that exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., Intel SIMD instruction extensions or NVIDIA Graphics Processing Unit (GPU)

Alternative Kinds of Parallelism: Multiple-Instruction/Multiple-Data Streams



- Multiple-Instruction, Multiple-Data streams (MIMD or "mim-dee")
  - Multiple autonomous processors simultaneously executing different instructions on different data.
  - MIMD architectures include multicore and Warehouse-Scale Computers
  - *(Discuss after midterm)*

Flynn* Taxonomy, 1966

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345 (Clovertown)

- In 2012, SIMD and MIMD most common parallelism in architectures – usually both in same system!
- Most common parallel processing programming style: Single Program Multiple Data ("SPMD")
  - Single program that runs on all processors of a MIMD
  - Cross-processor execution coordination through conditional expressions (thread parallelism after midterm)
- SIMD (aka hw-level *data parallelism*): specialized function units, for handling lock-step calculations involving arrays
  - Scientific computing, signal processing, multimedia (audio/ video processing)

*Prof. Michael Flynn, Stanford

Two kinds of Data-Level Parallelism (DLP)

- Lots of data in memory that can be operated on in parallel (e.g., adding together 2 arrays)
- Lots of data on many disks that can be operated on in parallel (e.g., searching for documents)

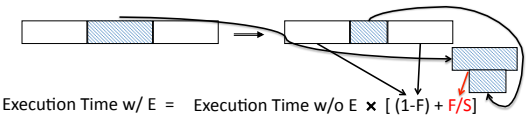
- 2nd/3rd lecture (and 1st project) did DLP across 10s of servers and disks using MapReduce
- Today's lecture (and 3rd project) does Data-Level Parallelism (DLP) in memory

Big Idea: Amdahl's (Heartbreaking) Law

- Speedup due to enhancement E is

$$\text{Speedup w/ E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/ E}}$$

- Suppose that enhancement E accelerates a fraction F (F <1) of the task by a factor S (S>1) and the remainder of the task is unaffected



Execution Time w/ E = Execution Time w/o E × [(1-F) + F/S]

Speedup w/ E = 1 / [(1-F) + F/S]

Big Idea: Amdahl's Law

Speedup =

Example: the execution time of half of the program can be accelerated by a factor of 2. What is the program speed-up overall?

Big Idea: Amdahl's Law

$$\text{Speedup} = \frac{1}{(1 - F) + \frac{F}{S}}$$

Non-speed-up part — Speed-up part

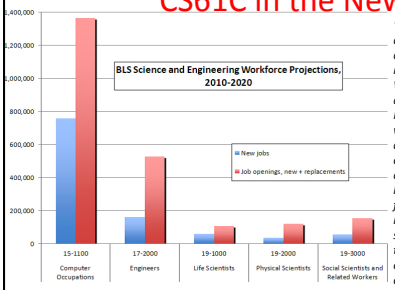
Example: the execution time of half of the program can be accelerated by a factor of 2. What is the program speed-up overall?

$$\frac{1}{0.5 + \frac{0.5}{2}} = \frac{1}{0.5 + 0.25} = 1.33$$

Administrivia

- Lab #6 posted
- Midterm Tuesday Oct 9, 8PM:
  - Two rooms: 1 Pimentel and 2050 LSB
  - Check your room assignment!
  - Covers everything through lecture today
  - Closed book, can bring one sheet notes, both sides
  - Copy of Green card will be supplied
  - No phones, calculators, …; just bring pencils & eraser
  - TA Review: Sun. Oct. 7, 3-5pm, 2050 VLSB

CS61C in the News



'Busy recruiter Nathan Ollestad, director of technology practice at Greythorne, lists his phone number prominently on LinkedIn. When called by a stranger, he answers right away. He has to. No matter how consumed with work he is, Ollestad can't miss a chance to grab precious engineering talent that might be on the other end of the phone line. Grab it, hold onto it, and find the company with just the right perks, incentives, and stimulating projects to entice a talented candidate to sign employment papers. It's not easy. "The industry we're in now is unlike any I've ever been in. It's the most competitive market I've ever seen...," he says.' Smartbear October 2012

"Computer Science is the most popular major at Stanford."
"45% of MIT undergrads enrolled in EECS."
Also now probably most popular major at UCB.

Example #1: Amdahl's Law

Speedup w/ E = 1 / [(1-F) + F/S]

- Consider an enhancement which runs 20 times faster but which is only usable 25% of the time
  Speedup w/ E = 1/(.75 + .25/20) = 1.31
- What if its usable only 15% of the time?
  Speedup w/ E = 1/(.85 + .15/20) = 1.17
- Amdahl's Law tells us that to achieve linear speedup with 100 processors, none of the original computation can be scalar!
- To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less
  Speedup w/ E = 1/(.001 + .999/100) = 90.99

Parallel Speed-up Example

$Z_0 + Z_1 + \ldots + Z_{10}$

$$\begin{bmatrix} X_{1,1} & X_{1,10} \end{bmatrix} + \begin{bmatrix} Y_{1,1} & Y_{1,10} \end{bmatrix}$$
$$\begin{bmatrix} X_{10,1} & X_{10,10} \end{bmatrix} \quad \begin{bmatrix} Y_{10,1} & Y_{10,10} \end{bmatrix}$$

Partition 10 ways and perform on 10 parallel processing units

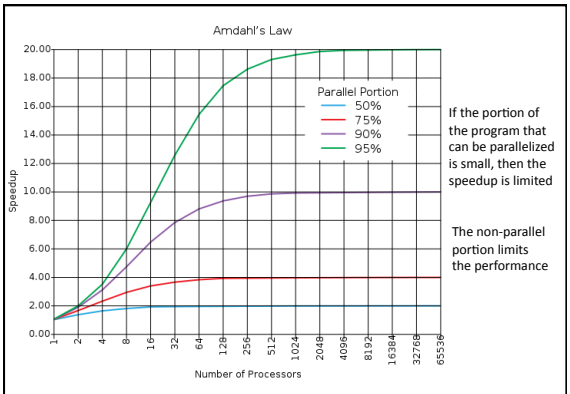
Non-parallel part Parallel part

- 10 "scalar" operations (non-parallelizable)
- 100 parallelizable operations
- 110 operations

Example #2: Amdahl's Law

Speedup w/ E = 1 / [(1-F) + F/S]

- Consider summing 10 scalar variables and two 10 by 10 matrices (matrix sum) on 10 processors
  Speedup w/ E = 1/(.091 + .909/10) = 1/0.1819 = 5.5
- What if there are 100 processors ?
  Speedup w/ E = 1/(.091 + .909/100) = 1/0.10009 = 10.0
- What if the matrices are 100 by 100 (or 10,010 adds in total) on 10 processors?
  Speedup w/ E = 1/(.001 + .999/10) = 1/0.1009 = 9.9
- What if there are 100 processors ?
  Speedup w/ E = 1/(.001 + .999/100) = 1/0.01099 = 91



If the portion of the program that can be parallelized is small, then the speedup is limited

The non-parallel portion limits the performance

Strong and Weak Scaling

- To get good speedup on a multiprocessor while keeping the problem size fixed is harder than getting good speedup by increasing the size of the problem.
  - *Strong scaling*: when speedup can be achieved on a parallel processor without increasing the size of the problem
  - *Weak scaling*: when speedup is achieved on a parallel processor by increasing the size of the problem proportionally to the increase in the number of processors
- Load balancing is another important factor: every processor doing same amount of work
  - Just one unit with twice the load of others cuts speedup almost in half

Suppose a program spends 80% of its time in a square root routine. How much must you speedup square root to make the program run 5 times faster?

Speedup w/ E = 1 / [(1-F) + F/S]

- ☐ 10
- ☐ 20
- ☐ 100
- ☐ None of the Above

27

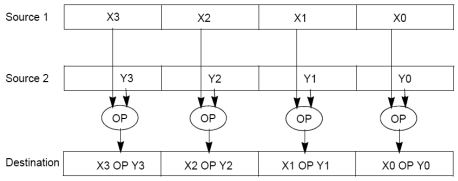
SIMD Architectures

- *Data parallelism*: executing one operation on multiple data streams
- Example to provide context:
  - Multiplying a coefficient vector by a data vector (e.g., in filtering)

    $y[i] := c[i] \times x[i], \ 0 \le i < n$
- Sources of performance improvement:
  - One instruction is fetched & decoded for entire operation
  - Multiplications are known to be independent
  - Pipelining/concurrency in memory access as well

"Advanced Digital Media Boost"

- To improve performance, Intel's SIMD instructions
  - Fetch one instruction, do the work of multiple instructions
  - MMX (MultiMedia eXtension, Pentium II processor family)
  - *SSE (Streaming SIMD Extension, Pentium III and beyond)*



Example: SIMD Array Processing

```
for each f in array
    f = sqrt(f)

for each f in array
{
    load f to the floating-point register
    calculate the square root
    write the result from the register to memory
}
```

```
for each 4 members in array
{
    load 4 members to the SSE register
    calculate 4 square roots in one operation
    store the 4 results from the register to memory
}
```

SIMD style

Data-Level Parallelism and SIMD

- SIMD wants adjacent values in memory that can be operated in parallel
- Usually specified in programs as loops

  for(i=1000; i>0; i=i-1)

  x[i] = x[i] + s;
- How can reveal more data-level parallelism than available in a single iteration of a loop?
- *Unroll loop* and adjust iteration rate

Looping in MIPS

Assumptions:

- $t1 is initially the address of the element in the array with the highest address
- $f0 contains the scalar value s
- 8($t2) is the address of the last element to operate on

CODE:

```
Loop:1. l.d     $f2,0($t1)    ; $f2=array element
     2. add.d   $f10,$f2,$f0  ; add s to $f2
     3. s.d     $f10,0($t1)   ; store result
     4. addui   $t1,$t1,#-8   ; decrement pointer 8 byte
     5. bne     $t1,$t2,Loop ;repeat loop if $t1 != $t2
```

Loop Unrolled

```
Loop: l.d    $f2,0($t1)
      add.d  $f10,$f2,$f0
      s.d    $f10,0($t1)
      l.d    $f4,-8($t1)
      add.d  $f12,$f4,$f0
      s.d    $f12,-8($t1)
      l.d    $f6,-16($t1)
      add.d  $f14,$f6,$f0
      s.d    $f14,-16($t1)
      l.d    $f8,-24($t1)
      add.d  $f16,$f8,$f0
      s.d    $f16,-24($t1)
      addui  $t1,$t1,#-32
      bne    $t1,$t2,Loop
```

NOTE:
1. Only 1 Loop Overhead every 4 iterations
2. This unrolling works if
   loop_limit(mod 4) = 0
3. (Different Registers eliminate stalls in pipeline; we'll see later in course)

Loop Unrolled Scheduled

```
Loop:l.d    $f2,0($t1)
     l.d    $f4,-8($t1)
     l.d    $f6,-16($t1)
     l.d    $f8,-24($t1)
     add.d  $f10,$f2,$f0
     add.d  $f12,$f4,$f0
     add.d  $f14,$f6,$f0
     add.d  $f16,$f8,$f0
     s.d    $f10,0($t1)
     s.d    $f12,-8($t1)
     s.d    $f14,-16($t1)
     s.d    $f16,-24($t1)
     addui  $t1,$t1,#-32
     bne    $t1,$t2,Loop
```

4 Loads side-by-side: Could replace with 4-wide SIMD Load

4 Adds side-by-side: Could replace with 4-wide SIMD Add

4 Stores side-by-side: Could replace with 4-wide SIMD Store

Loop Unrolling in C

- Instead of compiler doing loop unrolling, could do it yourself in C

```
for(i=1000; i>0; i=i-1)
      x[i] = x[i] + s;
```

- Could be rewritten

What is downside of doing it in C?

```
for(i=1000; i>0; i=i-4) {
      x[i]   = x[i]    + s;
      x[i-1] = x[i-1] + s;
      x[i-2] = x[i-2] + s;
      x[i-3] = x[i-3] + s;
      }
```


Generalizing Loop Unrolling

- A loop of **n iterations**
- **k copies** of the body of the loop
- **Assuming (n mod k) ≠ 0**

Then we will run the loop with 1 copy of the body **(n mod k)** times and

with k copies of the body **floor(n/k)** times

- (Will revisit loop unrolling again when get to pipelining later in semester)

Review

- Flynn Taxonomy of Parallel Architectures
  - *SIMD: Single Instruction Multiple Data*
  - *MIMD: Multiple Instruction Multiple Data*
  - SISD: Single Instruction Single Data (sequential machines)
  - MISD: Multiple Instruction Single Data (unused)
- Amdahl's Law
  - Strong versus weak scaling
- SIMD Extensions
  - Exploit data-level parallelism in loops