CS 61C:
Great Ideas in Computer Architecture
Cache Performance and Parallelism

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

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

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 @ one time
- Programming Languages

Harness Parallelism & Achieve High Performance

Hardware

Agenda

- Review
- Cache Performance
- Administrivia
- Parallel Processing
- Technology Break
- Amdahl's Law
- SIMD
- And in Conclusion, …

Agenda

- Review
- Cache Performance
- Administrivia
- Parallel Processing
- Technology Break
- Amdahl's Law
- SIMD
- And in Conclusion, …

Review

- Write-through versus write-back caches
- AMAT = Hit time + Miss rate x Miss penalty
- Larger caches reduce Miss rate via Temporal and Spatial Locality, but can increase Hit time
- Multilevel caches help Miss penalty

Caches Invisible to Software

- Load and store instructions just access large memory (32-bit addresses in MIPS); hardware automatically moves data in and out of cache
- Even if programmer writes applications not knowing about caches, we observe temporal and spatial locality in memory accesses
- Performance improves (over no caches) even when programmer unaware of cache's existence

CPI/Miss Rates/DRAM Access

SpecInt2006 on AMD Barcelona (64KB L1, 512KB L2)

		Data Only	Data Only	Instructions and Data
Name	CPI	L1 D cache misses/1000 instr	L2 D cache misses/1000 instr	DRAM accesses/1000 instr
perl	0.75	3.5	1.1	1.3
bzip2	0.85	11.0	5.8	2.5
gcc	1.72	24.3	13.4	14.8
mcf	10.00	106.8	88.0	88.5
go	1.09	4.5	1.4	1.7
hmmer	0.80	4.4	2.5	0.6
sjeng	0.96	1.9	0.6	0.8
libquantum	1.61	33.0	33.1	47.7
h264avc	0.80	8.8	1.6	0.2
omnetpp	2.94	30.9	27.7	29.8
astar	1.79	16.3	9.2	8.2
xalancbmk	2.70	38.0	15.8	11.4
Median	1.35	13.6	7.5	5.4

Agenda

- Review
- Cache Performance
- Administrivia
- Parallel Processing
- Technology Break
- Amdahl's Law
- SIMD
- And in Conclusion, …

Performance Programming:
Adjust software accesses to improve miss rate

- Now that understand how caches work, can revise program to improve cache utilization
- "Cache-Aware" performance optimizations
  - But code would still work even if no caches present

Performance of Loops over Arrays

- Array performance often limited by memory speed
- OK to access memory in different order as long as get correct result
- Goal: Increase performance by minimizing traffic from cache to memory
  - That is, reduce Miss rate by getting better reuse of data already in cache

Alternate Matrix Layouts in Memory

- A matrix is a 2-D array of elements, but memory addresses are "1-D" (0…MaximumMemoryAddress)
- Conventions for matrix layout
  - by column, or "column major" (Fortran default); A(i,j) at A+i+j*n
  - by row, or "row major" (C default) A[i][j] at A+i*n+j

Column major

0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

Row major

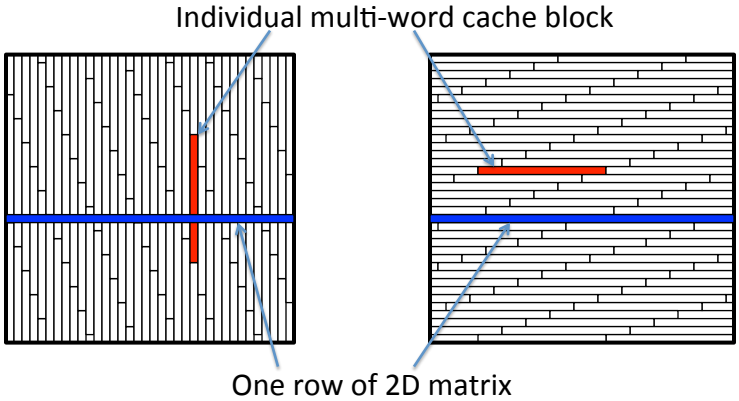
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

How a 4x5 Matrix is stored in memory, red numbers are memory addresses

Cache Blocks in Matrix

Column Major
(as used in FORTRAN)

Row Major
(as used in C)



*Cache Line is alternative name for Cache Entry or Block

Loop Interchange: Flashcard quiz

```
for(j=0; j < N; j++) {
  for(i=0; i < M; i++) {
     x[i][j] = 2 * x[i][j];
  }
}
```

```
for(i=0; i < M; i++) {
  for(j=0; j < N; j++) {
     x[i][j] = 2 * x[i][j];
  }
}
```

What kind of locality does this improve?

Spatial

Temporal

Both

Neither

13

Loop Fusion: Flashcard Quiz

```
for(i=0; i < N; i++)
     a[i] = b[i] * c[i];

for(i=0; i < N; i++)
     d[i] = a[i] * c[i];
```

```
for(i=0; i < N; i++)
{
      a[i] = b[i] * c[i];
      d[i] = a[i] * c[i];
}
```

What kind of locality does this improve?

Spatial

Temporal

Both

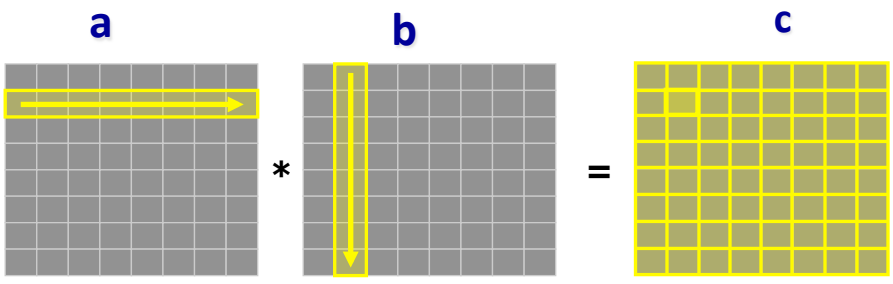
Neither

15

Cache Blocking (aka Cache Tiling)

- "shrink" problem by performing multiple iterations within smaller cache blocks
- Also known as cache *tiling*
- Don't confuse term "cache blocking" with:
  - cache blocks, i.e., individual cache entries or lines
  - (or later, blocking versus non-blocking caches)
- Use Matrix Multiply as example: Next Lab (and Project 3)

Matrix Multiplication



Simplest Algorithm

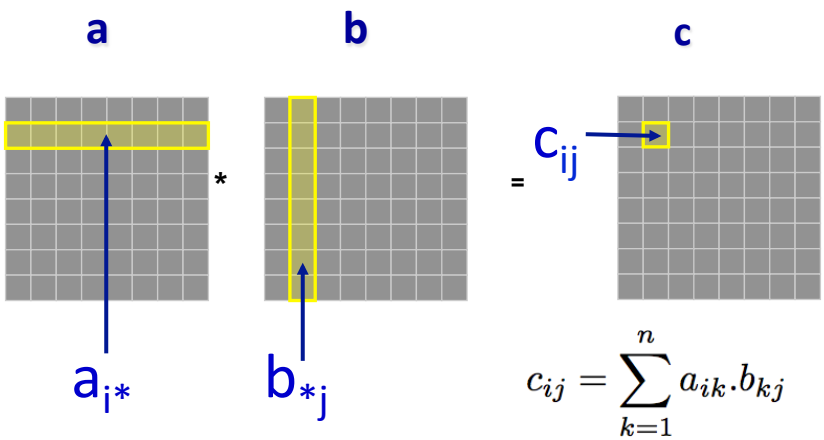
Assumption: the matrices are stored as 2-D NxN arrays

```
for (i=0;i<N;i++)
 for (j=0;j<N;j++)
   for (k=0;k<N;k++)
       c[i][j] += a[i][k] * b[k][j];
```

Advantage: code simplicity

Disadvantage: Marches through memory and caches

Matrix Multiplication

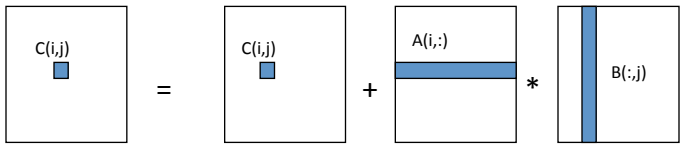


Simple Matrix Multiply - www.youtube.com/watch?v=yl0LTcDIhxc

100 x 100 Matrix, Cache 1000 blocks, 1 word/block

Improving reuse via Blocking: 1st "Naïve" Matrix Multiply

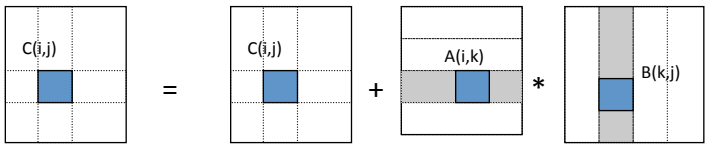
```
{implements C = C + A*B}
for i = 1 to n
 {read row i of A into cache}
  for j = 1 to n
     {read c(i,j) into cache}
     {read column j of B into cache}
     for k = 1 to n
        c(i,j) = c(i,j) + a(i,k) * b(k,j)
     {write c(i,j) back to main memory}
```



Blocked Matrix Multiply

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is called the block size

```
 for i = 1 to N
   for j = 1 to N
 {read block C(i,j) into cache}
 for k = 1 to N
            {read block A(i,k) into cache}
            {read block B(k,j) into cache}
             C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
 {write block C(i,j) back to main memory}
```



Blocked Matrix Multiply - www.youtube.com/watch?v=IFWgwGMMrh0

100 x 100 Matrix, 1000 cache blocks, 1 word/block, block 30x30

Blocked Algorithm

- The blocked version of the i-j-k algorithm is written simply as (A,B,C are submatricies of a, b, c)

```
for (i=0;i<N/r;i++)
  for (j=0;j<N/r;j++)
    for (k=0;k<N/r;k++)
      C[i][j] += A[i][k]*B[k][j]
```

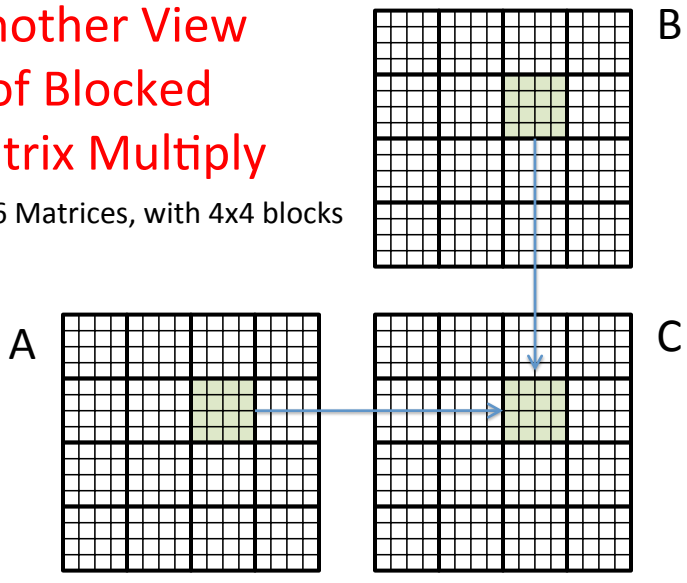
r x r matrix addition

r x r matrix multiplication

  - r = block (sub-matrix) size (Assume r divides N)
  - X[i][j] = a sub-matrix of X, defined by block row i and block column j

Another View of Blocked Matrix Multiply

16x16 Matrices, with 4x4 blocks



Maximum Block Size

- The blocking optimization works only if the blocks fit in cache.
- That is, 3 blocks of size r x r must fit in memory (for A, B, and C)
- M = size of cache (in elements/words)
- We must have: $3r^2 \approx M$, or $r \approx \sqrt{(M/3)}$
- Ratio of cache misses blocked vs. unblocked up to $\approx \sqrt{M}$

Simple Matrix Multiply Whole Thing - www.youtube.com/watch?v=f3-z6t_xIyw

1x1 blocks: 1,020,000 misses: read A once, read B 100 times, read C once

Blocked Matrix Multiply Whole Thing - www.youtube.com/watch?v=tgpmXX3xOrk

30x30 blocks: 90,000 misses = read A and B four times, read C once

"Only" 11X vs 30X Matrix small enough that row of A in simple version fits completely in cache (+ few odds and ends)

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

28

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

30

Agenda

- Review
- Cache Performance
- Administrivia
- Parallel Processing
- Technology Break
- Amdahl's Law
- SIMD
- And in Conclusion, …

Administrivia

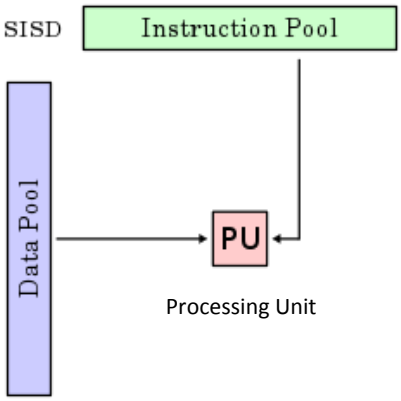
Agenda

- Review
- Cache Performance
- Administrivia
- Parallel Processing
- Technology Break
- Amdahl's Law
- SIMD
- And in Conclusion, …

Alternative Kinds of Parallelism: The Programming Viewpoint

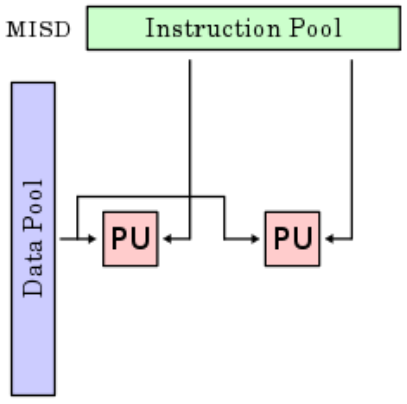
- Job-level parallelism/process-level parallelism
  - Running independent programs on multiple processors simultaneously
  - *Example?*
- Parallel-processing program
  - Single program that runs on multiple processors simultaneously
  - *Example?*

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



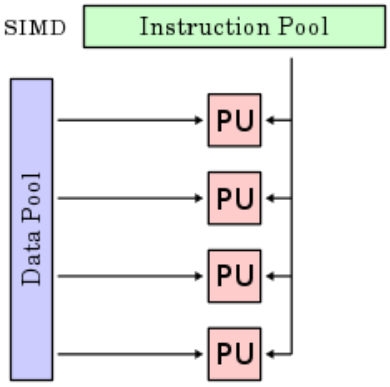
- Single Instruction, Single Data stream (SISD)
  - Sequential computer that exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are traditional uniprocessor machines

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



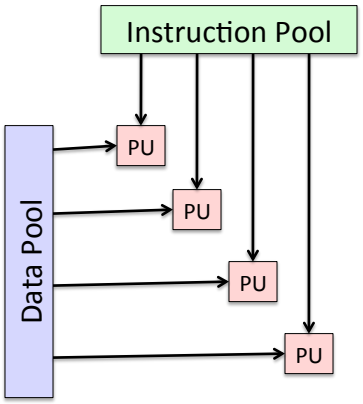
- Multiple-Instruction, Single-Data stream (MISD)
  - Computer that exploits multiple instruction streams against a single data stream for data operations that can be naturally parallelized. For example, certain kinds of array processors.
  - No longer commonly encountered, mainly of historical interest 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 2013, 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 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

Agenda

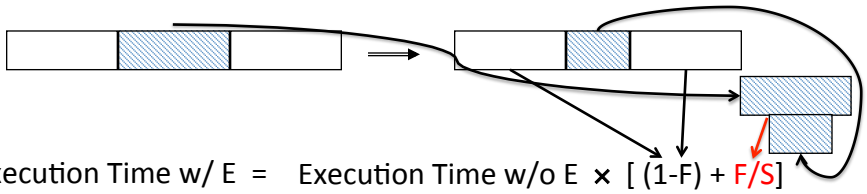
- Review
- Cache Performance
- Administrivia
- Parallel Processing
- Technology Break
- Amdahl's Law
- SIMD
- And in Conclusion, …

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?

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

Example #1: Amdahl's Law

Speedup w/ E =

- Consider an enhancement which runs 20 times faster but which is only usable 25% of the time.

  Speedup w/ E =

- What if its usable only 15% of the time?

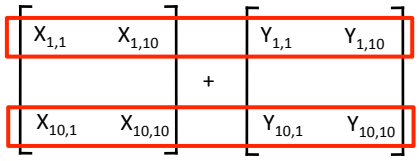
  Speedup w/ E =

- 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 =

Parallel Speed-up Example

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



Partition 10 ways and perform on 10 parallel processing units

Non-parallel part

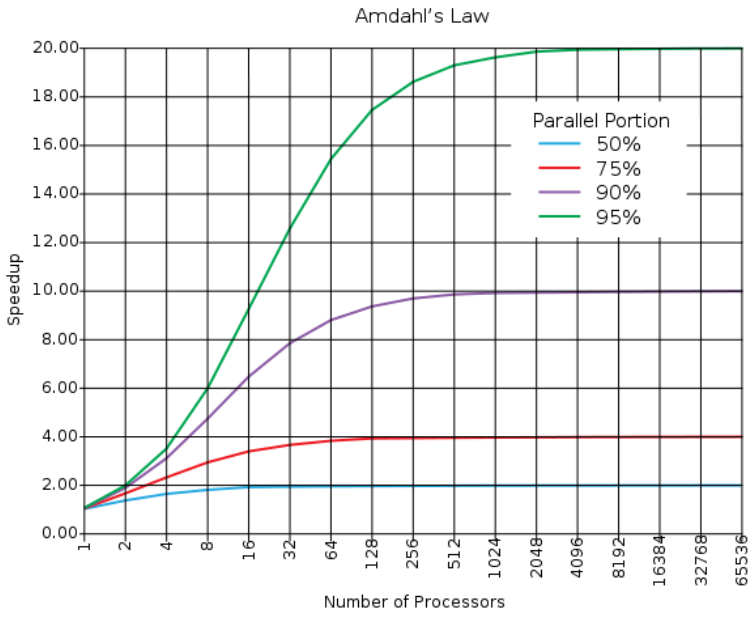
Parallel part

- 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 =
- What if there are 100 processors ?
  Speedup w/ E =
- What if the matrices are 100 by 100 (or 10,010 adds in total) on 10 processors?
  Speedup w/ E =
- What if there are 100 processors ?
  Speedup w/ E =



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

Agenda

- Review
- Cache Performance
- Administrivia
- Parallel Processing
- Technology Break
- Amdahl's Law
- SIMD
- And in Conclusion, …

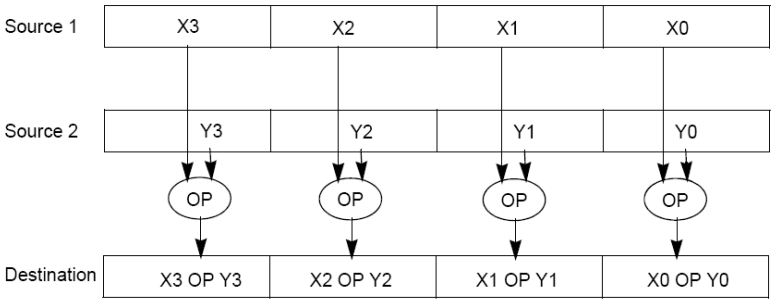
SIMD Architectures

- *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)

    $y[i] := c[i] \times x[i],\ 0 \leq i < n$
- 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)

And in Conclusion, …

- Although caches are software-invisible, a "cache-aware" performance programmer can improve performance by large factors by changing order of memory accesses
- 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