Great Ideas in Computer Architecture

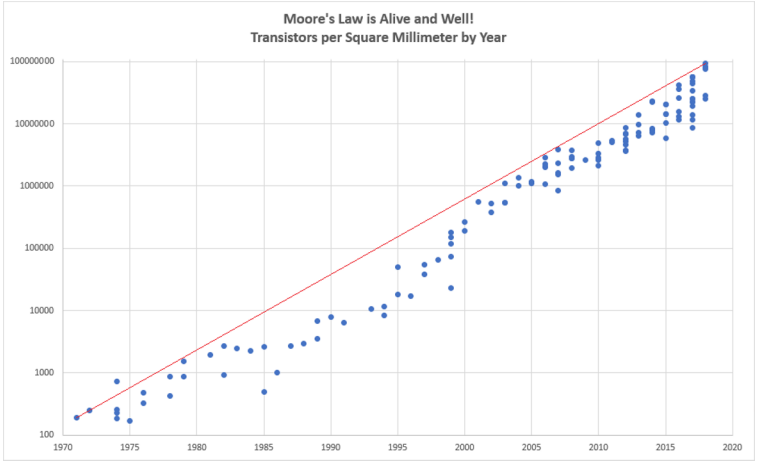
Flynn's Taxonomy and Data-level Parallelism

Instructor: Jenny Song

Agenda

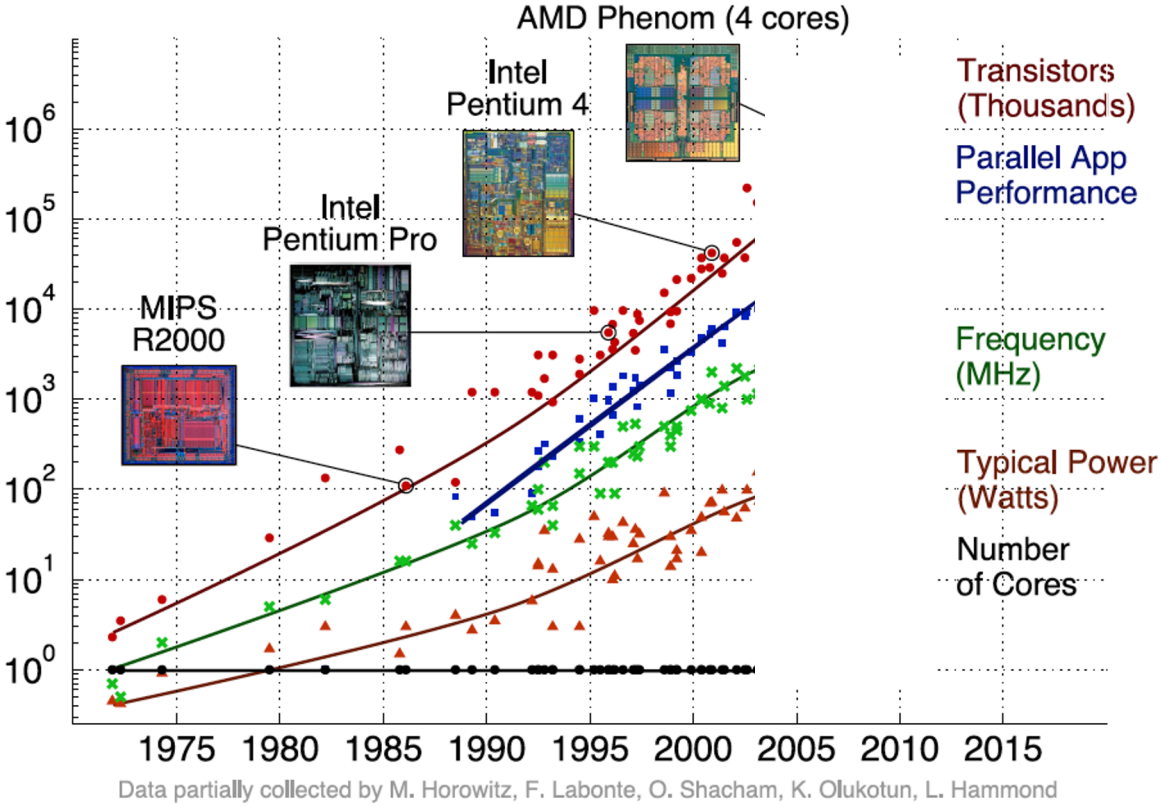
- **Intro**
- Parallelism and Flynn’s Taxonomy
- SIMD Architectures
- Loop Unrolling
- Summary

Moore’s Law

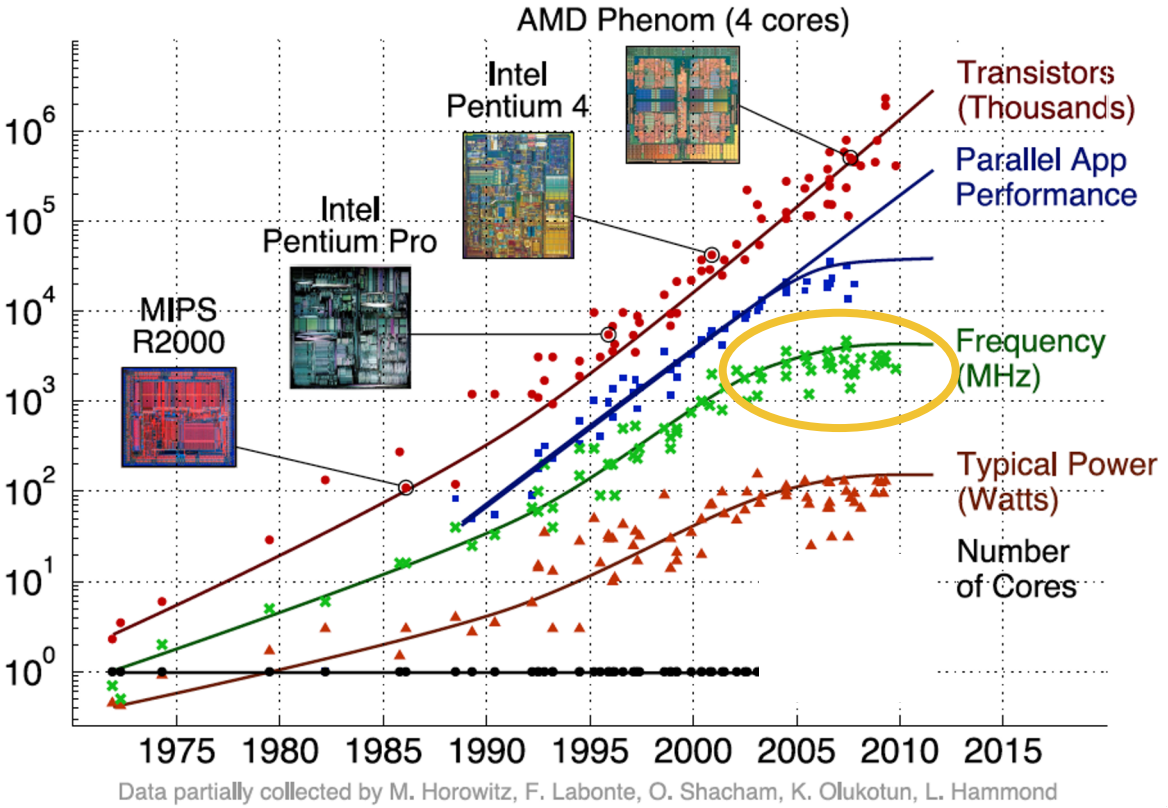


“Every two years, the number of transistors on a chip of a fixed size doubles”

Processors kept getting faster



Then they stopped getting faster



Dennard Scaling

- Moore's Law corollary: As transistors get smaller, the power density stays the same.
  - If Moore's Law holds true, we also get a doubling of "performance per watt" every two years!
  - *Manufacturers could raise the clock frequency between generations without more power consumption*
- This stopped around 2006!
  - Too much current leakage → **difficulty making transistors any faster!**

So… now what?

In summary: we can't make transistors faster due to current leakage, and because of that, *we can't reliably make performance better by waiting for clock speeds to increase.*

- How do we continue to get better performing hardware?

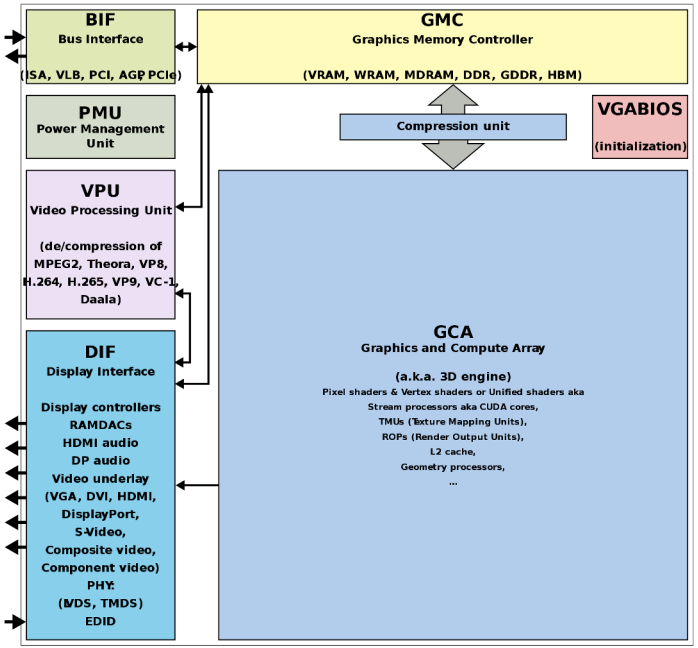
Domain-Specific Hardware

- Hardware designed for a particular workflow or task
- We can plan for a different “worst” and “best” case, and make smarter design decisions for our average use

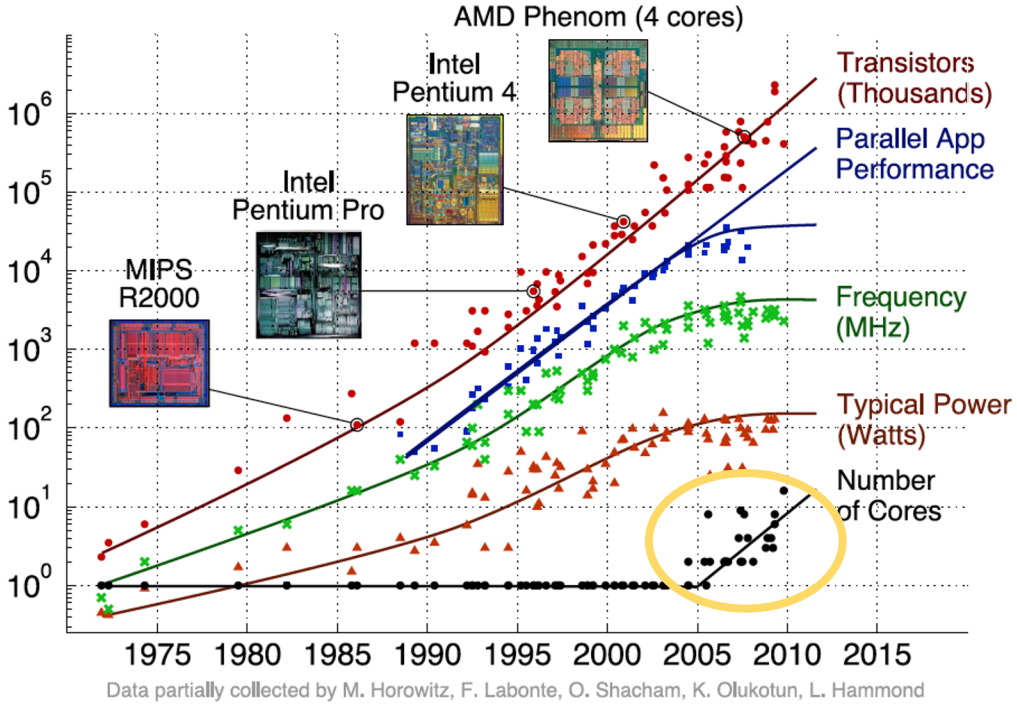
- “Do a few tasks, but extremely well”
  (Hennessy and Patterson)
- Achieve higher efficiency by tailoring the architecture to characteristics of the domain.

GPU's

Highly parallel processing units used (originally) for graphics and image processing



But what if we want to improve general computing?



Exploit Parallelism!

Great Idea #4: Parallelism

Software

- Parallel Requests
  - Assigned to computer
  - e.g. search "Garcia"
- Parallel Threads
  - Assigned to core
  - e.g. lookup, ads
- Parallel Data
  - > 1 data item @ one time
  - e.g. add of 4 pairs of words
- Parallel Instructions
  - > 1 instruction @ one time
  - e.g. 5 pipelined instructions
- Hardware descriptions
  - All gates functioning in parallel at same time

Leverage Parallelism & Achieve High Performance

Already covered

Hardware

Warehouse Scale Computer

Smart Phone

Computer

Core … Core

Memory

Input/Output

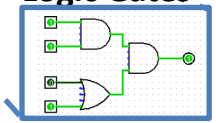
Core

Functional Unit(s)

A_0+B_0 A_1+B_1 A_2+B_2 A_3+B_3

Cache Memory

Logic Gates



Agenda

- **Parallelism and Flynn's Taxonomy**
- SIMD Architectures
- Loop Unrolling
- Summary

Parallelism Analogy

I want to peel 100 potatoes as fast as possible:

- I can learn to peel potatoes faster

  OR

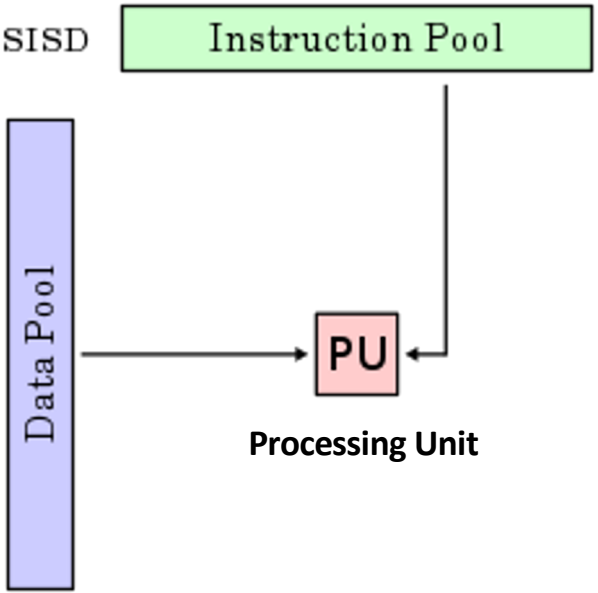
- I can get 99 friends to help me

Any time one result doesn't depend on another, doing the task in parallel can be a big win

Classes of Data-Level Parallelism

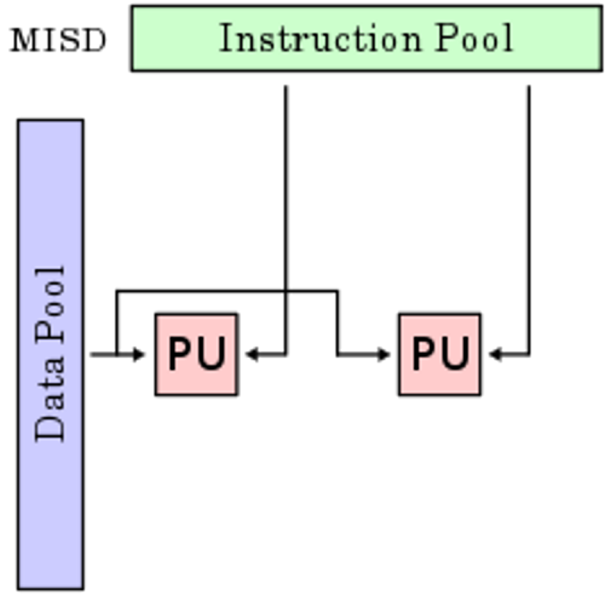
		Data Streams	
		Single	**Multiple**
Instruction Streams	**Single**	???	???
	Multiple	???	???

Single Instruction/Single Data Stream



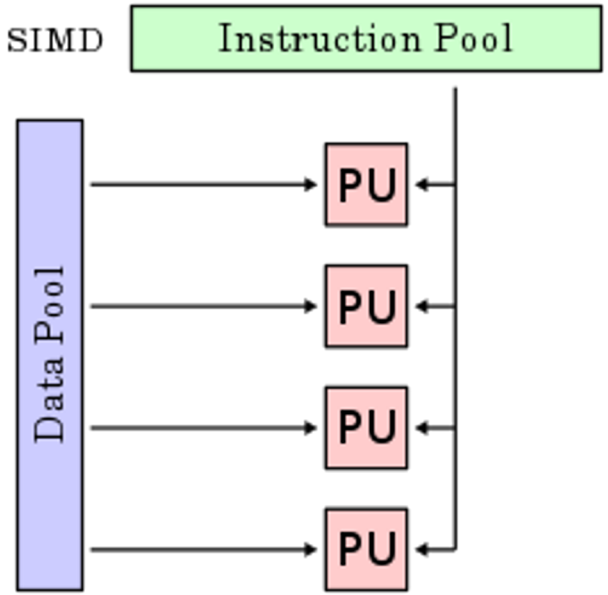
- Sequential computer that exploits no parallelism in either the instruction or data streams
- Examples of SISD architecture are traditional uniprocessor machines
  - Everything we've done!

Multiple Instruction/Single Data Stream



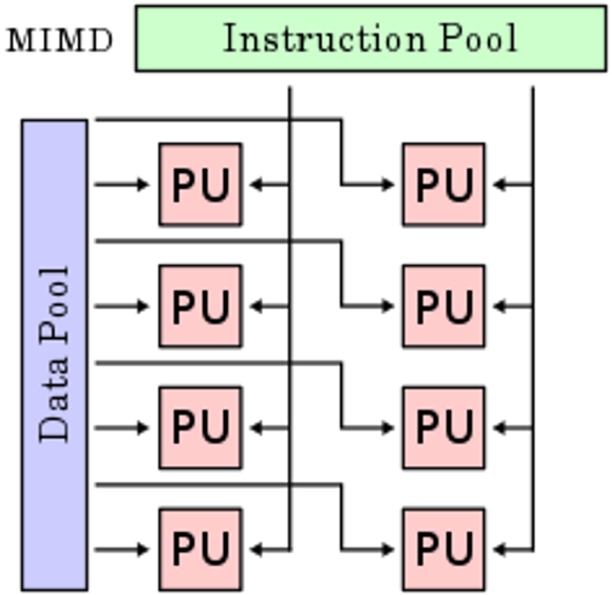
- Exploits multiple instruction streams against a single data stream for data operations that can be naturally parallelized (e.g. certain kinds of array processors)
- MISD no longer commonly encountered, mainly of historical interest only

Single Instruction/Multiple Data Stream



- Computer that applies a single instruction stream to multiple data streams for operations that may be naturally parallelized (e.g. SIMD instruction extensions or Graphics Processing Unit)
  - Today's topic

Multiple Instruction/Multiple Data Stream



- Multiple autonomous processors simultaneously executing different instructions on different data
- MIMD architectures include multicore and Warehouse Scale Computers
  - Tomorrow's Topic

Classes of Data-Level Parallelism

		Data Streams	
		Single	**Multiple**
Instruction Streams	**Single**	SISD: Single Stage Processor	SIMD: Vector Instructions
	Multiple	MISD: Nothing really here	MIMD: Multi-core Processors

Flynn's Taxonomy

*Prof. Michael Flynn, Stanford

When Parallelism Fails

Long chains of connected tasks do not perform better with parallelism

Analogy:

- Driving 10 separate cars can get 40 people somewhere faster than using a single car
- But the time to get 1 person to the location doesn't increase at all…
  - We can't travel each meter of the distance in parallel

Agenda

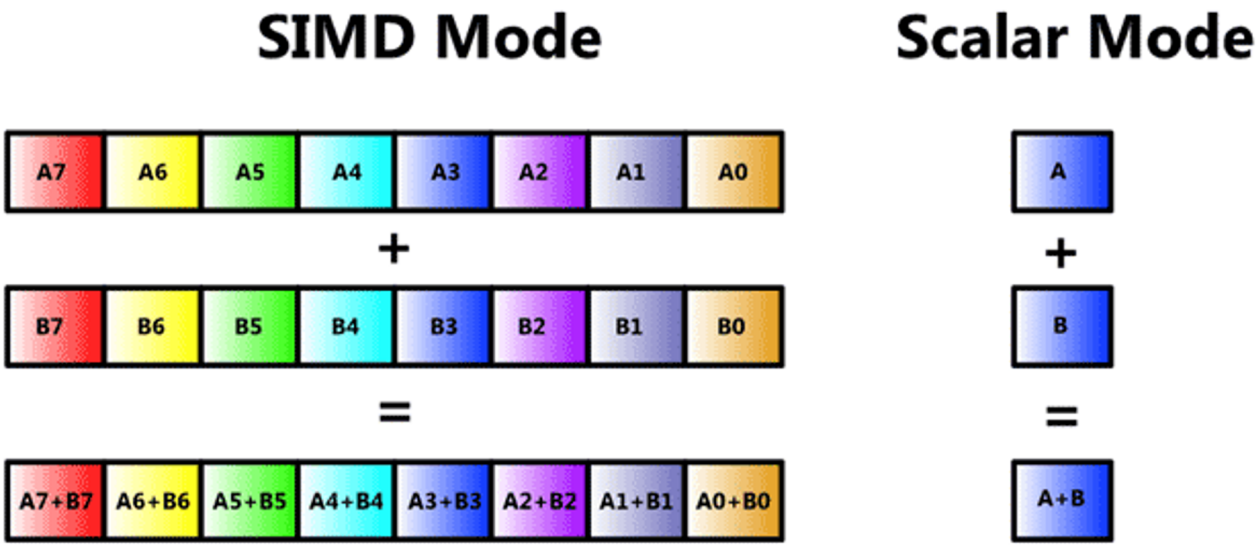
- Intro
- Parallelism and Flynn's Taxonomy
- **SIMD Architectures**
  - **Background**
- Loop Unrolling
- Summary

SIMD Architectures

- *Data-Level Parallelism (DLP):* Executing one operation on multiple data streams
- **Example:** Multiplying a coefficient vector by a data vector (e.g. in filtering)

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

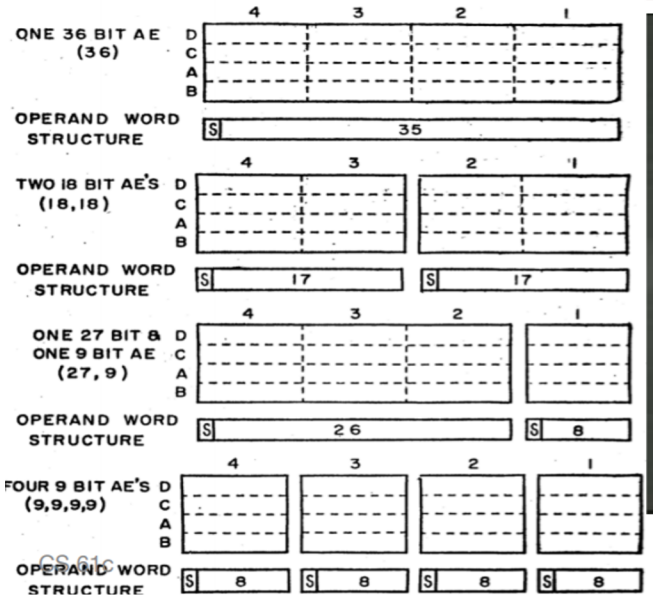
- Sources of performance improvement:
  - One instruction is fetched & decoded for entire operation
  - Multiplications are known to be independent
  - Pipelining/concurrency in memory access as well


SIMD Mode
A7
A6
A5
A4
A3
A2
A1
A0
+
B7
B6
B5
B4
B3
B2
B1
B0
=
A7+B7
A6+B6
A5+B5
A4+B4
A3+B3
A2+B2
A1+B1
A0+B0
Scalar Mode
A
+
B
=
A+B

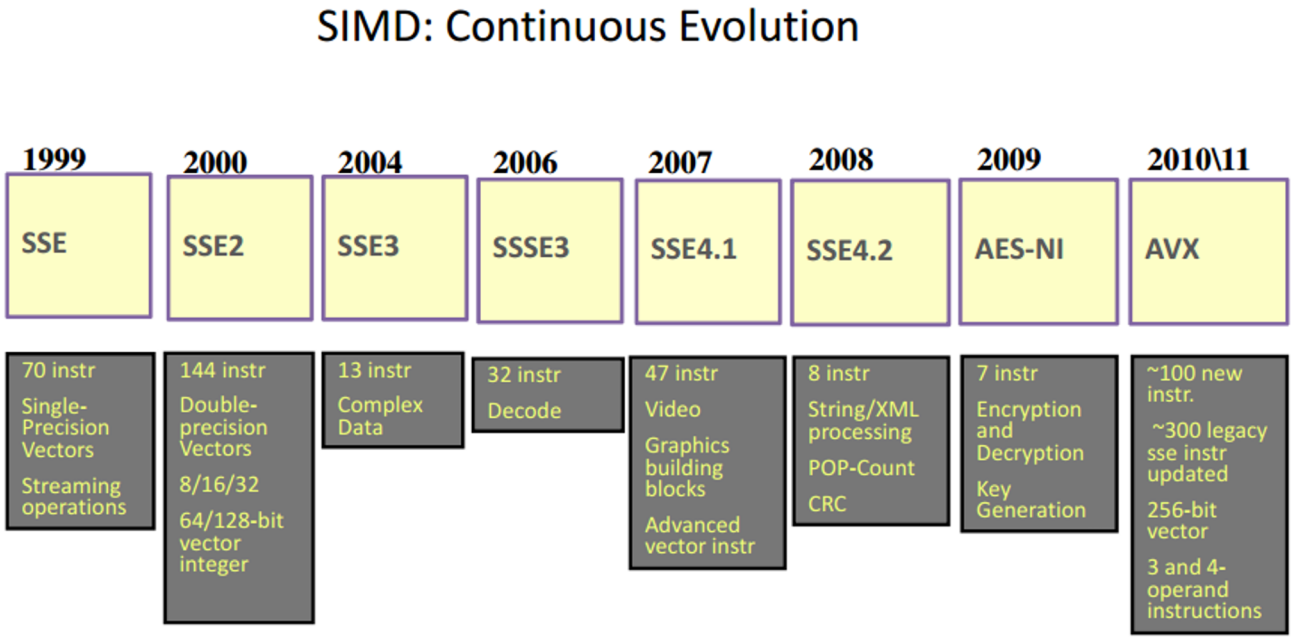
SIMD Applications & Implementations

- Applications
  - Scientific computing
    - Matlab, NumPy
  - Graphics and video processing
    - Photoshop
  - Big Data
    - Deep learning
  - Gaming
- Implementations
  - X86
  - ARM
  - RISC-V vector extensions
  - Video cards24

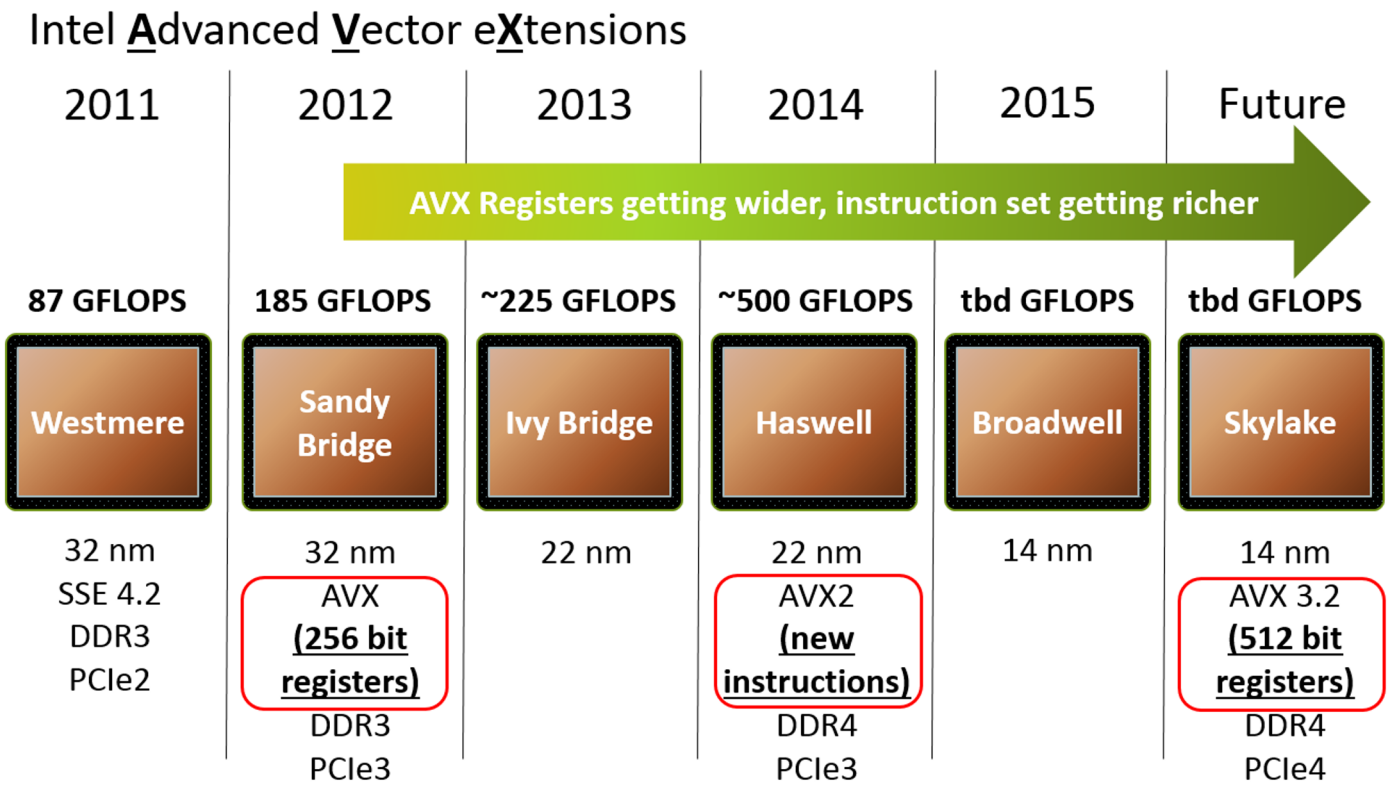
First SIMD Extensions: MIT Lincoln Labs TX-2, 1957



Intel SIMD has been continuously extended



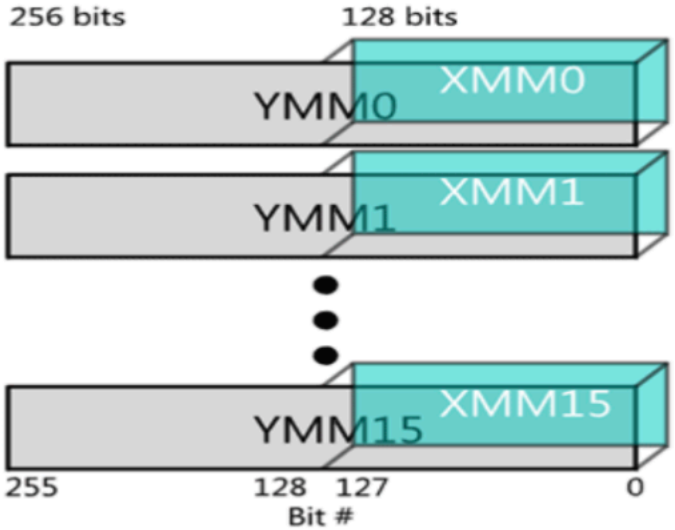
And it has increased in size a lot



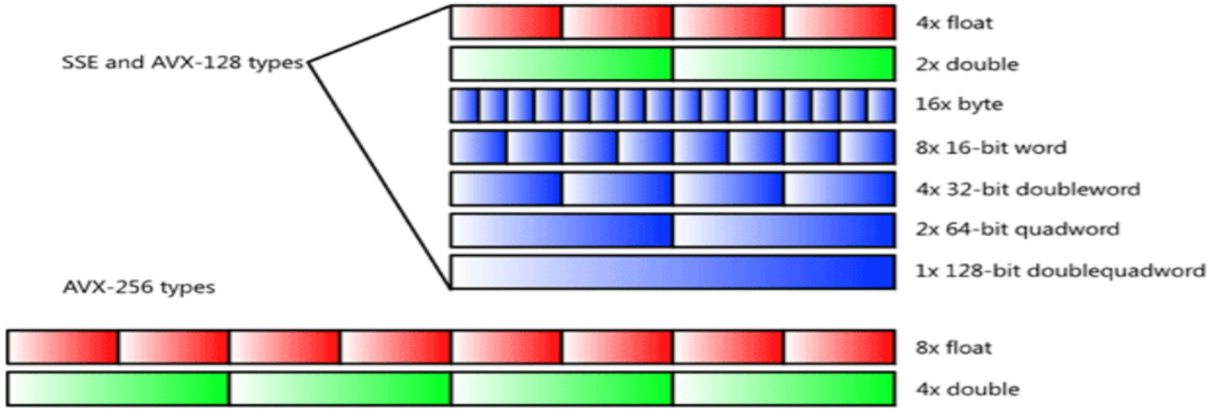
Laptop CPU Specs

```
$ sysctl -a | grep cpu
hw.physicalcpu: 2
hw.logicalcpu: 4

machdep.cpu.brand_string:
    Intel(R) Core(TM) i7-5557U CPU @ 3.10GHz

machdep.cpu.features: FPU VME DE PSE TSC MSR PAE MCE CX8 APIC SEP
    MTRR PGE MCA CMOV PAT PSE36 CLFSH DS ACPI MMX FXSR SSE SSE2 SS
    HTT TM PBE SSE3 PCLMULQDQ DTES64 MON DSCPL VMX EST TM2 SSSE3 FMA
    CX16 TPR PDCM SSE4.1 SSE4.2 x2APIC MOVBE POPCNT AES PCID XSAVE
    OSXSAVE SEGLIM64 TSCTMR AVX1.0 RDRAND F16C

machdep.cpu.leaf7_features: SMEP ERMS RDWRFSGS TSC_THREAD_OFFSET
    BMI1 AVX2 BMI2 INVPCID SMAP RDSEED ADX IPT FPU_CSDS
```

AVX SIMD Registers:
Greater Bit Extensions Overlap Smaller Versions



Intel SIMD Data Types



(Now also AVX-512 available (but not on Hive): 16x float and 8x double)

- In Intel Architecture (unlike RISC-V) a word is **16 bits**
  - Single precision FP: Double word (32 bits)
  - Double precision FP: Quad word (64 bits)

SIMD in the Real World

- Today's compilers can generate SIMD code!
  - But in some cases we get better results by hand
  - (See Project 4)

- RISC-V vector hardware isn't widely available
- So we'll study Intel's x86 SIMD instructions
  - Which have the benefit of being usable on hive machines
  - (and most of our own personal computers)

Agenda

- Intro
- Parallelism and Flynn's Taxonomy
- **SIMD Architectures**
  - **Usage**
- Loop Unrolling
- Summary

Intel SSE Intrinsics

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

How do we use these SIMD instructions?

- Intrinsics:
  - “function calls” that actually just execute an assembly instruction

Example:

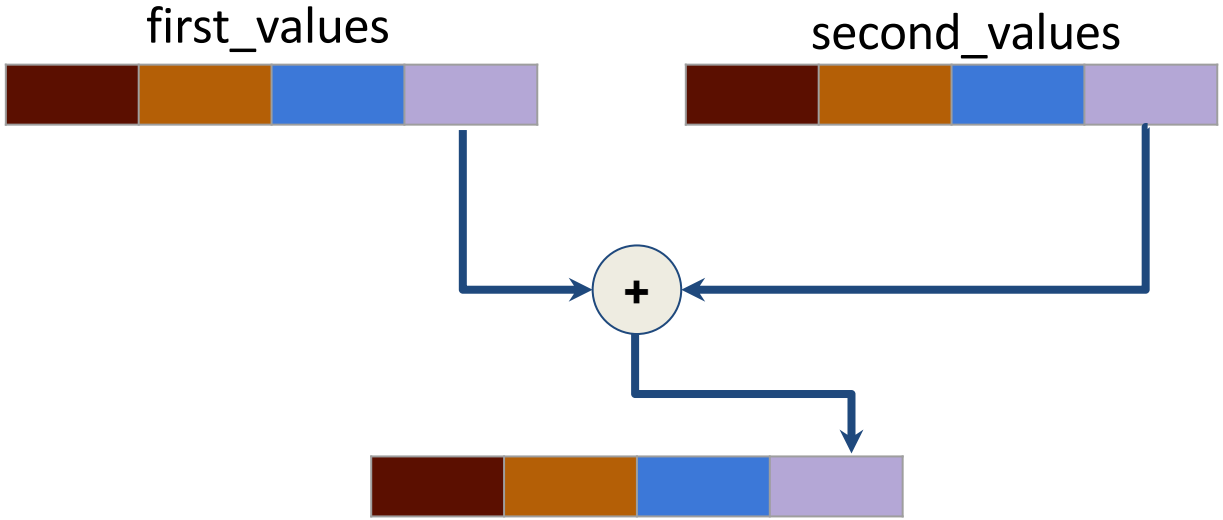
```
_mm_add_epi32(first_values, second_values);
```

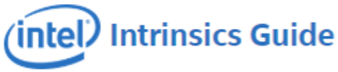
WHAT????

_mm_add_epi32(first_values, second_values)

Multi**M**edia extension
(They all start with this)

Arguments are
Extended **P**acked
Integers,
each **32**-bits in size
(signed)





Technologies

- MMX
- SSE
- SSE2
- SSE3
- SSSE3
- SSE4.1
- SSE4.2
- AVX
- AVX2
- FMA
- AVX-512
- KNC
- SVML
- Other

Categories

- Application-Targeted
- Arithmetic
- Bit Manipulation
- Cast
- Compare

mm_add_epi32

```
__m128i _mm_add_epi32 (__m128i a, __m128i b)
```

Synopsis

```
__m128i _mm_add_epi32 (__m128i a, __m128i b)
#include <emmintrin.h>
Instruction: paddd xmm, xmm
CPUID Flags: SSE2
```

Description

Add packed 32-bit integers in a and b, and store the results in dst.

Operation

```
FOR j := 0 to 3
        i := j*32
        dst[i+31:i] := a[i+31:i] + b[i+31:i]
ENDFOR
```

Performance

Architecture	Latency	Throughput (CPI)
Skylake	1	0.33
Broadwell	1	0.5
Haswell	1	0.5
Ivy Bridge	1	0.5

Soooooooo fast

X86 Intrinsics AVX Data Type

Type	Meaning
__m256	256-bit as eight single-precision floating-point values, representing a YMM register or memory location
__m256d	256-bit as four double-precision floating-point values, representing a YMM register or memory location
__m256i	256-bit as integers, (bytes, words, etc.)
__m128	128-bit single precision floating-point (32 bits each)
__m128d	128-bit double precision floating-point (64 bits each)

Intrinsics AVX Code nomenclature

Marking	Meaning
[s/d]	Single- or double-precision floating point
[i/u]nnn	Signed or unsigned integer of bit size *nnn*, where *nnn* is 128, 64, 32, 16, or 8
[ps/pd/sd]	Packed single, packed double, or scalar double
epi32	Extended packed 32-bit signed integer
si256	Scalar 256-bit integer

Sample of SSE Intrinsics

Arithmetic:

`__m128i _mm_and_si128(__m128i a, __m128i b)`:

Perform a bitwise AND of 128 bits in a and b, and return the result.

`__m128i _mm_add_epi32(__m128i a, __m128i b)`:

Return vector $(a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$

Load and store operations:

`void` `_mm_storeu_si128( __m128i *p, __m128i a)`:

Store 128-bit vector `a` at pointer `p`.

`__m128i _mm_loadu_si128( __m128i *p)`:

Load the 4 successive ints pointed to by `p` into a 128-bit vector.

Compare

`__m128i _mm_cmpeq_epi32(__m128i a, __m128i b)`:

The ith element of the return vector will be set to 0xFFFFFFFF if the ith elements of a and b are equal, otherwise it'll be set to 0.

Example: SIMD Array Processing

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

pseudocode

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

SISD

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

SIMD

Agenda

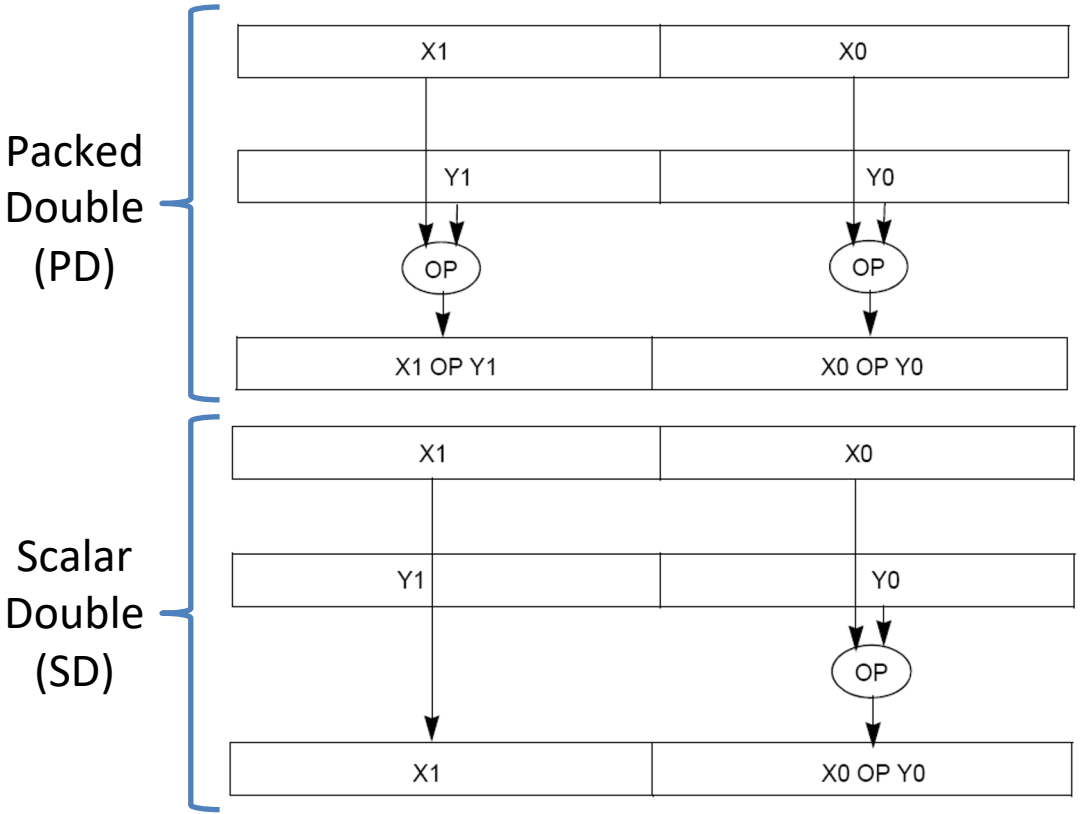
- Intro
- Parallelism and Flynn's Taxonomy
- **SIMD Architectures**
  - **Example**
- Loop Unrolling
- Summary

```
int add_no_SSE(int size, int *first_array, int *second_array) {
    for (int i = 0; i < size; ++i) {
        first_array[i] += second_array[i];
    }
}
```

```
int add_SSE(int size, int *first_array, int *second_array) {
    for (int i=0; i + 4 <= size; i+=4) { // only works if (size%4) == 0
        // load 128-bit chunks of each array
        __m128i first_values = _mm_loadu_si128((__m128i*) &first_array[i]);
        __m128i second_values = _mm_loadu_si128((__m128i*) &second_array[i]);

        // add each pair of 32-bit integers in the 128-bit chunks
        first_values = _mm_add_epi32(first_values, second_values);

        // store 128-bit chunk to first array
        _mm_storeu_si128((__m128i*) &first_array[i], first_values);
    }
            ...
}
```

https://www.cs.virginia.edu/~cr4bd/3330/F2017/simdref.html

You can do this with floating point numbers too!



Example: 2 × 2 Matrix Multiply

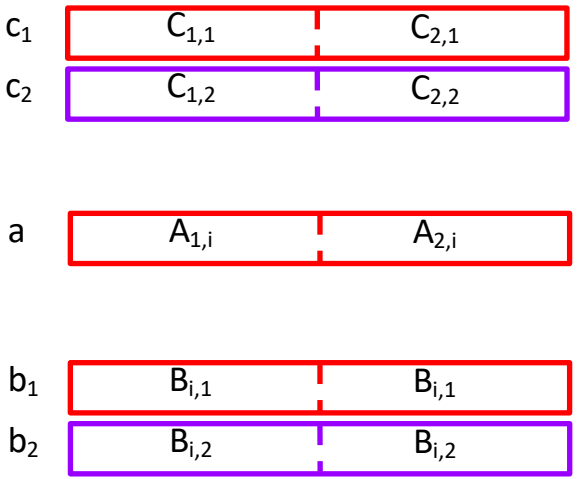
Definition of Matrix Multiply:

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

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

Example: 2 × 2 Matrix Multiply

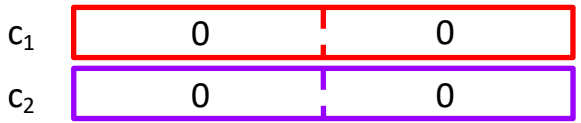
- Using the XMM registers
  - 64-bit/double precision/two doubles per XMM reg



Memory is column major

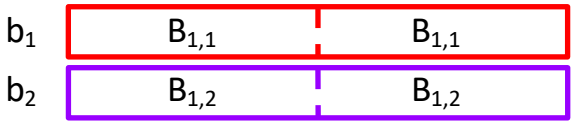
Example: 2 × 2 Matrix Multiply

- Initialization



- i = 1

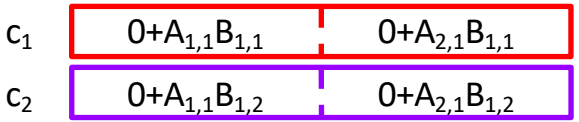
_mm_load_pd: Stored in memory in Column order



_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register

Example: 2 × 2 Matrix Multiply

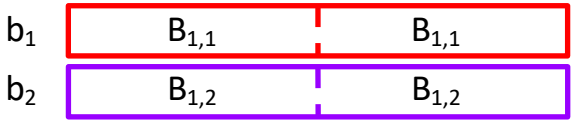
- First iteration intermediate result




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

- i = 1

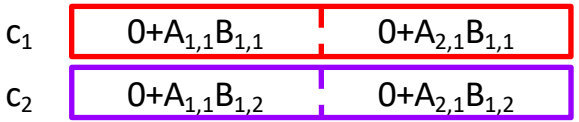
_mm_load_pd: Stored in memory in Column order



_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register

Example: 2 × 2 Matrix Multiply

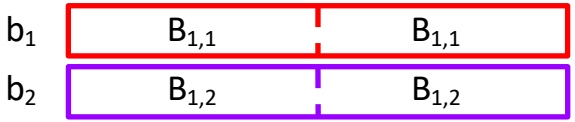
- First iteration intermediate result




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

- i = 2

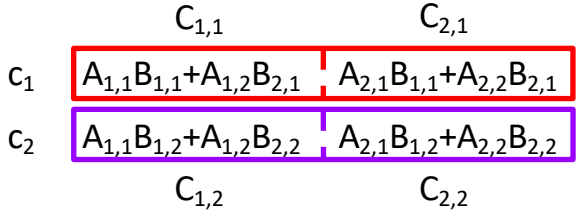
_mm_load_pd: Stored in memory in Column order



_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register

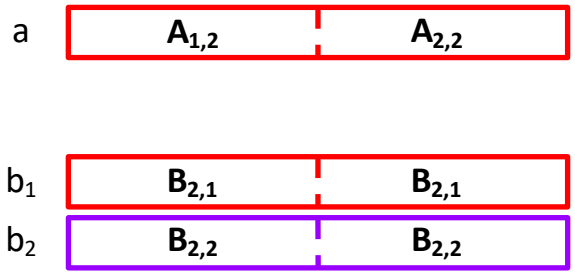
Example: 2 × 2 Matrix Multiply

- Second iteration intermediate result




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

- i = 2



_mm_load_pd: Stored in memory in Column order

_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register

2 x 2 Matrix Multiply Code (1/2)

```
#include <stdio.h>
// header file for SSE4.2 compiler intrinsics
#include <nmmintrin.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 a couple 128-bit vector variables
   __m128d c1,c2,a,b1,b2;

   /* 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;
/* continued on next slide */
```

2 x 2 Matrix Multiply Code (2/2)

```
// 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: [0 + a_11*b_11 | 0 + 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: [0 + a_11*b_12 | 0 + 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;
}
```

Agenda

- Intro
- Parallelism and Flynn’s Taxonomy
- SIMD Architectures
- **Loop Unrolling**
- Summary

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=0; i<1000; i++)
 x[i] = x[i] + s;
```

- How can we reveal more data level parallelism than is available in a single iteration of a loop?
  - *Unroll the loop* and adjust iteration rate

Looping in RISC-V

Assumptions:

s0 → initial address (top of array)

s1 → scalar value b

s2 → termination address (end of array)

```
Loop:
        lw      t0, 0(s0)
        addu    t0,t0,s1      # add b to array element
        sw      t0,0(s0)      # store result
        addi    s0,s0,4       # move to next element
        bne     s0,s2,Loop    # repeat Loop if not done
```

Loop Unrolled

Loop:

```
 lw   t0,0(s0)
add  t0,t0,s1
sw   t0,0(s0)
lw   t1,4(s0)
add  t1,t1,s1
sw   t1,4(s0)
lw   t2,8(s0)
add  t2,t2,s1
sw   t2,8(s0)
lw   t3,12(s0)
add  t3,t3,s1
sw   t3,12(s0)
addi s0,s0,16
bne  s0,s2,Loop
```

NOTE:

1. Loop overhead (addi, bne) encountered only once every 4 data iterations

2. This unrolling only works if (`loop_limit mod 4`)`= 0`

3. Using different registers allows us to eliminate stalls by reordering

4. Made code size larger...

Loop Unrolled and Reordered

Loop:

```
lw   t0,0(s0)
lw   t1,4(s0)
lw   t2,8(s0)
lw   t3,12(s0)
add  t0,t0,s1
add  t1,t1,s1
add  t2,t2,s1
add  t3,t3,s1
sw   t0,0(s0)
sw   t1,4(s0)
sw   t2,8(s0)
sw   t3,12(s0)
addi s0,s0,16
bne  s0,s2,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 the compiler doing loop unrolling, could do it yourself in C:

```
for(i=0; i<1000; i++)
  x[i] = x[i] + s;
```


```
for(i=0; i<1000; 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

- Take a loop of **n iterations** and perform a **k-fold** unrolling of the body of the loop:
  - First run the loop with k copies of the body **floor(n/k)** times
  - To finish leftovers, then run the loop with 1 copy of the body **n mod k** times (known as the *tail case*)

Drawbacks to Loop Unrolling

- Loop unrolling can greatly speed up your code but isn't perfect for a couple of reasons
  - If you are doing it by hand it's a really inefficient/tedious task
    - In reality you would want your compiler to do this but we want you to understand it
  - Loop unrolling increases your static code size
    - Static code size is important for accesses to your instruction cache
    - You might not want k to be too large
    - Try find a balance between less executed instructions and small static code size

Code Optimization

- Loop unrolling isn't really a form of parallelism but is instead an example of code optimization
  - Code is converted from a form easy to understand to one with better performance
- This is often the work of your compiler but it may not always be able to make the best optimizations
- Let's consider another example of how you can optimize your code

Loop Invariants

```
for (int i = 0; i < n; i++) {
    arr[i] = (f(x) - g(y)) * arr[i];
}
```

- This is an example of what we call a loop invariant
  - Invariant meaning does not change in the loop
- What happens if f and g are expensive?
  - Then f and g are computed each iteration, n times in total
  - But the loop recomputes the result

Loop Invariants

```
z = (f(x) - g(y))
for (int i = 0; i < n; i++) {
    arr[i] = z * arr[i];
}
```

- Solution: Move the code outside of the loop and only compute it once since it never changes
  - Now n expensive calls has become 1 expensive call
- But can we do better?

Loop Invariants

- What happens is f and/or g is really really expensive
  - We want compute it as little as possible
- Now we always compute it once
- But what happens if n <= 0
  - Then we compute the invariant once
  - But we never enter the loop so we never use it
- Solution: Add a check to avoid computing it if we don't enter the loop

Loop Invariants

```
if (n > 0) {
    z = f(x) - g(y);
    for (int i = 0; i < n; i++) {
            arr[i] += z * arr[i];
    }
}
```

Now we compute the invariant once if we enter the loop and otherwise not at all

Agenda

- Intro
- Parallelism and Flynn’s Taxonomy
- SIMD Architectures
- Loop Unrolling
- **Summary**

Summary

How do we get more performance?

- Can’t really do it by speeding clock up anymore
- Domain-specific hardware
- Parallelism!
  - Single Instruction Multiple Data examples
  - Loop unrolling optimizations