SIMD Programming

CS 240A, 2017

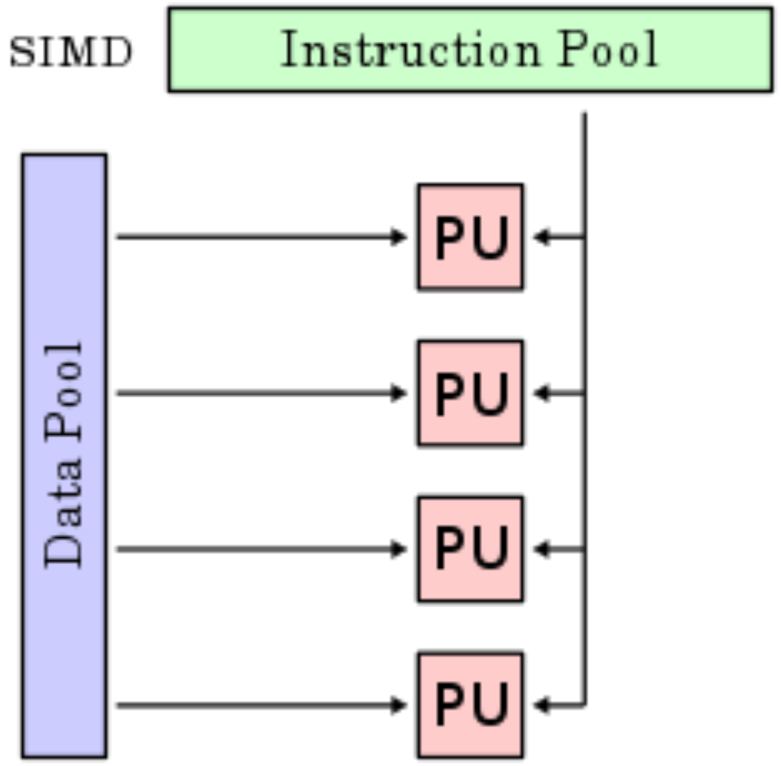
Flynn* Taxonomy, 1966

		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 using synchronization primitives
- 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

Single-Instruction/Multiple-Data Stream (SIMD or “sim-dee”)

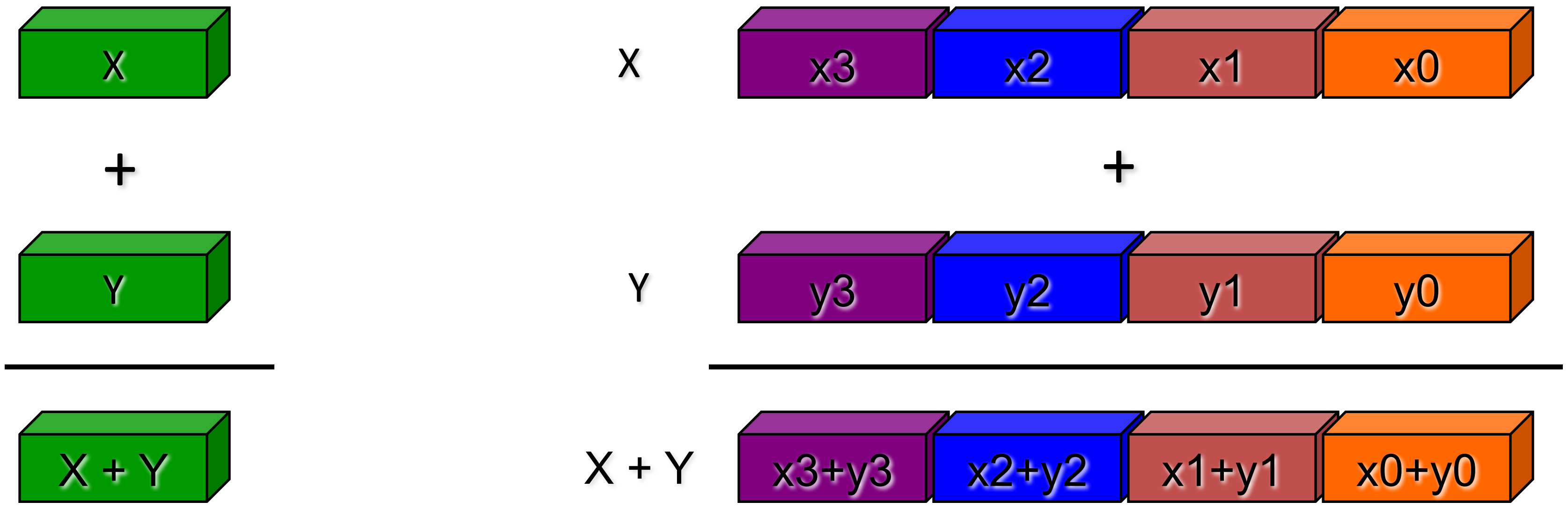


- SIMD computer exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., Intel SIMD instruction extensions or NVIDIA Graphics Processing Unit (GPU)

SIMD: Single Instruction, Multiple Data

- Scalar processing
  - traditional mode
  - one operation produces one result

- SIMD processing
  - With Intel SSE / SSE2
  - SSE = streaming SIMD extensions
  - one operation produces multiple results



Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

What does this mean to you?

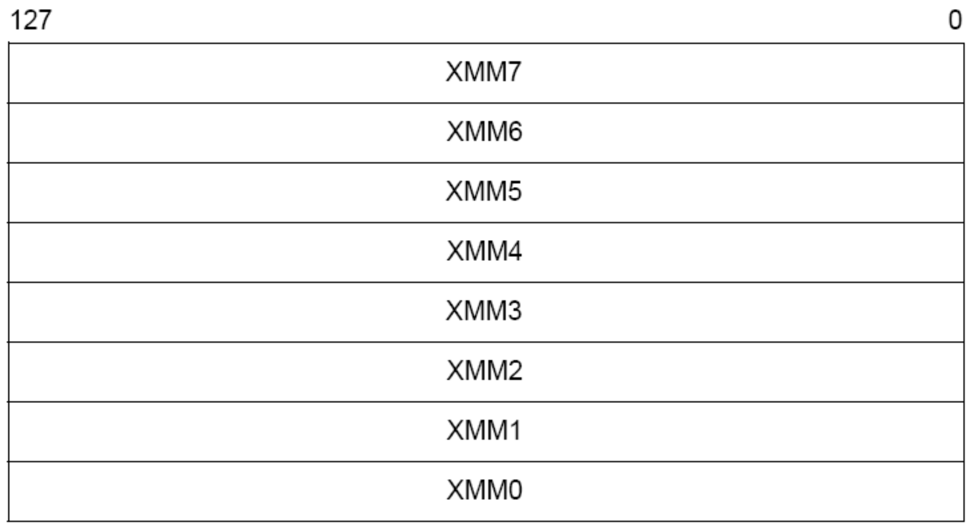
- In addition to SIMD extensions, the processor may have other special instructions
  - Fused Multiply-Add (FMA) instructions:

    $x = y + c * z$

    is so common some processor execute the multiply/add as a single instruction, at the same rate (bandwidth) as + or * alone
- In theory, the compiler understands all of this
  - When compiling, it will rearrange instructions to get a good "schedule" that maximizes pipelining, uses FMAs and SIMD
  - It works with the mix of instructions inside an inner loop or other block of code
- But in practice the compiler may need your help
  - Choose a different compiler, optimization flags, etc.
  - Rearrange your code to make things more obvious
  - Using special functions ("intrinsics") or write in assembly ☹

Intel SIMD Extensions

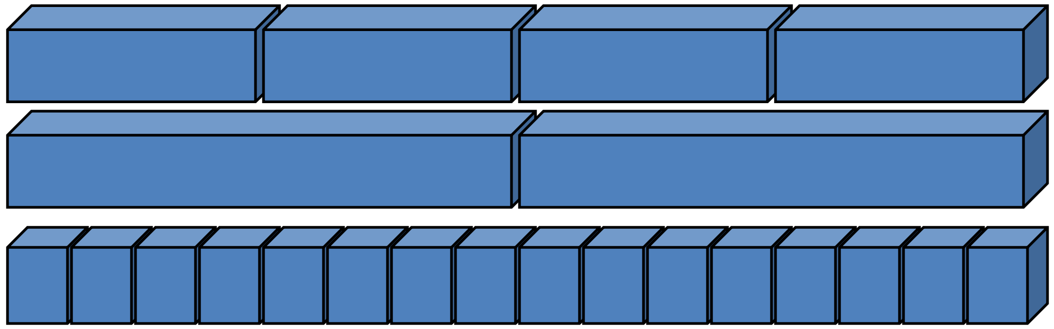
- MMX 64-bit registers, reusing floating-point registers [1992]
- SSE2/3/4, new 8 128-bit registers [1999]



- AVX, new 256-bit registers [2011]
  - Space for expansion to 1024-bit registers

SSE / SSE2 SIMD on Intel

- SSE2 data types: anything that fits into 16 bytes, e.g.,

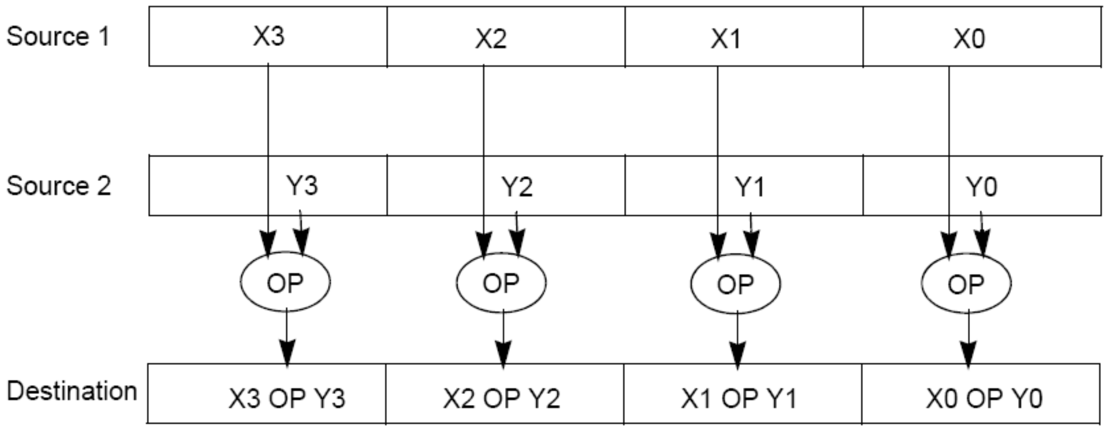


4x floats

2x doubles

16x bytes

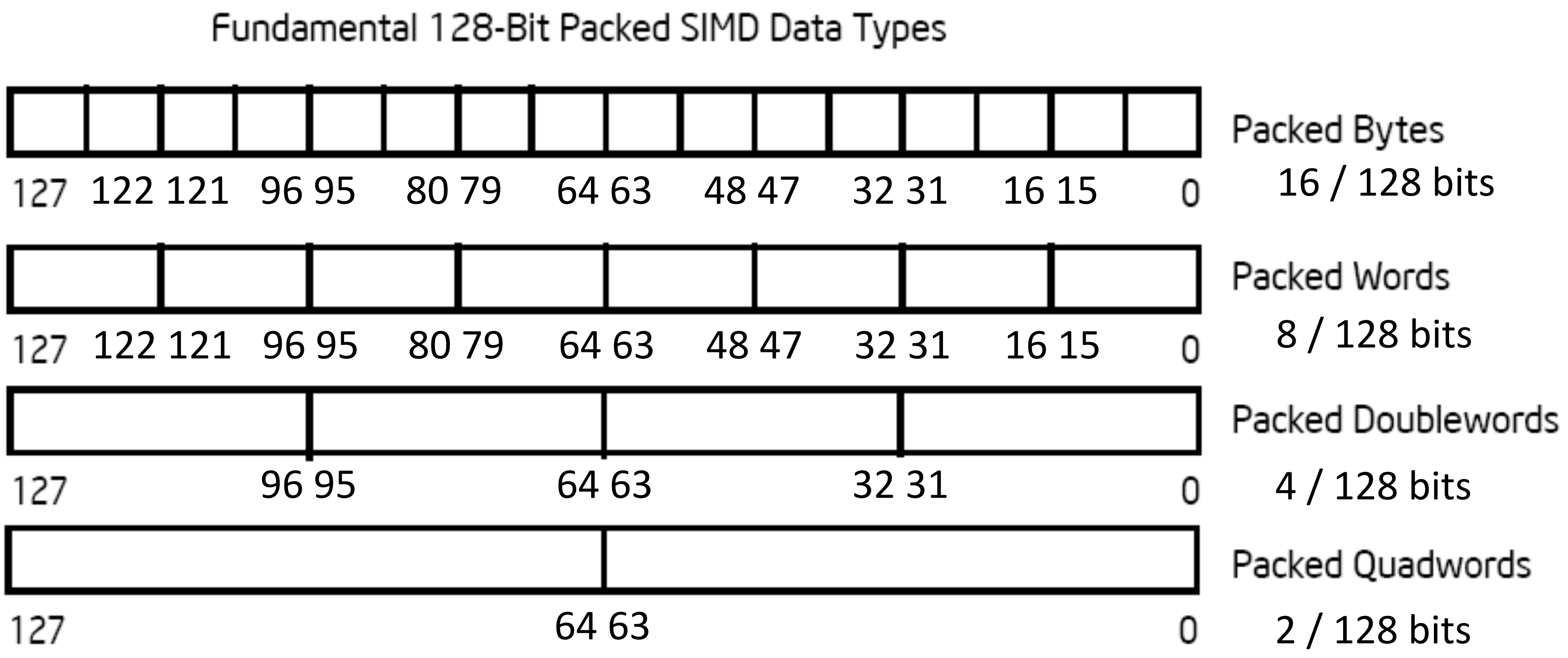
- Instructions perform add, multiply etc. on all the data in parallel



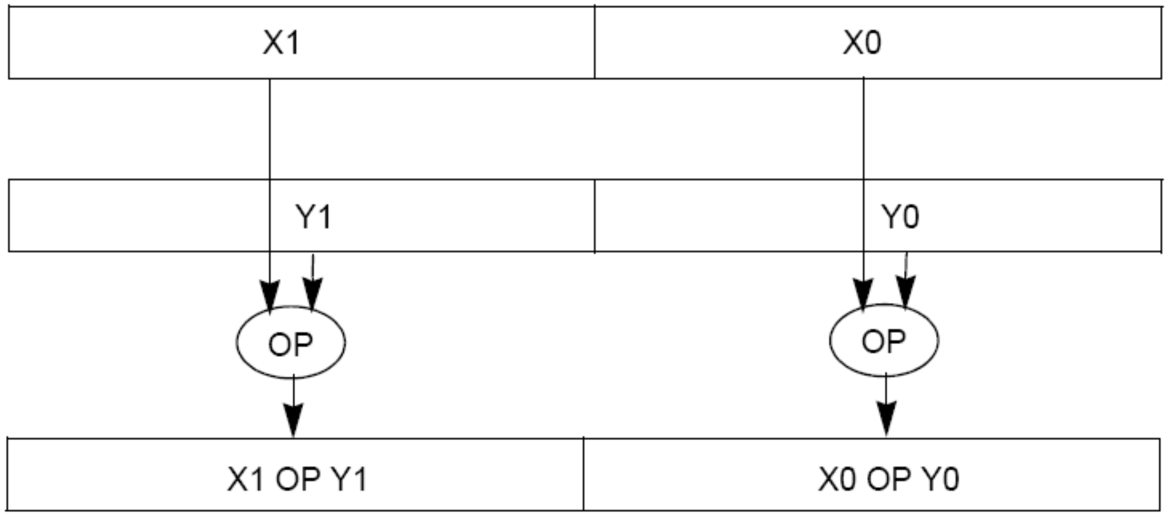
- Similar on GPUs, vector processors (but many more simultaneous operations)

Intel Architecture SSE2+ 128-Bit SIMD Data Types

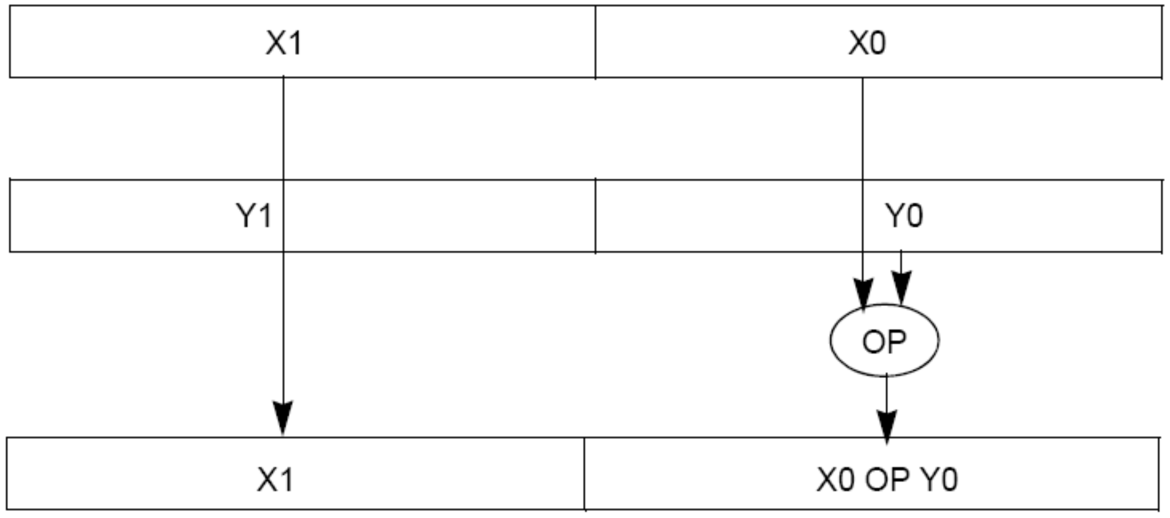
- Note: in Intel Architecture (unlike MIPS) a word is 16 bits
  - Single-precision FP: Double word (32 bits)
  - Double-precision FP: Quad word (64 bits)



Packed and Scalar Double-Precision Floating-Point Operations



Packed



Scalar

SSE/SSE2 Floating Point Instructions

Move does both load and store

Data transfer	Arithmetic	Compare
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD{SS/PS/SD/PD} xmm, mem/xmm	CMP{SS/PS/SD/PD}
	SUB{SS/PS/SD/PD} xmm, mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	MUL{SS/PS/SD/PD} xmm, mem/xmm	
	DIV{SS/PS/SD/PD} xmm, mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD} mem/xmm	

xmm: one operand is a 128-bit SSE2 register
mem/xmm: other operand is in memory or an SSE2 register
{SS} Scalar Single precision FP: one 32-bit operand in a 128-bit register
{PS} Packed Single precision FP: four 32-bit operands in a 128-bit register
{SD} Scalar Double precision FP: one 64-bit operand in a 128-bit register
{PD} Packed Double precision FP, or two 64-bit operands in a 128-bit register
{A} 128-bit operand is aligned in memory
{U} means the 128-bit operand is unaligned in memory
{H} means move the high half of the 128-bit operand
{L} means move the low half of the 128-bit operand

Example: SIMD Array Processing

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

```
for each f in array
{
    load f to 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

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

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

General Loop Unrolling with a Head

- Handing loop iterations indivisible by step size.

```
for(i=1003; i>0; i=i-1)
    x[i] = x[i] + s;
```

- Could be rewritten

```
for(i=1003;i>1000;i--)  //Handle the head (1003 mod 4)
    x[i] = x[i] + s;

for(i=1000; i>0; i=i-4) {// handle other iterations
    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;
 }
```

Tail method for general loop unrolling

- Handing loop iterations indivisible by step size.

```
for(i=1003; i>0; i=i-1)
      x[i] = x[i] + s;
```

- Could be rewritten

```
for(i=1003; i>0 && i> 1003 mod 4; 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;
    }
```

for(i= 1003 mod 4; i>0; i--) //special handle in tail

```
    x[i] = x[i] + s;
```

Another loop unrolling example

Normal loop	After loop unrolling

int x;
for (x = 0; x < 103; x++) {
    delete(x);
}
	int x;
	for (x = 0; x < 103/5*5; x += 5) {
	delete(x);
	delete(x + 1);
	delete(x + 2);
	delete(x + 3);
	delete(x + 4);
	}
	/*Tail*/
	for (x = 103/5*5; x < 103; x++) {
	    delete(x);
	}

Intel SSE Intrinsics

Intrinsics are C functions and procedures for inserting assembly language into C code, including SSE instructions

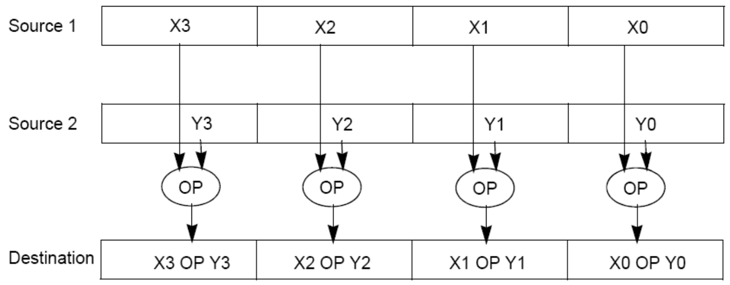
Instrinsics:	Corresponding SSE instructions:
• Vector data type:	
_m128d	
• Load and store operations:	
_mm_load_pd	MOVAPD/aligned, packed double
_mm_store_pd	MOVAPD/aligned, packed double
_mm_loadu_pd	MOVUPD/unaligned, packed double
_mm_storeu_pd	MOVUPD/unaligned, packed double
• Load and broadcast across vector	
_mm_load1_pd	MOVSD + shuffling/duplicating
• Arithmetic:	
_mm_add_pd	ADDPD/add, packed double
_mm_mul_pd	MULPD/multiple, packed double

Example 1: Use of SSE SIMD instructions

- For (i=0; i<n; i++) sum = sum+ a[i];
- Set 128-bit temp=0;

  For (i = 0; n/4*4; i=i+4){

    Add 4 integers with 128 bits from &a[i] to temp; }

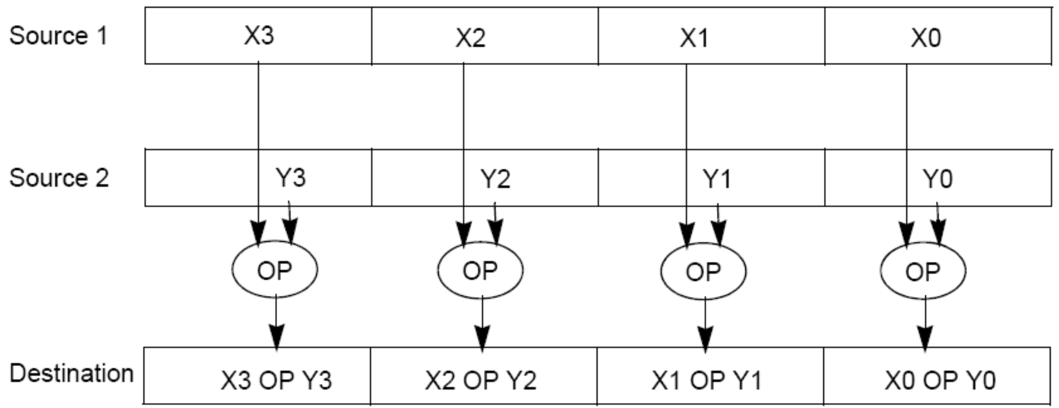


Tail: Copy out 4 integers of temp and add them together to sum.

For(i=n/4*4; i<n; i++) sum   += a[i];

Related SSE SIMD instructions

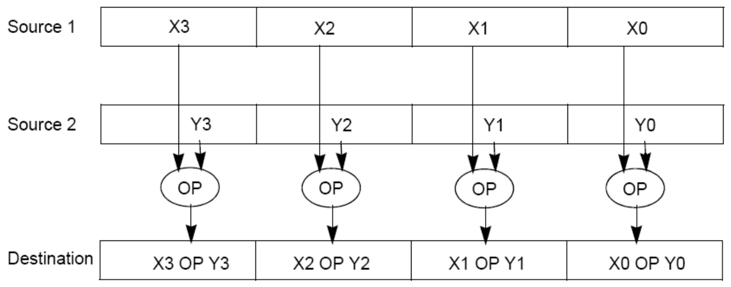
__m128i _mm_setzero_si128( )	returns 128-bit zero vector
__m128i _mm_loadu_si128(__m128i *p)	Load data stored at pointer p of memory to a 128bit vector, returns this vector.
__m128i _mm_add_epi32(__m128i a, __m128i b)	returns vector (a_0+b_0, a_1+b_1, a_2+b_2, a_3+b_3)
void _mm_storeu_si128(__m128i *p, __m128i a)	stores content off 128-bit vector ”a” ato memory starting at pointer p



Related SSE SIMD instructions

- Add 4 integers with 128 bits from &a[i] to temp vector with loop body temp = temp + a[i]
- Add 128 bits, then next 128 bits …

```
__m128i temp=_mm_setzero_si128();
__m128i temp1=_mm_loadu_si128((__m128i *)(a+i));
 temp=_mm_add_epi32(temp, temp1)
```



Example 2: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

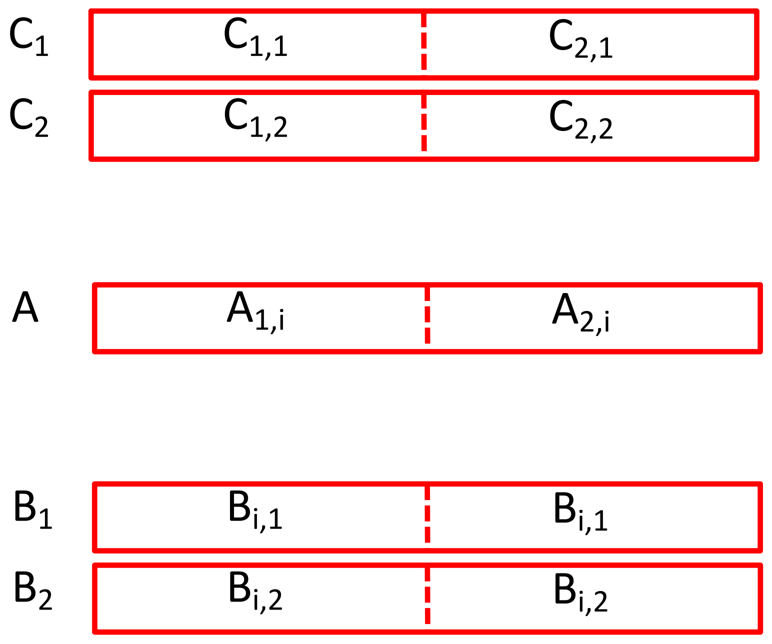
$$C_{i,j} = (A\times B)_{i,j} = \sum_{k=1}^{2} A_{i,k} \times B_{k,j}$$

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

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

Example: 2 x 2 Matrix Multiply

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



Stored in memory in Column order

$$
\begin{array}{cc}
\left[\begin{array}{cc} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{array}\right] \\
\begin{array}{cc} C_1 & C_2 \end{array}
\end{array}
$$

Example: 2 x 2 Matrix Multiply

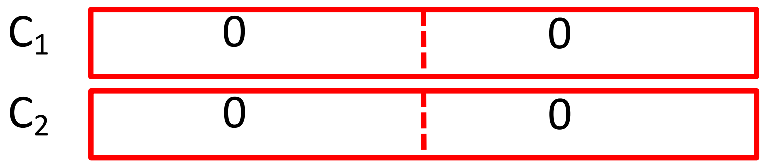
- Initialization

C_1	0	0
C_2	0	0

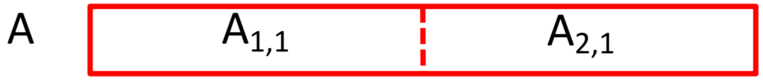
Example: 2 x 2 Matrix Multiply

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

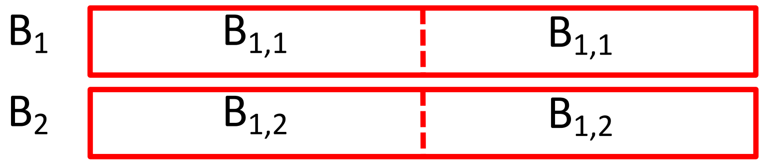
- Initialization



- I = 1



_mm_load_pd: Load 2 doubles into XMM reg, 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 (duplicates value in both halves of XMM)

Example: 2 x 2 Matrix Multiply

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

- First iteration intermediate result

C_1	$0+A_{1,1}B_{1,1}$	$0+A_{2,1}B_{1,1}$
C_2	$0+A_{1,1}B_{1,2}$	$0+A_{2,1}B_{1,2}$

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 = 1

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

_mm_load_pd: Stored in memory in Column order

B_1	$B_{1,1}$	$B_{1,1}$
B_2	$B_{1,2}$	$B_{1,2}$

_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)

Example: 2 x 2 Matrix Multiply

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

- First iteration intermediate result

C_1	$0+A_{1,1}B_{1,1}$	$0+A_{2,1}B_{1,1}$
C_2	$0+A_{1,1}B_{1,2}$	$0+A_{2,1}B_{1,2}$

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	$A_{1,2}$	$A_{2,2}$

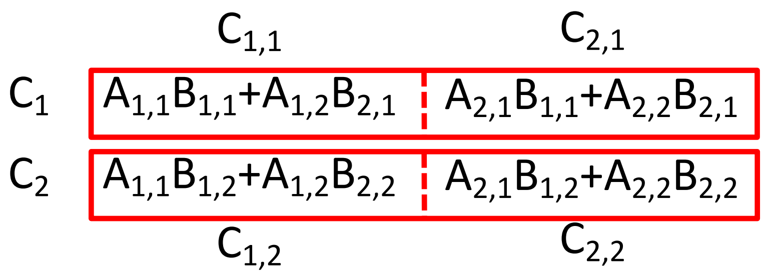
_mm_load_pd: Stored in memory in Column order

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)

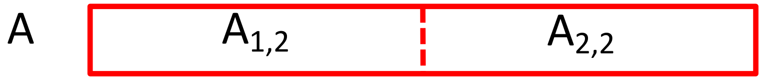
Example: 2 x 2 Matrix Multiply

- Second iteration intermediate result

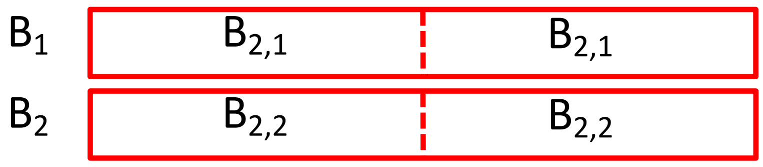

$C_{1,1}$
$C_{2,1}$
C_1
$A_{1,1}B_{1,1}+A_{1,2}B_{2,1}$
$A_{2,1}B_{1,1}+A_{2,2}B_{2,1}$
C_2
$A_{1,1}B_{1,2}+A_{1,2}B_{2,2}$
$A_{2,1}B_{1,2}+A_{2,2}B_{2,2}$
$C_{1,2}$
$C_{2,2}$

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



_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 (duplicates value in both halves of XMM)

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;
```

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;
}
```

Conclusion

- Flynn Taxonomy
- Intel SSE SIMD Instructions
  – Exploit data-level parallelism in loops
  – One instruction fetch that operates on multiple operands simultaneously
  – 128-bit XMM registers
- SSE Instructions in C
  – Embed the SSE machine instructions directly into C programs through use of intrinsics
  – Achieve efficiency beyond that of optimizing compiler