Instrucciones SIMD

Programación Paralela y Concurrente

Esteban Rodríguez Betancourt

Taxonomía de Flynn

		Flujo de datos	
		Único	**Múltiple**
Flujo de instrucciones	**Único**	SISD	SIMD
	Múltiple	MISD	MIMD

Single Instruction/Multiple Data (SIMD)

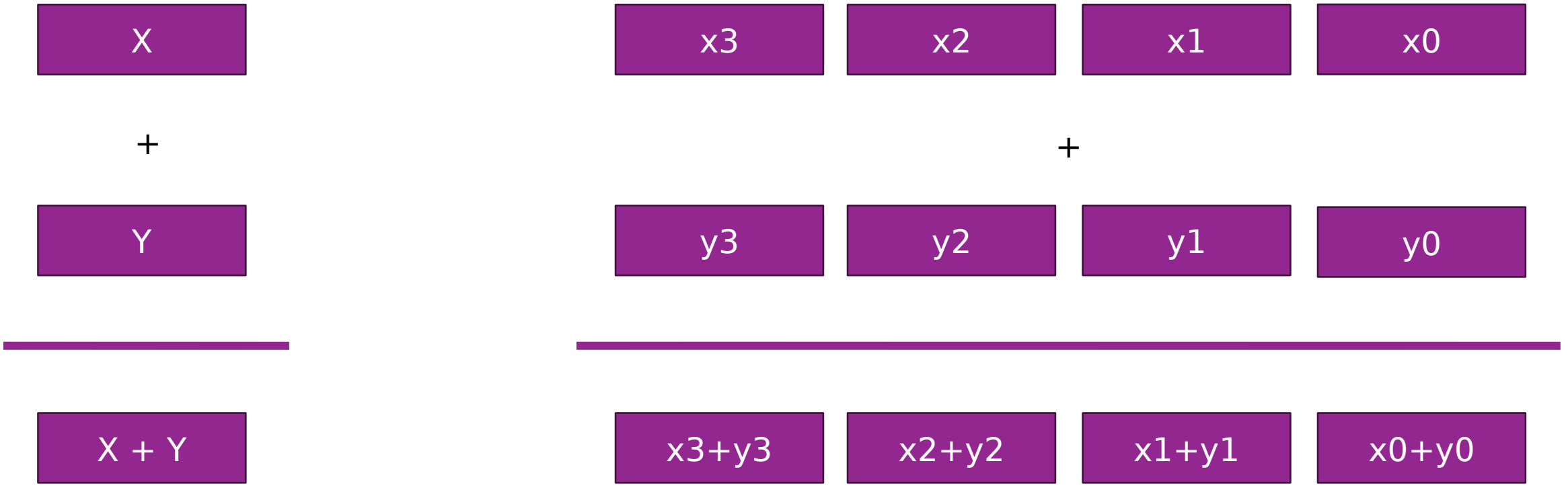
Se aplica la misma instrucción a múltiples flujos de datos

Disponible en CPUs

NO es paralelismo a nivel de tarea o instrucciones

Permite realizar más procesamiento por instrucción

Vs procesamiento escalar

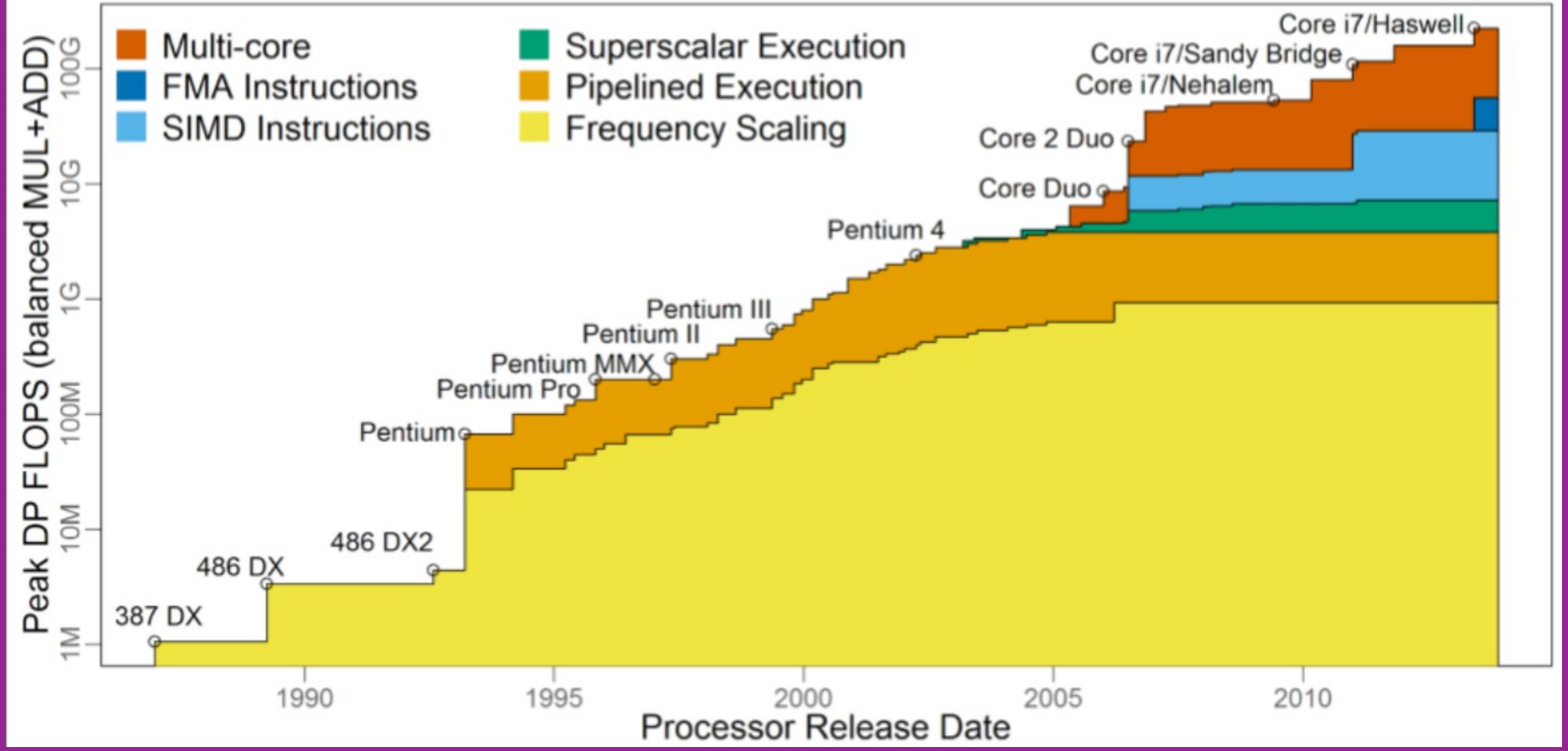


Implicaciones

- Más trabajo en menos instrucciones
- Instrucciones especiales como Fused Multiply Add
- Optimización del compilador
- Ayuda al compilador
  - Flags
  - Reordenar el código
  - Intrinsics
  - Ensamblador

Rendimiento del CPU

# Cores	• Paralelismo a nivel de instrucción
FLOPS / instrucción	• SIMD y FMA
Instrucciones por ciclo	• Paralelismo a nivel de instrucción
Ciclos por segundo	• Frecuencia



Componentes del rendimiento

Single Instruction – Multiple Data

Instrucciones especiales que funcionan con arreglos pequeños de tamaño fijo

Por ejemplo:

MMX soporta arrays de 4 elementos de 16 bits

SSE2 soporta arrays de 2 elementos de flotantes de doble precisión

¡SIMD es más eficiente!

Más instrucciones para procesar datos que en el conjunto de instrucciones típico

Abs, min, max, aritmética saturada, etc

Es posible ganar más rendimiento que el largo de los vectores

¡SIMD es más eficiente!

```
uint32_t a[4];
uint32_t b[4];

a[0] = a[0] > b[0] ? a[0] : b[0];
a[1] = a[1] > b[1] ? a[1] : b[1];
a[2] = a[2] > b[2] ? a[2] : b[2];
a[3] = a[3] > b[3] ? a[3] : b[3];
```

20 instrucciones RISC:

- 8 loads
- 4 cmp
- 4 move condicional
- 4 store

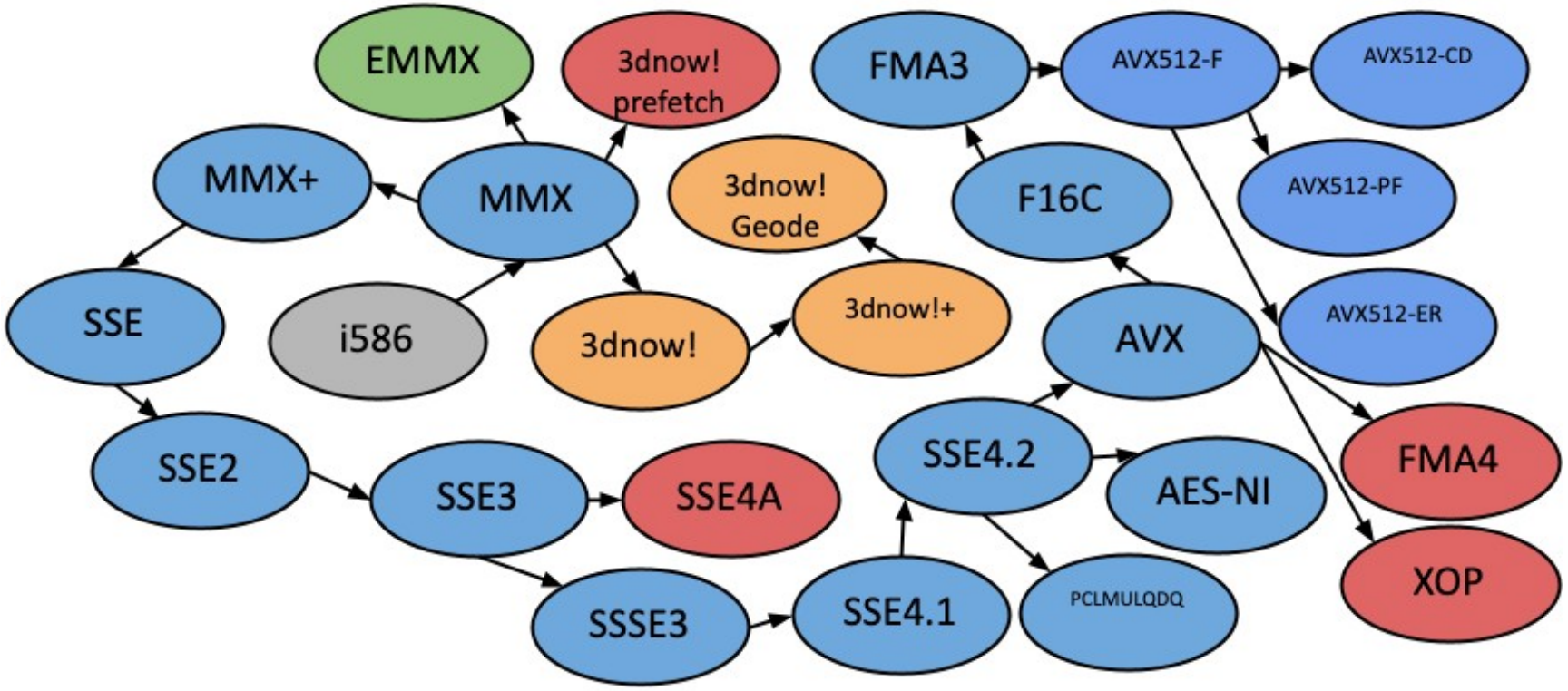
```
uint32x4_t a;
uint32x4_t b;

a = max(a, b); // Special instruction!
```

4 instrucciones RISC

- 2 SIMD load
- 1 SIMD max
- 1 SIMD store

Aunque puede ser algo confuso…



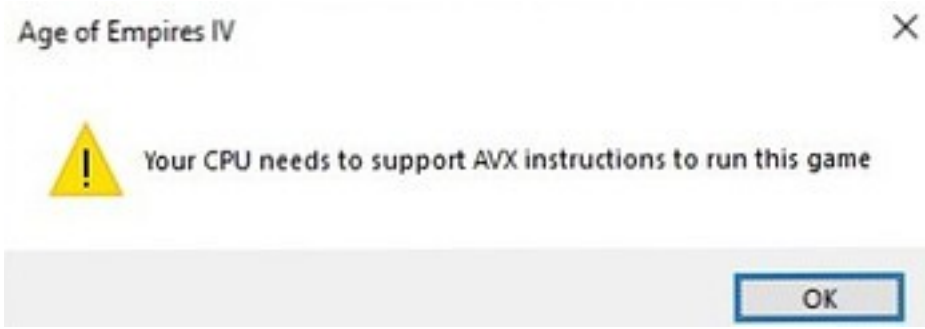
¿Cómo usarlo?

En gcc/clang, deben usar una arquitectura que soporte las instrucciones que deseen usar. Ejm:

-march=corei7

El compilador además intentará usar dichas instrucciones, de ser posible

NO pueden usar estas instrucciones en procesadores que no las soportan



Auto vectorización

Algunos compiladores intentarán usar instrucciones SIMD automáticamente
  gcc y clang lo hacen con la opción -O3 o -ftree-vectorize
  icc (Intel) lo hace por default

Lo anterior funciona para lenguajes como C, C++, Fortran

Lenguajes que usan el mismo backend, como Rust, también lo soportan

Otros lo han ido añadiendo recientemente
  .NET (C# y otros): https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-8/#vectorization

  JVM (Java y otros): Java Superword (https://eme64.github.io/blog/2023/02/23/SuperWord-Introduction.html)

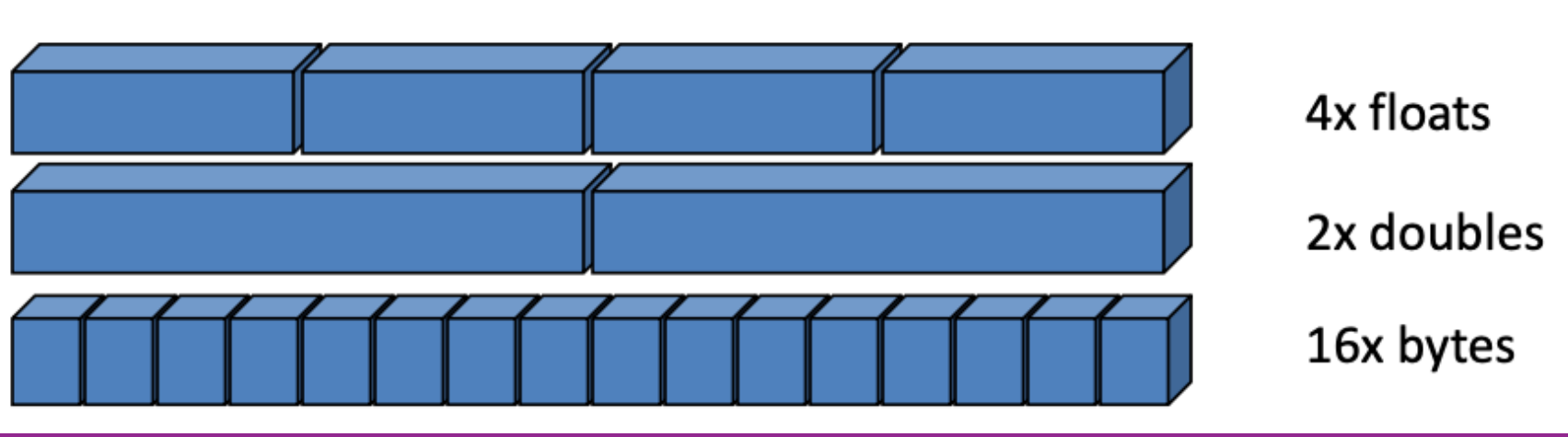
intrinsics

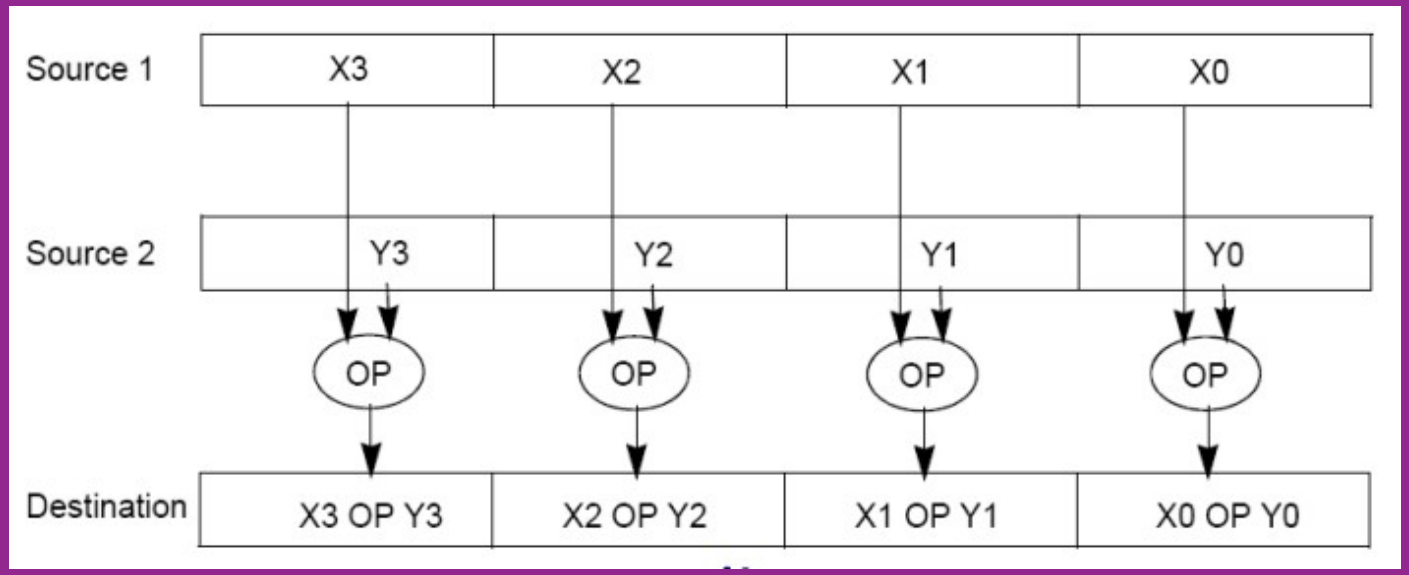
El compilador no siempre puede optimizar nuestro código usando SIMD

Requiere ayuda para seleccionar las instrucciones apropiadas

Se puede indicar qué instrucción en ensamblador usar usando intrinsics

Los intrinsics usualmente funcionan sobre tipos de datos especiales (vectores)





Registros SSE

Extensiones SIMD de Intel/AMD

MMX	• Registros de 64 bits, reusaba registros de punto flotante
SSE2/3/4	• 8 registros de 128 bits
AVX	• Registros de 256 bits
AVX512	• Registros de 512 bits

Registros Intel SSE2+ 128 bit SIMD

Word = 16 bits

Precisión simple FP: double word (32 bits)

Precisión doble FP: quad word (64 bits)

Divisiones:

- Packed bytes: 16 bytes
- Packed words: 8 words = 8x16bits
- Packed double words: 4 x 32 bits
- Packed quad words: 2 x 64 bits

Tipos de datos

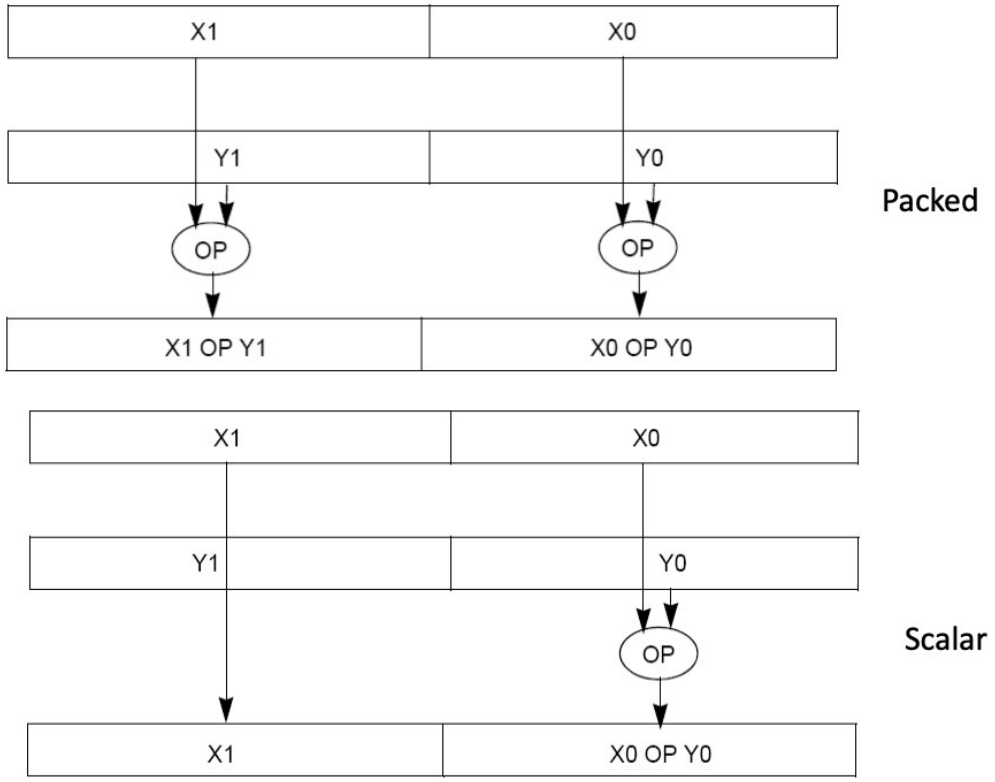
#include<x86intrin.h>

_m128
- 4 flotantes de precisión simple

_m128d
- 2 flotantes de precisión doble

_m128i
- 2 enteros de 64 bits
- 4 enteros de 32 bits
- 8 enteros de 16 bits
- 16 enteros de 8 bits

Packed vs scalar



Move does both load and store

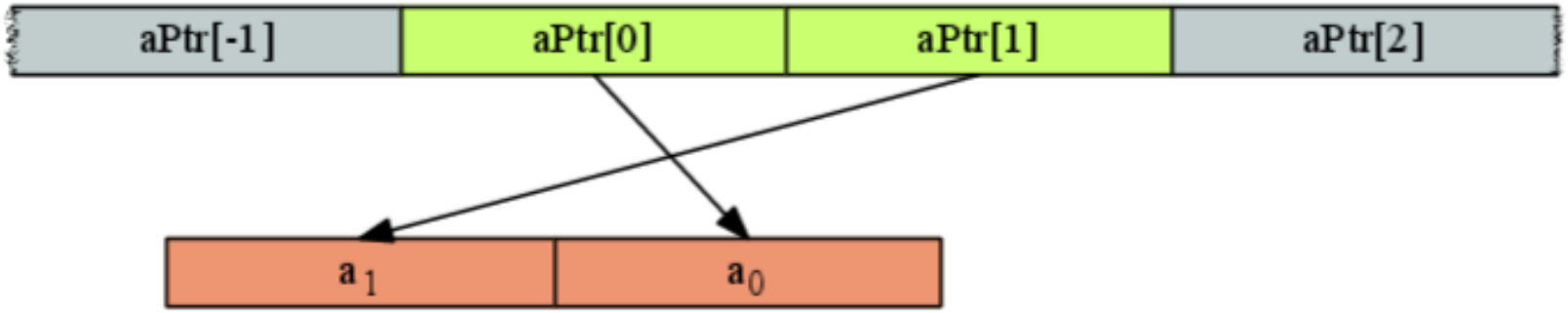
Data transfer	Arithmetic	Compare
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD{SS/PS/SD/PD} xmm, mem/xmm	CMP{SS/PS/SD/PD}
	SUB{SS/PS/SD/PD} xmm, mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	MUL{SS/PS/SD/PD} xmm, mem/xmm	
	DIV{SS/PS/SD/PD} xmm, mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD}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

Resumen de operaciones SSE/SSE2

En C++

```
double *aPtr = ...
// Load 2 doubles from memory
__m128d a = _mm_loadu_pd(aPtr);
// Store 2 doubles to memory
_mm_storeu_pd(aPtr, a);
```



En C++

Do operation <op> on a and b, write result to c:

__m128d c = _mm_<op>_pd(a, b);

Example operations:

// Addition: ci = ai + bi

c = _mm_add_pd(a, b);

// Subtraction: ci = ai - bi

c = _mm_sub_pd(a, b);

// Multiplication: ci = ai * bi

c = _mm_mul_pd(a, b);

// Division: ci = ai / bi

c = _mm_div_pd(a, b);

// Minimum: ci = fmin(ai, bi)

c = _mm_min_pd(a, b);

// Maximum: ci = fmax(ai, bi)

c = _mm_max_pd(a, b);

// Square root: ci = sqrt(ai)

c = _mm_sqrt_pd(a);

Transformar código a SIMD

Usualmente en loops

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

Transformar código a SIMD

Unroll loop!

```
For(i=0; i < 1000/4; 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
```

Transformar código a SIMD

Un loop de n iteraciones

K copias del cuerpo del loop (loop unrolling)

Caso de que n % k != 0:

Use el loop original para los últimos n % k elementos

O los primeros n % k. ¡¡¡Pero consideren el alineamiento de memoria!!!

Memoria alineada y rendimiento

Processor	Xeon E5-2670		Xeon X5550		Xeon E5440		Opteron 6282 SE		Opteron 8350		Opteron 244	
Frequency	2600 MHz		2667 MHz		2833 MHz		2600 MHz		2000 MHz		1800 MHz	
Microarchitecture	Sandy Bridge		Nehalem		Harpertown		Bulldozer		K10		K8	
Aligned	Yes	No	Yes	No	Yes	No	Yes	No	Yes	No	Yes	No
Naive	8.84	8.84	9.28	9.28	11.42	11.42	8.98	8.98	24.77	24.77	26.15	26.15
SSE2	1.36	1.36	1.56	1.56	1.99	1.99	1.2	1.23	1.78	1.78	2.18	2.18
SSE2 aligned load	1.36	1.37	1.54	1.54	1.97	1.99	1.21	1.23	1.78	1.86	1.66	1.75
AVX	0.78	0.89					0.83	0.9				
AVX aligned load	0.78	0.82					0.81	0.88				

SIMD es útil en muchos contextos

¡Usen su creatividad!

No todo es optimizar fórmulas matemáticas

On-Demand JSON: A Better Way To Parse Documents?
https://arxiv.org/abs/2312.17149

Number Parsing at a Gigabyte Per Second.
https://arxiv.org/abs/2101.11408

Validating UTF-8 in Less Than One Instruction Per Byte.
https://arxiv.org/abs/2010.03090

...

Referencias

SIMD Programming, UCSB CS 240A, 2017

Marat Dukhan SIMD Optimization

https://learn.microsoft.com/en-us/cpp/intrinsics/compiler-intrinsics?view=msvc-170