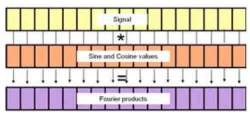
Vector Processing

(aka, Single Instruction Multiple Data, or SIMD)

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu



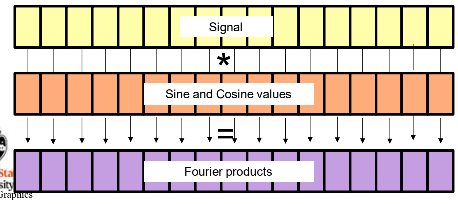
What is Vectorization/SIMD and Why do We Care?

Performance!

Many hardware architectures today, both CPU and GPU, allow you to perform arithmetic operations on multiple array elements simultaneously.

(Thus the label, “Single Instruction Multiple Data”.)

We care about this because many problems, especially scientific and engineering, can be cast this way. Examples include convolution, Fourier transform, power spectrum, autocorrelation, etc.



SIMD in Intel Chips

Year Released	Name	Width (bits)	Width (FP words)
1996	MMX	64	2
1999	SSE	128	4
2011	AVX	256	8
2013	AVX-512	512	16

Xeon Phi

Note: one complete cache line!
Also note: a 4x4 transformation matrix!

If you care:
- MMX stands for “MultiMedia Extensions”
- SSE stands for “Streaming SIMD Extensions”
- AVX stands for “Advanced Vector Extensions”

Sidebar: *Matrix* SIMD in Intel Chips

Intel has announced **AMX – the Advanced Matrix Extensions.**
It looks like this will multiply 16x16 matrices of data types fp16, int16, and int8.

AMX will be appearing starting with the 4th Generation Xeon Scalable Processors.

This is being billed as an “AI Acceleration Engine”. I suspect this is much like the Tensor Cores on Nvidia GPUs.

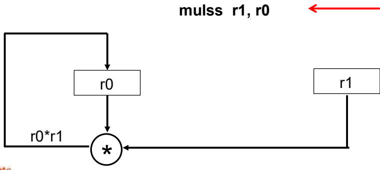
Intel SSE

Year Released	Name	Width (bits)	Width (FP words)
1996	MMX	64	2
1999	SSE	128	4
2011	AVX	256	8
2013	AVX-512	512	16

Intel and AMD CPU architectures support vectorization. The most well-known form is called Streaming SIMD Extension, or **SSE**. It allows four floating point operations to happen simultaneously.

Normally a *scalar* floating point multiplication instruction happens like this:

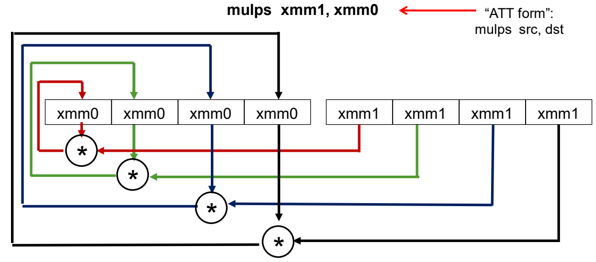
mulss r1, r0 ← “ATT form”: mulss src, dst



Intel SSE

The SSE version of the multiplication instruction happens like this:

mulps xmm1, xmm0 ← “ATT form”: mulps src, dst



SSE in the Kitchen? ☺

mulss r1, r0

mulps xmm1, xmm0

Requirements for a For-Loop to be SIMD'd

- If there are nested loops, the one to vectorize must be the inner one.
- There can be no jumps or branches. "Masked assignments" (an if-statement-controlled assignment) are OK, e.g.,

```
if( A[ i ] > 0. )
        B[ i ] = 1.;
```

- The total number of iterations must be known at runtime when the loop starts
- There can be no inter-loop data dependencies such as:

```
a[ i ] = a[ i-1 ] + 1.;
```

```
101st element            100th element
a[100]    =    a[99]   + 1.;   // this crosses an SSE boundary, so it is ok
a[101]    =    a[100] + 1.;    // this is within one SSE operation, so it is not OK
102nd element            101st element
```

- It helps performance if the elements have contiguous memory addresses.

This all sounds great! What is the catch?

The catch is that compilers haven't caught up to producing really efficient SIMD code. So, while there are great ways to express the desire for SIMD in code, you won't get the full potential speedup … yet.

One way to get a better speedup is to use assembly language.
Don't worry – *you* wouldn't need to write it.

Here are two assembly functions:

1. SimdMul: C[0:len] = A[0:len] * B[0:len]
2. SimdMulSum: return (ΣA[0:len] * B[0:len])

Warning – due to the nature of how different compilers and systems handle local variables, these two functions only work on *flip* and *rabbit* using gcc/g++, without any optimization !!!

Getting at the full SIMD power until compilers catch up


```
void
SimdMul( float *a, float *b,   float *c,   int len )
{
	int limit = ( len/SSE_WIDTH ) * SSE_WIDTH;
	__asm
	(
		".att_syntax\n\t"
		"movq    -24(%rbp), %r8\n\t"          // a
		"movq    -32(%rbp), %rcx\n\t"         // b
		"movq    -40(%rbp), %rdx\n\t"         // c
	);

	for( int i = 0; i < limit; i += SSE_WIDTH )
	{
		__asm
		(
			".att_syntax\n\t"
			"movups  (%r8), %xmm0\n\t"        // load the first sse register
			"movups  (%rcx), %xmm1\n\t"       // load the second sse register
			"mulps  %xmm1, %xmm0\n\t"         // do the multiply
			"movups  %xmm0, (%rdx)\n\t"       // store the result
			"addq $16, %r8\n\t"
			"addq $16, %rcx\n\t"
			"addq $16, %rdx\n\t"
		);
	}

	for( int i = limit; i < len; i++ )
	{
		c[ i ] = a[ i ] * b[ i ];
	}
}
```

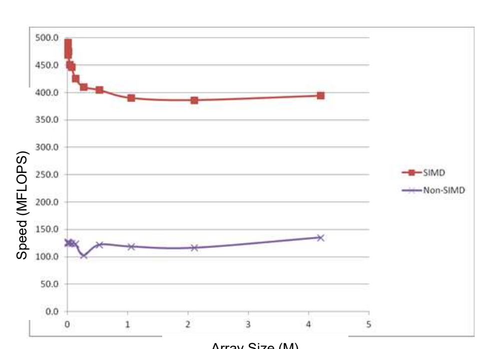
This only works on *flip* and *rabbit* using gcc/g++, without any optimization !!!

Getting at the full SIMD power until compilers catch up


```
float
SimdMulSum( float *a, float *b, int len )
{
	float sum[4] = { 0., 0., 0., 0. };
	int limit = ( len/SSE_WIDTH ) * SSE_WIDTH;

	__asm
	(
		".att_syntax\n\t"
		"movq    -40(%rbp), %r8\n\t"          // a
		"movq    -48(%rbp), %rcx\n\t"         // b
		"leaq     -32(%rbp), %rdx\n\t"        // &sum[0]
		"movups  (%rdx), %xmm2\n\t"           // 4 copies of 0. in xmm2
	);

	for( int i = 0; i < limit; i += SSE_WIDTH )
	{
		__asm
		(
			".att_syntax\n\t"
			"movups (%r8), %xmm0\n\t"         // load the first sse register
			"movups (%rcx), %xmm1\n\t"        // load the second sse register
			"mulps  %xmm1, %xmm0\n\t"         // do the multiply
			"addps  %xmm0, %xmm2\n\t"         // do the add
			"addq $16, %r8\n\t"
			"addq $16, %rcx\n\t"
		);
	}

	__asm
	(
		".att_syntax\n\t"
		"movups  %xmm2, (%rdx)\n\t"           // copy the sums back to sum[ ]
	);

	for( int i = limit; i < len; i++ )
	{
		sum[0] += a[ i ] * b[ i ];
	}

	return sum[0] + sum[1] + sum[2] + sum[3];
}
```

This only works on *flip* and *rabbit* using gcc/g++, without any optimization !!!

Array*Array Multiplication Speed



Avoiding Assembly Language: SIMD using the OpenMP SIMD Pragma

Array * Array

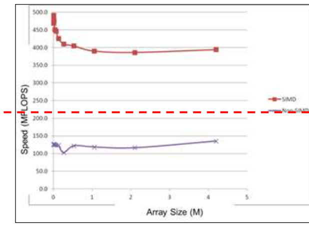
```
void
SimdMul( float *a, float *b,   float *c,   int len )
{
        #pragma omp simd
        for( int i= 0; i < len; i++ )
                c[ i ]  =  a[ i ] * b[ i ];
}
```

Array * Scalar

```
void
SimdMul( float *a, float b,  float *c,   int len )
{
        #pragma omp simd
        for( int i = 0; i < len; i++ )
                c[ i ]  =  a[ i ] * b;
}
```


Avoiding Assembly Language: SIMD using the OpenMP SIMD Pragma


```
#pragma omp simd
for( int i = 0; i < ArraySize; i++ )
{
        c[ i ] = a[ i ] * b[ i ];
}
```



#pragma omp simd

Avoiding Assembly Language: the Intel Intrinsics

Intel has a mechanism to get at the SSE SIMD without resorting to assembly language. These are called ***Intrinsics***.

Intrinsic	Meaning
__m128	Declaration for a 128 bit 4-float word
_mm_loadu_ps	Load a __m128 word from memory
_mm_storeu_ps	Store a __m128 word into memory
_mm_mul_ps	Multiply two __m128 words
_mm_add_ps	Add two __m128 words

SimdMul using Intel Intrinsics


```
#include <xmmintrin.h>
#define SSE_WIDTH          4

void
SimdMul( float *a, float *b,   float *c,   int len )
{
        int limit = ( len/SSE_WIDTH ) * SSE_WIDTH;
        register float *pa = a;
        register float *pb = b;
        register float *pc = c;
        for( int i = 0; i < limit; i += SSE_WIDTH )
        {
                _mm_storeu_ps( pc,  _mm_mul_ps( _mm_loadu_ps( pa ), _mm_loadu_ps( pb ) ) );
                pa += SSE_WIDTH;
                pb += SSE_WIDTH;
                pc += SSE_WIDTH;
        }

        for( int i = limit; i < len; i++ )
        {
                c[i] = a[i] * b[i];
        }
}
```


SimdMulSum using Intel Intrinsics


```
float
SimdMulSum( float *a, float *b, int len )
{
        float sum[4] = { 0., 0., 0., 0. };
        int limit = ( len/SSE_WIDTH ) * SSE_WIDTH;
        register float *pa = a;
        register float *pb = b;

        __m128  ss = _mm_loadu_ps( &sum[0] );
        for( int i = 0; i < limit; i += SSE_WIDTH )
        {
                ss = _mm_add_ps( ss,  _mm_mul_ps( _mm_loadu_ps( pa ),  _mm_loadu_ps( pb ) ) );
                pa += SSE_WIDTH;
                pb += SSE_WIDTH;
        }
        _mm_storeu_ps( &sum[0], ss );

        for( int i = limit; i < len; i++ )
        {
                sum[0] += a[ i ] * b[ i ];
        }

        return sum[0] + sum[1] + sum[2] + sum[3];
}
```


Intel Intrinsics

Each Core Has Its Own SIMD Unit! Thus, You Can Combine SIMD *and* Multicore

19

```
#define NUM_ELEMENTS_PER_CORE          ( ARRAYSIZE / NUMT )

. . .

omp_set_num_threads( NUMT );
double maxMegaMultsPerSecond = 0.;

double time0 = omp_get_wtime( );
#pragma omp parallel
{
        int thisThread = omp_get_thread_num( );
        int first = thisThread * NUM_ELEMENTS_PER_CORE;
        SimdMul( &A[first], &B[first],   &C[first],   NUM_ELEMENTS_PER_CORE );
}
double time1 = omp_get_wtime( );
double megaMultsPerSecond = (double)ARRAYSIZE / ( time1 - time0 ) / 1000000.;
. . .
```

The variable *first* is the first array element that *thisThread* is in charge of.

&A[first] is the memory address of *thisThread*'s first element.

Oregon State University Computer Graphics

Combining SIMD with Multicore

20

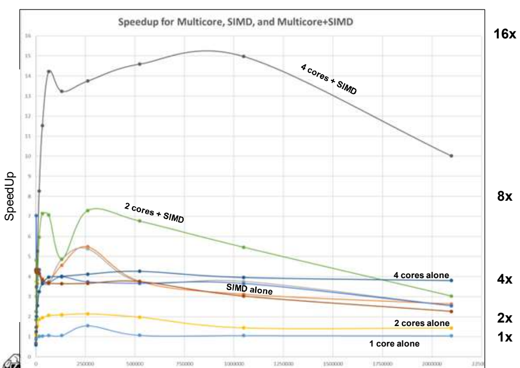
Notes:

- Remember that **#pragma omp parallel** creates a thread team and that *all* threads execute *everything* in the curly braces.
- The variable **thisThread** is the thread number of the thread who is executing this code right now. There will eventually be NUMT threads who get to execute this code. Thus, all the instances of **thisThread** will be between 0 and NUMT-1 .
- The variable **first** is the first array element number that **thisThread** will execute.
- Starting the SIMD multiplications at **&A[first], &B[first], &C[first]** gives each thread its very own set of contiguous array elements to work on. The **SimdMul** function depends on this.

Oregon State University Computer Graphics

Combining SIMD with Multicore

21



- Speedups are with respect to a for-loop with no multicore or SIMD.
- "cores alone" = a for-loop with "#pragma omp parallel for".
- "cores + SIMD" = as the code looks on last two slides

Oregon State University Computer Graphics

Prefetching

22

Prefetching is used to place a cache line in memory before it is to be used, thus hiding the latency of fetching from off-chip memory.

There are two key issues here:
1. Issuing the prefetch at the right time
2. Issuing the prefetch at the right distance

The right time:
If the prefetch is issued too late, then the memory values won't be back when the program wants to use them, and the processor has to wait anyway.

If the prefetch is issued too early, then there is a chance that the prefetched values could be evicted from cache by another need before they can be used.

The right distance:
The "prefetch distance" is how far ahead the prefetch memory is than the memory we are using right now.

Too far, and the values sit in cache for too long, and possibly get evicted.

Too near, and the program is ready for the values before they have arrived.

Oregon State University Computer Graphics

The Effects of Prefetching on SIMD Computations

23

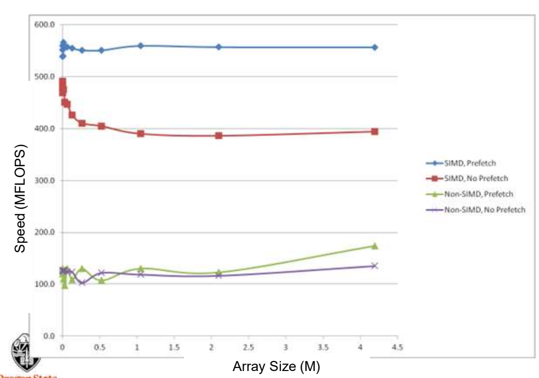
Array Multiplication
Length of Arrays (NUM): 1,000,000
Length per SIMD call (ONETIME): 256

```
for(  int i = 0;  i < NUM;  i += ONETIME )
{
        __builtin_prefetch ( &A[i+PD],  WILL_READ_ONLY,            LOCALITY_LOW );
        __builtin_prefetch ( &B[i+PD],  WILL_READ_ONLY,            LOCALITY_LOW );
        __builtin_prefetch ( &C[i+PD],  WILL_READ_AND_WRITE,  LOCALITY_LOW );

        SimdMul( A, B,  C,  ONETIME );
}
```

Oregon State University Computer Graphics

The Effects of Prefetching on SIMD Computations

24



Oregon State University Computer Graphics

- SIMD is an important way to achieve array-operation speed-ups on a CPU
- For now, you might have to write in assembly language to get to all of it
- I suspect that *#pragma omp simd* will catch up
- I suspect that *Intel Intrinsics* will catch up
- Prefetching can really help SIMD

Oregon State University
Computer Graphics