Vector Processing
(aka, Single Instruction Multiple Data, or SIMD)
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

<table>
<thead>
<tr>
<th>Year Released</th>
<th>Name</th>
<th>Width (bits)</th>
<th>Width (FP words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1996</td>
<td>MMX</td>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>1999</td>
<td>SSE</td>
<td>128</td>
<td>4</td>
</tr>
<tr>
<td>2011</td>
<td>AVX</td>
<td>256</td>
<td>8</td>
</tr>
<tr>
<td>2013</td>
<td>AVX-512</td>
<td>512</td>
<td>16</td>
</tr>
</tbody>
</table>

**Note:** one complete cache line!

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

\[
\text{mulss } r1, r0
\]

“ATT form”:

\[
\text{mulss src, dst}
\]

<table>
<thead>
<tr>
<th>Year Released</th>
<th>Name</th>
<th>Width (bits)</th>
<th>Width (FP words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1996</td>
<td>MMX</td>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>1999</td>
<td>SSE</td>
<td>128</td>
<td>4</td>
</tr>
<tr>
<td>2011</td>
<td>AVX</td>
<td>256</td>
<td>8</td>
</tr>
<tr>
<td>2013</td>
<td>AVX-512</td>
<td>512</td>
<td>16</td>
</tr>
</tbody>
</table>
The SSE version of the multiplication instruction happens like this:

```
mulps xmm1, xmm0
```

“ATT form”:

```
mulps src, dst
```
**SIMD Multiplication**

**Array * Array**

```c
void SimdMul( float *a, float *b, float *c, int len )
{
    c[0:len] = a[0:len] * b[0:len];
}
```

Note that the construct:

```
a[ 0 : ArraySize ]
```

is meant to be read as:

“The set of elements in the array *a* starting at index 0 and going for *ArraySize* elements”.

**not as:**

“The set of elements in the array *a* starting at index 0 and going through index *ArraySize*”.

**Array * Array**

```c
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];
}
```
SIMD Multiplication

Array * Scalar

```c
void SimdMul( float *a, float b, float *c, int len )
{
    c[0:len] = a[0:len] * b;
}
```

Array * Scalar

```c
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;
}
```
Array*Array Multiplication Speed

![Graph showing the speed of Array*Array multiplication for SIMD and Non-SIMD methods as a function of array size. The x-axis represents the array size (M), and the y-axis represents speed (MFLOPS). The graph compares the performance of SIMD and Non-SIMD methods across different array sizes.](image-url)
You would think it would always be 4.0 ± noise effects, but it’s not. Why?
#pragma omp simd

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

![Graph showing performance comparison between SIMD and Non-SIMD](image_url)

#pragma omp simd
• 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 cannot be any backward loop dependencies, like this:
  

• It helps if the elements have contiguous memory addresses.
Prefetching

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

```c
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 );
}
```
The Effects of Prefetching on SIMD Computations

![Graph showing the effects of prefetching on SIMD computations. The x-axis represents Array Size (M), and the y-axis represents Speed (MFLOPS). The graph compares different cases: SIMD, Prefetch; SIMD, No Prefetch; Non-SIMD, Prefetch; Non-SIMD, No Prefetch. Each line represents a different case, with distinct markers and colors for easy identification.]
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.

So, for the CPU SIMD project, we are going to investigate the potential speedup using assembly language. Don’t worry – you don’t need to write it.

You will be given two assembly functions:

1. SimdMul: \( C[0:len] = A[0:len] \times B[0:len] \)

2. SimdMulSum: return \( \sum A[0:len] \times B[0:len] \)

Warning – due to the nature of how different compilers and systems handle local variables, these two functions only work on flip using gcc/g++, without –O3 !!!
void SimdMul( float *a, float *b, float *c, int len )
{
    int limit = ( len/SSE_WIDTH ) * SSE_WIDTH;
    __asm
    {
        ".att_syntax
	"movq -24(%rbp), %r8\n" // a
	"movq -32(%rbp), %rcx\n" // b
	"movq -40(%rbp), %rdx\n" // c
    );

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

    for( int i = limit; i < len; i++ )
    {
        \n            c[i] = a[i] * b[i];
    }
}
Getting at the full SIMD power until compilers catch up

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

    __asm
    {
        "movq -40(%rbp), %r8
          movq -48(%rbp), %rcx
          leaq -32(%rbp), %rdx
          movups (%rdx), %xmm2
          movq -40(%rbp), %r8
          movq -48(%rbp), %rcx
          leaq -32(%rbp), %rdx
          movups (%rdx), %xmm2
        
        for( int i = 0; i < limit; i += SSE_WIDTH )
        {
            __asm
            {
                movups (%r8), %xmm0
                movups (%rcx), %xmm1
                mulps %xmm1, %xmm0
                addps %xmm0, %xmm2
            };
        };

        __asm
        {
            movups %xmm2, (%rdx)
        };

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

    }
}
```

This only works on float using gcc/g++, without –O3 !!!
When we get to OpenCL, we could compute projectile physics like this:

```
float4 pp; // p'
pp.x = p.x + v.x*DT;
pp.y = p.y + v.y*DT + .5*DT*DT*G.y;
pp.z = p.z + v.z*DT;
```

But, instead, we will do it like this:

```
float4 pp = p + v*DT + .5*DT*DT*G; // p'
```

We do it this way for two reasons:
1. Convenience and clean coding
2. Some hardware can do multiple arithmetic operations simultaneously
The whole thing will look like this:

```cpp
constant float4 G = (float4) ( 0., -9.8, 0., 0. );
constant float DT = 0.1;

kernel void Particle(  global float4 * dPobj,  global float4 * dVel,  global float4 * dCobj )
{
    int gid = get_global_id( 0 );  // particle #
    float4 p = dPobj[gid];        // particle #gid’s position
    float4 v = dVel[gid];         // particle #gid’s velocity

    float4 pp = p + v*DT + .5*DT*DT*G;  // p’
    float4 vp = v + G*DT;            // v’

    dPobj[gid] = pp;
    dVel[gid] = vp;
}
```

A preview of things to come:
OpenCL and CUDA have a data type called “float4”
• SIMD is an important way to achieve speed-ups on a CPU

• You might have to write in assembly language to get to all of it

• I suspect that `#pragma omp simd` will eventually catch up

• Prefetching can really help SIMD