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>

Xeon Phi

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

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

Intel SSE

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
```

r0*r1
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 Array * Array Multiplication Speed](image_url)

- **X-axis**: Array Size (M)
- **Y-axis**: Speed (MFLOPS)

Comparing SIMD and Non-SIMD methods.
You would think it would always be 4.0 ± noise effects, but it’s not. Why?

SIMD in OpenMP 4.0

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

Array*Array Multiplication Speedup

![Graph showing speedup of SIMD over Non-SIMD with Array Size (M) on the x-axis and Speedup of SIMD over Non-SIMD on the y-axis. The graph has a blue line with data points.]
Requirements for a For-Loop to be Vectorized

• 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.,
  \[
  \text{if( } A[ i ] > 0. ) \]
  \[
  B[ i ] = 1.;
  \]

• The total number of iterations must be known at runtime when the loop starts

• There can be no inter-loop data dependencies such as:
  \[
  a[ i ] = a[ i-1 ] + 1.;
  \]

- It helps performance 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.
The Effects of Prefetching on SIMD Computations

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 );
}
```

### Array Size (M)

<table>
<thead>
<tr>
<th>Speed (MFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>600.0</td>
</tr>
<tr>
<td>500.0</td>
</tr>
<tr>
<td>400.0</td>
</tr>
<tr>
<td>300.0</td>
</tr>
<tr>
<td>200.0</td>
</tr>
<tr>
<td>100.0</td>
</tr>
<tr>
<td>0.0</td>
</tr>
</tbody>
</table>

Array Size (M)

- SIMD, Prefetch
- SIMD, No Prefetch
- Non-SIMD, Prefetch
- Non-SIMD, No Prefetch
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:  \[ C[0:len] = A[0:len] \times B[0:len] \]
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 !!!

Getting at the full SIMD power until compilers catch up

```c
void SimdMul( float *a, float *b, float *c, int len )
{
    int limit = ( len/SSE_WIDTH ) * SSE_WIDTH;
    __asm
    {
        .att_syntax
        "movq -24(%rbp), %r8
         movq -32(%rbp), %rcx
         movq -40(%rbp), %rdx
         
        for( int i = 0; i < limit; i += SSE_WIDTH )
        {
            __asm
            {
                .att_syntax
                "movups (%r8), %xmm0
                movups (%rcx), %xmm1
                "mulps %xmm1, %xmm0
                "movups %xmm0, (%rdx)
                "addq $16, %r8
                "addq $16, %rcx
                "addq $16, %rdx
                
                for( int i = limit; i < len; i++ )
                {
                    c[i] = a[i] \times b[i];
                }
            }
        }
    }
}
```

This only works on *flip* using gcc/g++, without –O3 !!!
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
    {
        "att_syntax
        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
            {
                "att_syntax
                movups (%r8), %xmm0
                movups (%rcx), %xmm1
                mulps %xmm1, %xmm0
                addps %xmm0, %xmm2
                addq $16, %r8
                addq $16, %rcx
                
                movups %xmm2, (%rdx)
            }
            __asm
            {
                "att_syntax
                movups %xmm0, (%rdx)
            }
        }
        for( int i = limit; i < len; i++ )
        {
            sum[0] += a[ i ] * b[ i ];
        }
    }
}
```

This only works on flip using gcc/g++, without -O3 !!!

Combining SIMD with Multicore

```c
#define NUM_ELEMENTS_PER_CORE ARRAYSIZE / NUMT

omp_set_num_threads( NUMT );
maxMegaMultsPerSecond = 0.;
for( int t = 0; t < NUM_TRIES; t++ )
{
    double time0 = omp_get_wtime( );
    #pragma omp parallel
    {
        int first = omp_get_thread_num( ) * NUM_ELEMENTS_PER_CORE;
        SimdMul( &A[first], &B[first], &C[first], NUM_ELEMENTS_PER_CORE );
    }
    double time1 = omp_get_wtime( );
    double megaMultsPerSecond = (float)ARRAYSIZE / ( time1 - time0 ) / 1000000.;
    if( megaMultsPerSecond > maxMegaMultsPerSecond )
        maxMegaMultsPerSecond = megaMultsPerSecond;
}
```
Combining SIMD with Multicore

Speedup for Multicore, SIMD, and Multicore+SIMD

- 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 the previous page

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.

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>__m128</td>
<td>Declaration for a 128 bit 4-float word</td>
</tr>
<tr>
<td>_mm_loadu_ps</td>
<td>Load a __m128 word from memory</td>
</tr>
<tr>
<td>_mm_storeu_ps</td>
<td>Store a __m128 word into memory</td>
</tr>
<tr>
<td>_mm_mul_ps</td>
<td>Multiply two __m128 words</td>
</tr>
<tr>
<td>_mm_add_ps</td>
<td>Add two __m128 words</td>
</tr>
</tbody>
</table>
#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];
    }
}

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];
    }
}
Why do the Intrinsics do so well with a small dataset size?

It’s not due to the code in the inner-loop:

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

It’s actually due to the setup time. The intrinsics have a tighter coupling to the setting up of the registers. A smaller setup time makes the small dataset size speedup look better.
When we get to OpenCL, we could compute projectile physics like this:

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

```c
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:

```c
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;
}
```