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>

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
Computer Graphics

Intel SSE

The SSE version of the multiplication instruction happens like this:

mulps xmm1, xmm0

"ATT form": mulps src, dst

SIMD Multiplication

Array * Array

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

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

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

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;

Array * Array Multiplication Speed

Array * Array Multiplication Speed

Speed (MFLOPS)

Array Size (M)

<table>
<thead>
<tr>
<th>Size (M)</th>
<th>Speed MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5000</td>
</tr>
<tr>
<td>1</td>
<td>4500</td>
</tr>
<tr>
<td>2</td>
<td>4000</td>
</tr>
<tr>
<td>3</td>
<td>3500</td>
</tr>
<tr>
<td>4</td>
<td>3000</td>
</tr>
<tr>
<td>5</td>
<td>2500</td>
</tr>
</tbody>
</table>

Note: The graph shows the speed of array multiplication in MFLOPS as a function of array size in M. The data points indicate a decrease in speed with increasing array size, reflecting the efficiency of the SIMD multiplication operation.
You would think it would always be 4.0 ± noise effects, but it’s not. Why?

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.,
  ```c
  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:
  ```c
  a[i] = a[i+1] + 1.;
  ```
  10th element 11th element
  ```c
  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
  ```
• 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

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 );
}

SimdMul: C[0:len] = A[0:len] * B[0:len]
SimdMulSum: return ( SUM(A[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 \texttt{flip} using \texttt{gcc/g++}, without \texttt{–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;

    #ifdef _MIPS
        mov -40(%rbp), %r8               // a
        mov -48(%rbp), %rcx             // b
        leaq -32(%rbp), %rdx            // &sum[0]
        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
"
            );
    #endif

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

    __asm
        (".att_syntax
		mov -40(%rbp), %r8               // a
        mov -48(%rbp), %rcx             // b
        leaq -32(%rbp), %rdx            // &sum[0]
        addq %rdx, %r8

        "asm syntax"
        "movups (%rcx), %xmm1
        "addps %xmm1, %xmm0"
        "addq $16, %rcx"
    )

    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
        { 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;
}
```

Avoiding Assembly Language: the Intel 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>

Intel has a mechanism to get at the SSE SIMD without resorting to assembly language. These are called Intrinsics.
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[i] += 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:

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:

```plaintext
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:

```plaintext
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:

```plaintext
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;
}
```

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