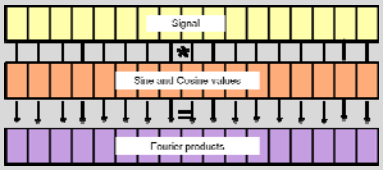
Vector Processing

(aka, Single Instruction Multiple Data)

Mike Bailey

mjb@cs.oregonstate.edu

Oregon State University



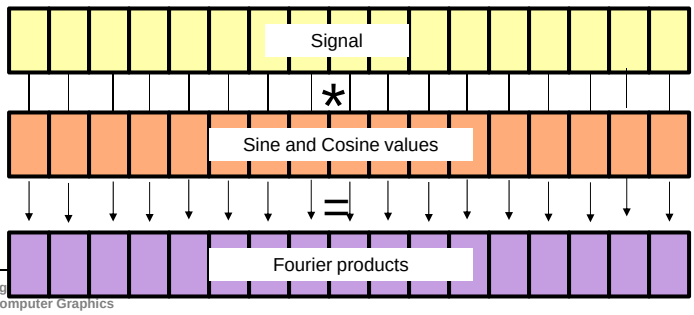
What is Vectorization/SIMD and Why do We Care?

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.



Remember Using float4's in OpenCL?

```
typedef  float4  point;
typedef  float4  vector;
typedef  float4  color;

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

kernel
void
Particle(  global point * dPobj,  global vector * dVel,  global color * dCobj  )
{
        int gid    = get_global_id( 0 );                    // particle #
        point p    = dPobj[gid];
        vector v = dVel[gid];

        point   pp   = p + v*DT + .5*DT*DT*G;               // p'
        vector vp   = v + G*DT;                             // v'

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

Oregon State University
Computer Graphics

Remember using float4's in OpenCL? Why did we do it that way?

Instead of doing this:

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

we could have done this instead:

```
        float 4 pp;
        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;
        pp.w     = 1.;
```

But, we did it the first way for two reasons:

1. Convenience and clean coding
2. Some hardware can do multiple arithmetic operations simultaneously

This is called Single Instruction-Multiple-Data (SIMD), or Vectorization.

Oregon State University
Computer Graphics

mjb – March 16, 2015

SIMD in Intel Chips

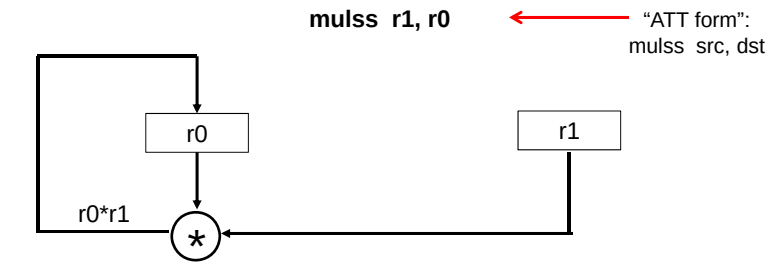
Year Released	Name	Width (bits)	Width (FP words)
1996	MMX	64	2
1999	SSE	128	4
2011	AVX	256	8
2013	AVX-512	512	16

Xeon Phi

Intel SSE

Intel architecture supports vectorization. The most well-known form is called Streaming SIMD Extension, or **SSE**. It allows four floating point operations to happen simultaneously.

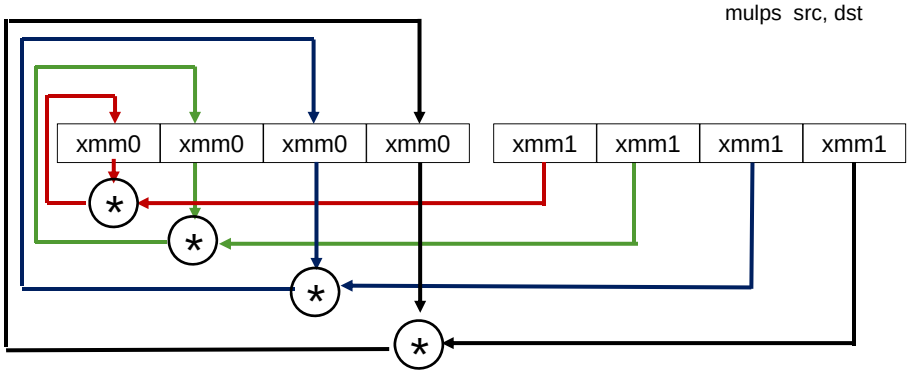
Normally a scalar floating point multiplication instruction happens like this:



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

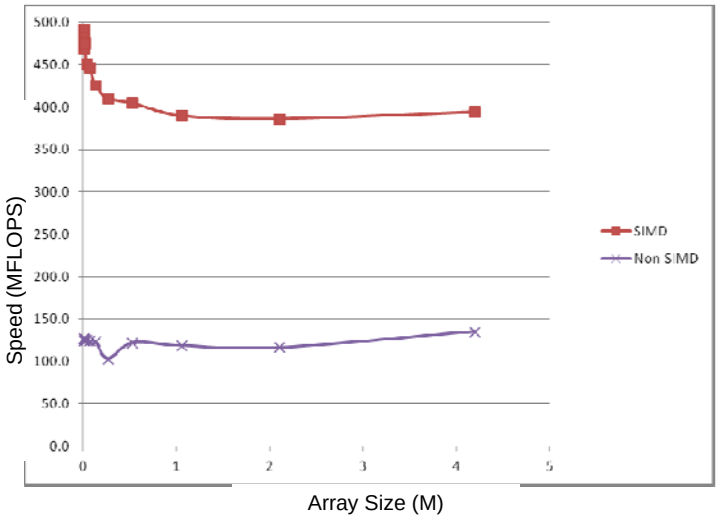
Array * Scalar

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

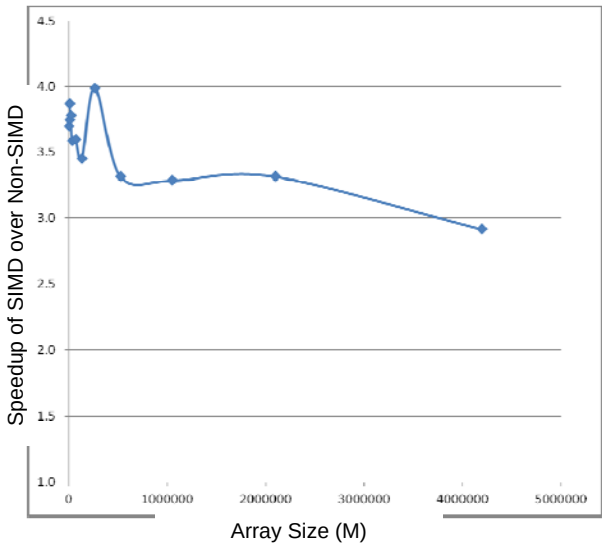
Array * Scalar

```
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 Speedup



You would think it would always be 4.0 ± noise effects, but it's not. Why? The answer is coming up.

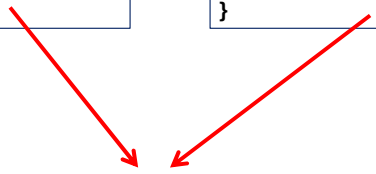
Combining SIMD with Parallel Threading in OpenMP 4.0

```
#pragma omp parallel for

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

```
#pragma omp simd

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



```
#pragma omp parallel for simd

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

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.

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

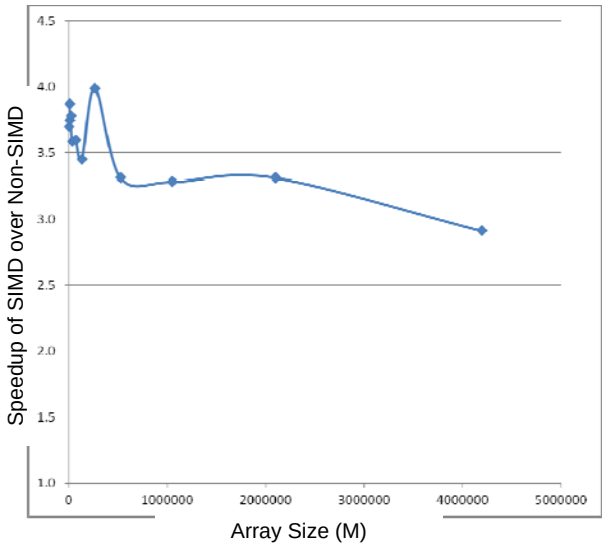
• There cannot be any backward loop dependencies.

• It helps if the elements have contiguous memory addresses.

• The C/C++ ***restrict*** keyword might help the compiler. ***restrict*** tells the compiler that the area of memory that this pointer is used to address does not overlap any other pointer's area. E.g.,

restrict float *p;

Remember the Array*Array Multiplication Speedup Graph?



You would think it would always be 4.0 ± noise effects, but it's not. Why? The answer is that racing through the arrays violates the idea of cache coherence.

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 );
}
```

The Effects of Prefetching on SIMD Computations