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

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.

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!

Intel SSE

The SSE version of the multiplication instruction happens like this:

**ATT form**: mulps xmm src, dst

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

Arrays * Array

Note that the construct:

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

SIMD Multiplication
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

You would think it would always be 4.0 ± noise effects, but it’s not. Why?

Combining SIMD with Parallel Threading in OpenMP 4.0

```c
#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, e.g.,
  ```c
  if(A[i] > 0.0) B[i] = 1.0;
  ```
- The total number of iterations must be known at runtime when the loop starts
- There cannot be any backward loop dependencies, like this:
  ```c
  A[i] = A[i-1] + 1.0;
  ```
- 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.
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);
}

Array Size (M)  Speed (MFLOPS)

This all sounds great! What is the catch?

The catch is that compilers haven’t caught up to producing 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:
2. 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 gcc/g++, without –O3 !!!

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
        movq -24(%rbp), %rbx
        movq -32(%rbp), %rcx
        movq -40(%rbp), %rdx
    );
    for( int i = 0; i < limit; i += SSE_WIDTH )
    {
        __asm
        (.att_syntax
            movups (%rbx), %xmm0
            movups (%rcx), %xmm1
            mulps %xmm1, %xmm0
            movups %xmm0, (%rdx)
            addq $16, %rbx
            addq $16, %rcx
            addq $16, %rdx
        );
    }
    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;
    __asm
    (.att_syntax
        movq -40(%rbp), %rbx
        movq -48(%rbp), %rcx
        leaq -32(%rbp), %rdx
        movups (%rdx), %xmm2
    );
    for( int i = 0; i < limit; i += SSE_WIDTH )
    {
        __asm
        (.att_syntax
            movups (%rbx), %xmm0
            movups (%rcx), %xmm1
            mulps %xmm1, %xmm0
            addps %xmm0, %xmm2
            addq $16, %rbx
            addq $16, %rcx
        );
    }
    __asm
    (.att_syntax
        movups %xmm2, (%rdx)
    );
    for( int i = limit; i < len; i++ )
    { sum[i-limit] += a[i] * b[i];
    }
}

This only works on \texttt{flip} using gcc/g++, without –O3 !!!

Getting at the full SIMD power until compilers catch up

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;

When we get to OpenCL, we could compute projectile physics like this:

But, instead, we will do it like this:

We do it this way for two reasons:
1. Convenience and clean coding
2. Some hardware can do multiple arithmetic operations simultaneously
A preview of things to come:

OpenCL has a data type called "float4"

```
constant float4 G               = (float4) ( 0., -9.8, 0., 0. );
constant float4 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;
}
```