Computer Graphics

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

Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

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!
Also note: a 4x4 transformation matrix!

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

Sidebar: Matrix SIMD in Intel Chips

Intel has announced **AMX – the Advanced Matrix Extensions**.
It looks like this will multiply 16x16 matrices of data types fp16, int16, and int8.

AMX will be appearing starting with the 4th Generation Xeon Scalable Processors.

This is being billed as an “AI Acceleration Engine”. I suspect this is much like the Tensor Cores on Nvidia GPUs.
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
```

The SSE version of the multiplication instruction happens like this:

```
mulps xmm1, xmm0
```

"ATT form":
```
mulps src, dst
```
Requirements for a For-Loop to be SIMD'd

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

  ![Example of memory address dependency]

  - It helps performance if the elements have contiguous memory addresses.
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: \[ \text{return } \left( \sum A[0:len] \times B[0:len] \right) \]

Warning – due to the nature of how different compilers and systems handle local variables, these two functions only work on flip and rabbit using gcc/g++, without any optimization !!!

```c
void SimdMul( float *a, float *b, float *c, int len )
{
  int limit = ( len/SSE_WIDTH ) * SSE_WIDTH;
  __asm
  {
    "att_syntax/init"
    "movq -24(%rbp), %r8"  // a
    "movq -32(%rbp), %rcx" // b
    "movq -40(%rbp), %rdx" // c
  };
  for( int i = 0; i < limit; i += SSE_WIDTH )
  {
    __asm
    {
      "att_syntax/init"
      "movups (%r8), %xmm0"  // load the first sse register
      "movups (%rcx), %xmm1"  // load the second sse register
      "mulps %xmm1, %xmm0"    // do the multiply
      "movups %xmm0, (%rdx)"  // store the result
      "addq $16, %rdx"       // store result
      "addq $16, %r8"        // load next sse register
      "addq $16, %rcx" 
    };
    for( int i = limit; i < len; i++ )
    {
      c[i] = a[i] * b[i];
    }
  }
```

Getting at the full SIMD power until compilers catch up

This only works on flip and rabbit using gcc/g++, without any optimization !!!
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
            
            }
        }
        __asm
        {
            "att_syntax
            "movups %xmm2, (%rdx)
        
        }
        for( int i = limit; i < len; i++ )
        {
            sum[0] += a[ i ] * b[ i ];
        }

    }

This only works on flip and rabbit using gcc/g++, without any optimization !!!
Avoiding Assembly Language: SIMD using the OpenMP SIMD Pragma

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 ];
}
```

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;
}
```

Avoiding Assembly Language: SIMD using the OpenMP SIMD Pragma

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

SimdMul using Intel Intrinsics

```c
#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, 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];
    }

}
#define NUM_ELEMENTS_PER_CORE           ( ARRAYSIZE / NUMT )

... 
omp_set_num_threads( NUMT );
double maxMegaMultsPerSecond = 0.;

double time0 = omp_get_wtime( );
#pragma omp parallel
{
    int thisThread = omp_get_thread_num( );
    int first = thisThread * NUM_ELEMENTS_PER_CORE;
    SimdMul( &A[first], &B[first], &C[first], NUM_ELEMENTS_PER_CORE );
}

double time1 = omp_get_wtime( );
double megaMultsPerSecond = (double)ARRAYSIZE / ( time1 - time0 ) / 1000000.;

Each Core Has Its Own SIMD Unit!
Thus, You Can Combine SIMD and Multicore

Notes:

- Remember that #pragma omp parallel creates a thread team and that all threads execute everything in the curly braces.

- The variable thisThread is the thread number of the thread who is executing this code right now. There will eventually be NUMT threads who get to execute this code. Thus, all the instances of thisThread will be between 0 and NUMT-1.

- The variable first is the first array element number that thisThread will execute.

- Starting the SIMD multiplications at &A[first], &B[first], &C[first] gives each thread its very own set of contiguous array elements to work on. The SimdMul function depends on this.
Combining SIMD with Multicore

- 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 last two slides

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

\[
\text{for}( \text{i} = 0; \text{i} < \text{NUM}; \text{i} += \text{ONETIME} ) \\
\{ \\
\quad \text{__builtin_prefetch (} \&\text{A}[\text{i}+\text{PD}], \text{WILL_READ_ONLY, LOCALITY_LOW);} \\
\quad \text{__builtin_prefetch (} \&\text{B}[\text{i}+\text{PD}], \text{WILL_READ_ONLY, LOCALITY_LOW);} \\
\quad \text{__builtin_prefetch (} \&\text{C}[\text{i}+\text{PD}], \text{WILL_READ_AND_WRITE, LOCALITY_LOW);} \\
\quad \text{SimdMul(A, B, C, ONETIME);} \\
\}
\]

The Effects of Prefetching on SIMD Computations

The graph shows the speed (in MFLOPS) of array multiplication with different array sizes (M). The x-axis represents the array size (M), and the y-axis represents the speed (MFLOPS). The graph compares different scenarios: SIMD, Prefetch, SIMD, No Prefetch, Non-SIMD, Prefetch, and Non-SIMD, No Prefetch.
• SIMD is an important way to achieve array-operation speed-ups on a CPU
• For now, you might have to write in assembly language to get to all of it
• I suspect that `#pragma omp simd` will catch up
• I suspect that *Intel Intrinsics* will catch up
• Prefetching can really help SIMD