Welcome!
Today’s Agenda:

- Recap
- Flow Control
- AVX, Larrabee, GPGPU
- Further Reading
SSE: Four Floats

```c
union {
    __m128 a4;
    float a[4];
};

a4 = _mm_sub_ps( val1, val2 );
__m128 b4 = _mm_sqrt_ps( a4 );
__m128 m4 = _mm_max_ps( a4, b4 );
```
Recap

SSE: Four Floats

- _mm_add_ps
- _mm_sub_ps
- _mm_mul_ps
- _mm_div_ps
- _mm_sqrt_ps
- _mm_rcp_ps
- _mm_rsqrt_ps
- _mm_add_epi32
- _mm_sub_epi32
- _mm_mul_epi32
- _mm_div_epi32
- _mm_sqrt_epi32
- _mm_rcp_epi32
- _mm_rsqrt_epi32
- _mm_cvtps_epi32
- _mm_cvtepi32_ps
- _mm_slli_epi32
- _mm_srai_epi32
- _mm_cmpeq_epi32

- _mm_add_epi16
- _mm_sub_epi16
- _mm_add_eppu8
- _mm_sub_epi8
- _mm_add_epi64
- _mm_sub_epi64
Recap

SIMD, Intel way: SSE2 / SSE4.x / AVX

- Separated streams
- Many different data types
- High performance

Remains one problem:
Stream programming is rather different from regular programming.
Recap

SSE: Four Floats

AOS

structure of arrays

SOA
Recap

SSE: Four Floats

```c
struct Particle {
    float x, y, z;
    int mass;
};
Particle particle[512];
```

AOS

```c
union {
    float x[512]; __m128 x4[128];
} x4union;
union {
    float y[512]; __m128 y4[128];
} y4union;
union {
    float z[512]; __m128 z4[128];
} z4union;
union {
    int mass[512]; __m128i mass4[128];
} mass4union;
```

SOA

structure of arrays
Recap

Vectorization:

“The Art of rewriting your algorithm so that it operates in four separate streams, rather than one.”

Note: compilers will apply SSE2/3/4 for you as well:

```c
vector3f A = { 0, 1, 2 };
vector3f B = { 5, 5, 5 };  
A += B;
```

This will marginally speed up *one line* of your code; manual vectorization is much more fundamental.
Recap

Streams – Data Organization

vector3f D = vector3f.Normalize( T - P );

float A = T.X - P.X
B = T.Y - P.Y
C = T.Z - P.Z
D = A * A
E = B * B
F = C * C
F += E
F += D
G = sqrt( F )
D.X = A / G
D.Y = B / G
D.Z = C / G
Recap

Streams – Data Organization

vector3f D = vector3f.Normalize(T - P);

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D1 = A1 * A1</td>
<td>D2 = A2 * A2</td>
<td>D3 = A3 * A3</td>
<td>D4 = A4 * A4</td>
</tr>
<tr>
<td>E1 = B1 * B1</td>
<td>E2 = B2 * B2</td>
<td>E3 = B3 * B3</td>
<td>E4 = B4 * B4</td>
</tr>
<tr>
<td>F1 = C1 * C1</td>
<td>F2 = C2 * C2</td>
<td>F3 = C3 * C3</td>
<td>F4 = C4 * C4</td>
</tr>
<tr>
<td>F1 += E1</td>
<td>F2 += E2</td>
<td>F3 += E3</td>
<td>F4 += E4</td>
</tr>
<tr>
<td>F1 += D1</td>
<td>F2 += D2</td>
<td>F3 += D3</td>
<td>F4 += D4</td>
</tr>
</tbody>
</table>

G1 = sqrt(F1)  G2 = sqrt(F2)  G3 = sqrt(F3)  G4 = sqrt(F4)
Recap

Streams – Data Organization

vector3f D = vector3f.Normalize( T - P );

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
</tr>
<tr>
<td>B1</td>
<td>B2</td>
<td>B3</td>
<td>B4</td>
</tr>
<tr>
<td>C1</td>
<td>C2</td>
<td>C3</td>
<td>C4</td>
</tr>
<tr>
<td>D1</td>
<td>D2</td>
<td>D3</td>
<td>D4</td>
</tr>
<tr>
<td>D1</td>
<td>A1 * A1</td>
<td>A2 * A2</td>
<td>A3 * A3</td>
</tr>
<tr>
<td>E1</td>
<td>E2</td>
<td>E3</td>
<td>E4</td>
</tr>
<tr>
<td>E1</td>
<td>B1 * B1</td>
<td>B2 * B2</td>
<td>B3 * B3</td>
</tr>
<tr>
<td>F1</td>
<td>F2</td>
<td>F3</td>
<td>F4</td>
</tr>
<tr>
<td>F1</td>
<td>C1 * C1</td>
<td>C2 * C2</td>
<td>C3 * C3</td>
</tr>
<tr>
<td>G1</td>
<td>G2</td>
<td>G3</td>
<td>G4</td>
</tr>
<tr>
<td>G1</td>
<td>sqrt( F1 )</td>
<td>sqrt( F2 )</td>
<td>sqrt( F3 )</td>
</tr>
</tbody>
</table>

Input:

TX = { T1.x, T2.x, T3.x, T4.x };  PX = { P1.x, P2.x, P3.x, P4.x };  TY = { T1.y, T2.y, T3.y, T4.y };  PY = { P1.y, P2.y, P3.y, P4.y };  TZ = { T1.z, T2.z, T3.z, T4.z };  PZ = { P1.z, P2.z, P3.z, P4.z };
Recap

```cpp
void Game::BuildBackdrop()
{
    Pixel* dst = m_Surface->GetBuffer();
    float fy = 0;
    for ( unsigned int y = 0; y < SCRHEIGHT; y++, fy++ )
    {
        float fx = 0;
        for ( unsigned int x = 0; x < SCRWIDTH; x++, fx++ )
        {
            float g = 0;
            for ( unsigned int i = 0; i < HOLES; i++ )
            {
                float dx = m_Hole[i]->x - fx, dy = m_Hole[i]->y - fy;
                float squaredist = ( dx * dx + dy * dy );
                g += (250.0f * m_Hole[i]->g) / squaredist;
            }
            if ( g > 1 ) g = 0;
            *dst++ = (int)(g * 255.0f);
        }
        dst += m_Surface->GetPitch() - m_Surface->GetWidth();
    }
}
```
Recap

```cpp
void Game::BuildBackdrop()
{
    Pixel* dst = m_Surface->GetBuffer();
    float fy = 0;
    for (unsigned int y = 0; y < SCRHEIGHT; y++, fy++)
    {
        float fx = 0;
        for (unsigned int x = 0; x < SCRWIDTH; x++, fx++)
        {
            float g = 0;
            for (unsigned int i = 0; i < HOLES / 4; i++)
            {
                float dx = m_Hole[i]->x - fx, dy = m_Hole[i]->y - fy;
                float squareddist = (dx * dx + dy * dy);
                g += (250.0f * m_Hole[i]->g) / squareddist;
            }
            if (g > 1) g = 0;
            *dst++ = (int)(g * 255.0f);
        }
        dst += m_Surface->GetPitch() - m_Surface->GetWidth();
    }
}
```
Recap

```cpp
void Game::BuildBackdrop()
{
    Pixel* dst = m_Surface->GetBuffer();
    float fy = 0;
    for ( unsigned int y = 0; y < SCRHEIGHT; y++, fy++ )
    {
        float fx = 0;
        for ( unsigned int x = 0; x < SCRWIDTH; x++, fx++ )
        {
            float g = 0;  // _m128 g4 = _mm_setzero_ps();
            for ( unsigned int i = 0; i < HOLES / 4; i++ )
            {
                __m128 dx4 = _mm_sub_ps( bhx4[i], fx4 );
                __m128 dy4 = _mm_sub_ps( bhy4[i], fy4 );
                __m128 sq4 = _mm_add_ps( _mm_mul_ps( dx4, dx4 ), _mm_mul_ps( dy4, dy4 ) );
                __m128 mulresult4 = _mm_mul_ps( _mm_set1_ps( 250.0f ), bhg4[i] );
                g4 = _mm_add_ps( g4, _mm_div_ps( mulresult4, sq4 ) );
            }
            if (g > 1) g = 0;
            *dst++ = (int)(g * 255.0f);
        }
    }
    dst += m_Surface->GetPitch() - m_Surface->GetWidth();
}
```
Recap

```cpp
void Game::BuildBackdrop()
{
    Pixel* dst = m_Surface->GetBuffer();
    float fy = 0;
    for ( unsigned int y = 0; y < SCRHEIGHT; y++, fy++ )
    {
        float fx = 0;
        for ( unsigned int x = 0; x < SCRWIDTH; x++, fx++ )
        {
            float g = 0; __m128 g4 = _mm_setzero_ps();
            for ( unsigned int i = 0; i < HOLESD / 4; i++ )
            {
                __m128 dx4 = _mm_sub_ps( bhx4[i], fx4 );
                __m128 dy4 = _mm_sub_ps( bhy4[i], fy4 );
                __m128 sq4 = _mm_add_ps( _mm_mul_ps( dx4, dx4 ), _mm_mul_ps( dy4, dy4 ) );
                __m128 mulresult4 = _mm_mul_ps( _mm_set1_ps( 250.0f ), bhg4[i] );
                g4 = _mm_add_ps( g4, _mm_div_ps( mulresult4, sq4 ) );
            }
            g += g_0 + g_1 + g_2 + g_3;
            if (g > 1) g = 0;
            *dst++ = (int)(g * 255.0f);
        }
    }
}
```

Recap

for ( uint i = 0; i < PARTICLES; i++ ) if ( m_Particle[i]->alive ) {
  m_Particle[i]->_x += m_Particle[i]->_vx;
  m_Particle[i]->_y += m_Particle[i]->_vy;
  if ( ((m_Particle[i]->_x < (2 * SCRWIDTH)) && (m_Particle[i]->_x > -SCRWIDTH) &&
       (m_Particle[i]->_y < (2 * SCRHEIGHT)) && (m_Particle[i]->_y > -SCRHEIGHT)) )
    SpawnParticle( i );
  continue;
}

for ( uint h = 0; h < HOLES; h++ )
{
  float dx = m_Hole[h]->_x - m_Particle[i]->_x;
  float dy = m_Hole[h]->_y - m_Particle[i]->_y;
  float sd = dx * dx + dy * dy;
  float dist = 1.0f / sqrtf( sd );
  dx *= dist, dy *= dist;
  float g = (250.0f * m_Hole[h]->_g * m_Particle[i]->_m) / sd;
  if ( g >= 1 ){
    SpawnParticle( i );
    break; }
  m_Particle[i]->_vx += 0.5f * g * dx;
  m_Particle[i]->_vy += 0.5f * g * dy;
}

int x = (int)m_Particle[i]->_x, y = (int)m_Particle[i]->_y;
if ( ((x >= 0) && (x < SCRWIDTH) && (y >= 0) && (y < SCRHEIGHT))
  m_Surface->GetBuffer()[x + y * m_Surface->GetPitch()] = m_Particle[i]->_c;
Today's Agenda:

- Recap
- Flow Control
- AVX, Larrabee, GPGPU
- Further Reading
Flow Control

Broken Streams

```cpp
bool respawn = false;

for ( uint h = 0; h < HOLES; h++ )
{
    float dx = m_Hole[h]->x - m_Particle[i]->x;
    float dy = m_Hole[h]->y - m_Particle[i]->y;
    float sd = dx * dx + dy * dy;
    float dist = 1.0f / sqrtf( sd );
    dx *= dist, dy *= dist;

    float g = (250.0f * m_Hole[h]->g * m_Particle[i]->m) / sd;
    if (g >= 1) {
        SpawnParticle( i );
        break;
    } respawn = true;

    m_Particle[i]->vx += 0.5f * g * dx;
    m_Particle[i]->vy += 0.5f * g * dy;
}

if (respawn) SpawnParticle( i );
```

FALSE == 0, TRUE == 1:

Masking allows us to run code unconditionally, without consequences.
Flow Control

Broken Streams

```c
__mm_cmpeq_ps ==
__mm_cmplt_ps <
__mm_cmpgt_ps >
__mm_cmple_ps <=
__mm_cmpge_ps >=
__mm_cmpne_ps !=
```
Flow Control

Broken Streams – Flow Divergence

Like other instructions, comparisons between vectors yield a vector of booleans.

```c
__m128 mask = _mm_cmpeq_ps( v1, v2 );
```

The mask contains a bitfield: 32 x ‘1’ for each TRUE, 32 x ‘0’ for each FALSE.

The mask can be converted to a 4-bit integer using _mm_movemask_ps:

```c
int result = _mm_movemask_ps( mask );
```

Now we can use regular conditionals:

```c
if (result == 0) { /* false for all streams */ }
if (result == 15) { /* true for all streams */ }
if (result < 15) { /* not true for all streams */ }
if (result > 0) { /* not false for all streams */ }
```
Flow Control

Streams – Masking

More powerful than ‘any’, ‘all’ or ‘none’ via movemask is *masking*.

```c
if (x >= 1 && x < PI) x = 0;
```

Translated to SSE:

```c
__m128 mask1 = _mm_cmpge_ps( x4, ONE4 );
__m128 mask2 = _mm_cmplt_ps( x4, PI4 );
__m128 fullmask = _mm_and_ps( mask1, mask2 );
__m128 x4 = _mm_andnot_ps( fullmask, x4 );
```
Flow Control

Streams – Masking

```c
float a[4] = { 1, -5, 3.14f, 0 };  
if (a[0] < 0) a[0] = 999;  
if (a[1] < 0) a[1] = 999;  
```

in SSE:

```c
__m128 a4 = _mm_set_ps( 1, -5, 3.14f, 0 );  
__m128 nine4 = _mm_set_ps1( 999 );  
__m128 zero4 = _mm_setzero_ps();  
__m128 mask = _mm_cmplt_ps( a4, zero4 );
```
Flow Control

Streams – Masking

```c
__m128 a4 = _mm_set_ps( 1, -5, 3.14f, 0 );
__m128 nine4 = _mm_set_ps1( 999 );
__m128 zero4 = _mm_setzero_ps();
__m128 mask = _mm_cmplt_ps( a4, zero4 );

__m128 part1 = _mm_and_ps( mask, nine4 );
// yields: { 0, 999, 0, 0 }

__m128 part2 = _mm_andnot_ps( mask, a4 );
// yields: { 1, 0, 3.14, 0 }

a4 = _mm_or_ps( part1, part2 );
// yields: { 1, 999, 3.14, 0 }

or simply:  a4 = _mm_blendv_ps( a4, nine4, mask );
```
Streams – Masking

Take-away:

- In vectorized code, stream divergence is not possible.
- We solve this by keeping all lanes alive.
- ‘Inactive lanes’ use masking to nullify actions.

This approach is used in SSE/AVX, as well as on GPUs.
Flow Control

Streams – Masking
Flow Control

```c
static union { float px[PARTICLES]; __m128 px4[PARTICLES / 4]; };
static union { float py[PARTICLES]; __m128 py4[PARTICLES / 4]; };
static union { float pvx[PARTICLES]; __m128 pvx4[PARTICLES / 4]; };
static union { float pvy[PARTICLES]; __m128 pvy4[PARTICLES / 4]; };
static union { float pm[PARTICLES]; __m128 pm4[PARTICLES / 4]; };
static bool pa[PARTICLES];
static union { uint pc[PARTICLES]; __m128i pc4[PARTICLES / 4]; };

... // convert to SoA
for( int i = 0; i < PARTICLES; i++ )
{
    px[i] = m_Particle[i]->x;
    py[i] = m_Particle[i]->y;
    pvx[i] = m_Particle[i]->vx;
    pvy[i] = m_Particle[i]->vy;
    pa[i] = m_Particle[i]->alive;
    pc[i] = m_Particle[i]->c;
    pm[i] = m_Particle[i]->m;
}
```
Today’s Agenda:

- Recap
- Flow Control
- AVX, Larrabee, GPGPU
- Further Reading
Beyond SSE

AVX*:


... etc.
Beyond SSE

AVX2*

Extension to AVX: adds broader __mm256i support, and FMA:

\[
\text{r8 = c8 + (a8*b8)}
\]

\[
\text{__m256 r8 = _mm256_fmadd_ps( a8, b8, c8 );}
\]

Emulate on AVX:

\[
\text{r8 = _mm256_add_ps( _mm256_mul_ps( a8, b8 ), c8 );}
\]

Benefits of \textit{fused multiply and add}:

- Even more work done for a single ‘fetch-decode’
- Better precision: rounding doesn’t happen between multiply and add

For a full list of instructions, see:


Beyond SSE

For a full list of instructions, see:  
Beyond SSE

AVX512*

__m512: 32 512-bit registers, as well as 7 opmask registers (__mmask16).

Example: __m512 r = __mm512_mask_add_ps( src, mask, a, b );

(uses src when mask bit is not set)

GPU Model

```c
__kernel void main( write_only image2d_t outimg )
{
    int column = get_global_id( 0 );
    int line = get_global_id( 1 );
    float red = column / 800.;
    float green = line / 480.;
    float4 color = { red, green, 0, 1 };
    write_imagef( outimg, (int2)(column, line), color );
}
```
GPU Model

```c
__kernel void main( write_only image2d_t outimg )
{
    int column = get_global_id( 0 );
    int line = get_global_id( 1 );
    float red, green, blue;
    if (column & 1)
    {
        red = column / 800.;
        green = line / 480.;
        color = { red, green, 0, 1 };
    }
    else
    {
        red = green = blue = 0;
    }
    write_imagef( outimg, (int2)(column, line), color );
}
```
Today's Agenda:

- Recap
- Flow Control
- AVX, Larrabee, GPGPU
- Further Reading
Practical SIMD Programming
Supplemental tutorial for INFOSEC, INFOMOV & INFOMAGR
Jaco Blikker, 2017

Introduction

Modern CPUs increasingly rely on parallelism to achieve peak performance. The most well-known form is task parallelism, which is supported at the hardware level by multiple cores. Hyper-threading and dedicated instructions supporting multithreading operating systems. Less known is the parallelism known as instruction level parallelism; the capability of a CPU to execute multiple instructions simultaneously, i.e., in the same cycles, in a single cycle. Older CPUs such as the original Pentium used this to execute instructions utilizing two pipelines, concurrently with high-latency floating point operations. Typically, this happens transparent to the programmer. Recent CPUs use a radically different form of instruction level parallelism. These CPUs deploy a versatile set of vector operations; instructions that operate on 4 or 8 inputs, yielding 4 or 8 results, often in a single cycle. This is known as SIMD: Single Instruction, Multiple Data. To leverage this compute potential, we can no longer rely on the compiler. Algorithms that exhibit extensive data parallelism benefit most from exploiting SIMD programming, with potential performance gains of 4x−6x and more. This document provides a practical introduction to SIMD programming in C++ and C#.

SIMD Concepts

A CPU uses registers to store data to operate on. A typical register stores 32 or 64 bits, and holds a single scalar value. CPU instructions typically operate on two operands. Consider the following code snippet:

```c
float length = sqrtf( velocity );
```

The line that calculates the length of the vector requires a significant number of vector

MONDAY OCTOBER 2: EXTENDED LAB
Today's Agenda:

- Recap
- Flow Control
- AVX, Larrabee, GPGPU
- Further Reading
END of “SIMD (2)”

next lecture: “Data Oriented”