Welcome!
Today’s Agenda:

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

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_add_epi32  _mm_add_epi16
_mm_sub_ps  _mm_sub_epi32  _mm_sub_epi16
_mm_mul_ps  _mm_mul_epi32  _mm_mul_epi16
_mm_div_ps  _mm_div_epi32
_mm_sqrt_ps _mm_sqrt_epi32
_mm_rcp_ps  _mm_rcp_epi32
_mm_sqrt_epi32
_mm_rsqrt_ps
_mm_add_epi64
_mm_sub_epi64
_mm_add_epi16
_mm_sub_epi16
_mm_add_epi32
_mm_rsqrt_epi32
_mm_div_epi32
_mm_sqrt_epi32
_mm_rsqrt_epi32
_mm_add_epi32
_mm_add_epi16
_mm_add_epi32
_mm_rsqrt_epi32
_mm_sqrt_epi32
_mm_add_epi64
_mm_sub_epi64
_mm_add_epi32
_mm_add_epi16
_mm_add_epi32
```

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

SOA

structure of arrays
Recap

SSE: Four Floats

```c
struct Particle {
    float x, y, z;
    int mass;
};
```

Particle particle[512];

AOS

union {
    float x[512]; __m128 x4[128];
};
union {
    float y[512]; __m128 y4[128];
};
union {
    float z[512]; __m128 z4[128];
};
union {
    int mass[512]; __m128i mass4[128];
};

SOA

structure of arrays

```c
struct Particle {
    float x, y, z;
    int mass;
};
```
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:

```cpp
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

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

```cpp
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 )

0 1 2 3
Recap

Streams – Data Organization

def 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 )

Input:

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

\[
D = \frac{A_{1}}{G_{1}} = \frac{A_{2}}{G_{2}} = \frac{A_{3}}{G_{3}} = \frac{A_{4}}{G_{4}}
\]
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 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; __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 < 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 ) );
            }
            g += g[0] + g[1] + g[2] + g[3];
            if (g > 1) g = 0;
            *dst++ = (int)(g * 255.0f);
        }
    }
    dst += m_Surface->GetPitch() - m_Surface->GetWidth();
}
```
Today’s Agenda:

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

Broken Streams

```c
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 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 );
```

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

Broken Streams

```c
_mm_cmpeq_ps ==
_mm_cmplt_ps <
_mm_cmpgt_ps >
_mm_cmplt_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 );
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:
__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);
```

```
000000000000000000000000000000001111111111111111111111111111111100000000000000000000000000000000000000000000000000000000000000
```

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:

```c
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

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.
AVX2*

Extension to AVX: adds broader _mm256i support, and FMA:

\[ r8 = c8 + (a8 \times b8) \]

\[ \_m256 \ r8 = \_mm256\_fmadd\_ps( \ a8, \ b8, \ c8 ) ; \]

Emulate on AVX:

\[ r8 = \_mm256\_add\_ps( \_mm256\_mul\_ps( \ a8, \ b8 ), \ c8 ) ; \]

Benefits of *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: [https://software.intel.com/sites/landingpage/IntrinsicsGuide](https://software.intel.com/sites/landingpage/IntrinsicsGuide)

Beyond SSE

For a full list of instructions, see: https://software.intel.com/sites/landingpage/IntrinsicsGuide
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)

Beyond SSE

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 );
}
```
Beyond SSE

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
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. Hyperthreading 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 cycle(s), in a single thread. 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 explicit SIMD programming, with potential performance gains of 4x - 8x 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:

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

The line that calculates the length of the vector requires a significant number of vector instructions.
END of “SIMD (2)"

next lecture: “Data-Oriented Design”