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\_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\_add\_epi16
- \_mm\_sub\_epi16
- \_mm\_add\_epu8
- \_mm\_sub\_epu8
- \_mm\_mul\_epi32
- \_mm\_mul\_epu32
- \_mm\_add\_epi16
- \_mm\_add\_epi64
- \_mm\_add\_epu8
- \_mm\_add\_epi64
- \_mm\_cvtps\_epi32
- \_mm\_cvtepi32\_ps
- \_mm\_cmpeq\_epi32
- \_mm\_slli\_epi32
- \_mm\_srai\_epi32
- \_mm\_cmpeq\_epi32
- \_mm\_cmpeq\_epi32
- \_mm\_cmpeq\_epi32
Recap

SSE: Four Floats

AOS

SOA

structure of arrays
SSE: Four Floats

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

### Recap

#### 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

```
float x[512];
float y[512];
float z[512];
int mass[512];
```
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();
    }
}
```
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

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
Flow

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

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

Broken Streams

```c
char a[4] = { 6, 7, 8, 9 };  
char c[4];  
*(uint*)c = *(uint*)a + *(uint*)b;
```

Masked addition:

```c
char a[4] = { 6, 7, 8, 9 };  
char mask[4] = { 255, 0, 255, 255 };  
char c[4];  
*(uint*)c = *(uint*)a + *(uint*)mask & *(uint*)b;
```

char a[4] = { 6, 7, 8, 9 };  
uint mask4 = 0xFFFF00FF;  
char c[4];  
*(uint*)c = *(uint*)a + *(uint*)b & mask4;
```
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_cmple_ps( x4, ONE4 );
__m128 mask2 = _mm_cmpgte_ps( x4, PI4 );
__m128 fullmask = _mm_and_ps( mask1, mask2 );

x4 = _mm_andnot_ps( fullmask, x4 );
```

(_mm_andnot_ps inverts the first argument.)
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 );
```
Streams – Masking

__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]; };  
```

```c
// 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*:

- _m256
- _mm256_add_ps
- _mm256_sqrt_ps

...etc.

Beyond SSE

**AVX2***

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

\[ r_8 = c_8 + (a_8 \times b_8) \]

```c
__m256 r8 = _mm256_fmadd_ps( a8, b8, c8 );
```

Emulate on AVX:

```c
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

Beyond SSE

AVX512*

16-wide SIMD, with 32 512-bit registers (__m512, __m512i).

Most AVX512 instructions can be masked:

```c
__m512 _mm512_maskz_add_ps( __mmask16 k, __m512 a, __m512 b )
```

"Add packed single-precision (32-bit) floating-point elements in a and b, and store the results in dst using zeromask k (elements are zeroed out when the corresponding mask bit is not set)."

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
Beyond SSE

GPU Model

__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

__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 / 800.0;
        green = line / 480.0;
        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
TL;DR

This OptimizedSummary is an adaptation of an existing tutorial for C++ and C# programmers, originally written for the Advanced graphics course of the Utrecht University.

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 multitasking 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, which it can then quickly load into memory. Consider a single integer or float.

<table>
<thead>
<tr>
<th>Integer</th>
<th>Floating-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code></td>
<td><code>float</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Single-precision</th>
<th>Double-precision</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>float32</code></td>
<td><code>float64</code></td>
</tr>
</tbody>
</table>

These data types can be used to store 4 single-precision or 2 double-precision floating-point numbers. The `SIMD::float4` type represents a 4-component vector, while the `SIMD::float2` type holds a single integer or float.
END of “SIMD (2)”

next lecture: “Data-Oriented Design”