Welcome!
Today's Agenda:

- Recap
- Flow Control
- AVX, Larrabee, GPGPU
- Assignment 2
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_mul_epi16
- _mm_div_epi16
- _mm_sqrt_epi16
- _mm_rcp_epi16
- _mm_rsqrt_epi16
- _mm_add_eppu8
- _mm_sub_eppu8
- _mm_add_eppu32
- _mm_div_eppu32
- _mm_sqrt_eppu32
- _mm_rcp_eppu32
- _mm_rsqrt_eppu32
- _mm_add_epi64
- _mm_sub_epi64
- _mm_mul_epi64
- _mm_div_epi64
- _mm_sqrt_epi64
- _mm_rcp_epi64
- _mm_rsqrt_epi64
- _mm_add_eppu64
- _mm_sub_eppu64
- _mm_add_eppu32
- _mm_div_eppu32
- _mm_sqrt_eppu32
- _mm_rcp_eppu32
- _mm_rsqrt_eppu32
- _mm_add_eppu64
- _mm_sub_eppu64
- _mm_add_eppu32
- _mm_div_eppu32
- _mm_sqrt_eppu32
- _mm_rcp_eppu32
- _mm_rsqrt_eppu32
- _mm_add_eppu64
- _mm_sub_eppu64
- _mm_add_eppu32
- _mm_div_eppu32
- _mm_sqrt_eppu32
- _mm_rcp_eppu32
- _mm_rsqrt_eppu32
- _mm_add_eppu64
- _mm_sub_eppu64
- _mm_add_eppu32
- _mm_div_eppu32
- _mm_sqrt_eppu32
- _mm_rcp_eppu32
- _mm_rsqrt_eppu32
- _mm_add_eppu64
- _mm_sub_eppu64
- _mm_add_eppu32
- _mm_div_eppu32
- _mm_sqrt_eppu32
- _mm_rcp_eppu32
- _mm_rsqrt_eppu32
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
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]; }
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**
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

Consider the following scalar code:

```
Vector3 D = Vector3.Normalize( T - P );
```

This is quite high-level. What the processor needs to do is:

```
Vector3 tmp = T - P;
float length = sqrt( tmp.x * tmp.x + tmp.y * tmp.y + tmp.z * tmp.z );
D = tmp / length;
```
Recap

Streams

Consider the following scalar code:

Vector3 D = Vector3.Normalize( T - P );

This is quite high-level. What the processor needs to do is:

```cpp
float tmp_x = T.x - P.x;
float tmp_y = T.y - P.y;
float tmp_z = T.z - P.z;
float sqlen = tmp_x * tmp_x + tmp_y * tmp_y + tmp_z * tmp_z;
float length = sqrt( sqlen );
D.x = tmp_x / length;
D.y = tmp_y / length;
D.z = tmp_z / length;
```
Recap

Streams

Consider the following scalar code:

Vector3 D = Vector3.Normalize( T - P );

Using vector instructions:

```c
__m128 A = T - P; // 75%
float B = dot( A, A ); // 75%
__m128 C = { B, B, B }; // 75%, overhead
__m128 D = A / C; // 75%
```
Recap

Streams

Consider the following scalar code:

```
Vector3 D = Vector3.Normalize( T - P );
```

```
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
```

```
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
```

```
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

Optimal utilization of SIMD hardware is achieved when we run the same algorithm four times in parallel. This way, the approach also scales naturally to 8-wide, 16-wide and 32-wide SIMD.
Recap

Streams – Data Organization

Vector3 D = Vector3.Normalize( T - P );

![Table showing calculations for D1, A1, B1, C1, D1, E1, F1, G1, A2, B2, C2, D2, E2, F2, G2, A3, B3, C3, D3, E3, F3, G3, A4, B4, C4, D4, E4, F4, G4)]
Recap

Streams – Data Organization

Vector3 D = Vector3.Normalize( T - P );

A1 = TX1 - PX1
B1 = TY1 - PY1
C1 = TZ1 - PZ1
D1 = A1 * A1
E1 = B1 * B1
F1 = C1 * C1
F1 += E1

G1 = sqrt( F1 )
DX1 = A1 / G1
DY1 = B1 / G1
DZ1 = C1 / G1

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 };

Vector3 E = Vector3.Normalize( T - P );

A2 = TX2 - PX2
B2 = TY2 - PY2
C2 = TZ2 - PZ2
D2 = A2 * A2
E2 = B2 * B2
F2 = C2 * C2
F2 += E2

G2 = sqrt( F2 )
DX2 = A2 / G2
DY2 = B2 / G2
DZ2 = C2 / G2

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 };

DX3 = A3 / G3
DY3 = B3 / G3
DZ3 = C3 / G3

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 };

DX4 = A4 / G4
DY4 = B4 / G4
DZ4 = C4 / G4
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
- Assignment 2
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; }
    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

```
_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.

```
__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:

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

Now we can use regular conditionals:

```
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 (g >= 1 && g < PI) g = 0;
```

Translated to SSE:

```c
__m128 mask1 = _mm_cmpge_ps( g4, ONE4 );
__m128 mask2 = _mm_cmplt_ps( g4, PI4 );
__m128 fullmask = _mm_and_ps( mask1, mask2 );
```

```c
if (_mm_movemask_ps( fullmask ) == 0) return NONE;
else if (_mm_movemask_ps( fullmask ) == 15) return ALL;
else return SOME;
```
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

\[
\text{__m128 } a4 = \text{ _mm_set_ps( 1, -5, 3.14f, 0 );}
\]
\[
\text{__m128 } \text{ nine4 } = \text{ _mm_set_ps1( 999 );}
\]
\[
\text{__m128 } \text{ zero4 } = \text{ _mm_setzero_ps();}
\]
\[
\text{__m128 } \text{ mask } = \text{ _mm_cmplt_ps( a4, zero4 );}
\]

\[
\text{__m128 part1 } = \text{ _mm_and_ps( mask, nine4 );}
\]
\[
\text{__m128 part2 } = \text{ _mm_andnot_ps( mask, a4 );}
\]
\[
\text{a4 } = \text{ _mm_or_ps( part1, part2 );}
\]
Flow Control

Streams – Masking

```c
Streams

int3 brdf = SampleDiffuse( diffuse, N, r1, r2, bb, pdf );

if ( pdf > 0 ) {
  E = brdf * ( dot( N, R ) / pdf );
}

Camera.

E = E + brdf; // E = E + diffuse;
```

Lecture 7 – “SIMD (2)”
Today’s Agenda:

- Recap
- Flow Control
- AVX, Larrabee, GPGPU
- Assignment 2
Beyond SSE

AVX*


**m256**  _mm256_add_ps  _mm256_sqrt_ps  ...etc.
Beyond SSE

AVX2*

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

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

\[
\text{__m256 r8} = \text{__mm256_fmadd_ps( a8, b8, c8 )};
\]

Emulate on AVX:

\[
\text{r8} = \text{__mm256_add_ps( _mm256_mul_ps( a8, b8 ), c8 )};
\]

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:  
Beyond SSE

**AVX512***

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

Example: __m512 r = __m512_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 );
}
```
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
- Assignment 2
Assignment 2

Optimize ‘rts’.

In preparation for the final assignment, you are encouraged to test your skills on the provided small real-time strategy game. Optimize this application using any means you see fit.

Important: apply the structured process.

You may work on this project in a team of max 3 students.

Fine print / details:

- This assignment serves as an exercise project in preparation for the final project, in which you optimize an application of your choice using the structured process mentioned in lecture 1 and 2.
- The assignment provides ample opportunities for high level, low level and SIMD optimizations.
- It is crucial that your optimizations are guided by profiling. Your report should make it abundantly clear that you followed this approach.
- The purpose of the optimization session is to support larger armies at higher frame rates.
- All optimizations are allowed (including using multiple cores and the GPU if you are able to do this). However, successful high level optimizations will yield the biggest performance increase; these will therefore also have the biggest impact on your grade.
- Executing the structured process correctly will yield a better score than extreme technical skills.

Deliverables:

For this assignment, a detailed report is required: analyze initial bottlenecks and scalability issues; propose algorithmic improvements; report on implemented algorithmic improvements; report on low level and SIMD improvements. Please also explain how the work was divided over the team members.
Assignment 2

Assignment 2 Practical Details

The deadline for assignment 2 is Tuesday, October 18, 23:59.
Date for ‘late delivery’ is Wednesday, October 19, 23:59.
Today's Agenda:

- Recap
- Flow Control
- AVX, Larrabee, GPGPU
- Assignment 2
END of “SIMD (2)"

next lecture: “Cache Oblivious”