/INFOMOV/

Optimization & Vectorization

J. Bikker - Sep-Nov 2018 - Lecture 6: “SIMD (2)”

Welcome!

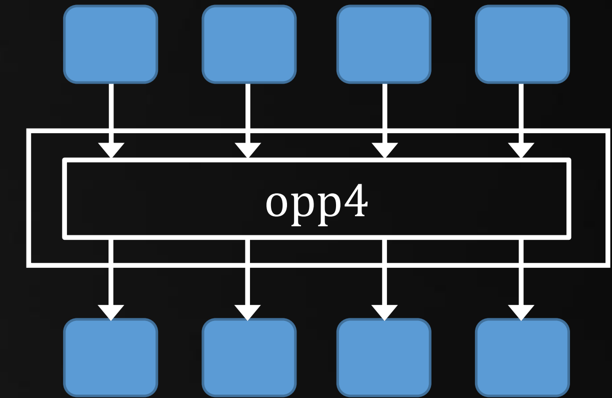
Today's Agenda:

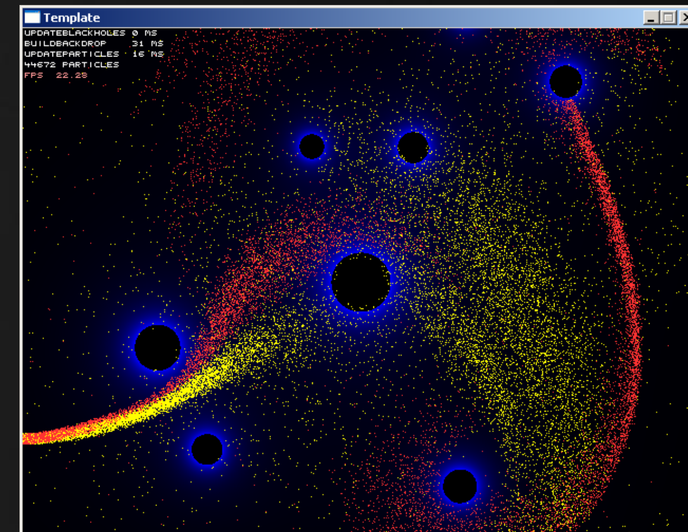
- Recap
- Flow Control
- AVX, Larrabee, GPGPU
- Further Reading

Recap

SSE: Four Floats

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

a4 = _mm_sub_ps( val1, val2 );
float sum = a[0] + a[1] + a[2] + a[3];

__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_epu8
_mm_sub_epu8

_mm_mul_epu32

_mm_add_epi64
_mm_sub_epi64

Recap

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

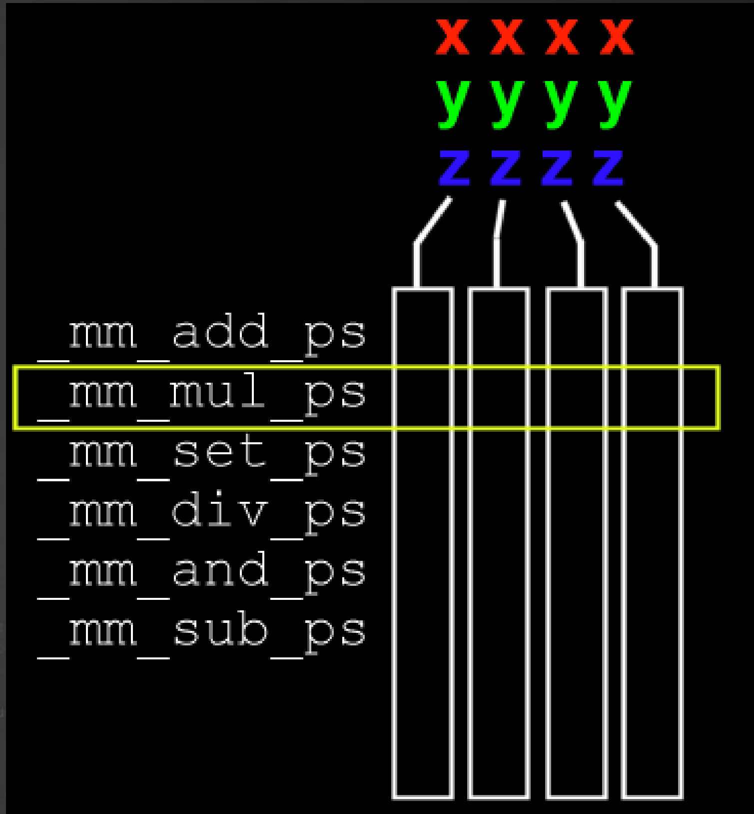
- Separated streams
- Many different data types
- High performance

Remains one problem:

Stream programming is rather different from regular programming.

Recap

SSE: Four Floats



AOS

SOA

structure of arrays

Recap

SSE: Four Floats

```
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

Recap

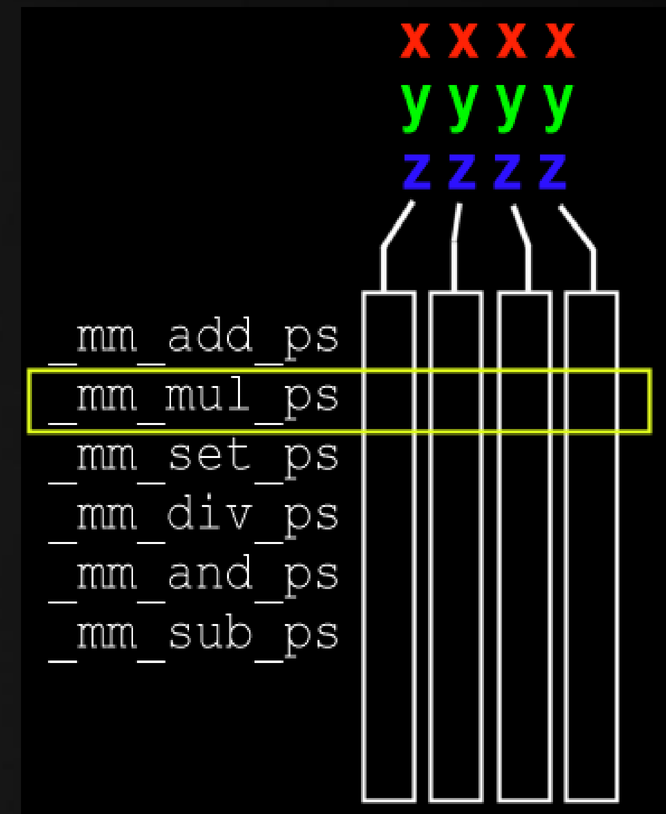
Vectorization:

“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:

```
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

```
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 );
```

float A[1..4]

__m128 A4

```
A1 = T1.X – P1.X      A2 = T2.X – P2.X      A3 = T3.X – P3.X      A4 = T4.X – P4.X
B1 = T1.Y – P1.Y      B2 = T2.Y – P2.Y      B3 = T3.Y – P3.Y      B4 = T4.Y – P4.Y
C1 = T1.Z – P1.Z      C2 = T2.Z – P2.Z      C3 = T3.Z – P3.Z      C4 = T4.Z – P4.Z
D1 = A1 * A1          D2 = A2 * A2          D3 = A3 * A3          D4 = A4 * A4
E1 = B1 * B1          E2 = B2 * B2          E3 = B3 * B3          E4 = B4 * B4
F1 = C1 * C1          F2 = C2 * C2          F3 = C3 * C3          F4 = C4 * C4
F1 += E1              F2 += E2              F3 += E3              F4 += E4
F1 += D1              F2 += D2              F3 += D3              F4 += D4
G1 = sqrt( F1 )       G2 = sqrt( F2 )       G3 = sqrt( F3 )       G4 = sqrt( F4 )
D1.X = A1 / G1        D2.X = A2 / G2        D3.X = A3 / G3        D4.X = A4 / G4
D1.Y = B1 / G1        D2.Y = B2 / G2        D3.Y = B3 / G3        D4.Y = B4 / G4
D1.Z = C1 / G1        D2.Z = C2 / G2        D3.Z = C3 / G3        D4.Z = C4 / G4

        0                     1                     2                     3
```

Recap

Streams – Data Organization

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

float A[1..4]

__m128 A4

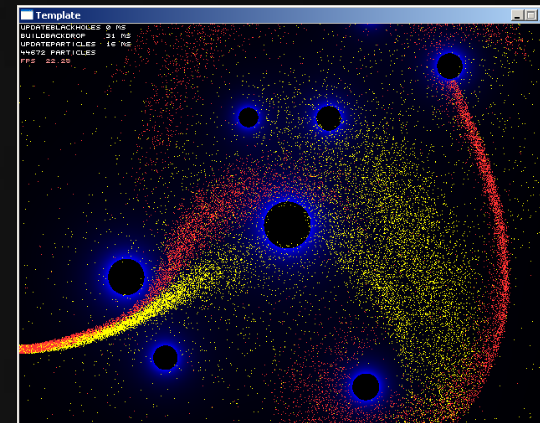
```
A1 = T1.X – P1.X     A2 = T2.X – P2.X     A3 = T3.X – P3.X     A4 = T4.X – P4.X
B1 = T1.Y – P1.Y     B2 = T2.Y – P2.Y     B3 = T3.Y – P3.Y     B4 = T4.Y – P4.Y
C1 = T1.Z – P1.Z     C2 = T2.Z – P2.Z     C3 = T3.Z – P3.Z     C4 = T4.Z – P4.Z
D1 = A1 * A1         D2 = A2 * A2         D3 = A3 * A3         D4 = A4 * A4
E1 = B1 * B1         E2 = B2 * B2         E3 = B3 * B3         E4 = B4 * B4
F1 = C1 * C1         F2 = C2 * C2         F3 = C3 * C3         F4 = C4 * C4
F1 += E1             F2 += E2             F3 += E3             F4 += E4
F1 += D1             F2 += D2             F3 += D3             F4 += D4
G1 = sqrt( F1 )      G2 = sqrt( F2 )      G3 = sqrt( F3 )      G4 = sqrt( F4 )
D1.X = A1 / G1       D2.X = A2 / G2       D3.X = A3 / G3       D4.X = A4 / G4
D1.Y = B1 / G1       D2.Y = B2 / G2       D3.Y = B3 / G3       D4.Y = B4 / G4
D1.Z = C1 / G1       D2.Z = C2 / G2       D3.Z = C3 / G3       D4.Z = C4 / G4
```

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 };
```

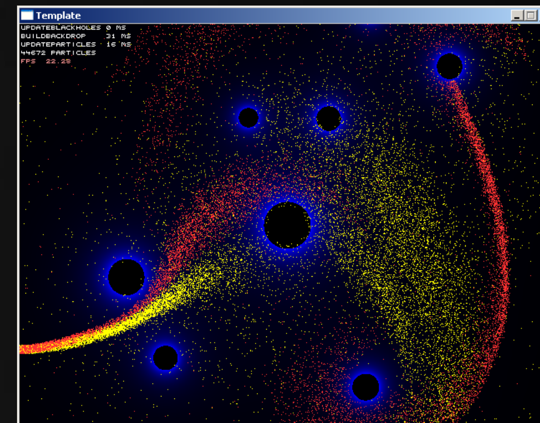
Recap

```
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 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();
    }
}
```



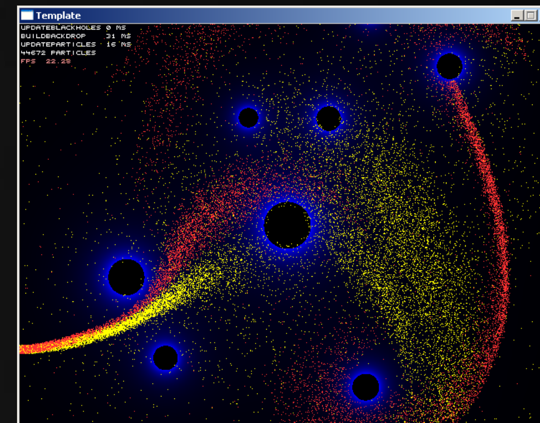
Recap

```
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();
    }
}
```



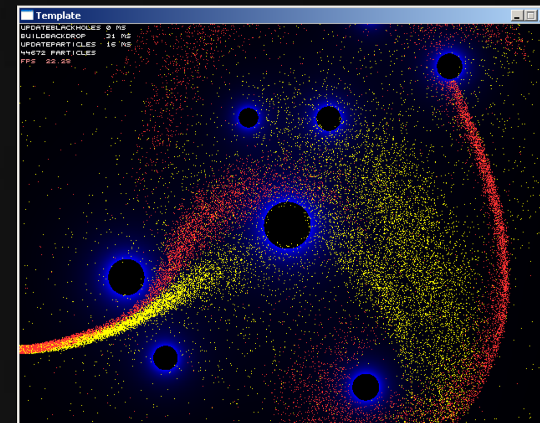
Recap

```
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

```
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

FALSE == 0, TRUE == 1:

Masking allows us to run code unconditionally, without consequences.

Broken Streams

```
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; * !respawn;
    m_Particle[i]->vy += 0.5f * g * dy; * !respawn;
}
if (respawn) SpawnParticle( i );
```

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*.

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

Translated to SSE:

```
__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

```
float a[4] = { 1, -5, 3.14f, 0 };
if (a[0] < 0) a[0] = 999;
if (a[1] < 0) a[1] = 999;
if (a[2] < 0) a[2] = 999;
if (a[3] < 0) a[3] = 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 );
```

00000000000000000000000000000000111111111111111111111111111111110000000000000000000000000000000000000000000000000000000000000000

Flow Control

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 );
```

00000000000000000000000000000000111111111111111111111111111111110000000000000000000000000000000000000000000000000000000000000000

```
__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 );` ☺

Flow Control

Streams – Masking

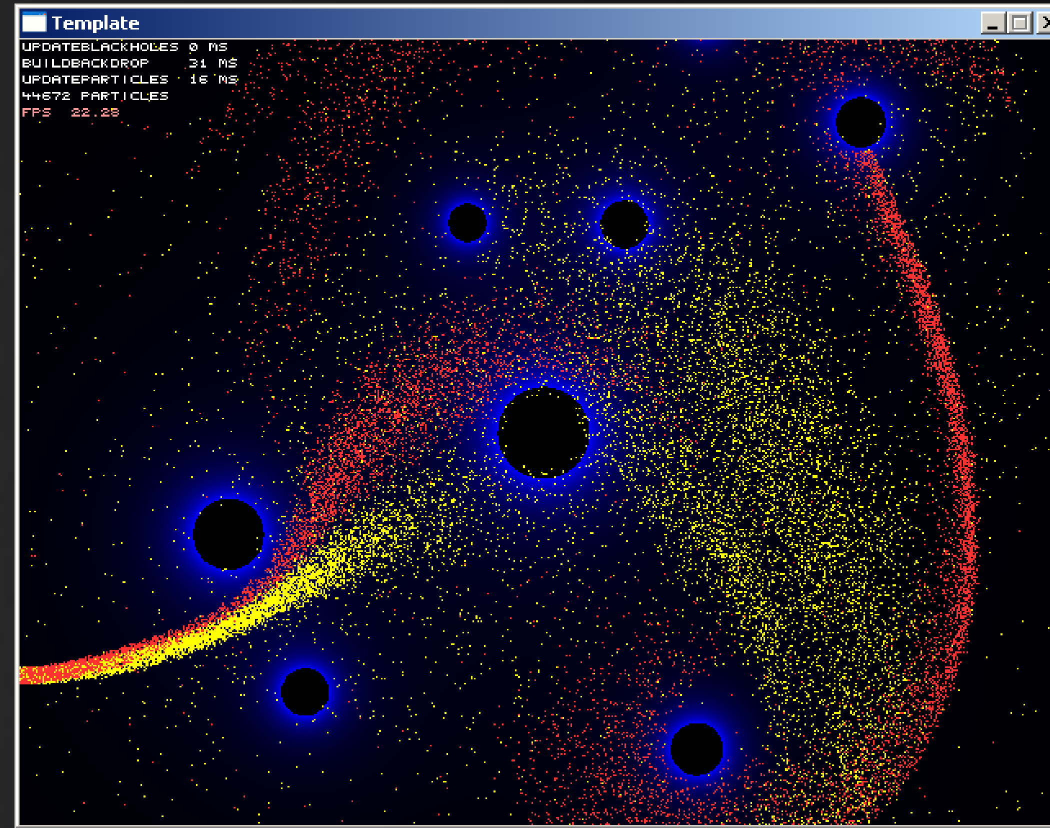
Take-away:

- 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

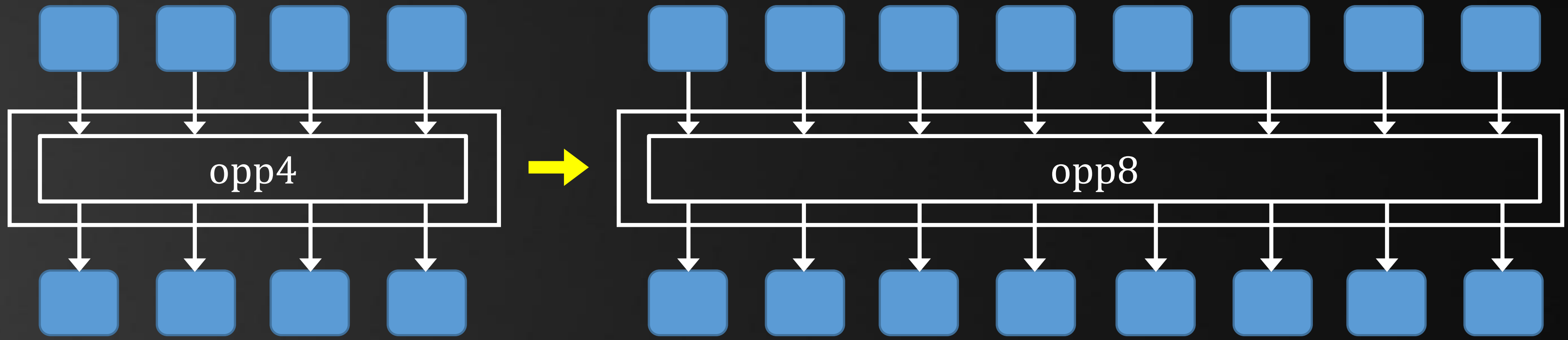
```
static union { float px[PARTICLES]; __m128 px4[PARTICLES / 4]; };
static union { float py[PARTICLES]; __m128 py4[PARTICLES / 4]; };
static union { float pvx[PARTICLES]; __m128 pvx4[PARTICLES / 4]; };
static union { float pvy[PARTICLES]; __m128 pvy4[PARTICLES / 4]; };
static union { float pm[PARTICLES]; __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*




```
__m256        _mm256_add_ps
              _mm256_sqrt_ps
              ...etc.
```

*: On: ‘Sandy Bridge’ (Intel, 2011), ‘Bulldozer’ (AMD, 2011).

Beyond SSE

AVX2*

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

```
r8 = c8+(a8*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

*: On: ‘Haswell’ (Intel, 2013), ‘Carrizo’ and ‘Zen’ (AMD, 2015, 2017).

Beyond SSE

The Intel Intrinsics Guide is an interactive reference tool for Intel intrinsic instructions, which are C style functions that provide acces many Intel instructions - including Intel® SSE, AVX, AVX-512, and more - without the need to write assembly code.

Technologies

- MMX
- SSE (checked)
- SSE2
- SSE3
- SSSE3
- SSE4.1
- SSE4.2
- AVX
- AVX2
- FMA
- AVX-512
- KNC
- SVML
- Other

Categories

- Application-Targeted
- Arithmetic
- Bit Manipulation
- Cast
- Compare
- Convert
- Cryptography
- Elementary Math Functions
- General Support
- Load
- Logical
- Miscellaneous
- OS-Targeted
- Probability/Statistics
- Random
- Set
- Shift
- Special Math Functions
- Store
- String Compare
- Swizzle

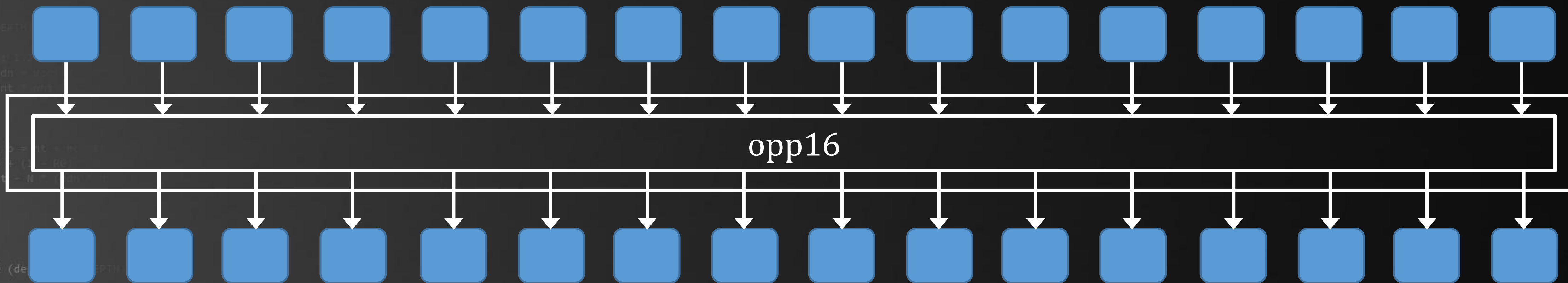
_mm_search

```
__m128 _mm_add_ps (__m128 a, __m128 b)
__m128 _mm_add_ss (__m128 a, __m128 b)
__m128 _mm_and_ps (__m128 a, __m128 b)
__m128 _mm_andnot_ps (__m128 a, __m128 b)
__m64 _mm_avg_pu16 (__m64 a, __m64 b)
__m64 _mm_avg_pu8 (__m64 a, __m64 b)
__m128 _mm_cmpeq_ps (__m128 a, __m128 b)
__m128 _mm_cmpeq_ss (__m128 a, __m128 b)
__m128 _mm_cmpge_ps (__m128 a, __m128 b)
__m128 _mm_cmpge_ss (__m128 a, __m128 b)
__m128 _mm_cmpgt_ps (__m128 a, __m128 b)
__m128 _mm_cmpgt_ss (__m128 a, __m128 b)
__m128 _mm_cmple_ps (__m128 a, __m128 b)
__m128 _mm_cmple_ss (__m128 a, __m128 b)
__m128 _mm_cmplt_ps (__m128 a, __m128 b)
__m128 _mm_cmplt_ss (__m128 a, __m128 b)
__m128 _mm_cmpneq_ps (__m128 a, __m128 b)
__m128 _mm_cmpneq_ss (__m128 a, __m128 b)
__m128 _mm_cmpnge_ps (__m128 a, __m128 b)
__m128 _mm_cmpnge_ss (__m128 a, __m128 b)
__m128 _mm_cmpngt_ps (__m128 a, __m128 b)
__m128 _mm_cmpngt_ss (__m128 a, __m128 b)
__m128 _mm_cmpnle_ps (__m128 a, __m128 b)
__m128 _mm_cmpnle_ss (__m128 a, __m128 b)
__m128 _mm_cmpnlt_ss (__m128 a, __m128 b)
__m128 _mm_cmpord_ps (__m128 a, __m128 b)
__m128 _mm_cmpord_ss (__m128 a, __m128 b)
__m128 _mm_cmpunord_ps (__m128 a, __m128 b)
__m128 _mm_cmpunord_ss (__m128 a, __m128 b)
int _mm_comieq_ss (__m128 a, __m128 b)
int _mm_comige_ss (__m128 a, __m128 b)
```

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

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 );`

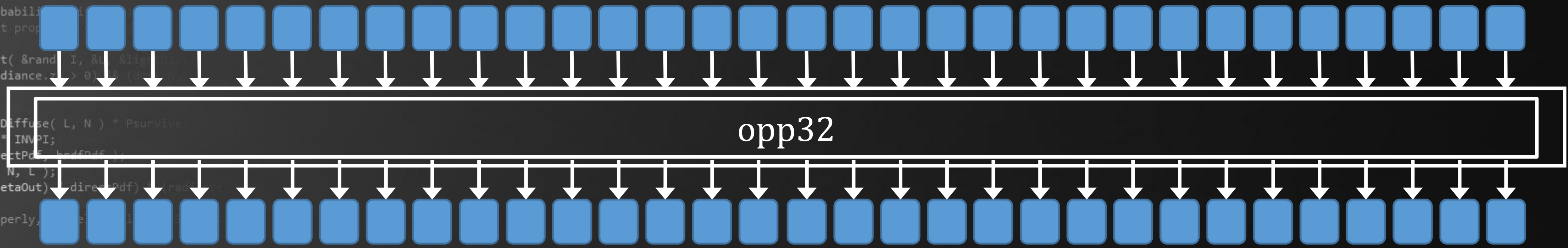
(uses `src` when mask bit is not set)

*: On: ‘Knights Landing’ (Intel, 2016).

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.;
        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 INFOB3CC, INFOMOV & INFOMAGR
Jacco Bikker, 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, 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[1], 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[2], and holds a single scalar value. CPU instructions typically operate on two operands. Consider the following code snippet:

```
vec3 velocity = GetPlayerSpeed();
float length = sqrtf( velocity );
```

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

```
x2 = velocity.x * velocity.x
y2 = velocity.y * velocity.y
z2 = velocity.z * velocity.z
sum = x2 + y2
sum = sum + z2
length = ...
```

/INFOMOV/

END of “SIMD (2)”

next lecture: “Data-Oriented Design”