Welcome!
Today's Agenda:

- Introduction
- Intel: SSE
- Streams
- Vectorization
Introduction

Consistent Approach

(0.) Determine optimization requirements
1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize / vectorize / use GPGPU
6. Profile again.
7. Apply low level optimizations to hotspots
8. Repeat steps 7 and 8 until time runs out

Rules of Engagement

1. Avoid Costly Operations
2. Precalculate
3. Pick the Right Data Type
4. Avoid Conditional Branches
5. Early Out
6. Use the Power of Two
7. Do Things Simultaneously
Introduction

S.I.M.D.

Single Instruction Multiple Data:

Applying the same instruction to several input elements.

In other words: if we are going to apply the same sequence of instructions to a large input set, this allows us to do this in parallel (and thus: faster).

SIMD is also known as instruction level parallelism.

Examples:

"Give me four random bytes, but they can’t be all zero"

```c
union {
    uint a4; unsigned char a[4];
};
do {
    a[0] = RandomByte();
    a[1] = RandomByte();
    a[2] = RandomByte();
    a[3] = RandomByte();
    while (a4 == 0);

    unsigned char a[4] = { 1, 2, 3, 4 };,
    unsigned char b[4] = { 5, 5, 5, 5 };
    unsigned char c[4];
    *(uint*)c = *(uint*)a + *(uint*)b;
    // c is now { 6, 7, 8, 9 }.
```
S.I.M.D.  
Single Instruction Multiple Data:  
Applying the same instruction to several input elements.  
In other words: if we are going to apply the same sequence of instructions to a large input set, this allows us to do this in parallel (and thus: faster).  
SIMD is also known as instruction level parallelism.

Examples:

"Give me four random bytes, but they can't be all zero"

```c
union {
  uint a4;
  unsigned char a[4];
};
do {
  a[0] = RandomByte();
  a[1] = RandomByte();
  a[2] = RandomByte();
  a[3] = RandomByte();
} while (a4 == 0);
```

```c
unsigned char a[4] = { 1, 2, 3, 4 };
unsigned char b[4] = { 5, 5, 5, 5 };
unsigned char c[4];
*(uint*)c = *(uint*)a + *(uint*)b;
```

```c
void Game::Tick( float deltaTime )
{
  union {
    dword ptr [rsp+10h], xmm1
    qword ptr [rsp+8h], rcx
  }
push rdi
  sub rsi, 90h
  mov rdi, rsi
  mov ecx, 24h
  mov eax, 0CCCCCCCH
  rep stos dword ptr [rdi]
  mov rcx, qword ptr [this]
  union {
    uint a4;
    unsigned char a[4];
  }
do {
  a[0] = RandomByte();
  a[1] = RandomByte();
  a[2] = RandomByte();
  a[3] = RandomByte();
} while (a4 == 0);
```
Introduction

Examples:

"Give me four random bytes, but they can't be all zero"

```c
union { uint a4; unsigned char a[4]; }
    do
    { 
        a[0] = RandomByte(); 
        a[1] = RandomByte(); 
        a[2] = RandomByte(); 
        a[3] = RandomByte(); 
    } 
    while (a4 == 0);

unsigned char a[4] = { 1, 2, 3, 4 }; 
unsigned char b[4] = { 5, 5, 5, 5 }; 
unsigned char c[4]; 
*(uint*)c = *(uint*)a + *(uint*)b; 
// c is now { 6, 7, 8, 9 }.
```
uint = unsigned char[4]

Pinging google.com yields: 74.125.136.101
Each value is an unsigned 8-bit value (0..255).
Combing them in one 32-bit integer:

101 +
256 * 136 +
256 * 256 * 125 +
256 * 256 * 256 * 74 = 1249740901.

Browse to: http://1249740901 (works!)

Evil use of this:

We can specify a user name when visiting a website, but any username will be accepted by google. Like this:

http://infomov@google.com

Or:

http://www.ing.nl@1249740901

Replace the IP address used here by your own site which contains a copy of the ing.nl site to obtain passwords, and send the link to a 'friend'.

http://infomov@google.com
Example: color scaling

Assume we represent colors as 32-bit ARGB values using unsigned ints:

To scale this color by a specified percentage, we use the following code:

```c
uint ScaleColor( uint c, float x ) // x = 0..1
{
    uint red = (c >> 16) & 255;
    uint green = (c >> 8) & 255;
    uint blue = c & 255;
    red = red * x, green = green * x, blue = blue * x;
    return (red << 16) + (green << 8) + blue;
}
```
Example: color scaling

```c
uint ScaleColor( uint c, float x ) // x = 0..1
{
    uint red = (c >> 16) & 255, green = (c >> 8) & 255, blue = c & 255;
    red = red * x, green = green * x, blue = blue * x;
    return (red << 16) + (green << 8) + blue;
}
```

Improved:

```c
uint ScaleColor( uint c, uint x ) // x = 0..255
{
    uint red = (c >> 16) & 255, green = (c >> 8) & 255, blue = c & 255;
    red = (red * x) >> 8;
    green = (green * x) >> 8;
    blue = (blue * x) >> 8;
    return (red << 16) + (green << 8) + blue;
}
```
Example: color scaling

```c
uint ScaleColor( uint c, uint x ) // x = 0..255
{
    uint red = (c >> 16) & 255, green = (c >> 8) & 255, blue = c & 255;
    red = (red * x) >> 8, green = (green * x) >> 8, blue = (blue * x) >> 8;
    return (red << 16) + (green << 8) + blue;
}
```

Improved:

```c
uint ScaleColor( const uint c, const uint x ) // x = 0..255
{
    uint redblue = c & 0x00FF00FF;
    uint green = c & 0x0000FF00;
    redblue = ((redblue * x) >> 8) & 0x00FF00FF;
    green = ((green * x) >> 8) & 0x0000FF00;
    return redblue + green;
}
```

7 shifts, 3 ands, 3 muls, 2 adds

2 shifts, 4 ands, 2 muls, 1 add
Example: color scaling

```c
uint ScaleColor( uint c, uint x ) // x = 0..255
{
    uint red = (c >> 16) & 255, green = (c >> 8) & 255, blue = c & 255;
    red = (red * x) >> 8, green = (green * x) >> 8, blue = (blue * x) >> 8;
    return (red << 16) + (green << 8) + blue;
}
```

Further improved:

```c
uint ScaleColor( const uint c, const uint x ) // x = 0..255
{
    uint redblue = c & 0x00FF00FF;
    uint green = c & 0x0000FF00;
    redblue = (redblue * x) & 0xFF00FF00;
    green = (green * x) & 0x000000FF;
    return (redblue + green) >> 8;
}
```

7 shifts, 3 ands, 3 muls, 2 adds (15 ops)

1 shift, 4 ands, 2 muls, 1 add (8 ops)
Introduction

Other Examples

Rapid string comparison:

```c
char a[] = "optimization skills rule";
char b[] = "optimization is so nice!";
bool equal = true;
int l = strlen(a);
for (int i = 0; i < l; i++)
{
    if (a[i] != b[i])
    {
        equal = false;
        break;
    }
}
```

Likewise, we can copy byte arrays faster.

```c
char a[] = "optimization skills rule";
char b[] = "optimization is so nice!";
bool equal = true;
int q = strlen(a) / 4;
for (int i = 0; i < q; i++)
{
    if (((int*)a)[i] != (((int*)b)[i]))
    {
        equal = false;
        break;
    }
}
Other Examples

Rapid string comparison:

```c
char a[] = "optimization skills rule";
char b[] = "optimization is so nice!";
bool equal = true;
int q = strlen(a) / 4;
for (int i = 0; i < q; i++)
{
    if (((int*)a)[i] != ((int*)b)[i])
    {
        equal = false;
        break;
    }
}
```

Likewise, we can copy byte arrays faster.
SIMD using 32-bit values - Limitations

Mapping four chars to an int value has a number of limitations:

\[
\begin{align*}
\{100, 100, 100, 100\} + \{1, 1, 1, 200\} &= \{101, 101, 102, 44\} \\
\{100, 100, 100, 100\} \times \{2, 2, 2, 2\} &= \{\ldots\} \\
\{100, 100, 100, 200\} \times 2 &= \{200, 200, 201, 144\}
\end{align*}
\]

In general:

- Streams are not separated (prone to overflow into next stream);
- Limited to small unsigned integer values;
- Hard to do multiplication / division.
Introduction

SIMD using 32-bit values - Limitations

Ideally, we would like to see:

- Isolated streams
- Support for more data types (char, short, uint, int, float, double)
- An easy to use approach

Meet SSE!
Today’s Agenda:

- Introduction
- Intel: SSE
- Streams
- Vectorization
A Brief History of SIMD

Early use of SIMD was in vector supercomputers such as the CDC Star-100 and TI ASC (image).

Intel's MMX extension to the x86 instruction set (1996) was the first use of SIMD in commodity hardware, followed by Motorola's AltiVec (1998), and Intel's SSE (P3, 1999).

**SSE:**
- 70 assembler instructions
- Operates on 128-bit registers
- Operates on vectors of 4 floats.
SIMD Basics

C++ supports a 128-bit vector data type: __m128
Henceforth, we will pronounce to this as ‘quadfloat’. 😊

__m128 literally is a small array of floats:

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

Alternatively, you can use the integer variety __m128i:

```cpp
union { __m128i a4; int a[4]; }; 
```
SSE

SIMD Basics

We operate on SSE data using *intrinsics*: in the case of SSE, these are keywords that translate to a single assembler instruction.

Examples:

```c
__m128 a4 = __mm_set_ps( 1, 0, 3.141592f, 9.5f );
__m128 b4 = __mm_setzero_ps();
__m128 c4 = __mm_add_ps( a4, b4 ); // not: __m128 = a4 + b4;
__m128 d4 = __mm_sub_ps( b4, a4 );
```

Here, '_ps' stands for *packed scalar*. 

---

INFOMOV – Lecture 4 – “SIMD (1)” 21
SSE

SIMD Basics

Other instructions:

```c
__m128 c4 = _mm_div_ps( a4, b4 ); // component-wise division
__m128 d4 = _mm_sqrt_ps( a4 ); // four square roots
__m128 d4 = _mm_rcp_ps( a4 ); // four reciprocals
__m128 d4 = _mm_rsqrt_ps( a4 ); // four reciprocal square roots (!)
__m128 d4 = _mm_max_ps( a4, b4 );
__m128 d4 = _mm_min_ps( a4, b4 );
```

Keep the assembler-like syntax in mind:

```c
__m128 d4 = dx4 * dx4 + dy4 * dy4;
__m128 d4 = _mm_add_ps(
    _mm_mul_ps( dx4, dx4 ),
    _mm_mul_ps( dy4, dy4 )
);```
SSE

SIMD Basics

In short:

- Four times the work at the price of a single scalar operation (if you can feed the data fast enough)
- Potentially even better performance for min, max, sqrt, rsqrt
- Requires four independent streams.

And, with AVX we get __m256...
Today's Agenda:

- Introduction
- Intel: SSE
- Streams
- Vectorization
SIMD According To Visual Studio

vec3 A( 1, 0, 0 );
vec3 B( 0, 1, 0 );
vec3 C = (A + B) * 0.1f;
vec3 D = normalize( C );

The compiler will notice that we are operating on 3-component vectors, and it will use SSE instructions to speed up the code. This results in a modest speedup. Note that one lane is never used at all.

To get maximum throughput, we want four independent streams, running in parallel.

Streams

Streams

Agner Fog:

"Automatic vectorization is the easiest way of generating SIMD code, and I would recommend to use this method when it works. Automatic vectorization may fail or produce suboptimal code in the following cases:
- when the algorithm is too complex.
- when data have to be re-arranged in order to fit into vectors and it is not obvious to the compiler how to do this or when other parts of the code needs to be changed to handle the re-arranged data.
- when it is not known to the compiler which data sets are bigger or smaller than the vector size.
- when it is not known to the compiler whether the size of a data set is a multiple of the vector size or not.
- when the algorithm involves calls to functions that are defined elsewhere or cannot be inlined and which are not readily available in vector versions.
- when the algorithm involves many branches that are not easily vectorized.
- when floating point operations have to be reordered or transformed and it is not known to the compiler whether these transformations are permissible with respect to precision, overflow, etc.
- when functions are implemented with lookup tables."
SIMD According To Visual Studio

```c
float Ax = 1, Ay = 0, Az = 0;
float Bx = 0, By = 1, Bz = 0;
float Cx = (Ax + Bx) * 0.1f;
float Cy = (Ay + By) * 0.1f;
float Cz = (Az + Bz) * 0.1f;
float l = sqrtf( Cx * Cx + Cy * Cy + Cz * Cz);
float Dx = Cx / l;
float Dy = Cy / l;
float Dz = Cz / l;
```
Streams

SIMD According To Visual Studio

```c
float Cx[4] = {...};
float Cy[4] = {...};
float Cz[4] = {...};
float l[4] = {...};
float Dx[4] = {...};
float Dy[4] = {...};
float Dz[4] = {...};
```
Streams

SIMD According To Visual Studio

```c
__m128 Ax4 = {...}, Ay4 = {...}, Az4 = {...};
__m128 Bx4 = {...}, By4 = {...}, Bz4 = {...};
__m128 Cx4 = ...
__m128 Cy4 = ...
__m128 Cz4 = ...
__m128 l4 = ...
__m128 Dx4 = ...
__m128 Dy4 = ...
__m128 Dz4 = ...
```

---

INFOMOV – Lecture 4 – “SIMD (1)”
SIMD According To Visual Studio

```c
__m128 Ax4 = {...}, Ay4 = {...}, Az4 = {...};
__m128 Bx4 = {...}, By4 = {...}, Bz4 = {...};
__m128 X4 = _mm_set1_ps( 0.1f );
__m128 Cx4 = _mm_mul_ps( _mm_add_ps( Ax4, Bx4 ), X4 );
__m128 Cy4 = _mm_mul_ps( _mm_add_ps( Ay4, By4 ), X4 );
__m128 Cz4 = _mm_mul_ps( _mm_add_ps( Az4, Bz4 ), X4 );
__m128 l4 = ...
__m128 Dx4 = ...
__m128 Dy4 = ...
__m128 Dz4 = ...
```

Streams
SIMD According To Visual Studio

```c
__m128 Ax4 = _mm_set_ps( Ax[0], Ax[1], Ax[2], Ax[3] );
__m128 Ay4 = _mm_set_ps( Ay[0], Ay[1], Ay[2], Ay[3] );
__m128 Az4 = _mm_set_ps( Az[0], Az[1], Az[2], Az[3] );
__m128 Bx4 = {...}, By4 = {...}, Bz4 = {...};
__m128 X4 = _mm_set1_ps( 0.1f );
__m128 Cx4 = _mm_mul_ps( _mm_add_ps( Ax4, Bx4 ), X4 );
__m128 Cy4 = _mm_mul_ps( _mm_add_ps( Ay4, By4 ), X4 );
__m128 Cz4 = _mm_mul_ps( _mm_add_ps( Az4, Bz4 ), X4 );
__m128 l4 = ...
__m128 Dx4 = ...
__m128 Dy4 = ...
__m128 Dz4 = ...
```

Streams
Streams

SIMD Friendly Data Layout

Consider the following data structure:

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

Particle particle[512];
```

AoS  SoA
Streams

SIMD Data Naming Conventions

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]; };;

Notice that SoA is breaking our OO...

Consider adding the struct name to the variables:

float particle_x[512];

Or put an amount of particles in a struct.

Also note the convention of adding ‘4’ to any SSE variable.
Today's Agenda:

- Introduction
- Intel: SSE
- Streams
- Vectorization
Vectorization

Converting your Code

1. Locate a significant bottleneck in your code (converting is going to be labor-intensive, be sure it’s worth it)

2. Keep a copy of the original code (use #ifdef) (you may want to compile on some other platform later)

3. Prepare the scalar code (add a ‘for( int stream = 0; stream < 4; stream++ )’ loop)

4. Reorganize the data (make sure you don’t have to convert all the time)

5. Union with floats

6. Convert one line at a time, verifying functionality as you go

7. Check MSDN for exotic SSE instructions (some odd instructions exist that may help your problem)
END of “SIMD (1)"

next lecture: “SIMD (2)”