

/INFOMOV/

Optimization & Vectorization

J. Bikker & D. Alexandridis - April - June 2022 - Lecture 4: "SIMD (1)"

Welcome!



```
ics
& (depth < MAXDEPTH)
{
    if ( ! inside )
        return;
    int nt = nt / nc, nct = nct / nc;
    double r0s2t = 1.0f + nnt * nnt;
    double D, N );
}

at a = nt - nc, b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * r0s2t);
Tr) R = (D * nnt - N * (D0 + D1 * r0s2t));

E * diffuse;
= true;

-
efl + refr)) && (depth < MAXDEPTH)
{
    D, N );
    refl * E * diffuse;
    = true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) && (depth < MAXDEPTH)
{
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance

random walk - done properly, closely following
ive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

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
9. Report.

```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * ddn;
        cos2t = 1.0f - nnt;
        D, N );
    }
}

at a = nt - nc, b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) *
Tr) R = (D * nnt - N * (ddn
);

E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
{
    D, N );
    refl * E * diffuse;
    = true;
}

MAXDEPTH)
survive = SurvivalProbability( diffuse,
estimation - doing it properly, closely
if;
radiance = SampleLight( &rand, I, &L, &light
e.x + radiance.y + radiance.z) > 0) && (rand
);
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
);

random walk - done properly, closely following
ive)
);
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
    
```



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*.

```

...ics
& (depth < MAXDEPTH)
...
= inside ? 1.0f : 0.0f;
nt = nt / nc; ddn = ddn / nc;
ps2t = 1.0f - nnt * ddn;
D, N );
)
...
at a = nt - nc; b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * R);
Tr) R = (D * nnt - N * ddn) * ps2t;
...
E * diffuse;
= true;
...
efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;
...
MAXDEPTH)
survive = SurvivalProbability( diffuse, 1.0f - diffuse);
estimation - doing it properly, closely following the
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
...
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following the
survive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```

Examples:

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

```

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 }.

```



```

void Game::Tick( float deltaTime )
{
00000000140002C40  movss      dword ptr [rsp+10h],xmm1
00000000140002C46  mov       qword ptr [rsp+8],rcx
00000000140002C4B  push     rdi
00000000140002C4C  sub      rsp,90h
00000000140002C53  mov      rdi,rsp
00000000140002C56  mov      ecx,24h
00000000140002C5B  mov      eax,0CCCCCCCCh
00000000140002C60  rep stos dword ptr [rdi]
00000000140002C62  mov      rcx,qword ptr [this]
    unsigned char a[4] = { 1, 2, 3, 4 };
00000000140002C6A  mov      byte ptr [a],1
00000000140002C6F  mov      byte ptr [rsp+35h],2
00000000140002C74  mov      byte ptr [rsp+36h],3
00000000140002C79  mov      byte ptr [rsp+37h],4
    unsigned char b[4] = { 5, 5, 5, 5 };
00000000140002C7E  mov      byte ptr [b],5
00000000140002C83  mov      byte ptr [rsp+55h],5
00000000140002C88  mov      byte ptr [rsp+56h],5
00000000140002C8D  mov      byte ptr [rsp+57h],5
    unsigned char c[4];
    *(uint*)c = *(uint*)a + *(uint*)b;
00000000140002C92  mov      eax,dword ptr [b]
00000000140002C96  mov      ecx,dword ptr [a]
00000000140002C9A  add     ecx,eax
00000000140002C9C  mov      eax,ecx
00000000140002C9E  mov     dword ptr [c],eax

```

Examples:

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

```

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 }.

```



Introduction

```

void Game::Tick( float deltaTime )
{
00000000140002250  movss      dword ptr [rsp+10h],xmm1
00000000140002256  mov       qword ptr [rsp+8],rcx
0000000014000225B  sub      rsp,38h
    unsigned char a[4] = { 1, 2, 3, 4 };
    unsigned char b[4] = { 5, 5, 5, 5 };
0000000014000225F  mov      dword ptr [rsp+40h],5050505h
    unsigned char c[4];
    *(uint*)c = *(uint*)a + *(uint*)b;
00000000140002267  mov      edx,dword ptr [b]
0000000014000226B  mov      dword ptr [rsp+48h],4030201h
00000000140002273  add      edx,dword ptr [a]
00000000140002277  mov      ecx,edx
00000000140002279  mov      eax,edx

```

Examples:

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

```

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 }.

```



Introduction

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’.



Introduction

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:

```
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;
}
```



Introduction



Example: color scaling

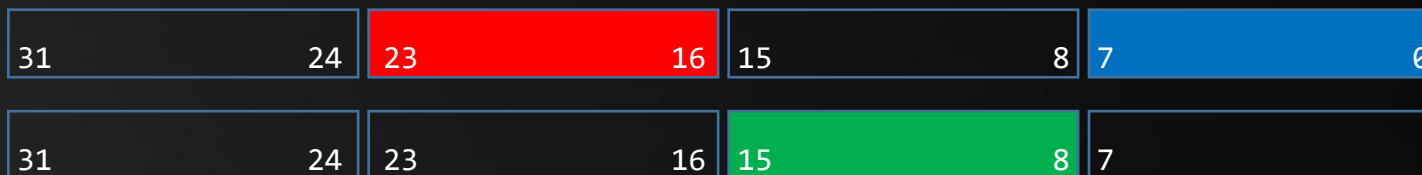
```
uint ScaleColor( uint c, float x ) // x = 0..1
{
    uint red = (c >> 16) & 255, green = (c >> 8) & 255, blue = c & 255;
    red = red * x, green = green * x, blue = blue * x;
    return (red << 16) + (green << 8) + blue;
}
```

Improved:

```
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;
}
```



Introduction



Example: color scaling

```
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;
}
```

7 shifts, 3 ands, 3 muls, 2 adds

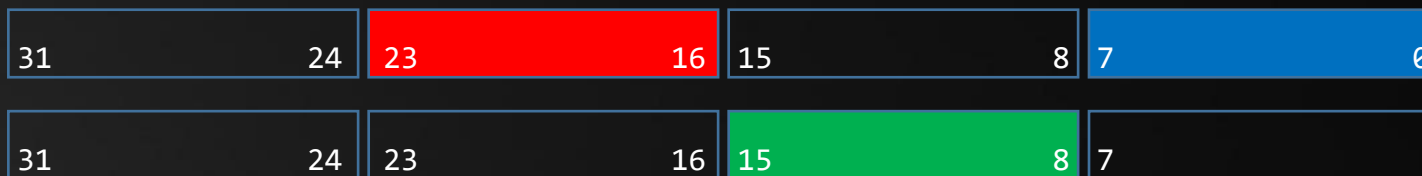
Improved:

```
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;
}
```

2 shifts, 4 ands, 2 muls, 1 add



Introduction



Example: color scaling

```
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;
}
```

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

Further improved:

```
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) & 0x00FF0000;
    return (redblue + green) >> 8;
}
```

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



Introduction

Other Examples

Rapid string comparison:

```

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.

```

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;
    }
}

```



```

    for (int i = 0; i < q; i++)
000000014000228D movsxd    rdx,eax
00000001400022C0 test     eax,eax
00000001400022C2 jle     Tmpl8::Game::Tick+87h (01400022D7h)
00000001400022C4 xor     eax,eax
    {
        if (((int*)a)[i] != ((int*)b)[i])
00000001400022C6 mov     ecx,dword ptr b[rax*4]
00000001400022CA cmp     dword ptr [rsp+rax*4],ecx
00000001400022CD jne     Tmpl8::Game::Tick+87h (01400022D7h)
        for (int i = 0; i < q; i++)
00000001400022CF inc     rax
00000001400022D2 cmp     rax,rdx
00000001400022D5 jl     Tmpl8::Game::Tick+76h (01400022C6h)
            {
                equal = false;
                break;
            }
    }
}

```

```

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;
    }
}

```



Introduction

SIMD using 32-bit values - Limitations

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

$$\{ 100, 100, 100, 100 \} + \{ 1, 1, 1, 200 \} = \{ 101, 101, 102, 44 \}$$

$$\{ 100, 100, 100, 100 \} * \{ 2, 2, 2, 2 \} = \{ \dots \}$$

$$\{ 100, 100, 100, 200 \} * 2 = \{ 200, 200, 201, 144 \}$$

In general:

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

```

ics
& (depth < MAXDEPTH)
{
    int inside = 1;
    int nt = nt / nc;
    double ddn = dot(N, D);
    double nnt = 1.0f - nnt * nnt;
    double NdotN = dot(N, N);
    double R = (D * nnt - N * (ddn * nnt));
    double E * diffuse;
    bool = true;
    double refl + refr;
    double D, N;
    double refl * E * diffuse;
    bool = true;
    double MAXDEPTH);
    double survive = SurvivalProbability( diffuse );
    double estimation - doing it properly, closely following;
    double radiance = SampleLight( &rand, I, &L, &light );
    double e.x + radiance.y + radiance.z > 0) && (survive);
    bool = true;
    double brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    double factor = diffuse * INVPI;
    double weight = Mis2( directPdf, brdfPdf );
    double cosThetaOut = dot( N, L );
    double E * ((weight * cosThetaOut) / directPdf) * (radiance);
    double random walk - done properly, closely following;
    double survive);
    double brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    double survive;
    double pdf;
    double n = E * brdf * (dot( N, R ) / pdf);
    double mission = true;
    
```



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!

```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * 2;
        ps2t = 1.0f - nnt * nnt;
        D, N );
    }
}

at a = nt - nc, b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * ddn);
at R = (D * nnt - N * (ddn * nnt));

E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
{
    D, N );
    refl * E * diffuse;
    = true;
}

MAXDEPTH)
{
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    if;
    radiance = SampleLight( &rand, I, &L, &light);
    e.x + radiance.y + radiance.z) > 0) && (depth <
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
    random walk - done properly, closely following
    survive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
}

```



```
...ics
& (depth < MAXDEPTH)
{
    if ( ! inside )
        return;
    int nt = nt / nc, ddn = ddn / nc;
    double r0s2t = 1.0f - nnt * nnt;
    double D, N );
}

double a = nt - nc, b = nt + nc;
double Tr = 1 - (R0 + (1 - R0) * ddn);
double R = (D * nnt - N * (ddn > 0 ? 1 : -1));

E * diffuse;
= true;

-
refl + refr)) && (depth < MAXDEPTH)
{
    D, N );
    refl * E * diffuse;
    = true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light,
e.x + radiance.y + radiance.z) > 0) && (depth <
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance

random walk - done properly, closely following
vive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
urvive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

Today's Agenda:

- Introduction
- Intel: SSE
- Streams
- Vectorization



SSE

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.



SSE

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:

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

Alternatively, you can use the integer variety `__m128i`:

```
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:

```

__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*.



SSE

SIMD Basics

Other instructions:

```

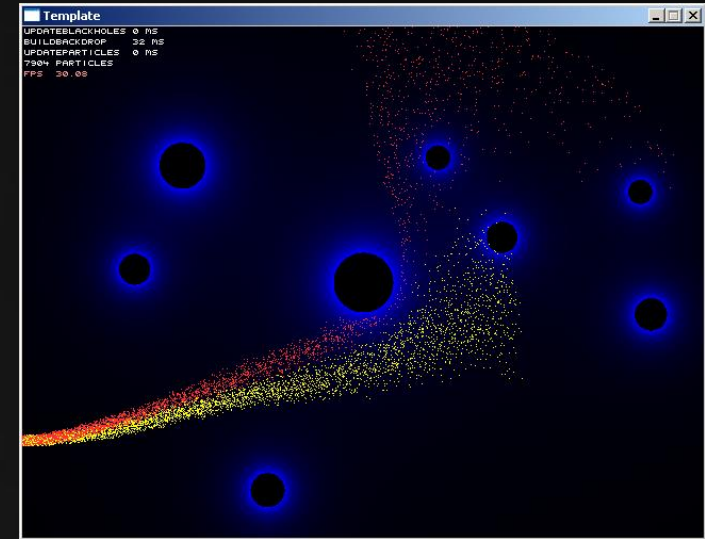
__m128 c4 = _mm_div_ps( a4, b4 ); // component-wise division
__m128 d4 = _mm_sqrt_ps( a4 ); // four square roots
__m128 d4 = _mm_rcp_ps( a4 ); // four reciprocals
__m128 d4 = _mm_rsqrt_ps( a4 ); // four reciprocal square roots (!)

__m128 d4 = _mm_max_ps( a4, b4 );
__m128 d4 = _mm_min_ps( a4, b4 );
    
```

Keep the assembler-like syntax in mind:

```

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



CODING TIME



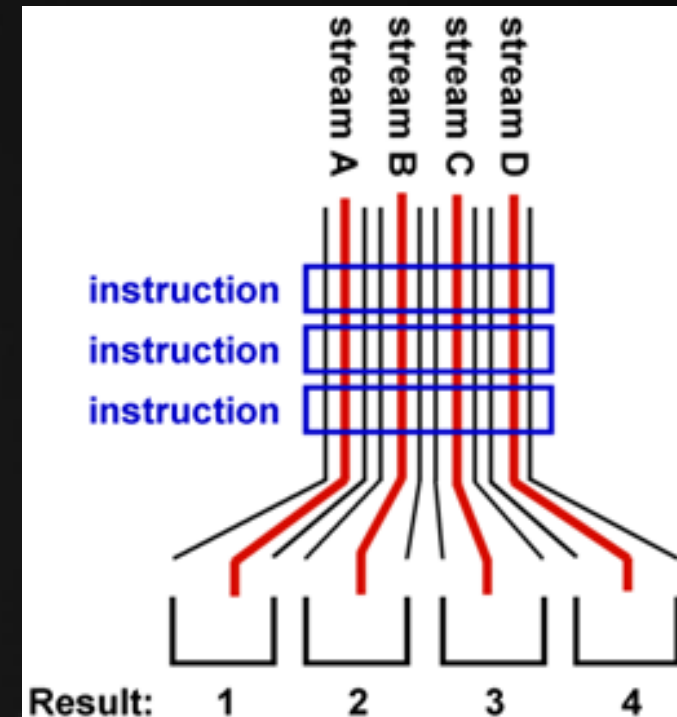
SSE

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...`



```
ics
& (depth < MAXDEPTH)
{
    if ( ! inside )
        return;
    int nt = nt / nc, nct = nct / nc;
    double r0s2t = 1.0f + nnt * nnt;
    double D, N );
    }
    double a = nt - nc, b = nt + nc;
    double Tr = 1 - (R0 + (1 - R0) * r0s2t);
    double R = (D * nnt - N * (Dn));
    double E * diffuse;
    bool = true;
    -
    refl + refr)) && (depth < MAXDEPTH)
    {
        D, N );
        refl * E * diffuse;
        = true;
    }
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    if;
    radiance = SampleLight( &rand, I, &L, &light);
    e.x + radiance.y + radiance.z) > 0) && (abs(
    w = true;
    double brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    double factor = diffuse * INVPI;
    double weight = Mis2( directPdf, brdfPdf );
    double cosThetaOut = dot( N, L );
    double E * ((weight * cosThetaOut) / directPdf) * (radiance
    random walk - done properly, closely following
    survive)
    ;
    double brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
}
```

Today's Agenda:

- Introduction
- Intel: SSE
- Streams
- Vectorization



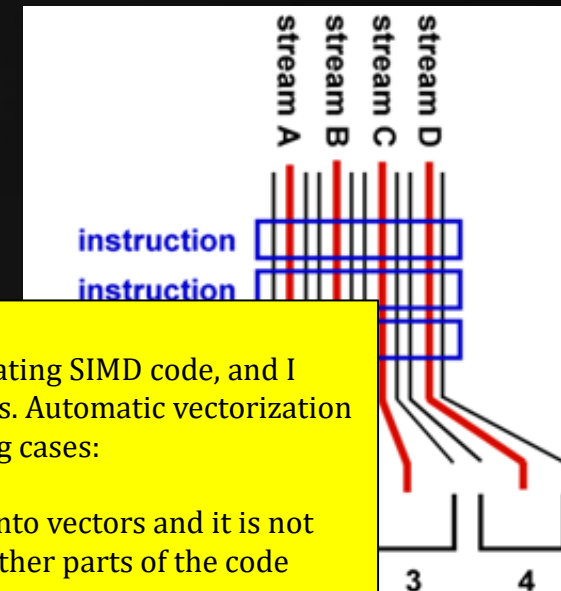
Streams

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 vectors, and it will use SSE instructions. This results in a modest speedup. Note that operating on vectors

To get maximum throughput, we want for each instruction running in parallel.



- 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.

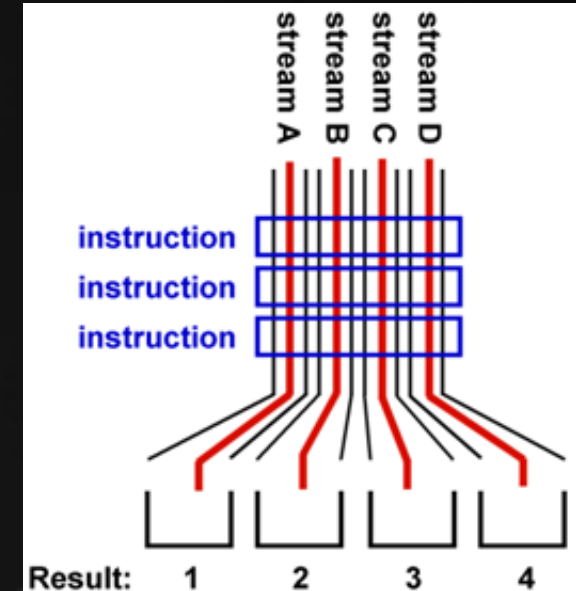


Streams

SIMD According To Visual Studio

```

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

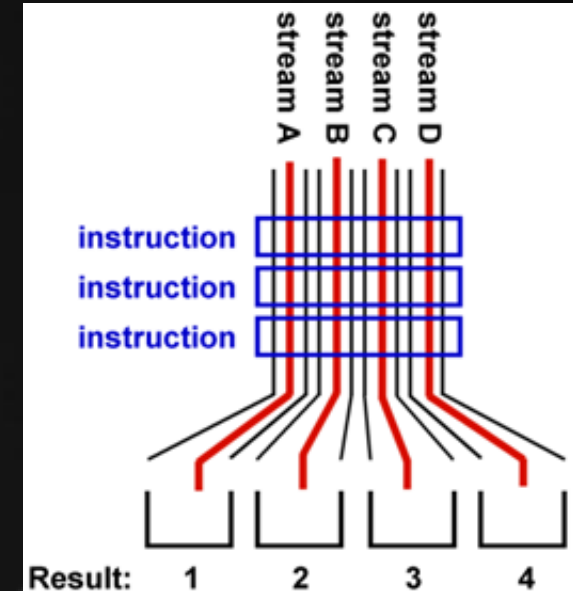
SIMD According To Visual Studio

```

...
    & (depth < MAXDEPTH)
...
    = inside ? 1.0f : 0.0f;
    nt = nt / nc; ddn = ddn * ddn;
    ps2t = 1.0f - nnt * ddn;
    D, N );
    )
...
    at a = nt - nc; b = nt + nc;
    at Tr = 1 - (R0 + (1 - R0) * ps2t);
    Tr) R = (D * nnt - N * (ddn * ps2t));
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (rand < survive);
...
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following
(ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, $pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
    
```

```

float Ax[4] = {...}, Ay[4] = {...}, Az[4] = {...};
float Bx[4] = {...}, By[4] = {...}, Bz[4] = {...};
float Cx[4] = ...;
float Cy[4] = ...;
float Cz[4] = ...;
float l[4] = ...;
float Dx[4] = ...;
float Dy[4] = ...;
float Dz[4] = ...;
    
```



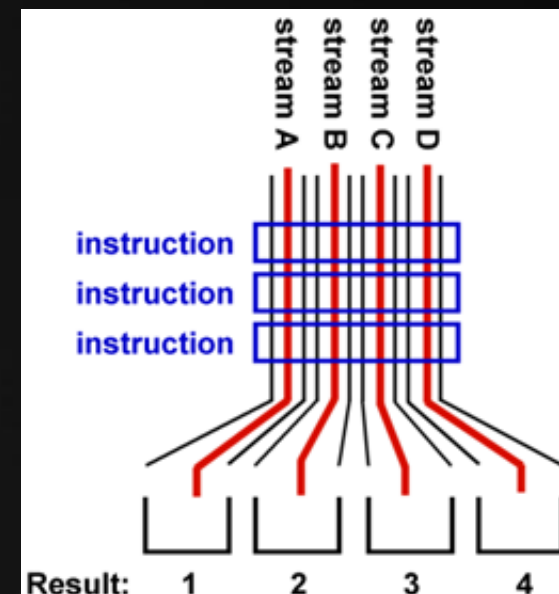
Streams

SIMD According To Visual Studio

```

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

```



Streams

SIMD According To Visual Studio

```

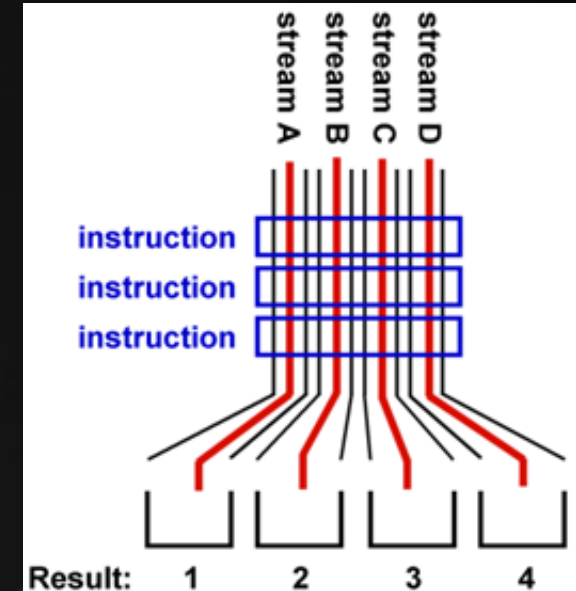
...
    & (depth < MAXDEPTH)
...
    = inside ? 1.0f : 0.0f;
    nt = nt / nc; add = add * nt;
    ps2t = 1.0f - nnt * nt;
    D, N );
    )
...
    at a = nt - nc, b = nt - nc;
    at Tr = 1 - (R0 + (1 - R0) * nt);
    Tr) R = (D * nnt - N * (add
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
    D, N );
    refl * E * diffuse;
    = true;
...
    MAXDEPTH)
...
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    if;
    radiance = SampleLight( &rand, I, &L, &light;
    e.x + radiance.y + radiance.z) > 0) && (add
...
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
...
    random walk - done properly, closely following
    (survive)
...
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf;
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```

```

__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

```

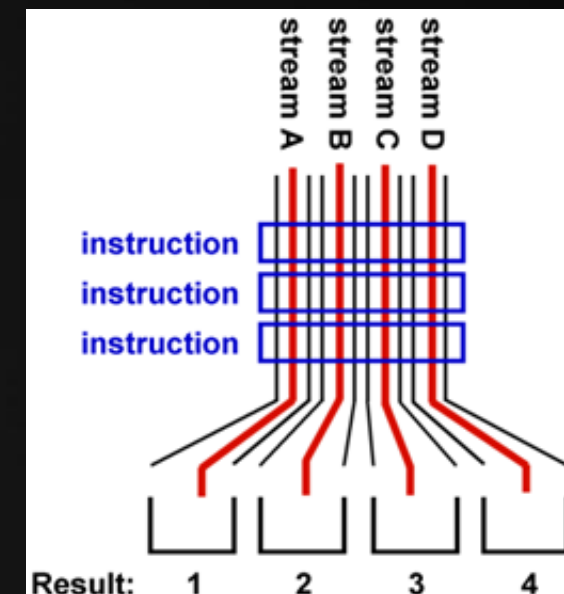
...
& (depth < MAXDEPTH)
...
inside ? 1 : 0;
nt = nt / nc;
ps2t = 1.0f - nnt;
D, N );
)
at a = nt - nc, b = nt - nc;
at Tr = 1 - (R0 + (1 - R0)
Tr) R = (D * nnt - N * (D0
E * diffuse;
= true;
...
efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;
MAXDEPTH)
survive = SurvivalProbability( diffuse,
estimation - doing it properly, closely
if;
radiance = SampleLight( &rand, I, &L, &light
e.x + radiance.y + radiance.z) > 0) && (depth <
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf;
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```

```

__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

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.



```
...ics
& (depth < MAXDEPTH)
{
    if ( ! inside )
        return;
    int nt = nt / nc, nct = nct / nc;
    double r0s2t = 1.0f + nnt * nnt;
    double D, N );
}

at a = nt - nc, b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * r0s2t);
Tr) R = (D * nnt - N * (D0 + D1 * r0s2t));

E * diffuse;
= true;

-
refl + refr)) && (depth < MAXDEPTH)
{
    D, N );
    refl * E * diffuse;
    = true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) && (abs(radiance.x) > 0) && (abs(radiance.y) > 0) && (abs(radiance.z) > 0)
{
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance.x * cosThetaOut + radiance.y * sinThetaOut * cosPhiOut + radiance.z * sinThetaOut * sinPhiOut);
}

random walk - done properly, closely following
survive)
{
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
}
```

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)



/INFOMOV/

END of “SIMD (1)”

next lecture: “SIMD (2)”

```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * ddn;
        cos2t = 1.0f - nnt * nnt;
        D, N );
    }

    at a = nt - nc, b = nt + nc;
    at Tr = 1 - (R0 + (1 - R0) * r);
    (Fr) R = (D * nnt - N * (ddn * ddn));

    E * diffuse;
    = true;

    refl + refr)) && (depth < MAXDEPTH)
    D, N );
    refl * E * diffuse;
    = true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (cosThetaOut > 0)
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance

random walk - done properly, closely following
ive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
urvive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

