Welcome!
Meanwhile, on ars technica

Crystalwell Architecture

Unlike previous eDRAM implementations in game consoles, Crystalwell is true 4th level cache in the memory hierarchy. It acts as a victim buffer to the L3 cache, meaning anything evicted from L3 cache immediately goes into the L4 cache. Both CPU and GPU requests are cached. The cache can dynamically allocate its partitioning between CPU and GPU use. If you don’t use the GPU at all (e.g., discrete GPU installed), CPU requests. That’s right, Haswell CPUs equipped with connection to Crystalwell other than to say that it’s a next generation 50GB/s bi-directional bandwidth (100GB/s aggregate) of 30 - 32ns, nicely in between an L3 and main memory.

Latency vs. Access Range (Sandra 2013 SP3)

The eDRAM clock tops out at 1.6GHz.

There’s only a single size of eDRAM offered this generation: 128MB. Since it’s a cache and not a buffer (and not at that), Intel found that hit rate rarely dropped below 95%. It turns out that for current workloads, Intel didn’t see much benefit beyond a 32MB eDRAM however it wanted the design to be future proof. Intel
Meanwhile, the job market

We only have jobs for people who get bored easily.

In return we offer a solution to boredom, as performance engineering is hard.

As the market demand for affordable high performance software grows, StreamHPC continuously looks for people to join the team. With upcoming products and new markets like OpenCL on low-power ARM processors, we expect continuous growth for the years to come.

To apply send your motivation and a recent version of your CV to jobs@streamhpc.com. If you have LinkedIn, you can easily build a CV with LinkedIn Resume Builder and send us the link.

- OpenCL/CUDA export (Also accepting freelancers)
- Sales support

The procedure for the technical roles is as follows:

- You send a CV and tell us why you are the perfect candidate.
- After that you are invited for a longer online test. You show your skills on C/C++ and algorithms. You will receive a PDF with useful feedback.
- If you selected GPGPU or mentioned it, we send you a GPU assignment. You need to pick out the right optimisations, code it and explain your decisions. (Hopefully under 30 minutes)
- If all goes well, you’ll have a video chat with Vincent (CEO) on personal and practical matters. You can also ask us anything, to find out if we fit you. (Around 1 hour)
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:

```c
union { uint a4; unsigned char a[4]; };
do
{
    GetFourRandomValues( a );
}
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:

```c
union { uint a4; unsigned char a[4]; };
do {
    GetFourRandomValues( a );
}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;
```

```assembly
void Game::Tick( float deltaTime )
{
    movss    dword ptr [rsp+10h],xmm1
    mov      qword ptr [rsp+8],rcx
    push     rdi
    sub      rsp,90h
    rdi,rsp
    mov      ecx,24h
    mov      eax,0CCCCCCCCCh
    rep stos  dword ptr [rdi]
    rcx,qword ptr [this]
    union char a[4] = { 1, 2, 3, 4 };
    byte ptr [a],1
    byte ptr [rsp+35h],2
    byte ptr [rsp+36h],3
    byte ptr [rsp+37h],4
    unsigned char b[4] = { 5, 5, 5, 5 };
    byte ptr [b],5
    byte ptr [rsp+55h],5
    byte ptr [rsp+56h],5
    byte ptr [rsp+57h],5
    unsigned char c[4];
    *(uint*)c = *(uint*)a + *(uint*)b;
    eax,dword ptr [b]
    ecx,dword ptr [a]
    ecx,eax
    eax,ecx
    eax,eax
    dword ptr [c],eax
}
```

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:

```c
union {
    uint a4;
    unsigned char a[4];
};
do {
    GetFourRandomValues( a );
}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](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](http://infomov@google.com)

Or:

[http://www.ing.nl@1249740901](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’.
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) & 0x00FF0000;
    return (redblue + green) >> 8;
}
```
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.

```assembly
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 \} = \{ \ldots \}
\{ 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.
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 this as ‘quadfloat’.

__m128 literally is a small array of floats:

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

Alternatively, you can use the integer variety __m128i:

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

SIMD Basics

Other instructions:

\[
\_ \_ m 1 2 8 \quad c 4 = \_ \_ m m _ { d i v} _ { p s} ( a 4 , b 4 ) ; \quad // \quad c o m p o n e n t - w i s e \quad d i v i s i o n
\]

\[
\_ \_ m 1 2 8 \quad d 4 = \_ \_ m m _ { s q r t} _ { p s} ( a 4 ) ; \quad // \quad f o u r \quad s q u a r e \quad r o o t s
\]

\[
\_ \_ m 1 2 8 \quad d 4 = \_ \_ m m _ { r c p} _ { p s} ( a 4 ) ; \quad // \quad f o u r \quad r e c i p r o c a l s
\]

\[
\_ \_ m 1 2 8 \quad d 4 = \_ \_ m m _ { r s q r t} _ { p s} ( a 4 ) ; \quad // \quad f o u r \quad r e c i p r o c a l \quad s q u a r e \quad r o o t s \quad (!)
\]

\[
\_ \_ m 1 2 8 \quad d 4 = \_ \_ m m _ { m a x} _ { p s} ( a 4 , b 4 ) ;
\]

\[
\_ \_ m 1 2 8 \quad d 4 = \_ \_ m m _ { m i n} _ { p s} ( a 4 , b 4 ) ;
\]

Keep the assembler-like syntax in mind:

\[
\_ \_ m 1 2 8 \quad d 4 = \_ \_ m m _ { a d d} _ { p s} ( d x 4 , d y 4 ) ;
\]

\[
\_ \_ m 1 2 8 \quad d 4 = \_ \_ m m _ { m u l} _ { p s} ( d x 4 , d x 4 ) ,
\]

\[
_ \_ m m _ { m u l} _ { p s} ( d y 4 , d y 4 )
\]
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

```cpp
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

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

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

__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

```cpp
__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 = ...
```
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**

```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]; };
```
SIMD Data Naming Conventions

```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]; }; 
```

Notice that SoA is breaking our OO...

Consider adding the struct name to the variables:

```c
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)”
Assignment P2 – Cache Simulator

Formal assignment description for P2 - INFOMOV
Jacco Bakker, 2019

Introduction

This document describes the requirements for the second assignment for the INFOMOV course. For this assignment, you will extend a simple cache simulator, which currently implements a fully associative cache.

Base Code

The base code in game.cpp renders a spiral. The contents of a simulated memory system are visualized in real-time: bright colors are cached; darker ones reside in ‘DRAM’. The default cache uses a random eviction policy, so pixels of the spiral will randomly leave the cache, resulting in an attractive spiky trail.

An alternative chunk of code renders a Mandelbrot fractal (note: historically Genshi is more accurate). This is a fractal similar to the Mandelbrot fractal, and consists of the set of points in the complex plane for which the sequence \( z_{n+1} = z_n^2 + c \) does not tend to infinity for \( z_0 = 0 \) (as described by Wikipedia). The actual implementation is as mysterious as the previous sentence. One part of the code matters: the two lines that read and write data to iteratively update the image buffer.

Memory access

The two code paths have very different memory access patterns:

- The code for the spiral tends to access memory in a more random order.
- The application has regular memory access patterns with memory traffic patterns that are more cache-friendly.

In the course of development you may replace the default code with your own implementation. You may also use different heuristics when choosing cache replacement policies.