2019/2020, 1st quarter

INFOMOV: Optimization & Vectorization / *OptmzdSummary*

#3 – Practical SIMD

Author: Jacco Bikker

TL;DR

This OptmzdSummary is an adaptation of an existing tutorial for C++ and C# programmers, originally written for the Advanced Graphics course of the Utrecht University.

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 integer or floating point scalar value.

Consider the following code snippet:

```
vec3 velocity = GetPlayerSpeed();
float length = velocity.Length();
```

CPU instructions typically operate on two scalar operands. The first line, which calculates the length of the vector, can be broken down in a series of scalar operations:

---

[1] AVX512, available in Intel's Knights Landing architecture, supports 16 inputs. This technology is not yet available in consumer level CPUs.

[2] For the sake of simplicity, we ignore the fact that some registers can be split in 16-bit halves, or even in single bytes. Floating point numbers may be stored in 80-bit registers. For details see OptmzdSummary #1.

```
x2 = velocity.x * velocity.x
y2 = velocity.y * velocity.y
z2 = velocity.z * velocity.z
sum_of_xyzsquared = x2 + y2
sum_of_xyzsquared = sum_of_xyzsquared + z2
length = sqrtf( sum_of_xyzsquared )
```

Vector registers store 4 (SSE) or 8 (AVX) scalars. This means that the C# or C++ vector remains a vector at the assembler level: rather than storing three separate values in three registers, we store four values (x, y, z and a dummy value) in a single vector register. And, rather than squaring x, y and z separately, we use a single SIMD instruction to square the three values (as well as the dummy value).

This simple example illustrates a number of issues we need to deal with when writing SIMD code:

- When operating on three-component vectors, we do not use the full compute potential of the vector processor: we waste 25% (for SSE) or 62.5% (for AVX) of the 'slots' in the SIMD register.
- Storing three scalars in the vector register is not free: the cost depends on a number of factors which we will discuss later. This adds some overhead to the calculation.
- The square root on the last line is still performed on a single value. So, although this is the most expensive line, it doesn't benefit from the vector hardware, limiting our gains.

There is a reliable way to mitigate these concerns. Suppose our application is actually a four-player game:

```
for( int i = 0; i < 4; i++ )
{
   vec3 velocity = GetPlayerSpeed();
   float length = velocity.Length();
}
```

In this scenario, we can operate on four vectors at the same time:

```
x4 = GetPlayerXSpeeds();
y4 = GetPlayerYSpeeds();
z4 = GetPlayerZSpeeds();
x4squared = x4 * x4;
y4squared = y4 * y4;
z4squared = z4 * z4;
sum4 = x4squared + y4squared;
sum4 = sum4 + z4squared;
length4 = sqrtf4( sum4 );
```

Note that we have completely decoupled the C++/C# vector concept from the SIMD vectors: we simply use the SIMD vectors to execute the original scalar functionality four times in parallel. Every line now uses a SIMD instruction, at 100% efficiency (granted, we need 8 players for AVX…), even the square root is now calculated for four numbers.

There is one important thing to notice here: in order to make the first three lines efficient, player speeds must already be stored in a 'SIMD-friendly' format, i.e.: xxxx, yyyy, zzzz. Data organized like this can be directly copied into a vector register.

This also means that we can not possibly expect the compiler to do this for us automatically. Efficient SIMD code requires an efficient data layout; this *must* be done manually.

Data parallelism

The example with four player speeds would waste 50% of the compute potential on AVX machines. Obviously, we need more jobs. Efficient SIMD code requires massive *data parallelism*, where a sequence of operations is executed for a large number of inputs. Reaching 100% efficiency requires that the input array size is a multiple of 4 or 8; however for any significant input array size we get very close to this optimum and AVX performance simply becomes twice the SSE performance.

For a data-parallel algorithm, each of the scalars in a SIMD register holds the data for one 'thread'. We call the slots in the register *lanes*. The input data is called a *stream*.

Into the Mud

If you are a C++ programmer, you are probably familiar with the basic types: char, short, int, float, and so on. Each of these have specific sizes: 8 bits for a char, 16 for short, 32 for int and float. Bits are just bits, and therefore the difference between a float and an int is in the interpretation. This allows us to do some nasty things:

```
int a;
float& b = (float&)a;
```

This creates one integer, and a float reference, which points to a. Since variables a and b now occupy the same memory location, changing a changes b, and vice versa. An alternative way to achieve this is using a union:

```
union { int a; float b; };
```

Again, a and b reside in the same memory location. Here's another example:

```
union { unsigned int a4; unsigned char a[4]; };
```

This time, a small array of four chars overlaps the 32-bit integer value a4. We can now access the individual bytes in a4 via array `a[4]`. Note that a4 now basically has four 1-byte 'lanes', which is somewhat similar to what we get with SIMD. We could even use a4 as 32 1-bit values, which is an efficient way to store 32 boolean values.

An SSE register is 128 bit in size, and is named `__m128` if it is used to store four floats, or `__m128i` for ints. For convenience, we will pronounce `__m128` as 'quadfloat', and `__m128i` as 'quadint'. The AVX versions are `__m256` ('octfloat') and `__m256i` ('octint'). To be able to use the SIMD types, we need to include some headers:

```
#include "nmmintrin.h" // for SSE4.2
#include "immintrin.h" // for AVX
```

A `__m128` variable contains four floats, so we can use the union trick again:

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

Now we can conveniently access the individual floats in the `__m128` vector.

We can also create the quadfloat directly:

```
__m128 a4 = _mm_set_ps( 4.0f, 4.1f, 4.2f, 4.3f );
__m128 b4 = _mm_set_ps( 1.0f, 1.0f, 1.0f, 1.0f );
```

To add them together, we use `_mm_add_ps`:

```
__m128 sum4 = _mm_add_ps( a4, b4 );
```

The `__mm_set_ps` and `_mm_add_ps` keywords are called *intrinsics*. SSE and AVX intrinsics all compile to a single assembler instruction; using these means that we are essentially writing assembler code directly in our program. There is an intrinsic for virtually every scalar operation:

```
_mm_sub_ps( a4, b4 );
_mm_mul_ps( a4, b4 );
_mm_div_ps( a4, b4 );
_mm_sqrt_ps( a4 );
_mm_rcp_ps( a4 ); // reciprocal
```

For AVX we use similar intrinsics: simply prepend with `_mm256` instead of `_mm`, so: `_mm256_add_ps( a4, b4 )`, and so on.

A full overview of SSE and AVX instructions can be found here:

https://software.intel.com/sites/landingpage/IntrinsicsGuide/

You can safely assume that any CPU produced after 2000 supports SSE up to 4.2. AVX and especially AVX2 are more recent technologies; check Wikipedia for a list of supporting processors:

https://en.wikipedia.org/wiki/Advanced_Vector_Extensions

SIMD in C#

The previous section assumed the use of C++. Luckily, SIMD is also available in C#, although the implementation is not great.

SIMD support can be found in the System.Numerics.Vectors package. First, you need to add the latest version of the assembly (4.3.0 at the time of writing) via the Nuget Package Manager.

Here is a small C# program that we will use to do some experiments:

```
 1  using System.Numerics;
 2  namespace test { class P { static void Main(string[] args)
 3  {
 4      var lanes = Vector<int>.Count;
 5      int[] a = { 1, 2, 3, 4, 5, 6, 7, 8 };
 6      int[] b = { 1, 1, 1, 1, 1, 1, 1, 1 };
 7      int[] c = new int[a.Length];
 8      for( int i = 0; i < a.Length; i += lanes )
 9      {
10          var a8 = new Vector<int>(a, i);
11          var b8 = new Vector<int>(b, i);
12          (a8 + b8).CopyTo(c, i);
13  }}}}
```

The first line in function `Main` determines the number of lanes supported by your hardware. C# doesn't let you pick SSE or AVX; instead things are designed to be hardware independent. Lanes could either be 4 or 8, and on future hardware perhaps 16. If we put a breakpoint on line 5, lanes turns out to be 4, even on a very recent CPU. To resolve this, go to Project -> Properties, select the Build tab, and disable 'Prefer 32-bit'. Now lanes will be 8 on an AVX2-capable processor.

With eight lanes, `Vector<int>` is an *octint*. In the loop, octint a8 is filled with 8 values from array a, and b8 is filled with 8 values from array b. They are added together, and the result is copied to array c, which holds the final result.

Question is: did the C# compiler actually produce SIMD code? To determine this, we need to dig deep. Behold the generated x86 assembler for the loop that does the actual adding:

```
    for( int i = 0; i < a.Length; i += lanes )
00007FFBEC7B0E0F  xor      r15d,r15d
00007FFBEC7B0E12  test     esi,esi
00007FFBEC7B0E14  jle      00007FFBEC7B0E70
    {
        var a8 = new Vector<int>(a, i);
00007FFBEC7B0E16  cmp      r15d,esi
00007FFBEC7B0E19  jae      00007FFBEC7B0E90
00007FFBEC7B0E1B  lea      r12d,[r15+7]
00007FFBEC7B0E1F  cmp      r12d,esi
00007FFBEC7B0E22  jae      00007FFBEC7B0E90
00007FFBEC7B0E24  vmovupd  ymm0,[rdi+r15*4+10h]
        var b8 = new Vector<int>(b, i);
00007FFBEC7B0E2B  cmp      r15d,ebx
00007FFBEC7B0E2E  jae      00007FFBEC7B0E90
00007FFBEC7B0E30  cmp      r12d,ebx
00007FFBEC7B0E33  jae      00007FFBEC7B0E90
00007FFBEC7B0E35  vmovupd  ymm1,[rbp+r15*4+10h]
        var c8 = a8 + b8;
00007FFBEC7B0E3C  vpaddd   ymm6,ymm0,ymm1
        c8.CopyTo(c, i);
00007FFBEC7B0E52  mov      eax,dword ptr [r14+8]
00007FFBEC7B0E56  cmp      r15d,eax
00007FFBEC7B0E59  jae      00007FFBEC7B0E95
00007FFBEC7B0E5B  cmp      r12d,eax
00007FFBEC7B0E5E  jae      00007FFBEC7B0E9A
00007FFBEC7B0E60  vmovupd  [r14+r15*4+10h],ymm6
```

If we look closely at the generated assembler, we see that it indeed matches the C# code:

- Register r15d represents variable i, which is reset to 0 by xor'ing it with itself;
- Register esi contains the number of loop iterations, i.e. a.Length;
- 'test' does a bitwise AND; it verifies that esi is not zero;
- 'jle' jumps out of the loop if the number of loop iterations is in fact zero.

Skipping to the line that contains the generated assembler for 'c8 = a8 + b8', we see something interesting: there is a *single instruction*, namely vpaddd. Intel's Intrinsics Guide tells us this corresponds to the _mm256_add_epi32(a, b) instruction, which returns an __m256i. Similarly, the assembler line just above it uses vmovupd, which corresponds to _mm256_loadu_pd, which loads *unaligned* data to an octint or octfloat. More on alignment later on in this document.

There's something else to note: before reading unaligned data, each array access does two comparisons and conditional jumps (cmp / jae). The target address is interesting: it contains a call to JIT_RngChkFail. In other words: *every* array access in C# checks that we do not read before the start or beyond the end of the array, even if this is clearly not the case, like here. This safety comes at a price, and is one of the reasons C# is slower (but also more secure) than C++.

A Practical Example: C++

The following code renders a Mandelbrot fractal:

```
// based on https://www.shadertoy.com/view/Mdy3Dw
float scale = 1 + cosf( t );
t += 0.01f;
for( int y = 0; y < SCRHEIGHT; y++ )
{
   float yoffs = ((float)y / SCRHEIGHT - 0.5f) * scale;
   float xoffs = -0.5f * scale, dx = scale / SCRWIDTH;
   for( int x = 0; x < SCRWIDTH; x++, xoffs += dx )
   {
      float ox = 0, oy = 0, py;
      for( int i = 0; i < 99; i++ ) px = ox, py = oy,
         oy = -(py * py - px * px - 0.55f + xoffs),
         ox = -(px * py + py * px - 0.55f + yoffs);
      int r = min( 255, max( 0, (int)(ox * 255) ) );
      int g = min( 255, max( 0, (int)(oy * 255) ) );
      screen->Plot( x, y, (r << 16) + (g << 8) );
```

```
    }
}
```

Note that this code is quite well-optimized, and very compute intensive. We could easily run this code on multiple cores: there are no dependencies between pixels, so this algorithm is *embarrassingly parallel*. We should use multiple cores, but for optimal performance, we also need to use instruction level parallelism. That means that each scalar operation should be executed for four input elements. The heavy-lifting happens in the inner loop, so if we just optimize that we should see a decent speedup. Let's consider our options: there is a loop dependency in the inner loop, so we can't run iterations in parallel. We could however process four pixels in parallel.

We will now translate the existing scalar code step by step to vectorized code. I will use SSE, but with minor modifications the same process applies to AVX.

STEP 1: make a backup of the original code

The best way to do this is using an `#if 1 … #else … #endif` block. This way the original code is within reach, should anything go wrong, or just for reference.

STEP 2: create four streams

We start out by simulating the use of four streams. Working on four pixels at a time means that x increases in steps of 4. Inside this loop we will loop over the lanes. Besides this, we need four copies of the ox and oy variables, as these will now be calculated for four pixels in parallel.

```
for( int x = 0; x < SCRWIDTH; x += 4, xoffs += dx * 4 )
{
    float ox[4] = { 0, 0, 0, 0 }, oy[4] = { 0, 0, 0, 0 };
    for( int lane = 0; lane < 4; lane++ )
```

The contents of the inner loop barely change: we do the same work, but instead of operating on ox and oy, we now operate on array elements:

```
for( int i = 0; i < 99; i++ ) px = ox[lane], py = oy[lane],
    oy[lane] = -(py * py - px * px - 0.55f + xoffs + lane * dx),
    ox[lane] = -(px * py + py * px - 0.55f + yoffs);
```

And finally, we need to plot the four pixels. Let's do this in a separate loop, so we can either not convert that loop to SIMD, or do the conversion separately:

```
for( int lane = 0; lane < 4; lane++ )
{
    int r = min( 255, max( 0, (int)(ox[lane] * 255) ) );
    int g = min( 255, max( 0, (int)(oy[lane] * 255) ) );
    screen->Plot( x + lane, y, (r << 16) + (g << 8) );
}
```

STEP 3: create the SIMD data structure

This is a simple step: we already have data for the four lanes in `ox[4]` and `oy[4]`, which means we have two sets of four floats, which is precisely what is stored in a quadfloat.

```
union { __m128 ox4; float ox[4]; };
union { __m128 oy4; float oy[4]; };
ox4 = oy4 = _mm_setzero_ps();
```

The last line uses an intrinsic to set the 128-bit vectors to zero.

STEP 4: check functionality

We are making some quite invasive changes to our code, so make sure regularly that everything still works as intended!

STEP 5: convert the inner loop

Since the stream conversion has already been prepared, the final conversion is straightforward:

```
for( int i = 0; i < 99; i++ ) px4 = ox4, py4 = oy4,
    oy4 = -(py4 * py4 – px4 * px4 - 0.55f + xoffs4),
    ox4 = -(px4 * py4 + py4 * px4 - 0.55f + yoffs4);
```

This code does not work, but it does give us a clear idea of where we want to go. The loop over the streams disappeared, because we do these in parallel now. The use of `ox[lane]` and `oy[lane]` is replaced by ox4 and oy4. Variables px4 and py4 now also should be quadfloats. Some problems remain:

- one does not simply multiply two quadfloats using the * operator;
- the contents of `xoffs4` are a bit complex.

Regarding `xoffs4`: variable `xoffs` used to be incremented by dx at every iteration. So, what we are looking for is an array of four floats, containing `{ xoffs, xoffs + dx, xoffs + 2 * dx, xoffs + 3 * dx }`:

```
__m128 xoffs4 = _mm_set_ps( xoffs, xoffs + dx, xoffs + dx * 2, xoffs + dx * 3 );
```

Variable `yoffs4` contains the same value for each of the four pixels:

```
__m128 yoffs4 = _mm_set_ps( yoffs, yoffs, yoffs, yoffs );
```

That leaves us with the operators. We need to replace every multiplication by `_mm_mul_ps`, every subtraction by `_mm_sub_ps`, and so on. Let's do this for oy4:

```
oy4 = -(py4 * py4 - px4 * px4 - 0.55f + xoffs4);
```

becomes:

```
oy4 =
_mm_sub_ps(
    _mm_setzero_ps(),
    _mm_add_ps(
       _mm_sub_ps(
          _mm_sub_ps(
             _mm_mul_ps( py4, py4 ),
             _mm_mul_ps( px4, px4 )
          ),
          _mm_set1_ps( 0.55f )
    ),
    xoffs4 )
);
```

Bringing everything together, we get the final vectorized program:

```
for( int y = 0; y < SCRHEIGHT; y++ )
{
    float yoffs = ((float)y / SCRHEIGHT - 0.5f) * scale;
    float xoffs = -0.5f * scale, dx = scale / SCRWIDTH;
    for( int x = 0; x < SCRWIDTH; x += 4, xoffs += dx * 4 )
    {
        union { __m128 ox4; float ox[4]; };
        union { __m128 oy4; float oy[4]; };
```

```
        ox4 = oy4 = _mm_setzero_ps();
        __m128 xoffs4 = _mm_setr_ps( xoffs, xoffs + dx,
                        xoffs + dx * 2, xoffs + dx * 3 );
        __m128 yoffs4 = _mm_set_ps1( yoffs );
        for( int i = 0; i < 99; i++ )
        {
            __m128 px4 = ox4, py4 = oy4;
            oy4 = _mm_sub_ps( _mm_setzero_ps(), _mm_add_ps( _mm_sub_ps(
                  _mm_sub_ps( _mm_mul_ps( py4, py4 ), _mm_mul_ps( px4, px4 ) ),
                  _mm_set_ps1( 0.55f ) ), xoffs4 ) );
            ox4 = _mm_sub_ps( _mm_setzero_ps(), _mm_add_ps( _mm_sub_ps(
                  _mm_add_ps( _mm_mul_ps( px4, py4 ), _mm_mul_ps( py4, px4 ) ),
                  _mm_set_ps1( 0.55f ) ), yoffs4 ) );
        }
        for( int lane = 0; lane < 4; lane++ )
        {
            int r = min( 255, max( 0, (int)(ox[lane] * 255) ) );
            int g = min( 255, max( 0, (int)(oy[lane] * 255) ) );
            screen->Plot( x + lane, y, (r << 16) + (g << 8) );
        }
    }
}
```

This code runs, as promised, almost four times faster than the original. Here is the assembler generated by the compiler for the C++ code:

```
0040121D  mov         eax,63h
            for( int i = 0; i < 99; i++ )
            {
                __m128 px4 = ox4, py4 = oy4;
00401222  movaps      xmm2,xmm3
                oy4 = _mm_sub_ps( _mm_setzero_ps(), _mm_add_ps( _mm_sub_ps( _mm_sub_ps( _mm_mul_ps( py4, py4 ),
00401225  movaps      xmm0,xmm4
                ox4 = _mm_sub_ps( _mm_setzero_ps(), _mm_add_ps( _mm_sub_ps( _mm_add_ps( _mm_mul_ps( px4, py4 ),
00401228  mulps       xmm2,xmm4
0040122B  movaps      xmm1,xmm3
0040122E  mulps       xmm1,xmm3
00401231  xorps       xmm3,xmm3
00401234  mulps       xmm0,xmm4
00401237  xorps       xmm4,xmm4
0040123A  addps       xmm2,xmm2
0040123D  subps       xmm1,xmm0
00401240  subps       xmm2,xmm7
00401243  subps       xmm1,xmm7
00401246  addps       xmm2,xmmword ptr [yoffs4]
0040124A  addps       xmm1,xmm5
0040124D  subps       xmm4,xmm2
00401250  subps       xmm3,xmm1
00401253  movaps      xmmword ptr [ebp-50h],xmm4
00401257  movaps      xmmword ptr [ebp-60h],xmm3
0040125B  sub         eax,1
0040125E  jne         Tmpl8::Game::Tick+132h (0401222h)
            }
```

On the first line, the loop counter is initialized (63 hexadecimal equals 99 decimal). After that, we only see SIMD instructions: xmm3 contains ox4, and oy4 is in xmm4. These are copied to px and py (xmm2 and xmm0). After that we see our sequence of SIMD instructions: mulps, addps, subps and so on. The _mm_setzero_ps is implemented using the xorps function: xor'ing a register with itself yields zero, and a float that has all its bits set to zero happens to be 0.0. On the last lines the loop counter (eax) is decreased by 1. If this does not yield zero, a jump is made to address 00401222, which is the first line of the loop.

This inspection of the generated assembler reveals a problem just before the end of the loop: the union forced the compiler to store xmm4 and xmm3 (ox4 and oy4) to memory. The compiler could have done this after the loop, but it missed this opportunity. We can help the compiler by removing the union. We do however still need to access ox[4] and oy[4] in the second loop, so after the first loop, we copy ox4 and oy4 to a second pair of quadfloats, which we union with ox[4] and oy[4]:

```
union { __m128 tmp1; float ox[4]; }; tmp1 = ox4;
union { __m128 tmp2; float oy[4]; }; tmp2 = oy4;
```

This shaves off two instructions in the critical inner loop. The effect on final performance is minimal: writing to the same memory address 99 times in rapid succession is pretty much guaranteed to be a level-1 cache operation, which is very fast.

A Practical Example: C#

For completeness we will now also convert the C# version of this code to use SIMD. Here is a direct port of the C++ code:

```
public void Tick()
{
    // based on https://www.shadertoy.com/view/Mdy3Dw
    float scale = 1 + (float)Math.Cos( t );
    t += 0.01f;
    for( int y = 0; y < screen.height; y++ )
    {
        float yoffs = ((float)y / screen.height - 0.5f) * scale;
        float xoffs = -0.5f * scale, dx = scale / screen.width;
        for( int x = 0; x < screen.width; x++, xoffs += dx )
        {
            float ox = 0, oy = 0, px, py;
            for( int i = 0; i < 99; i++ )
            {
                px = ox; py = oy;
                oy = -(py * py - px * px - 0.55f + xoffs);
                ox = -(px * py + py * px - 0.55f + yoffs);
            }
            int r = Math.Min( 255, Math.Max( 0, (int)(ox * 255) ) );
            int g = Math.Min( 255, Math.Max( 0, (int)(oy * 255) ) );
            screen.pixels[x + y * screen.width] = (r << 16) + (g << 8);
        }
    }
}
```

We start the conversion by creating the streams. The number of lanes is determined by the hardware that runs the code, so we can not just assume this number is four.

```
int lanes = Vector<float>.Count;
for( int x = 0; x < screen.width; x += lanes, xoffs += dx * lanes )
{
    for( int lane = 0; lane < lanes; lane++ )
    {
        ox[lane] = 0;
        oy[lane] = 0;
        for( int i = 0; i < 99; i++ )
```

```
        {
            float px = ox[lane], py = oy[lane];
            oy[lane] = -(py * py - px * px - 0.55f + xoffs + dx * lane);
            ox[lane] = -(px * py + py * px - 0.55f + yoffs);
        }
    }
```

Next, we create the SIMD data structure. The lanes will reside in `Vector<float>` variables, which we create outside the x-loop for efficiency:

```
var ox8 = new Vector<float>();
var oy8 = new Vector<float>();
var px8 = new Vector<float>();
var py8 = new Vector<float>();
var C8 = new Vector<float>( 0.55f );
var yoffs8 = new Vector<float>( yoffs );
```

Variable C8 contains the constant used in the calculation. Passing only a single float parameter to the `Vector<float>` constructor fills all lanes with this value. We do the same to obtain `yoffs8`.

Now we can convert the inner loop. This is actually much easier than in C++:

```
ox8 = oy8 = Vector<float>.Zero;
for( int lane = 0; lane < lanes; lane++ ) tmp[lane] = xoffs + lane * dx;
var xoffs8 = new Vector<float>( tmp, 0 );
for( int i = 0; i < 99; i++ )
{
    px8 = ox8; py8 = oy8;
    oy8 = -(py8 * py8 - px8 * px8 - C8 + xoffs8);
    ox8 = -(px8 * py8 + py8 * px8 - C8 + yoffs8);
}
```

C# lets us use regular operators, which is in fact somewhat confusing: every multiplication here is actually a vector multiplication. Also notice how `xoffs8` is filled with correct values: this needs to happen via a temporary float array; there is no way to directly write to individual lanes in the vector.

Finally, we need to copy the vector data back to regular float arrays for the pixel plotting loop:

```
ox8.CopyTo( ox );
oy8.CopyTo( oy );
```

After this, functionality is restored, and the code now runs a lot faster. Here is the final source code:

```
for( int y = 0; y < screen.height; y++ )
{
    float yoffs = ((float)y / screen.height - 0.5f) * scale;
    float xoffs = -0.5f * scale, dx = scale / screen.width;
    Vector<float> ox8 = new Vector<float>(), oy8 = new Vector<float>();
    Vector<float> px8 = new Vector<float>(), py8 = new Vector<float>();
    Vector<float> C8 = new Vector<float>( 0.55f );
    Vector<float> yoffs8 = new Vector<float>( yoffs );
    for( int x = 0; x < screen.width; x += lanes, xoffs += dx * lanes )
    {
        ox8 = oy8 = Vector<float>.Zero;
        for( int lane = 0; lane < lanes; lane++ ) tmp[lane] = xoffs + lane * dx;
        var xoffs8 = new Vector<float>( tmp, 0 );
        for( int i = 0; i < 99; i++ )
        {
```

```
            px8 = ox8; py8 = oy8;
            oy8 = -(py8 * py8 - px8 * px8 - C8 + xoffs8);
            ox8 = -(px8 * py8 + py8 * px8 - C8 + yoffs8);
        }
        ox8.CopyTo( ox );
        oy8.CopyTo( oy );
        for( int lane = 0; lane < lanes; lane++ )
        {
            int r = Math.Min( 255, Math.Max( 0, (int)(ox[lane] * 255) ) );
            int g = Math.Min( 255, Math.Max( 0, (int)(oy[lane] * 255) ) );
            screen.pixels[x + lane + y * screen.width] = (r << 16) + (g << 8);
        }
    }
}
```

For completeness, here's the assembler output of the C# compiler for the line that updates ox8:

```
                        ox8 = -(px8 * py8 + py8 * px8 - C8 + yoffs8);
00007FFE7726C2D0  vmulps      ymm0,ymm13,ymm12
00007FFE7726C2D5  vmulps      ymm12,ymm12,ymm13
00007FFE7726C2DA  vaddps      ymm0,ymm0,ymm12
00007FFE7726C2DF  vsubps      ymm0,ymm0,ymm10
00007FFE7726C2E4  vmovupd     ymm11,ymmword ptr [rsp+30h]
00007FFE7726C2EB  vaddps      ymm0,ymm0,ymm11
00007FFE7726C2F0  vxorps      ymm1,ymm1,ymm1
00007FFE7726C2F5  vsubps      ymm13,ymm1,ymm0
```

As you can see, C# produced optimal SIMD code for this part of the main loop. This is confirmed by looking at raw performance figures: without SIMD, the code runs in 182 milliseconds per frame on my machine (Intel i3-7100 @ 3.9Ghz), versus 175 milliseconds for the C++ version. With SIMD, the frame time drops to 29 milliseconds, which is ~6.3 times faster.

The benefits of C# over C++ are quite obvious in this case: the vectorized code is much closer to the original code, and, more importantly, the code is the same for SSE and AVX, as well as possible future hardware capabilities. On the other hand: hiding hardware details as is done here makes it hard to write optimal vectorized code, unless we know precisely what we are doing. In this case this is not a problem: we just converted C++ code, and it turns out that we can use the same approach in C#. Without the fresh experience of converting similar C++ code, this would have been much harder.

Conditional Code & SIMD

Code vectorization is the process of converting existing code to independent scalar flows which can be executed in parallel, where each task executes the same instructions. This way, four or eight (or more) scalar flows can be executed concurrently, using 'single instruction multiple data' instructions.

The code we have vectorized so far was relatively simple: all pixels of the image can be calculated independently, in any order, as well as in parallel, and for each pixel, we execute exactly the same instructions. But what if things are not so simple? The most common complication is *conditional code*: anything involving an *if* statement, conditional expressions such as `a=b>a?a:b`, but also loops with a variable number of iterations, switch statements and so on. Clearly anything that is conditional may lead to scalar flows *not* executing the same code.

Consider the second loop in our vectorized Mandelbrot example:

```
for( int lane = 0; lane < 4; lane++ )
{
    int r = min( 255, max( 0, (int)(ox[lane] * 255) ) );
    int g = min( 255, max( 0, (int)(oy[lane] * 255) ) );
    screen->Plot( x + lane, y, (r << 16) + (g << 8) );
}
```

The min and max functions used here hide some conditional code. Min could be implemented as:

```
int min( a, b ) { if (a < b) return a; else return b; }
```

Or using a conditional expression:

```
#define min(a,b) ((a)<(b)?(a):(b));
```

For the specific case of min and max, SSE and AVX offer an efficient solution:

```
__m128 c4 = _mm_min_ps( a4, b4 );
__m128 c4 = _mm_max_ps( a4, b4 );
```

The existence of these instructions sometimes causes SSE code to *exceed* the expected optimal 400% efficiency: conditional code can cause delays on a CPU, but in SSE and AVX, min and max are not conditional at all.

We can now vectorize part of the pixel plotting loop:

```
__m128 C4 = _mm_set_ps1( 255.0f );
ox4 = _mm_min_ps( C4, _mm_max_ps( _mm_setzero_ps(), _mm_mul_ps( ox4, C4 ) ) );
oy4 = _mm_min_ps( C4, _mm_max_ps( _mm_setzero_ps(), _mm_mul_ps( oy4, C4 ) ) );
for( int lane = 0; lane < 4; lane++ )
{
    int r = (int)ox[lane];
    int g = (int)oy[lane];
    screen->Plot( x + lane, y, (r << 16) + (g << 8) );
}
```

Note that the constant `255.0f` is stored in a variable, so that we don't have to execute the `_mm_set1_ps` instruction four times, but just once.

We can in fact go a step further: the conversion from float to int can also be done using an SSE instruction:

```
union { __m128i tmp1; int oxi[4]; }; tmp1 = _mm_cvtps_epi32( ox4 );
union { __m128i tmp2; int oyi[4]; }; tmp2 = _mm_cvtps_epi32( oy4 );
```

Note that the union now combines a quadint and an integer array.

In C# we have similar functionality. The `Vector<>` classes have Min and Max methods, which compile to the expected minps and maxps assembler instructions.

There is now a single line left in the second loop, which plots the pixel. Plot is a method of the surface class, and it is implemented as follows:

```
void Surface::Plot( int x, int y, Pixel c )
{
    if ((x >= 0) && (y >= 0) && (x < m_Width) && (y < m_Height))
        m_Buffer[x + y * m_Pitch] = c;
}
```

Here, 'Pixel' is simply a 32-bit unsigned int, and `m_Width` and `m_Height` are the width and height of the surface. The if-statement prevents pixels from being plotted off-screen. In the Mandelbrot application, this never happens, but obviously other applications might need this functionality.

An SSE version of `Surface::Plot` might look like this:

```
void Surface::Plot4( __m128i x4, __m128i y4, __m128i c4 )
{
    if ((x4 >= 0) && (y4 >= 0) && (x4 < m_Width) && (y4 < m_Height))
        ...
}
```

This time we have a problem. SSE and AVX do not have instructions equivalent to if statements, and for a good reason: the boolean expression that we see in the scalar code would become a 'quadbool' expression, and the conditional code (storing something in the pixel buffer) may have to be executed for none, some, or all of the lanes.

I just wrote SSE and AVX do not have *if* statements, but they do in fact have comparison instructions. These do not yield 'quadbools', but they do return something useful: *bitmasks*. Here is an example:

```
__m128 mask = _mm_cmpge_ps( x4, _mm_setzero_ps() ); // if (x4 >= 0)
```

This line takes x4 and a quadfloat containing zeroes, and checks if the first operand is greater or equal than the second operand. Similar comparison instructions exist for greater (_mm_cmpgt_ps), less (_mm_cmplt_ps), less or equal (_mm_cmple_ps), equal (_mm_cmpeq_ps) and not equal (_mm_cmpne_ps).

The mask value is a 128-bit value. After the comparison, its contents reflect the result: 32 zeroes for a 'false', and 32 ones for a 'true'.

We can also combine comparisons:

```
__m128 mask1 = _mm_cmpge_ps( x4, _mm_setzero_ps() ); // if (x4 >= 0)
__m128 mask2 = _mm_cmpge_ps( y4, _mm_setzero_ps() ); // if (y4 >= 0)
__m128 mask = _mm_and_ps( mask1, mask2 ); // if (x4 >= 0 && y4 >= 0)
```

None of this is actually conditional: we unconditionally calculate bitmasks. The resulting bitmasks can be used in two distinctive ways. The first way is to break the vector instruction flow, and switch to scalar code to handle the result of the comparisons. For this, we use the _mm_movemask_ps instruction. This instruction takes a mask, and returns a 4-bit value, where each bit is set to 1 if the 32 bits for a lane are 1, and 0 otherwise. Now we can test the bits individually:

```
int result = _mm_movemask_ps( mask );
if (result & 1) { … } // result for first lane is true
if (result & 2) { … } // result for second lane is true
if (result & 4) { … } // result for third lane is true
if (result & 8) { … } // result for fourth lane is true
```

The benefit is that we now at least did the comparisons using vector code. We didn't solve the actual problem though: the conditional code still breaks our vector flow.

To resolve that, we need to use the masks differently: *to disable functionality for lanes*.

Consider the actual conditional code:

```
m_Buffer[x + y * m_Pitch] = c;
```

This line writes an unsigned int to an address in the screen buffer. Now, what if we replaced this address by some other safe location, for instance the address of a dummy variable? We would still perform the write, but this time it does not yield a visible pixel.

Let's consider an even more pragmatic solution: if a pixel happens to be off-screen, we write it to location (0,0). Of course, this pixel will contain nonsense, as it will be overwritten by all off-screen pixels, but for the sake of this example, we will consider that acceptable. To implement this, we replace the pixel address calculation `x+y*m_Pitch` by `(x+y*m_Pitch)*0`. Whatever the value of x, y and `m_Pitch` are, the result of this equation will be 0. And this sort of operation is precisely what the masks are designed for.

Let's compute the full mask for the plot statement:

```
__m128 mask1 = _mm_cmpge_ps( x4, _mm_setzero_ps() );
__m128 mask2 = _mm_cmpge_ps( y4, _mm_setzero_ps() );
__m128 mask3 = _mm_cmplt_ps( x4, _mm_set_ps1( m_Width ) );
__m128 mask4 = _mm_cmplt_ps( y4, _mm_set_ps1( m_Height ) );
__m128 mask = _mm_and_ps( _mm_and_ps( _mm_and_ps( mask1, mask2 ), mask3 ), mask4 );
```

We can calculate the four pixel addresses as follows:

```
__m128i address4 = _mm_add_epi32( _mm_mullo_epi32( y4, m_Pitch4 ), x4 );
address4 = _mm_and_si128( address, *(__m128i*)&mask ) );
```

A few remarks about these lines:

- Multiplying two 32-bit integers yields a 64-bit integer, which doesn't fit in a 32-bit lane. The `_mm_mullo_epi32` instruction discards the top 32-bit, which is fine in this case.
- There is no `_mm_and_epi32` instruction; instead doing a bitwise and to integers operates directly on the 128 bits using `_mm_and_si128`.
- Our mask is a quadfloat, while `_mm_and_si128` expects a quadint mask. We thus convert it on-the-fly to the correct type.
- The second line uses the calculated mask to reset all off-screen pixel addresses to 0, as we planned to do.

Now there is one thing left to do: plotting the four pixels to the addresses stored in quadint `address4`. The write we want to do is known as a scatter: the four addresses may be right next to each other, but they may also be all over the screen. There are no SSE and AVX instructions that support this, so our only option is to use four 32-bit writes to do this. Although this breaks our vector flow, none of this is conditional.

The final `Plot4` method:

```
void Surface::Plot4( __m128 x4, __m128 y4, __m128i c4 )
{
    __m128 mask1 = _mm_cmpge_ps( x4, _mm_setzero_ps() );
    __m128 mask2 = _mm_cmpge_ps( y4, _mm_setzero_ps() );
    __m128 mask3 = _mm_cmplt_ps( x4, _mm_set_ps1( (float)m_Width ) );
    __m128 mask4 = _mm_cmplt_ps( y4, _mm_set_ps1( (float)m_Height ) );
    __m128 mask = _mm_and_ps( _mm_and_ps( _mm_and_ps( mask1, mask2 ), mask3 ), mask4 );
    union { __m128i address4; int address[4]; };
    __m128i m_Pitch4 = _mm_set1_epi32( m_Pitch );
    __m128i x4i = _mm_cvtps_epi32( x4 ), y4i = _mm_cvtps_epi32( y4 );
    address4 = _mm_add_epi32( _mm_mullo_epi32( y4i, m_Pitch4 ), x4i );
    for( int i = 0; i < 4; i++ ) m_Buffer[address[i]] = c4.m128i_i32[i];
}
```

Note that the function now takes quadfloats for x4 and y4; this is because the set of SSE instructions for quadints is more limited than for quadfloats. In particular, _mm_cmpge_epi32 is missing. This functionality could have been emulated, but this would make the code less clear.

Fun with Masks

In the previous section we used 128-bit masks to nullify calculations. We did this by using an integer 'and' using _mm_and_si128. We applied it to a quadint variable containing addresses (actually: offsets from the start of the screen buffer), but the same trick works for floats. For this, we 'abuse' an interesting property of the floating point number 0.0f: it's binary representation is 32 zeroes. That means that if we 'and' a floating point number with 32 zeroes, we reset all its bits, which makes the floating point value 0.0f. 'And'ing with 32 ones does nothing: we simply keep the original floating point number. An example:

```
__m128 mask = ...; // some comparison
a4 = _mm_and_ps( a4, mask );
```

The second line sets the lanes of quadfloat a4 to 0.0f if the corresponding lane in the mask is 'false'.

It may happen that we want to put something else than zero in some lanes, depending on a condition. Consider the following conditional expression:

```
float a = b == 0 ? b : c;
```

…which replaces a by b if its value is zero, and by c otherwise. One way to do this is again with masks:

```
__m128 mask = _mm_cmpeq_ps( a4, _mm_setzero_ps() );
__m128 part1 = _mm_and_ps( mask, b4 );
__m128 part2 = _mm_andnot_ps( mask, c4 );
a4 = _mm_or_ps( part1, part2 );
```

Here, part1 will contain zeroes for each lane where mask is false, and values from b4 where mask is true. Quadfloat part2 uses the inverted mask, and selects from c4. Note that part1 and part2 have no overlap: if a lane is non-zero in part1, it will be zero in part2, and vice versa. The two parts can thus be safely blended to obtain the final result.

A more direct way to obtain this result is to use the _mm_blendv_ps instruction:

```
__m128 mask = _mm_cmpeq_ps( a4, _mm_setzero_ps() );
a4 = _mm_blendv_ps( b4, c4, mask );
```

The _mm_blendv_ps intrinsic selects values from b4 and c4 depending on mask: if a value in mask is set to true, the value in c4 will be selected, otherwise the value in b4 will be selected.

Again, the same functionality is available in C#: the method to use is ConditionalSelect. The comparison instructions are also conceptually the same: e.g., method Equals produces a vector of 'true' and 'false' values depending on the equality of the components of the vector operands. See https://msdn.microsoft.com/en-us/library/system.numerics.vector(v=vs.111).aspx for full details.

Optimizing and Debugging SIMD Code, Hints

In the previous sections we have seen how code can be vectorized, and how to deal with conditional code. In this section we will discuss some common opportunities for improving the efficiency of SIMD code.

Instruction count In principle every intrinsic compiles to a single compiler instruction. That means that shorter source code results in a smaller program, which will most of the time run faster. Sometimes advanced instructions such as `_mm_blendv_ps` can replace a sequence of simpler instructions. It can thus be beneficial to familiarize yourself with the available instructions.

Floating point versus integer Floating point support in SSE and AVX is much better than integer support. Sometimes a temporary conversion to floats can make your code more efficient, even if it means that you need to convert back later. Floating point arithmetic will definitely make your life easier: many integer intrinsics are quite obscure (see e.g. `_mm_mullo_epi32`).

Reduce the use of _mm_set_ps You will frequently need constants in your vectorized code, as we have seen in the Mandelbrot example. It may be tempting to create quadfloats on the spot for these. However, `_mm_set_ps` is an expensive function, as it takes four operands. Consider caching the result: calculate the quadfloat outside loops, so you can use it many times without penalty inside the loop. Similarly, if you need to expand scalars to quadfloats (like `m_Pitch` in the `Plot` method), consider caching the expanded version in the class.

Avoid gather operations An additional pitfall related to `_mm_set_ps` is that the data you feed it comes from locations scattered though memory. The fastest way to get data from memory to a quadfloat is when it is already stored in memory as a quadfloat, i.e. in 16 consecutive bytes.

Data alignment One thing to keep in mind is that a quadfloat in memory must *always* be stored at an address that is a multiple of 16. Failure to do so will result in a crash. This is why C# used a slow *unaligned read* for SSE/AVX data: C# cannot guarantee data alignment. In C++, variables created on the stack will automatically obey this rule. Variables allocated using new however may be unaligned, causing unexpected crashes. If you do experience a crash, check if the data that is being processed is properly aligned: the (hexadecimal) address should always end with a zero.

C++ debugger Support for SIMD is well-integrated in the Visual Studio debugger. You can e.g. effortlessly inspect individual values in SIMD variables.

AVX/AVX2 support If your processor happens to be the latest and greatest AMD and Intel have to offer, be aware that some code you produce may not work on your neighbour's laptop. In C# this is not really a problem: unavailable instructions will simply be emulated, which may be slower than not using vectorization in the first place, but at least everything will work. In C++, it is perfectly possible to produce an .exe that will not work if e.g. AVX2 is not available. Make sure you keep target hardware in mind, or provide an alternative implementation for older hardware. An example of this problem: an early crack of Metal Gear V required obscure SSE instructions not available on some AMD hardware, even though this hardware was perfectly capable of running the game itself.

Vectorize bottlenecks only In the Mandelbrot example, we vectorized the Plot method, even though it consumes only a tiny portion of time. Don't do this in a real-world situation: vectorization is *hard* and you want to focus your effort on bottlenecks only. In the Mandelbrot example, the massive loop that updates ox and oy is an excellent example: a lot of work focused in a tiny portion of code, screaming for close-to-the-metal optimization.

Evade fancy SIMD libraries Vectorization is hard, and it feels unnatural to write `_mm_mul_ps(a,b)` when you meant to write `a * b`. Resist the urge to write your own operators, at least until you are proficient in SSE; get used to the raw intrinsics. Anything more complex is bound to hide inefficiencies or even introduce them. Besides, some SIMD in your code makes it look like wizardry (which it is, in fact). If you must use something convenient, consider Agner Fog's vector library: http://www.agner.org/optimize/#vectorclass . Also read the rest of his site, the man is a guru of software optimization.

The End

This concludes the SIMD tutorial. This material is part of the Optimization & Vectorization course of the MGT master program at the Utrecht University in the Netherlands. More information at:

http://www.cs.uu.nl/docs/vakken/mov.

Utrecht University