Using the CPU to Improve Performance in 3D Applications

NCCA
Peter Smith
i9055645

August 2011

Abstract

Many applications in the film and game industries require multiple calculations to be performed on vast data sets. Any of these tools that are required to run in real-time, and be used interactively, must be developed with performance in mind. The following paper aims to explain how the Central Processing Unit can be utilised effectively in order to speed up an existing application. It explores programs written using both SSE intrinsic instructions and Intels new SPMD programming compiler, as well as conventional the conventional serial approach.

Contents

List of Figures

Chapter 1

The Need for Speed

1.1 Introduction

In the multimedia, educational and medical industries, applications are consistently developed that require 3D visualisations. These applications often require real-time frame rates. This is due to the demand for interactive feedback from video games, simulations and other computer graphics (CG) techniques. In recent years there has been a rise in programability of the Graphics Processing Unit (GPU). This has followed a vast increase in the speed of the GPU's floating point arithmetic capabilities and allows developers to store vertex and texture data into GPU memory to be processed more quickly. This can be done by use of various shading languages, [1] which are beginning to allow total customisation of the rasterisation process. In addition to this, a trend has emerged for General-Purpose computation on Graphics Processing Units (GPGPU). This allows more general computations and simulations to be performed on the GPU using geometry processing and kernels. This technique has seen many complex simulations and rendering methods, such as physically based shading, become possible at real-time rates.

Whilst the use of GPU speed has gained much popularity in computer graphics, the *Central Processing Unit* (CPU) is still the de-facto 'brain' of the machine and has also seen hardware improvements. In contrast to the GPU however, programability of the CPU is not very accessible for developers. There are features of CPU architecture that quite often are not taken advantage of in conventional software development. The opening section of this paper will analyse what makes up a CPU and how conventional applications use it. I will then look at how the CPU can be more fully utilised. The implementation of several tests and testing tools will then be explored to motivate discussion on how to measure the success of performance tuning. Finally, a simple mesh deformer will be used as a case study on implementing integrated speed-up techniques.

It should be noted that the aim of this paper is not to explore algorithmic inefficiencies or programming faults that lead to slow applications, but only to assess how

[1] GLSL, HLSL, CG etc.

hardware can be used effectively.

1.2 The CPU

The architecture of the CPU varies from chip to chip. For this reason, the model referred to in this paper will be a simplified abstraction, loosely analogous of modern Intel processors. In conventional computer architecture, the CPU is the workhorse of the machine. It consists of:

- Multi-Level Caches
- The Registers
- Arithmetic Logic Unit (ALU)
- Floating Point Unit (FPU)

1.2.1 Multi-Level Caches

The processor uses multi-level caches to very quickly access data and instructions. The reason for this is that it is static memory and is the first place that will be checked for data to be processed. The first cache to be checked is the level one (L1) cache. If the required data cannot be found here, the L2 cache is checked followed by the L3 cache and so on [2]. Application data is 'fetched' into the caches from the Random Access Memory (RAM) for use in operations. The reason for this is that RAM is a larger bank of dynamic memory and needs to be loaded into static memory for processing. The process of fetching from RAM is performed at a much lower clock speed than loading data from caches. Modern CPU's will compensate for this by intelligently fetching data before it is used in the application. Issues surrounding this process can greatly slow down applications, see section 1.4.2.

1.2.2 Registers

The registers of the modern CPU's are commonly 128 bits wide. These registers are a key part of the processor use to load data from the cache and submit this data to the ALU or FPU. They are the highest performance memory in the processor and are used for quick access and used in machine instructions. Figure 1.1 shows a diagram representing a CPU register:

For most graphical applications, 32 bit floating point data provides a sufficient precision-to-speed ratio; a 128 bit register can hold four of these values.

[2] modern processors usually have 2 or 3 caches

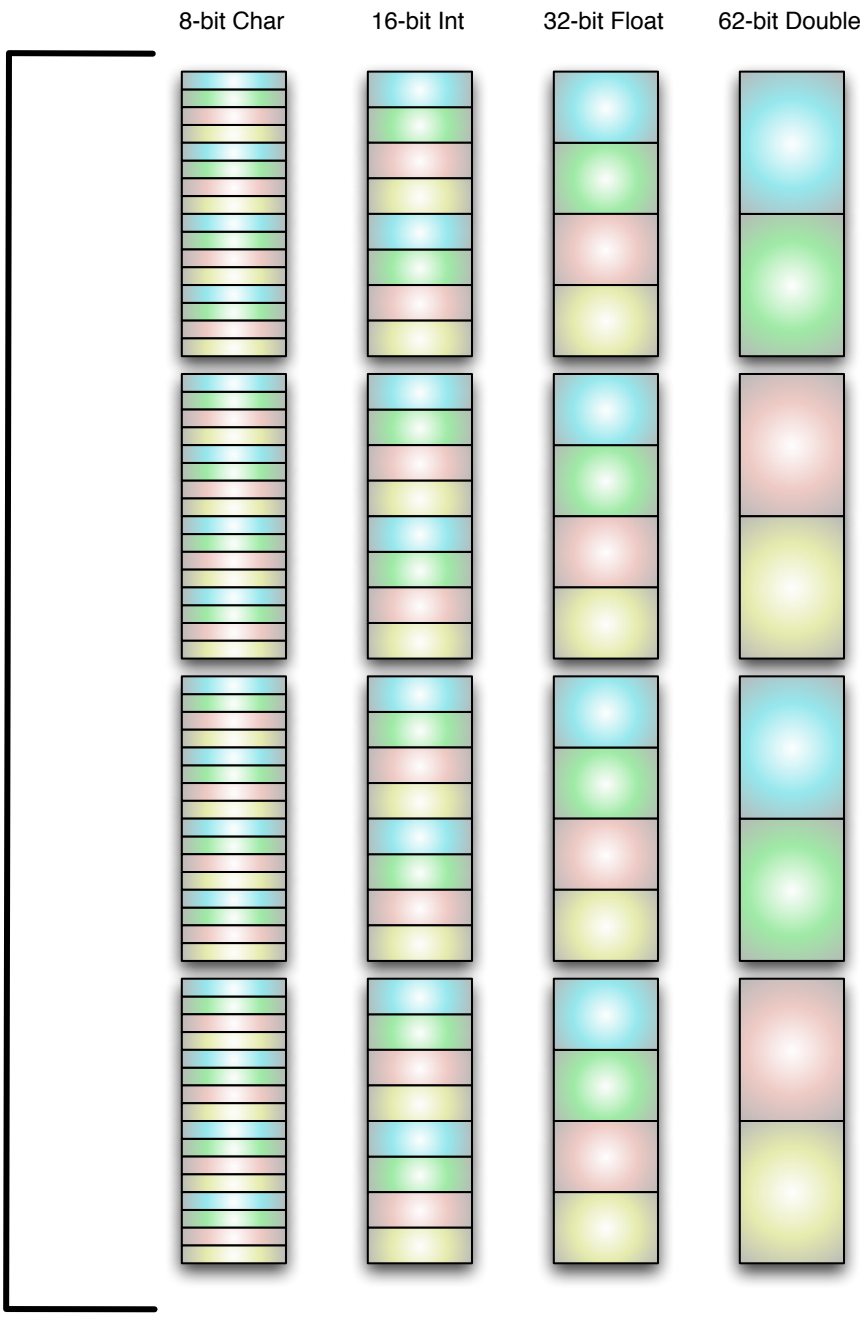


Figure 1.1: Typical Memory Layout of a 128 bit register

1.2.3 ALU and FPU

The Arithmetic Logic Unit (ALU) is in charge of arithmetic and logical operations. This refers to processes from addition and subtraction to 'exclusive or' and other control flow operations. The Floating Point Unit (FPU) is a more recent addition to modern machines and is designed specifically to handle operations on 32 bit floating point data. Depending on the type of operation requested, a different unit is used.

Figure 1.2 is a diagram showing an approximation of the pipeline of data through the CPU:

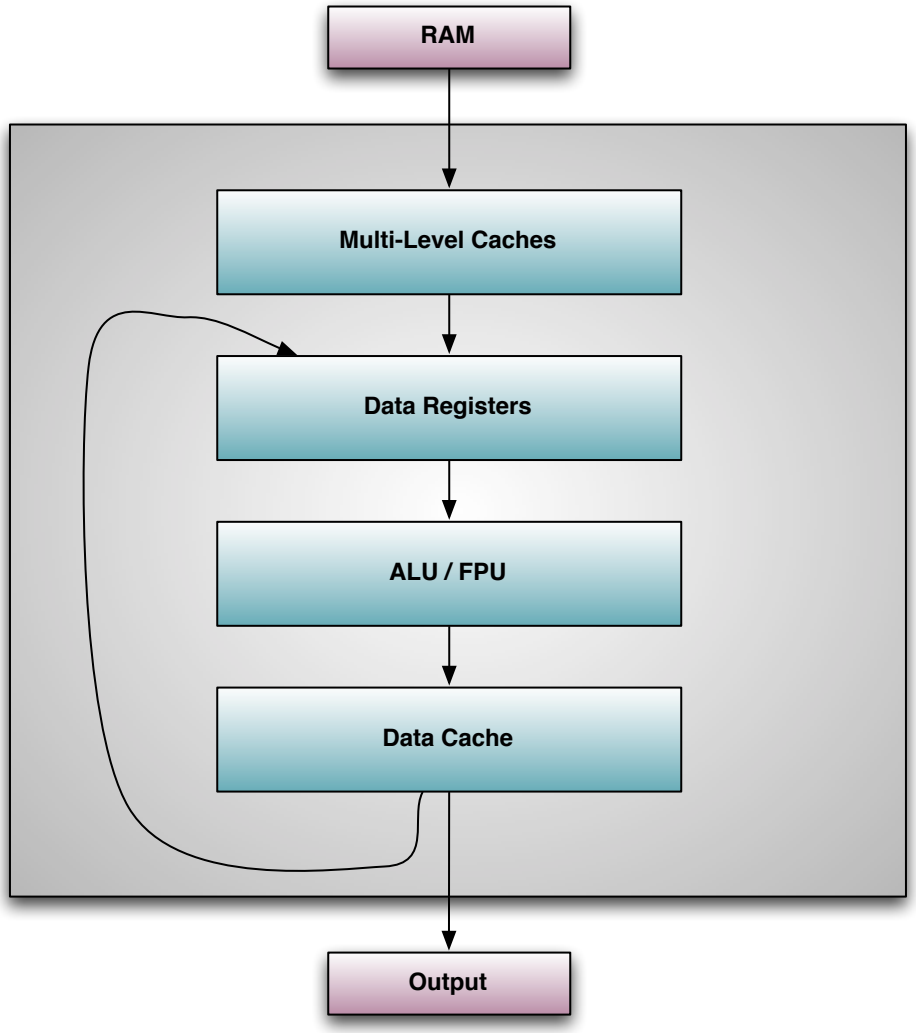


Figure 1.2: A high level representation of data flow through the CPU

The flow of data through the CPU begins in the caches, which take data from RAM, then moves into the registers before being submitted to the ALU or FPU for processing. After this it is held in cache memory to be output in some form.

1.3 Conventional Implementations

Despite the aforementioned features of modern CPU's, many applications do not utilise the hardware supplied by these processors. Quite often the 'Multiple Instruction Multiple Data' (MIMD) model is used for processing operations. The reason for this is that when using a compiler with no optimisations, coupled with developing in a 'naive' way, data is submitted for machine instructions one chunk at a time. For example, when

adding two floating point vectors (represented as XYZW), the conventional method is as follows:

```
result.x = v1.x + v2.x
result.y = v1.y + v2.y
result.z = v1.z + v2.z
result.w = v1.w + v2.w
```

In the worst case, this code will be compiled to execute as it is written; one instruction after the other. The CPU registers will load and execute one element at a time. More often than not, 3D applications have to perform hundreds of calculations on thousands, perhaps millions of these floating point vectors. Most significantly, they have to do this many times per second. Figure 1.3 symbolises the MIMD architecture.

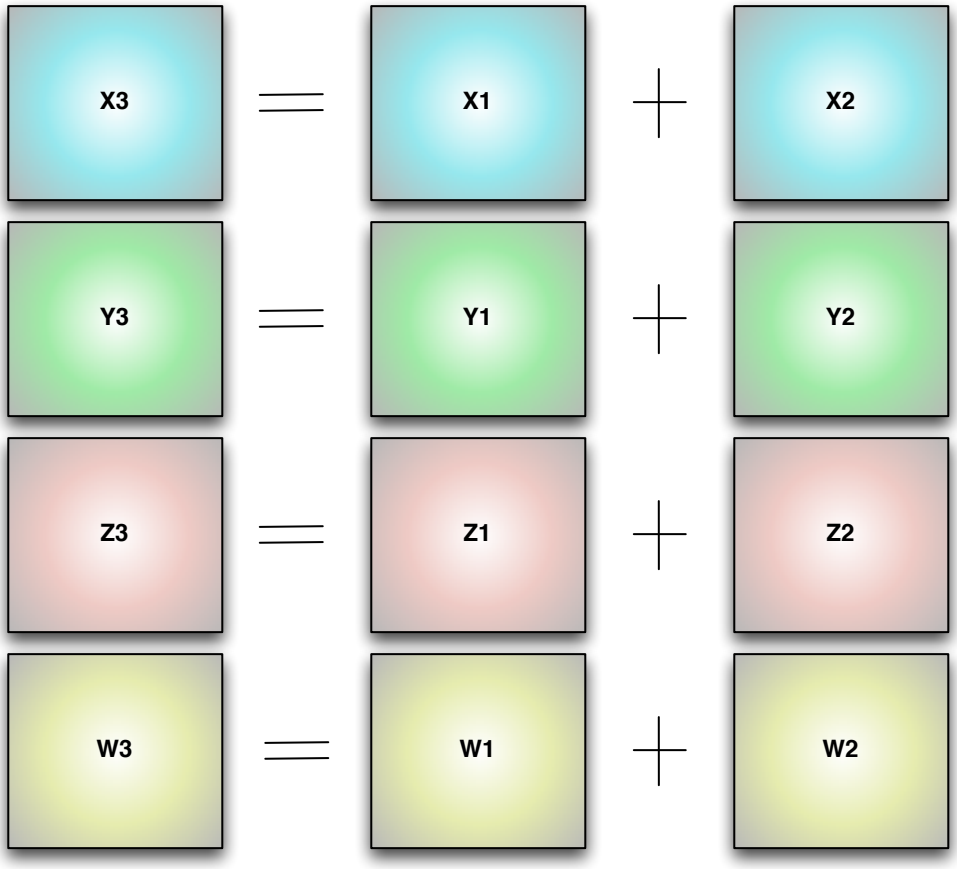


Figure 1.3: The use of registers in operations using the MIMD architecture

MIMD is one of four architectures in Flynn's taxonomy. Flynn, writing for the Institute of Electrical and Electronics Engineers (IEEE), recognised that taking advantage of certain types of processor architecture could lead to vastly better performance. Figure 1.4 shows a diagram of Flynns 4 classifications of architectures.

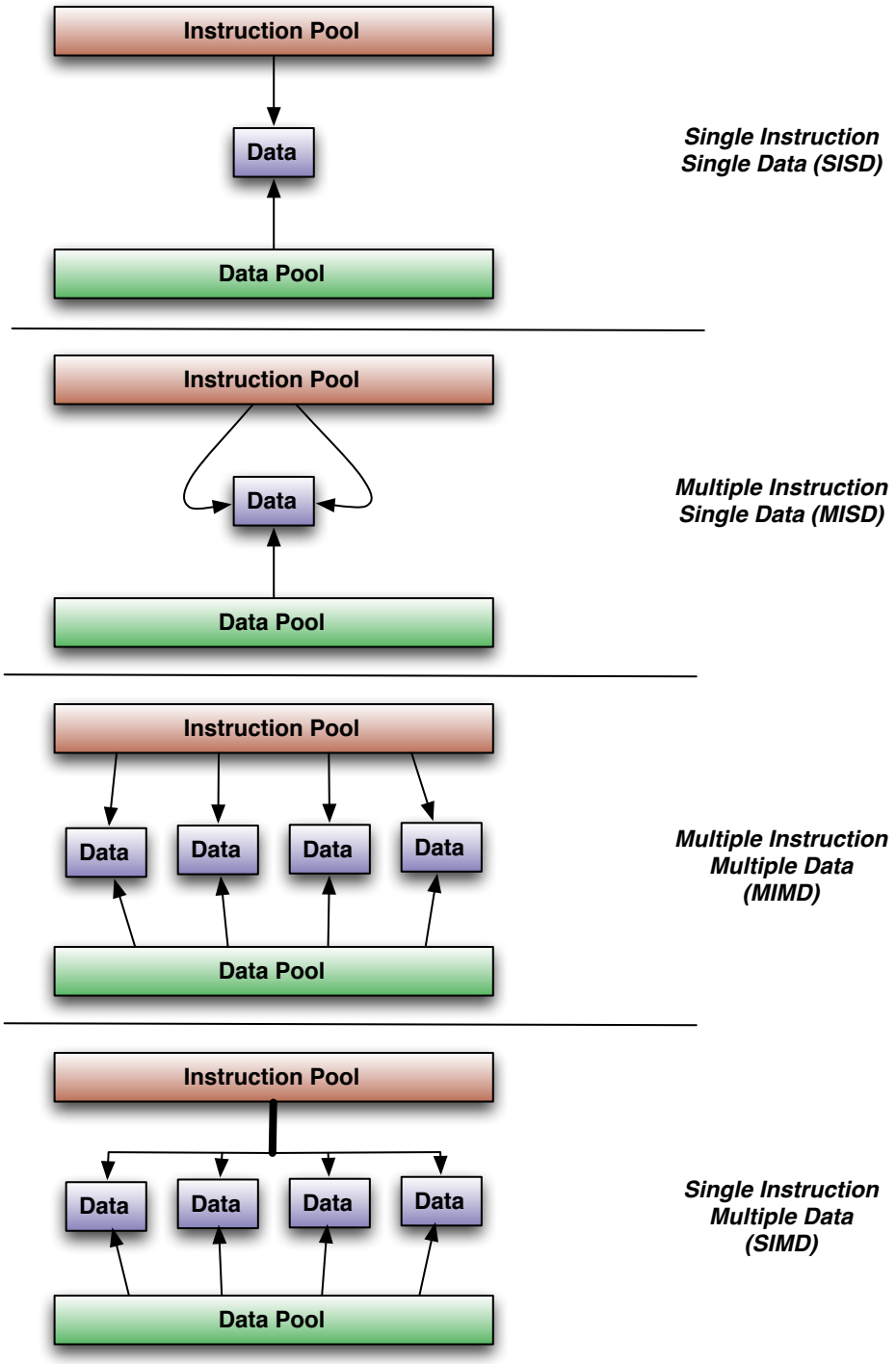


Figure 1.4: Diagram showing some classifications in Flynns taxonomy

SISD and MISD architectures are most commonly found in specialist or legacy systems, the final two of Flynns classifications on this diagram represent the two most common models of application programming today.

1.4 Other Performance Issues

There are additional features of computer architecture that can lead to reduced performance when attempting to process a large data set with a lot of calculations.

1.4.1 Threading

Many applications execute on a single thread, meaning that all operations are executed linearly. However, modern CPU's often have multiple *cores* each having many *threads*. This means that applications can perform heavy computations, concurrently, over these threads. For example, in a 4-core machine each core acts like a separate CPU, on which 2 threads can be run. Therefore the developer can take advantage by chunking his data into 8 parts and passing one chunk to each thread. In theory, some processes should speed up eight-fold. [3] The most significant thing to consider when writing multi-threaded systems is the state of the data. Developers must consider both:

- Accessing non-constant data from other chunks of threaded data
- After execution, waiting for each thread to finish to ensure data is synchronised

Developers must be careful with their memory access between threads, because whilst operating on data in one thread, another will be processing in parallel. Fortunately, there are several libraries that aid multi-threaded programming. [4] These help to ensure application data is in a 'safe state' for operating over many threads.

Typically the way this is achieved is through the use of a construct called a *mutex* [5]. This construct can be used to make sure that the process that locked the mutex is the only one that can unlock it. In multi-threaded systems, data pools being shared across threads will use this construct and check whether a mutex is already locked, if it isn't that thread can access and, in turn, lock it.

1.4.2 Memory Bandwith and Cache Misses

Some applications suffer from frequent cache misses, this means that when data is needed for processing in the CPU, it is fetched from RAM. This is far slower than loading data from the multi-level caches. When held in the cache, this data can be read very quickly by the registers to be operated on. If data is requested for processing and found in the L1 cache, this is a 'hit'. If it is not in memory there then a L1 cache 'miss' has occurred, and an attempt will be made to retrieve it from the L2 cache. This process is repeated down to the L3 cache. If *none* of the multi-caches hold the data, then it must be retrieved from RAM. This is a very undesirable situation and will slow

[3] although in practice threading overheads reduce this
[4] Intels Threading Building Blocks, Boost, OpenMP, POSIX etc.
[5] Short form for Mutually Exclusive

down applications if re-occurring. Fortunately, modern compilers will 'intelligently' predict the data that will required for incoming operations and fetch the relevant data into cache memory. This goes some way to avoiding cache misses. Even if data is not in the L1 cache, retrieving data from L2 or L3 memory, is far more preferable to a cache miss. For examples and testing surrounding this limitation see Chapter 4.

It should be noted that a common tuning technique is to manually *pre-fetch* data into cache memory. However, in most cases, the pre-fetching performed by modern compilers is sufficient.

Chapter 2

Improving Perfomance

2.1 Using CPU Width

The most performant of Flynn's taxonomy of architectures is Single Instruction Multiple Data (SIMD). In the context of a modern CPU, this means that the entire register is submitted to the ALU or FPU at once and is processed with a single instruction. Every chunk of data inside the register is processed in parallel. The size of the register is often referred to as the 'width of the lane'. In the MIMD model, data is processed as a linear stream of instructions. Improvements made to CPU clock speed, previously centred around increasing the throughput of data. In contrast to this idea of 'going faster', it is becoming more important for developers to 'go wider'; to use the full lane size of SIMD registers. Figure 2.1 shows the same vector addition as in Section 1.3, this time using SIMD.

2.2 SSE Instructions

In the late 1990's, Intel released Streaming SIMD Extensions (SSE) for the Pentium III processor. These extensions were then developed to incorporate additional arrangements and types of data, such as double precision floating point numbers. This new architecture was constructed with SIMD instruction units which allow SIMD to be fully utilised. Modern CPU's now typically support the latest version of extensions, SSE4. SSE gained popularity with its ability to compute fast floating point mathematics.

These extensions supply assembly instructions for an array of processes. The examples in the list below are for floating point values:

- Memory / Register data movement e.g. MOVAPS (Move Aligned Packed Single-Precision Floating-Point Values)
- Arithmetic e.g. ADDPS (Packed Single-Precision Floating-Point Add)
- Comparisons e.g. CMPSS (Compare Scalar Single-Precision Floating-Point Values)

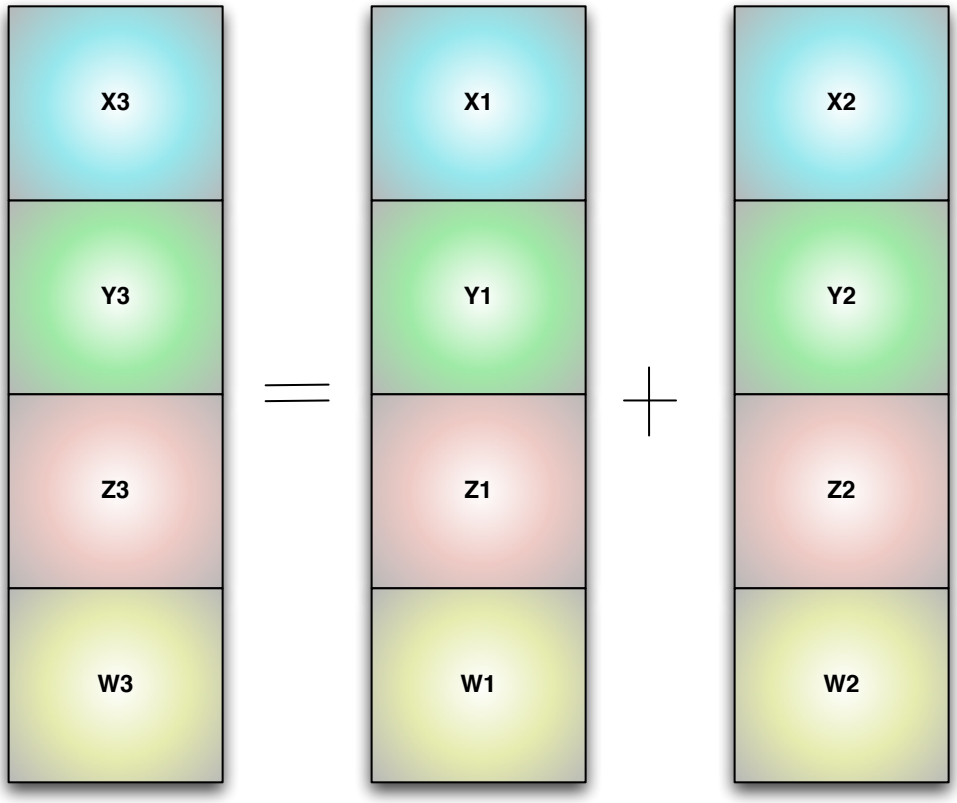


Figure 2.1: The use of registers in operations using the SIMD architecture

- Shuffling and Unpacking e.g. SHUFPS (Shuffle Single-Precision Floating-Point Values)
- Data conversion e.g. CVTSI2SS (Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value)
- Bitwise Logical Operations e.g. XORPS (Bitwise Logical XOR for Single-Precision Floating-Point Values)

Instructions like these are used to perform operations on entire registers[1] at a time. The speed of these processes is a major feature of modern CPU's, which have many registers built into them. Therefore in multimedia industries, that sometimes have 16-core machines, vast speed-gains are possible. One of the problems is, that writing these instructions in hard-to-maintain assembly language is not a viable long-term programability on the CPU.

It is important to note that whilst SSE instructions are not easily used, modern compilers will often 'guess' at parts of application that could be vectorised for SSE. Most compilers will automate some SSE 1, 2 and 3 instruction sets. Despite this,

[1] more of which were added to CPU's when SSE was released

the full benefit of SIMD architecture can only be realised is developers can explicitly program for it.

2.3 Intrinsics

In order to bypass the assembly instructions supplied by SSE, compiler intrinsics are provided for use within application code. Intrinsic functions are built into the compiler as appose to regular functions that are built into libraries or applications. There are a different set of functions available depending upon the SSE instructions the developer requires. There are header files provided that include special data types which define SIMD vectors for floating points, double precision and integer amongst others. These come with assignment operators for casting from serial to SIMD types. The type referred to throughout this paper will be *__m128* , which is a 4 chunk register of 32 bit floating point values.

Using these intrinsics, the assembly code highlighted in the previous section can be written in application as shown:

- MOVAPS : _mm_load_ps(float*)
- ADDPS : _mm_add_ps(__m128, __m128)
- CMPSS : _mm_cmpeq_ss(_m128, _m128)
- SHUFPS : _mm_shuffle_ps(_m128, _m128, int)
- CVTSI2SS : _mm_cvtsi32_ss(__m128, int)
- XORPS : _mm_xor_ps(__m128, __m128)

Using instructions like these has made programmability of the CPU more accessible for developers; which is especially useful in 3D applications. The reason for this is that data structures like four-by-four matrices and floating point vectors, representing co-ordinates, are used for cartesian transformations. For example, transforming a point from model to screen space by using the model-view-projection matrix. Not only can an __m128 type represent parts of these, but float pointers can be cast into float __vector__ pointers (__m128) for easier processing of large data sets. See appendix A.1 for an example.

This not only allows less loop iterations than a serial implementation, but the developer is able to take advantage of the multiple lanes of a SIMD architecture.

2.4 Intrinsic Difficulties

2.4.1 Data Arrangements

Although intrinsics allow the use of SIMD processing units there is an overhead to programming using these instructions; code maintainability. These compiler intrinsics are comparatively difficult to write when compared to serial applications. This is because constant awareness of data layout and alignment is required on the part of the developer. For example, an__m128 type should be 16-byte aligned to achieve optimum speed from SIMD code. The is because processors have a level of granularity for their memory lookups. They will look up memory at a power-of two memory address. This is done so that lots of time isn't spent looking for data on a byte-by-byte basis. Therefore, if the applications data is not aligned, the CPU has to load several chunks of memory and discard the bytes it doesn't need. (IBM, 2011) Aligning properly can become particularly difficult to achieve when constructing arrays of complicated data structures.

In addition to this, even if the data is aligned, many calculations require individual elements of SIMD lanes be operated on together, for instance when calculating vector length:

```
length = sqrt( x*x + y*y + z*z )
```

In this example, not only is the W component not required, but the separate memory chunks of the float vector need to be accessed individually.

In order to do this we need to get the individual elements of a float vector. Accessing individual elements of the vector is known as gathering. This term refers to reading the data based on its index within a vector lane. Storing data by index is known as scattering. This technique of read/write operations is together known as gather-scatter. The cost of doing this is very high and often renders using SIMD redundant.

To resolve issues of this nature, SSE includes shuffle instructions that allow fast data copying into new registers. So, in this example, for every four __m128 registers they can be re-arranged between the following layouts:

The price of using gather-scatter techniques is far more costly than shuffling memory and then operating on the data. However, the code to generate this shuffle requires knowledge of the internal data format of the application and the CPU being used; it is hard coded. In other words, if the SIMD lane size is 4, the shuffle instructions are written explicitly to reflect this. See Appendix A.2. This code can quickly become hard to maintain and is not generic or re-usable.

2.4.2 Hardware Dependence

As highlighted above the hardware dependence of different SSE instructions can become problematic. When programming for 128-bit registers, we can use the __m128 data

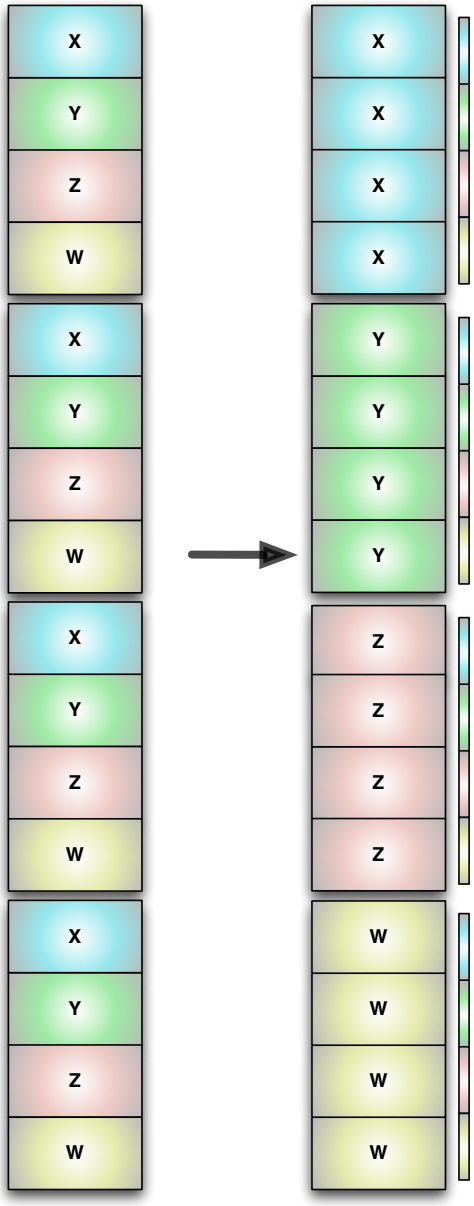


Figure 2.2: A diagram showing the process of shuffling from AOS to SOA

type, whereas on CPU's with 256-bit registers [2] a larger data set can be used in SIMD. Whilst future proof software is rarely guaranteed, an __m128 and __m256 data type are so conceptually similar that switching to more advanced software could be made easier. This is a problem that Intel have tried to overcome, as discussed below.

[2]expected in the coming years in Intels AVX processors

2.5 Intel SPMD Program Compiler

Released in mid 2011 was an LLVM[3] based compiler called the Intel SPMD Program Compiler (ISPC). The aim of this compiler is to take ISPC code and compile it to SSE assembly code. SPMD stands for Single Process, Multiple Data and is analogous to SIMD.

The idea is that developers can write "programs as if they were operating on a single data elements" (ISPC, 2011) then the underlying processor and runtime system executes several invocations of it concurrently with different inputs. This process is similar to the pixel shaders in a programmable GPU pipeline. A program is written for a single pixel, then compiled to tell the hardware to work on many pixels at a time. Not only does this language remove some of the more complex and un-maintainable aspects of SSE intrinsics but it is also, in general, future proof. It allows the developer to easily tie in the application (C++) code with ISPC (SSE) code. Two key ideas in ISPC are *uniform* and *varying* data types and the *programCount* and *programIndex* variables.

2.5.1 Uniform and Varying

A uniform data type is one corresponding to regular C++ types. For example a *uniform float []* is a 32 bit floating point array. This is the typical type used for passing in a pointer to your applications data-set. Varying data types are variables that change over different program instances (see below).

Using variables in ISPC is similar to the use of the same data-types in C++. The main thing to remember is that when using a *float* data type, the ISPC compiler will generate SSE assembly to make this into a *float __vector__* (__m128) type. Without any extra thought on the part of the developer a uniform float can be assigned to a varying float, the compiler will handle the conversion automatically. In addition, ISPC supports short vector types. Simply including:

```
typedef float<3> V3f;
```

allows for 3 lanes of SIMD floats to be used as X,Y and Z components, allowing more natural programability for 3D developers. These can then be accessed via swizzling techniques common to many shading languages.

2.5.2 Global Variables

In ISPC, the global variable *programCount* corresponds to the width of the SIMD lanes on the machine, for SSE this is four. Whilst *programIndex* can be used to retrieve the number of the currently running program instance.The reason for this construct

[3]LLVM stands for Low Level Virtual Machine but in actual fact has little to do with virtual machines. It is a "collection of modular and reusable compiler and toolchain technologies" (LLVM, 2011)

is to allow future processor architectures to run the same ISPC code at speeds native to their own specifications. This logic for vectorisation is analogous to that of CUDA programming.

However, the current version of ISPC still requires some shuffle code for converting between AOS and SOA, which removes this benefit of hardware agnostic code. This means that shuffle code written for SSE will have to be re-written for Intels new AVX processors. [4]

A simple example kernel is shown below. In this example, two float arrays a multiplied together and the result is stored in the first one. The *export* keyword marks the function to be made accessible in application. The keyword *uniform* tells the compiler that conventional non-vector types are being passed in. [5] The for loop has conventional form, significantly, i is incremented by *programCount* (4) each cycle. Then within the loop *programIndex* is used as an index that varies over the width of the SIMD lane. The operation is then performed using this index.

```
export void vmul(
                uniform float a[],
                uniform float b[],
                uniform int count )
{
        for (uniform int i = 0; i < count; i += programCount)
        {
                int index = i + programIndex;
                a[index] = a[index] * b[index];
        }
}
```

[4] These processors are not yet released and have 8-wide SIMD lanes

[5] all parameters in an exported function must be uniform

Chapter 3

Measuring Success

In order to fully understand the benefits of certain techniques, gathering meaningful data is paramount. This can be done in a number of ways. For CPU performance, two methods were explored; benchmark timings and profiling.

3.1 Benchmarking

A a 'naive' method for testing the speed of an application or operation is to time it. Simply recording when the process started and when it finished, can give a good impressions of speed and help to compare different methods of computation. The reason this method is not considered robust is that, especially with smaller data sets, the overhead of other small processes can skew results. There are ways to lessen this inaccuracy. Developers can perform many cycles of the process and accumulate our results, alternatively data-sizes can be increased to a degree that produces meaningful results. Although as mentioned in section 1.4.2, this can lead to memory bandwidth issues.

The reason that this method is often referred to as benchmarking is that it does supply a good estimation of the speed of an operation. It is important to note the use of the term 'operation'. Benchmarking is rarely used to measure the performance of an entire application. Its strength is in testing smaller, atomic operations or algorithms. For a comprehensive test of application-wide performance, profiling is preferable.

A basic benchmark timer was developed in order to quickly gather data about the execution time of different processes for the testing shown in Chapter 4. This implementation was then developed further to be integrated into a working application shown in Chapter 5.

3.2 Profiling

CPU profiling allows an application to run with some performance loss, whilst sampling the stack at regular intervals to see which function is in control. The key feature of

this kind of profiling is that the profiler samples n times every second and records the method it found running at the time. If the sample size is large enough to be representative (i.e. the application has lasted long enough to take more than just a few samples) the methods in an application that are most commonly in control can be found.

3.2.1 Open Source Profilers

There are two main open source profiling tools that have been explored; *Google Perftools* and *Valgrind* . The key difference between these two tools is the method in which they take samples.

Valgrind uses a technique called dynamic binary re-compilation. This means that the *Valgrind* core will disassemble an applications machine code into an intermediate representation. The tool being used [1] will then instrument it with analysis code. After this *Valgrind* recompiles it to machine code. (Valgrind, 2011) None of the original application code is actually run and as a result the execution of the program will usually take at least ten times longer.

Perftools on the other hand, has the profile code compiled inside the application at compile-time and as a result does not receive a very large slow down. This means that the execution speed more closely represents the actual performance of an application. It also allows automatic viewing of call-graphs in the web browser as SVG. This is very useful for speedy viewing of profiles. The command line program 'pprof allows quick customisation of what nodes to focus on and which to ignore.

While *Valgrind* is quicker and easier to get setup and profiling applications, a *Perftools* profile, once obtained, can better be used to get an impression of overall application performance. This is greatly helped by using the *Pprof* tool with the top# option to see total samples and heaviest functions.

With both tools it is possible to compile and profile debug builds. Whilst this may be useful to get a more in depth look at what is being called, the sample weighting does not reflect the execution times when built optimised.

In addition to these tools there is a popular free visualisation tool called *kCachegrind*.

kCachegrind is the de-facto profile viewer for *Valgrind* output, but can also be used with *Google Perfomance Tools*. *kCachegrind* allows developers to view and interact with their profiles to more easily target problem areas. It also allows viewing detailed call-graph information and can show the profile by as source file, function, class or compiled object. It allows an in depth look at how much time different sections of an application are being processed.

A very powerful feature of this tool is that if you compile using the -g flag[2], a profile

[1] For CPU profiles this is callgrind

[2]

can be viewed within the source code. This means that in the Graphical User Interface (GUI) developers can debug through the call graph on a line-by-line basis seeing how many samples each line had.

3.3 An Example

Consider the program in Appendix A.3. In this program data is generated then clamped within the range of 0.0 - 1.0, then remapped it to the range 10.0 - 100.0. The clamp and remap implementations are found in Appendix A.4

Using Google Performance Tools this text version of the profile can be output:

```
Total: 1751 samples
 761  43.5%  43.5%     1625  92.8% main
 459  26.2%  69.7%      535  30.6% remap
 316  18.0%  87.7%      316  18.0% clamp
 214  12.2%  99.9%      214  12.2% _wordcopy_fwd_aligned
   1   0.1% 100.0%        1   0.1% madvise
   0   0.0% 100.0%        1   0.1% ::do_free
   0   0.0% 100.0%        1   0.1% ::do_free_with_callback
   0   0.0% 100.0%        1   0.1% TCMalloc_SystemRelease
   0   0.0% 100.0%     1625  92.8% __libc_start_main
   0   0.0% 100.0%     1625  92.8% _start
```

This shows that, as expected, main is responsible for initiating most samples. After this point it is clear that the main areas to be improved are remap, clamp and _wordcopy_fwd_aligned. The first 2 functions have been defined in our application, but the last is, to begin with, unknown. Viewing the call-graph in kCachegrind the origin of this function is revealed.

It is now evident that _wordcopy_fwd_aligned is being called in the memory allocation for the std::vector

3.3.1 Remap

Remap can now be targeted as an area to improve. With some shuffling around, a constant can be calculated for each remap.

```
scale = ( newMax - newMin ) / ( oldMax - oldMin )
```

Remap is then changed to use less operations.

```
( in - oldMin ) * scale + newMin;
```

After profiling again, the top 5 slowest processors are as follows:

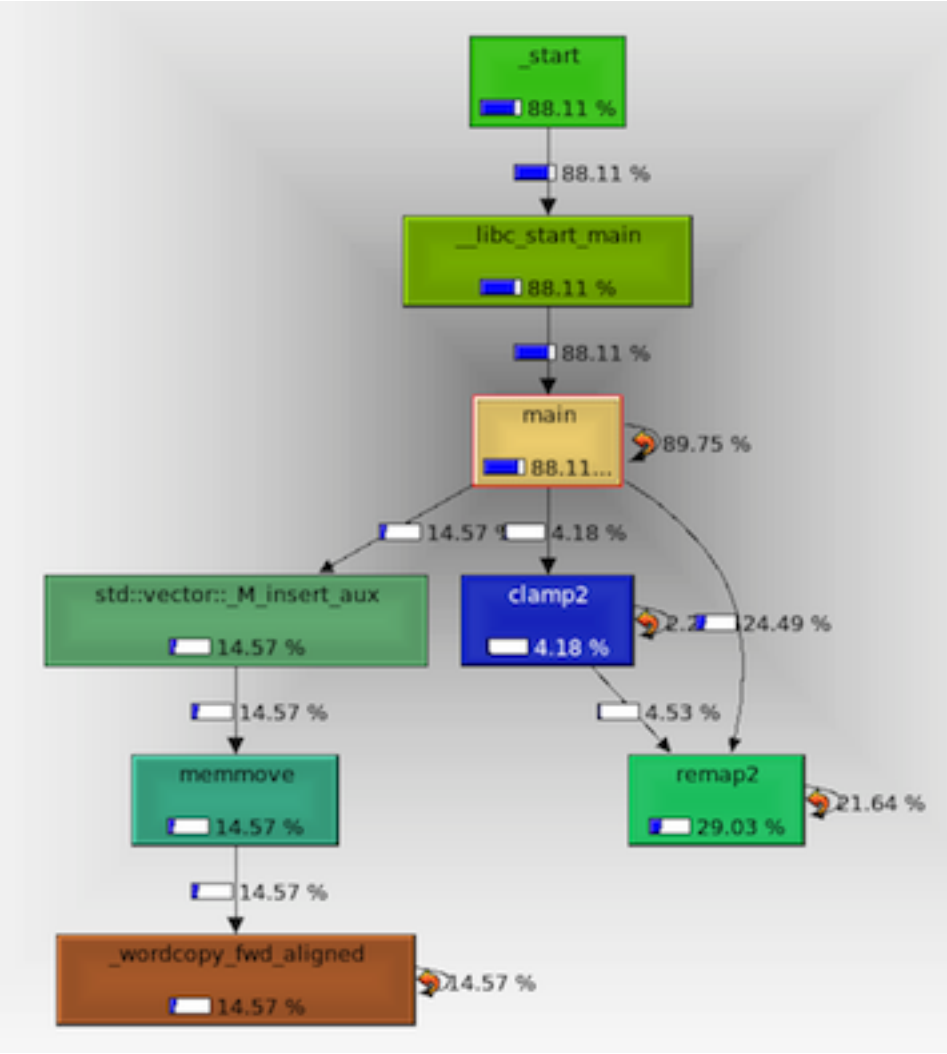


Figure 3.1: A helpful Call-graph in kCachegrind

```
Total: 1598 samples
 779  48.7%  48.7%     1470  92.0% main
 298  18.6%  67.4%      298  18.6% clamp
 289  18.1%  85.5%      354  22.2% remap2
 232  14.5% 100.0%      232  14.5% _wordcopy_fwd_aligned
   0   0.0% 100.0%     1470  92.0% __libc_start_main
```

By altering the remap method the samples have been cut from 1751 to 1598 and so the execution time is improved by 1.5s

3.3.2 Clamp

The next highest cost is the clamp (min/max) implementation. This is a conventional method for clamping, a new method can be attempted however, this is shown in Appendix A.5

After Profiling:

```
Total: 1540 samples
 722  46.9%  46.9%     1340  87.0% main
 492  31.9%  78.8%      492  31.9% remap2
 218  14.2%  93.0%      218  14.2% _wordcopy_fwd_aligned
 107   6.9%  99.9%      260  16.9% clamp2
   1   0.1% 100.0%        1   0.1% ProfilerStart
```

Here the samples are cut from 1598 to 1540, so there is a gain of 0.5s. This is a small improvement but is still a gain.

3.3.3 std::vector allocation

Relying on push_back to allocate memory is usually not a good idea. This is because of the way push_back allocates memory. Using Pprof to generate callgrind output the source code profile can be viewed for our allocation. See figure 3.2. This shows that around 20% (line 50) of samples taken from main occur in this loop. One option is to try reserving memory then pushing to the back of the list.

The new profile after this change is:

```
Total: 1315 samples
 790  60.1%  60.1%     1130  85.9% main
 386  29.4%  89.4%      386  29.4% remap2
 138  10.5%  99.9%      260  19.8% clamp2
   1   0.1% 100.0%        1   0.1% madvise
   0   0.0% 100.0%        1   0.1% ::do_free
```

The allocation of data no longer appears on the top 5 slowest operations list and 2.2s have been take off.

The source code profile (figure 3.2) now shows that the application actually spends longer inside this allocation loop, seconds may have been saved elsewhere but the push_back calls are expensive. Experimenting with resize yields the following results.

Profile:

```
Total: 1243 samples
 503  40.5%  40.5%     1074  86.4% main
 476  38.3%  78.8%      476  38.3% remap2
 139  11.2%  89.9%      139  11.2% std::vector::_M_insert_aux
 124  10.0%  99.9%      238  19.1% clamp2
   1   0.1% 100.0%        1   0.1% madvise
```

Despite _M_fill_insert appearing to eat up execution time, the key figure is the sample size. The execution time has gone from 1315 to 1243 samples, which means a 0.7s gain. The final source code profile shown in figure 3.2 highlights the effect of this.

The loop has been targeted exclusively using *kCachegrind* , and its enabled further optimisations of allocations. By filling the vector before hand, and *then* inserting the execution time has been reduced by just under a second. It seems that, in this case, resizing works best.

3.3.4 Summary

Over the course of the profiling for this sample application, the execution time has gone from 1751 samples to 1243. Using *Perftools* and *kCachegrind* just under 6 seconds have been shaved off, that's a 30% speed gain. It is clear why using a profiler can be a valuable practice for performance-minded developers. Although this example was an isolated and simple one and a 30% gain may not always be achieved; it shows how targeting specific weaknesses in applications could noticeably cut execution times.

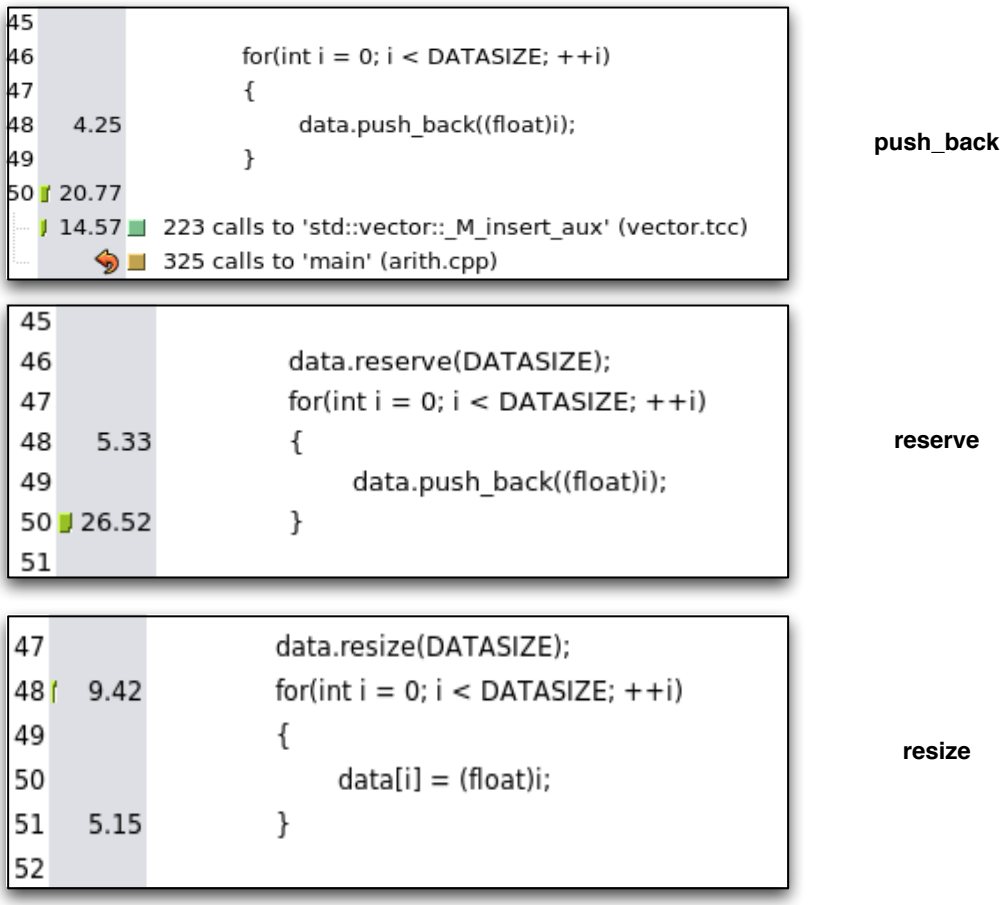


Figure 3.2: The source code profiles of different methods of allocating memory in a std::vector

Chapter 4

Benchmark Comparisons: Serial, SSE Intrinsics and ISPC

4.1 Initial Tests

A large part of implementation for this paper surrounded the extensive and thorough testing of basic operations using the three different methods; Serial, SSE Intrinsics and ISPC. The reason for this was twofold. Firstly, a benchmark of the possible speed increases of using intrinsics was desired and secondly a comparison of ISPC with Serial to verify the fidelity of Intels claims regarding the performance of ISPC. In order to do these a simple benchmark timer was developed, the interface if which is shown in Appendix A.6.

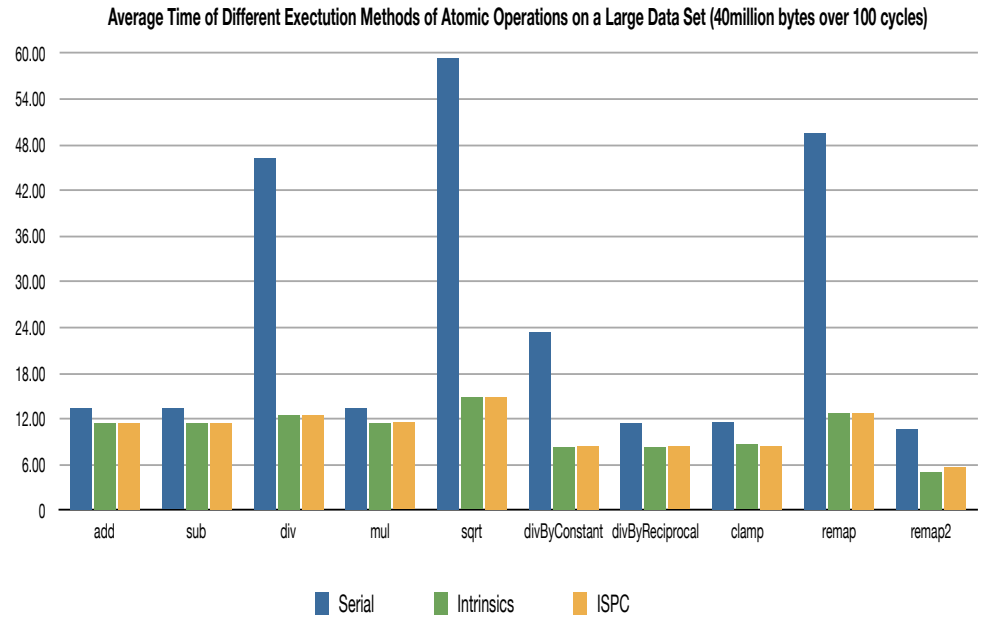
The initial set of testing I did was sending large data sets to be operated on using SSE, Serial and ISPC. Figure 4.1 show my results.

The first thing to notice is the peaks of blue. These are either notoriously expensive operations like divide and square root, or they are custom compound operations like remap. This shows us that the more SIMD operations computed in a single pass, the larger the benefit. This is because the memory access for data already loaded into registers is the fastest possible way to read and process data.

Another important feature to notice is just how closely ISPC and Intrinsics perform. The average execution time on each operation is virtually indistinguishable.

There is one disappointment revealed in these results however. The speed increase from SIMD on the atomic operations is sometimes only just over one times. The reason for this is related to memory bounds and cache missing. As explained in Chapter 1, the memory bandwidth is being exceeded. This means that with 40 million bytes of data, the caches are filling up and large chunks of memory need to be fetched from RAM, drastically slowing process speed down. This over head of fetching from RAM outweighs much of the advantage of vectorised instructions. The main bottleneck in this case is how quickly the processor can load data, not how quickly it operates on it; choosing MIMD or SIMD is almost irrelevant.

To find a solution to this problem a second set of tests was designed.



	Serial	Intrinsics	ISPC
add	13.4279	11.4624	11.4678
sub	13.3653	11.4722	11.4619
div	46.2344	12.524	12.5136
mul	13.3949	11.4776	11.5353
sqrt	59.3836	14.8081	14.8434
divByConstant	23.427	8.28946	8.42286
divByReciprocal	11.48	8.2904	8.38224
clamp	11.6159	8.60712	8.46062
remap	49.4987	12.743	12.7645
remap2	10.6998	5.01708	5.68488

Figure 4.1: Initial Tests of Performance of Several Simple operations using Serial, SSE and ISPC implementations

The aim of further testing was to monitor how an atomic operation like addition, performed over varying data sizes. Based on the results from the first set of tests only Serial and ISPC were chosen to test. This was due to the extremely close performance of intrinsics and ISPC on both atomic and advanced operations. (see figure 4.1) ISPC appears to be both fast and easily programmable.

Figure 4.2 shows the amplitude of gains achieved from using ISPC to sum two arrays

of floating point numbers at varying sizes.

DataSize(Kb)	Times Gain
0.516	2.4263666755
0.664	2.5658468266
0.86	2.7877894777
1.112	2.2430964972
1.436	2.5422300496
1.856	3.1799335979
2.396	3.331232493
3.096	3.0880005752
4	3.4466213915
5.164	3.4363260266
6.672	3.7291991852
8.616	3.6719224898
11.128	3.5180965088
14.372	3.1086374622
18.564	2.8356093695
23.976	2.8586039671
30.968	2.8565304266
40	2.843723564
51.66	2.7806822367
66.724	2.6857413077
86.176	2.4100036996
111.3	2.272944755
143.752	2.0285447656
185.66	1.9552730994
239.792	1.9237396329
309.704	1.9266034237
400	1.925692936
516.616	1.9279513396
667.24	1.9318243663
861.772	1.9302245066
1113.02	1.9278146195
1437.524	1.9157894182
1856.632	1.8433277862
2397.936	1.6256282289
3097.052	1.2884353685
4000	1.2072760819
5166.196	1.1957732228
6672.4	1.1809634229
8617.736	1.1831155644
11130.236	1.1838136282
14375.252	1.2115782883
18566.356	1.2014440785
23979.368	1.1689749298
30970.548	1.356370006
40000	1.3041466601

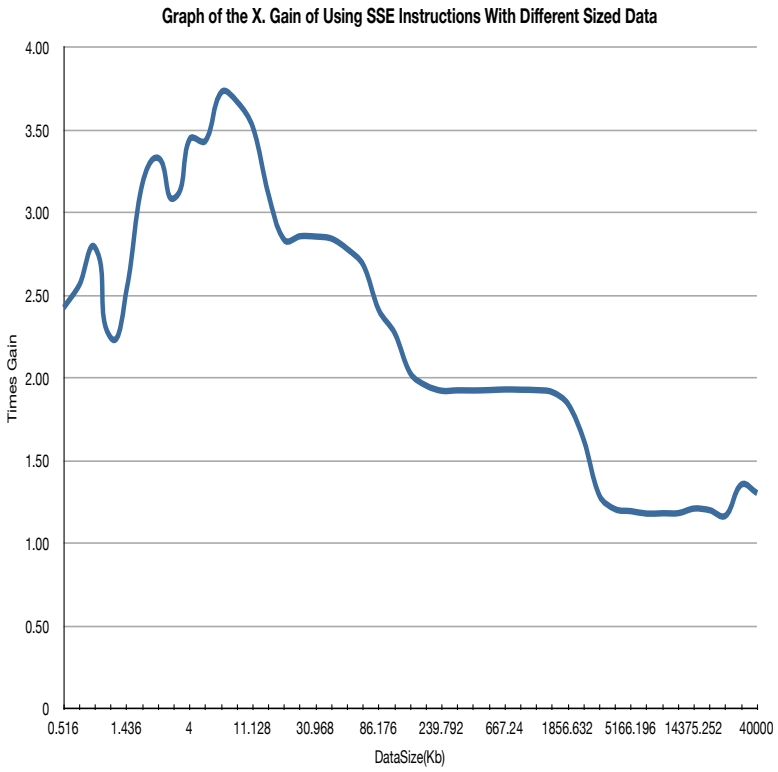


Figure 4.2: The Gain of Using ISPC to Sum to Float arrays at Growing Data Size

This clearly shows three distinct fall offs that correspond to the different cache levels of the processor.

- $\hat{=}$ 9kb: At this stage the CPU is able to fetch everything into the L1 Cache in time for fast SIMD processing.
- $\hat{=}$ 20kb: Here, the CPU is having to fetch data from the L2 cache into its registers, the size of data being fetched from here has been increasing since 9Kb

- $\hat{=}$ 170Kb The level of Data available in the L2 cache has been falling and now regular fetches from the L3 are being made
- $\hat{=}$ 3800Kb The data is now being fetched from RAM a lot and the gains of SIMD instructions are lost

This shows that at a certain point, depending on the machine, the overhead of exceeding memory bandwidth and cache misses will overshadow any gains made from using SIMD. This suggests two important practices when making applications for use with large data sets.

1. Chunk data into sizes that correspond to the "sweet spot" of you CPUs memory bandwidth.
2. If you have to pass a large data set in one call, and cannot split it up, perform as many of the operations needed in a single pass. This utilises good cache efficiency.

4.2 Measuring Gather-Scatter

As mentioned in Chapter 2, using gather-scatter techniques to read and write to individual elements of a SIMD register can have significant performance hits. A set of tests was written to measure the impact of these techniques when doing some simple inter-vector operations. The results, shown in figure 4.3, give a convincing argument for processing data as structure of arrays (SOA) and appose to arrays of structures (AOS).

The form of the operation accumulate is to sum each individual element of the vector and store that value back to each element, like so:

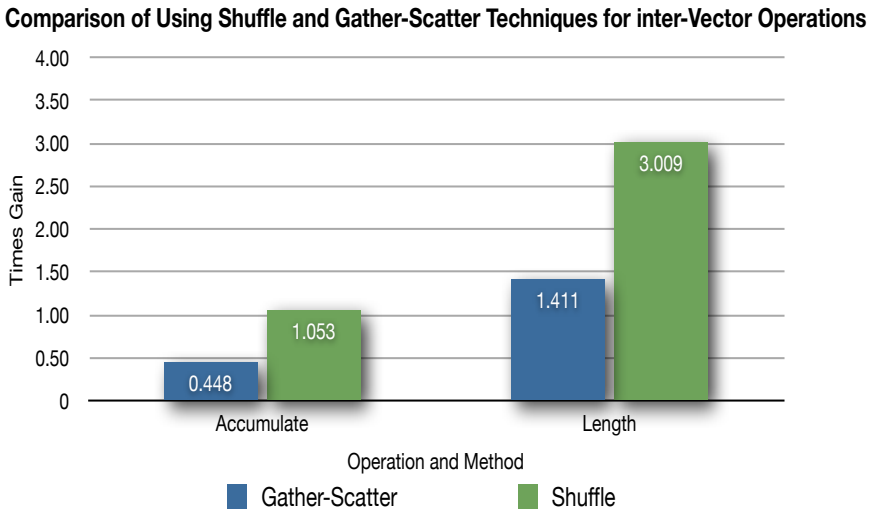
```
v = v.x+v.y+v.z+v.w
```

The length operations simply calculates the length of the vector and stores it back to each index.

```
v = sqrt(v.x*v.x + v.y*v.y + v.z*v.z)
```

This clearly shows us that the overhead of shuffling into SOA is worth the eventual payoff of being able to use SIMD instructions. The cost of gathering even renders the accumulate operations slower than the serial implementation! In the ideal case developers should consider storing there data initially as SOA in order to try avoiding the shuffling process altogether; although this is not often easy to do.

The code for the above tests is included with this paper and provided a strong basis from which to more vigorously test ISPC in a 3D application.



Serial

Actions	Average
Accumulate	32.1332
Length	101.133

Gather-Scatter

Actions	Average	Gain
Accumulate	71.7531	0.4478301286
Length	71.6955	1.4105906228

Shuffle

Actions	Average	Gain
Accumulate	30.5066	1.0533196095
Length	33.615	3.0085676037

Figure 4.3: Gather-scatter and shuffle tests done over 100 cycles on 10 million floats

Chapter 5

Mesh Deformer: Case Study

The aim of the following case study is to show the gains the can be achieved from using ISPC for 3D applications. Although the example is contrived it demonstrates the gains that can be achieved by utilising SIMD instruction architecture.

5.1 The Application

The application itself is a mesh deformer preview window. It currently supports two different types of mesh deformations. Perlin Noise and Radial Distance. The deformation of meshes is a procedure common in the film industry. It can contribute to specialist visual effects needed for a large variety of projects. Deformation of a large number of points can become a costly process and is not always very usable for artists.

5.1.1 Radial Deformer

The radial deformer is a simple one that uses the distance of the point from a spherical field origin as a way of moving. The formula for this deformation is:

$$\delta P = \sqrt[3]{r^3 + \mid P - O \mid^3}.\frac{\mid P - O \mid}{P - O} + O$$

Where r is the radius of the sphere field and o is the origin. P is the point being deformed.

The effect of this deformation on a grid of points is shown in figure 5.1. This figure shows the output of a serial implementation of the radial deformer. It should be noted that this effect scales to a 3D mesh reliably.

5.1.2 Perlin Noise

In 1983 Ken Perlin set about creating "a primitive space-filling signal" that had variation and looked random [19]. Importantly however, this noise had to be controllable so it could be used to design different aesthetic. This means that a table of indices or the permutation table can be used to lookup into another table of pseudo-random

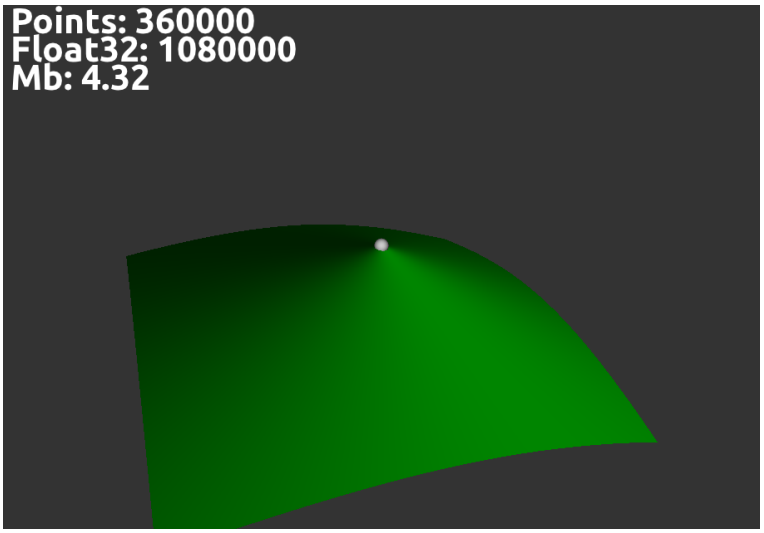


Figure 5.1: Visual output of Radial deformer on grid. Serial Implementation.

numbers; in this case between -1.0 and 1.0. Perlin originally used this range to easily "peturb things" using the values to "fuzz out"[19] the values close to zero

Figure 5.2 shows the effect of a Perlin Noise deformer applied as a height field to a grid, implemented in Serial.

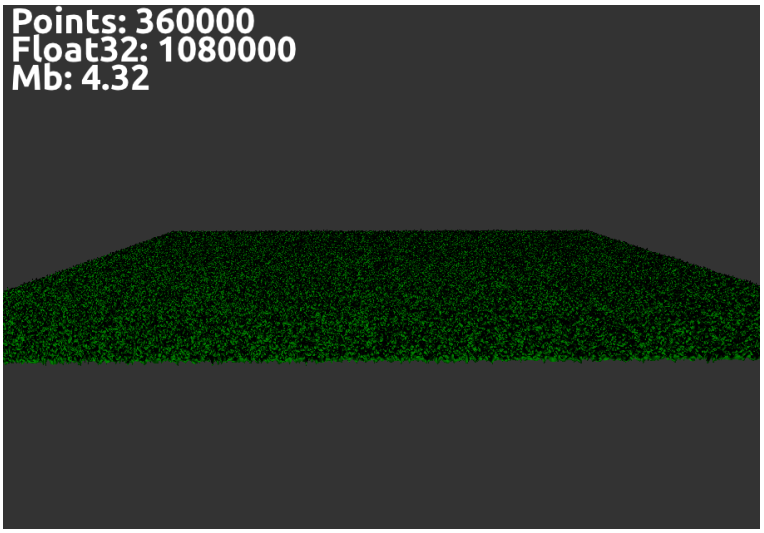


Figure 5.2: Visual output of noise deformer on grid. Serial Implementation

5.1.3 A Quick Profile

A profile of the serial implementations of the two deformers shows what areas consume most of the applications execution time. The data was collected by simply profiling *main()* and performing each deformation 20 times in succession.

- Total - 2025 samples

- Radial - 342 samples
- Noise - 633 samples

This profile shows that the problem areas are as expected, the most computationally heavy sections of code.

5.2 Injecting ISPC

In order to speed up the calculation of the deformations ISPC algorithms were implemented to utilise the speed that the CPU offers. The main criteria for the output of these implementations was the fidelity of the effect in comparison to the serial original. Figures 5.3 and 5.4 show the radial and noise deformers implemented in ISPC.

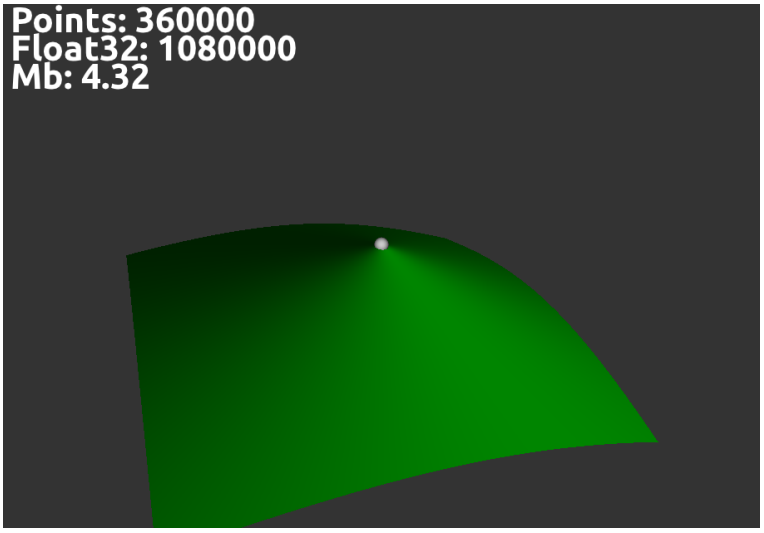


Figure 5.3: ISPC implementation of Radial Deformer

The level of aesthetic similarity to the serial implementations is evident. Having achieved results aesthetically, the calculation speed must be assessed. Initially, benchmark tests can be compared between both implementations. Figures 5.5 and 5.6 shows the application output of benchmark timings for the deformers.

The radial deformer has gone from to while to noise deformer has gone from to .

To support this overwhelming improvement, the new profile can be taken for the ISPC version that reflect these results.

- Total - 1566 samples
- Radial - 102 samples
- Noise - 189 samples

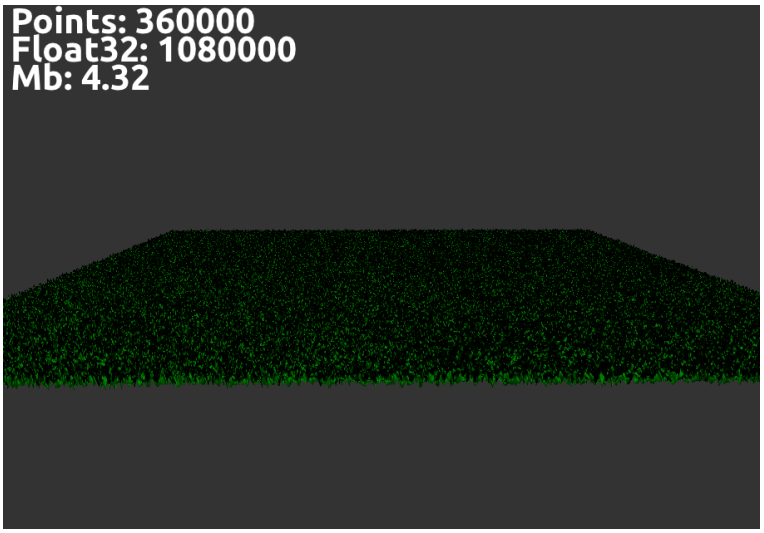


Figure 5.4: ISPC implementation of Noise Deformer

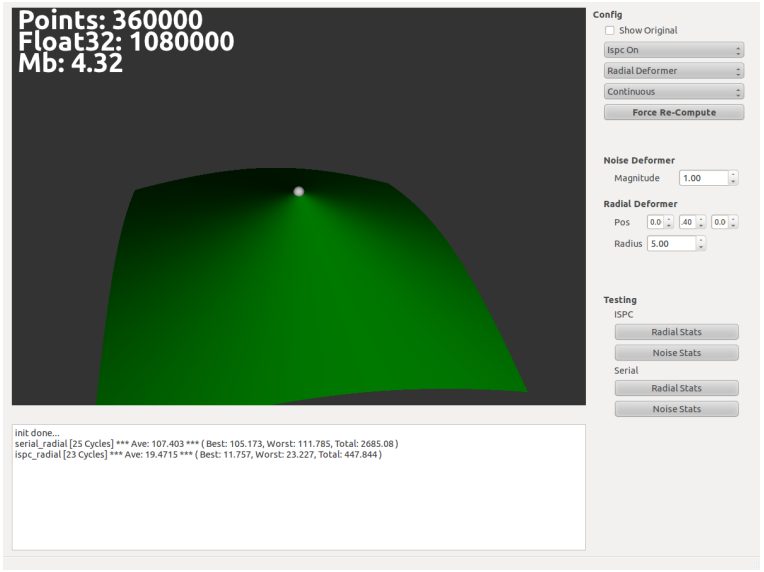


Figure 5.5: Benchmark timing of Radial Deformer

It is clear to see that from using the width and speed of the CPU to calculate operations of some algorithmic complexity massive performance gains can be seen. By compounding several atomic operations on a large data set, as shown in calculating Perlin Noise, the CPU can be cache efficient and use SIMD instructions effectively.

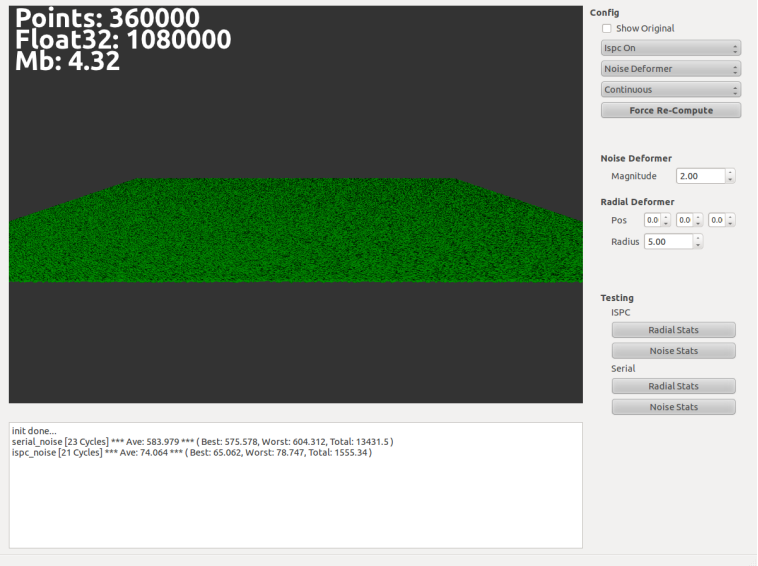


Figure 5.6: Benchmark timing of Noise Deformer

Chapter 6

Conclusion

Overall, the findings of the research conducted for this paper were fairly conclusive. Initially it was identified that SSE intrinsics are often difficult to develop and maintain; they are also even more difficult to integrate with existing applications. For this reason the recently released ISPC was chosen as the focus for exploration. The idea of a programmable CPU being analogous to GPGPU is, at first somewhat counter-intuitive. However, once developers fully grasp how machine architecture *can* be used, a compiler like ISPC begins to make more sense. It has been shown that ISPC aims to be extendable to future architectures and although some shuffling procedures limit this facility, future versions are expected to automate this process.

Initial benchmark testing showed that, given the optimum memory size SIMD instructions can perform between three and four times quicker than serial operations. Using structures of arrays to avoid complicated data accessing or shuffling also speed the process up and ultimately a 3D application like the one shown in Chapter 5 can be sped up considerably.

Among the recent trend toward GPGPU, the CPU can often get forgotten about as a utility for speed. Much like the GPU however, its capacity for performance is growing. In the next few years Intels AVX, 256-bit wide architectures will become accessible for developers, and there is a lot of potential for advanced CPU usage. One example of this has already been developed by Intel with their real-time ray tracer Embree [13].

Ultimately, the kind of speed gains shown in Chapter 5 are integral for the film industry. Rendering is notoriously expensive and costs thousands of server-hours from production to production. Pixar, in their latest release of Renderman implemented a large SIMD backend to improve performance. Many companies have code bases that are becoming legacy and tools that do not operate as quickly as desired. [1] This paper shows that through General-Purpose CPU programming using ISPC, 3D applications can benefit enormously.

[1] Ideally real-time

Chapter 7

Appendices

Appendix A

1. **Conventional Method for Loading Aligned Data for Processing in SIMD**

```
// load a large data set to 128 bit registers
// to more quickly square them
float * data = getData();
__m128 packedData = (__m128*) data;

for( int i = 0; dataSize / 4; ++i)
{
        packedData[i] = _mm_mul_ps( packedData[i],
                                    packedData[i] );
}

float * processData;
_mm_load_ps( processedData, packedData );
```

2. **Typical Shuffling Code**

```
shuffle code
```

3. **Example Program to Demonstrate Profiling**

```
#define DATASIZE 100000000

int main()
{
    ProfilerStart("path/to/sample.prof");

    // loops to run the program
        //again for larger sample size
    for(int cycle = 0; cycle < 20; ++cycle)
    {
        std::vector<float> data;

        for(int i = 0; i < DATASIZE; ++i)
```

```
        {
            data.push_back((float)i);
        }

        // the range of our clamped data
        float oldMax = 1.0;
        float oldMin = 0.0;

        // the range we want to map to
        float newMin = 10.0;
        float newMax = 100.0;

        for(int i = 0; i < DATASIZE; ++i)
        {
            data[i] = clamp(data[i], 0.0, 1.0);
            data[i] = remap( data[i],
                        oldMin, oldMax,
                        newMin, newMax);
        }
    }

    ProfilerStop();
    ProfilerFlush();

    return 0;
}
```

4. **Naive Clamp and Remap Implementations**

```
float clamp(float a, float min, float max)
{
    float tmp = std::max(min, a);
    return std::min(max, tmp);
}

float vremap(
        float in,
        float oldMin,
        float oldMax,
        float newMin,
        float newMax
        )
{
    return      ( in - oldMin ) / ( oldMax - oldMin ) *
                (newMax - newMin) + newMin;
}
```

5. **New Form Clamp Without Min/Max**

```
float ret = 0.0;
```

```
if( a > max )
        ret = max;
else if( a < min )
        ret = min;
else
        ret = a;

return ret;
```

6. **Interface for a Simple Robust Benchmark Timer**

```
class Record
{
public:
    Record();
    float m_secs, m_best, m_worst, m_total, m_cycles;
    timeval m_startT, m_endT;
};

class Timer
{
public:
    Timer();

    void startCycle( const std::string &_recordName );
    void endCycle( const std::string &_recordName );

    inline float peek( const std::string &_recordName )
    {...}

    void addRecord( const std::string &_name );

    void writeToCSV(
                const std::string &_file,
                const std::string &_recordName );

    void writeAllToCSV(const std::string &_file );
    std::string getRecord( const std::string &_recordName);
    std::string getAll();

private:
    std::map<std::string, Record> m_records;
};
```

Bibliography

[1] Gddeke, D. (2002).
General-Purpose Computation on Graphics Hardware.
Available: http://gpgpu.org/about
Last accessed 3rd August 2011.

[2] Gupta, Satish Chandra. (2005).
Need for speed - Eliminating performance bottlenecks.
Available: http://www.ibm.com/developerworks/rational/library/05/1004_gupta/
Last accessed 25th July 2011.

[3] Flynn, Michael. J -
Some Computer Organizations and Their Effectiveness IEEE Transactions on Computers
21(9):948--960, September, 1972.
Available at: http://www.cs.utah.edu/~kirby/classes/cs6230/Flynn.pdf

[4] Anon. (unknown).
Basics of SIMD Programming.
Available: http://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/
ps3-linux-docs-latest/CellProgrammingTutorial/BasicsOfSIMDProgramming.html
Last accessed 11th August 2011.

[5] Stokes, J. (2000).
SIMD architectures.
Available: http://arstechnica.com/old/content/2000/03/simd.ars
Last accessed 28th July 2011.

[6] Intel ISPC. (2011).
User's Guide.
Available: http://ispc.github.com/
Last accessed 3rd August 2011.

[7] Engelen, R. (2009).
Floating Point Operations and Streaming SIMD Extensions.
Available: http://www.cs.fsu.edu/~engelen/courses/HPC-adv/FP.pdf
Last accessed 1st August 2011.

[8] Fisher, R. (1997).
General-Purpose SIMD Within A Register: Parallel Processing On Consumer Microprocessors
Available: http://www.enlight.ru/docs/arch/prelim.pdf.
Last accessed 4th August 2011.

[9] StackExchange. (2011).
Aligned memory management?.
Available: http://stackoverflow.com/questions/5061392/aligned-memory-management
Last accessed 6th August 2011.

[10] Anon. (2002).
Optimizing for SSE: A Case Study.
Available: http://www.cortstratton.org/articles/OptimizingForSSE.php
Last accessed 5th August 2011.

[11] Anon. (2010).
SSE Intrinsics Tutorial.
Available: http://www.formboss.net/blog/2010/10/sse-intrinsics-tutorial/
Last accessed 10th August 2011.

[12] Stewart, J. (2005).
An Investigation of SIMD instruction set.
Available: http://noisymime.org/blogimages/SIMD.pdf
Last accessed 8th August 2011.

[13] Woop, S. Ernst, M. (2011).
Embree: Photo-Realistic Ray Tracing Kernels
Available: http://software.intel.com/en-us/articles/
embree-highly-optimized-visibility-algorithms-for-monte-carlo-ray-tracing/
Last accessed 10th August 2011.

[14] Song A, (2005).
Data Alignment

Available: http://www.songho.ca/misc/alignment/dataalign.html
Last accessed 8th August 2011.

[15] Rentzsch, J. (2005)
Data alignment: Straighten up and fly right
Available: http://www.ibm.com/developerworks/library/pa-dalign/
Last accessed 5th August 2011.

[16] Microsoft (2011)
Compiler Intrinsics
Available: http://msdn.microsoft.com/en-us/library/26td21ds.aspx
Last accessed 10th August 2011.

[17] Alvaro, W. Kurzak, J. Dongarra, J. (2008)
Fast and Small Short Vector SIMD Matrix Multiplication Kernels
for the Synergistic Processing Element of the CELL Processor
Available: http://www.netlib.org/lapack/lawnspdf/lawn189.pdf
Last accessed 9th August 2011.

[18] Abel, J. et al. (1999)
Applications Tuning for Streaming SIMD Extensions
Available: http://download.intel.com/technology/itj/Q21999/PDF/apps_simd.pdf
Last accessed 10th August 2011.

[19] Perlin, K (1999)
Making Noise
Available: http://www.noisemachine.com/talk1/6.html
Last accessed 9th August 2011.