University of Ballarat
School of Information Technology and Mathematical Sciences

An Investigation of SIMD instruction sets

By Josh Stewart (josh@noisymime.org)

27/11/05

Supervisor: Greg Simmons

Overview

The aim of this project was to investigate Single Instruction Multiple Data (SIMD) instructions sets and perform both a theoretical and practical comparison of two implementations. This is to be done with two purposes:

1. Learn the basics of SIMD programming, including its history, on different platforms; and
2. Investigate how the performance of two SIMD implementations compares in a common task

The research has focused around SIMD on the desktop and included in this report is an overview of what SIMD is, a brief history and a practical comparison of two implementations.

Theoretical Comparison

SIMD Overview

The SIMD concept is a method of improving performance in applications where highly repetitive operations need to be performed. Simply put, SIMD is a technique of performing the same operation, be it arithmetic or otherwise, on multiple pieces of data simultaneously.

Traditionally, when an application is being programmed and a single operation needs to be performed across a large dataset, a loop is used to iterate through each element in the dataset and perform the required procedure. During each iteration, a single piece of data has a single operation performed on it. This is known as Single Instruction Single Data (SISD) programming. SISD is generally trivial to implement and both the intent and method of the programmer can quickly be seen at a later time. Loops such as this, however, are typically very inefficient, as they may have to iterate thousands, or even millions of times. Ideally, to increase performance, the number of iterations of a loop needs to be reduced.

Once method of reducing iterations is known as loop unrolling. This takes the single operation that was being performed in the loop, and carries it out multiple times in each iteration. For example, if a loop was previously performing a single operation and taking 10,000 iterations, its efficiency could be improved by performing this operation 4 times in each loop and only having 2500 iterations.

The SIMD concept takes loop unrolling one step further by incorporating the multiple actions in each loop iteration, and performing them simultaneously. With SIMD, not only can the number of loop iterations be reduced, but also the multiple operations that are required can be reduced to a single, optimised action.

SIMD does this through the use of 'packed vectors' (hence the alternate name of vector processing). A packed vector, like traditional programming vectors or arrays, is a data structure that contains multiple pieces of basic data. Unlike traditional vectors, however, a SIMD packed vector can then be used as an argument for a specific instruction (For example an arithmetic operation) that will then be performed on all elements in the vector simultaneously (Or very close to). Because of this, the number of values that can be loaded into the vector directly affects performance; the more values being processed at once, the faster a complete dataset can be completed. This size depends on two things:

1. The data type being used (ie int, float, double etc)
2. The SIMD implementation

When values are stored in packed vectors and 'worked upon' by a SIMD operation, they are actually moved to a special set of CPU registers where the parallel processing takes place. The size and number of these registers is determined by the SIMD implementation being used.

The other area that dictates the usefulness of a SIMD implementation (Other than the level of hardware performance itself) is the instruction set. The instruction set is the list

of available operations that a SIMD implementation provides for use with packed vectors. These typically include operations to efficiently store and load values to and from a vector, arithmetic operations (add, subtract, divide, square root etc), logical operations (AND, OR etc) and comparison operations (greater than, equal to etc). The more operations a SIMD implementation provides, the simpler it is for a developer to perform the required function. SIMD operations are available directly when writing code in assembly however not in the C language. To simplify SIMD optimisation in C, *intrinsics* can be used that are essentially a header file containing functions that translate values to their corresponding call in assembler.

SIMD Example

The best way to demonstrate the effectiveness of SIMD is through an example. One area where SIMD instructions are particularly useful is within image manipulation. When a raster-based image, for example a photo, has a filter of some kind applied to it, the filter has to process the colour value of each pixel and return the new value. The larger the image, the more pixels that need to be processed. The operation of calculating each new pixel value, however, is the same for every pixel. Put another way, there is a single operation to be performed and multiple pieces of data on which it must be completed. Such a scenario is perfect for SIMD optimisation.

In this case, a SIMD optimised version of the filter would still have a main loop to go through the entire pixel array, however the number of iterations would be significantly reduced because in each pass, the loop would be transforming multiple pixels.

SIMD History

SIMD instruction sets have been utilized within computing since the early 1980's. These units, however, were almost exclusively used in large supercomputers and mainframes. Also during this period there were a number of attempts to produce a pure SIMD processor. This, however, was largely unsuccessful as SIMD, whilst being very well suited to some applications, is not as flexible as a traditional SISD processor and many

common programming algorithms cannot be reproduced easily or efficiently in a SIMD manner.

The introduction of SIMD processors onto the desktop platform was not something that the chipmakers immediately saw as necessary. When the Information Technology (IT) boom of the mid-nineties pushed PC's into millions of homes, it became clear that multimedia applications were going to become ubiquitous on PC's. It was during this period that Intel released the first desktop CPU with SIMD capabilities, the Pentium MMX. Whilst severely limited in both its technical capabilities and uptake, MMX opened the door for future SIMD technologies.

Since the release of MMX, all the desktop CPU manufacturers (With the exception of the now defunct Cyrix) have released chips with SIMD instructions. In some cases these have been the manufacturers own design while in other instances specifications have been licensed for use in a manufacturers chips. The table below gives a brief timeline of desktop SIMD implementations:

1996	Intel release Pentium chips featuring MMX. MMX goes widely unused due to its technical limitations.
1997	Motorola introduce the Altivec instructions into the last of the G3 PowerPC chips. A more complete version is released shortly afterwards in the new G4 PowerPC. Developers welcome Altivec and uptake on the Apple platform is fast.
1997 / 1998	Intel release MMX2 (Pentium 2) to address some of the limitations of the original specification. Uptake is still slow.
1999	Intel release SSE on the Pentium 3 chip. Adding significant improvements to MMX, SSE enjoys more success than its predecessor, however its use is still small compared to Altivec.
2000	AMD release Athlon chips containing 3DNow! instructions. 3DNow! is an extension to MMX that is very similar to SSE. Uptake

	is small as 3DNow! is proprietary to AMD, whose market was relatively small.
2002	Intel releases the Pentium 4 containing SSE2 instructions. SSE2 addresses many of the problems with previous implementations from Intel. SSE2 is very successful and used across many applications in the multimedia field.
2002	IBM releases the G5 PowerPC that includes the Altivec instructions licensed from Motorola. Whilst the instructions remain the same, IBM improve the hardware backend of Altivec, further increasing performance.
2003	AMD introduces Advanced 3DNow! (Or 3DNow!2). These instructions are roughly in line with SSE2 however it does not enjoy the same success, primarily due to AMD's low market share.
2004	Intel launches SSE3 in new Pentium 4 chips. SSE3 includes a large and flexible instruction set and improved performance on all SSE/2 operations. Intel claims better performance than Altivec.
2004	AMD license SSE and SSE2 instructions for inclusion in their new Athlon and Athlon 64 processors.

As SIMD on the desktop becomes both more common and more technically advanced, the number of cases where it can be used has increased dramatically. Despite this, many developers do not have the required knowledge or the development time to implement these features. For this reason compiler developers have slowly introduced a process known as auto-vectorisation.

Auto-vectorisation is the use of SIMD optimizations in code that was not originally written with it in mind. The programmer may not even be aware that SIMD is being used as the compiler performs all the required transformations when it detects an area of code that will benefit from such optimizations. Obviously, auto-vectorisation does not give the

same level of performance increase as an experienced programmer, but it does return noticeably improved binaries in many instances. It also means that code can benefit from SIMD optimizations without the need for multiple versions to be written and maintained for different platforms. Auto-vectorisation is currently implemented in the following compilers:

- GCC-4.0 (GNU C Compiler) and above
- ICC-8.1 (Intel C Compiler) and above
- Code Warrior 9 and above

Altivec

Altivec is the SIMD implementation used by Apple on its PowerPC based Mac PC's and laptops. Originally developed by Motorola for the G4 PowerPC processor it is now being used by the IBM manufactured G5 processors. Altivec was the first desktop SIMD instruction set to gain widespread acceptance by the developer community, being released 4 years before the comparable SSE2 implementation.

Altivec is comprised of 32 128-bit registers that are used to hold packed vectors for processing. Altivec supports up to 32-bit integers (signed or unsigned) or floats meaning that a single vector can contain one of the following values:

- 16 8-bit values
- 8 16-bit values
- 4 32-bit values

A significant weakness within Altivec is the lack of support for 64-bit values within registers, meaning that only single precision floats are available.

Despite this disadvantage, however, Altivec has many other strengths. The technology is now very mature when compared to other SIMD implementations and not only does this mean that the hardware has been refined, but also that there is significant documentation

and example content available for developers. Additionally Apple has gone to great lengths to ensure that Altivec is as simple as possible for a developer by adding intrinsics to the GCC compiler as well as integrating it into its XCode development environment.

Altivec's instruction set has remained virtually the same since its first version. It is the largest of the desktop SIMD instruction sets with many methods of loading and storing as well as comparison available. Strangely, however, Altivec does not contain some basic arithmetic operations such as divide, square root and reciprocal. Apple provides a utility header file that provides these functions by using other available SIMD instructions (still giving quite reasonable performance) however it is sometimes frustrating that these are not implemented at the hardware level.

SSE

SSE is the name given to a family of SIMD implementations developed by Intel. These have their roots in the MMX instruction set that first bought SIMD onto the desktop in the Pentium chip. SSE has gone through 3 revisions (SSE, SSE2 and SSE3) that have each added functionality to both the hardware and software sides of SSE. SSE first appeared in the Pentium 3 range of chips release in 1999 however, despite being a significant improvement over MMX, was still limited in a number of areas. The introduction of the Pentium 4 family of CPUs bought with it SSE2 and a much more complete SIMD implementation. Currently SSE3 is only available in the latest Pentium 4 CPUs (known as the Prescott core) and adds a number of additional operations for working with floating point vectors.

SSE2/3 is a very similar technology to Altivec. They are both 128-bit registers allowing the same amount of data per vector. Crucially, however, SSE2 adds support for 64-bit variables (ie double precision floats), which are missing from Altivec.

The instruction set of SSE has changed at each version and by SSE2 contained the most common operations used by Altivec developers as well as a number of additional

arithmetic instructions (divide, square root etc). Whilst SSEs instruction set, even in its newest form, is only 60 to 70% of Altivec's size, Intel, by being later to market with SSE, have had the advantage of seeing what functions are used, what functions are redundant and what extra functions are required.

Intel has not been as thorough documenting SSE as Apple have been with Altivec and this can cause delays when first attempting to utilise this. Interestingly, since the announcement that they would be switching to Intel x86 CPUs, Apple have released a large amount of documentation (more than Intel) that is primarily aimed at Altivec developers migrating their code, however, it is also an excellent starting point for new users of the system. Similar to Apple, Intel has release intrinsics for SSE on both the GNU and Intel C Compilers (GCC and ICC).

With each revision of SSE, Intel has also altered the hardware unit that performs SIMD processing. With SSE2 and 3 the parallel processing capabilities of the CPU have been improved providing significant performance improvements over Intel's earlier efforts.

Practical Comparison

With the goals of learning SIMD programming and performing a comparison of two SIMD implementations, a practical evaluation was required. In this case a program was written and optimised in the same way for both Altivec and SSE(2). The program was then benchmarked on various platforms and the results compared. This section will outline this program and analyse the results from it.

Method

Algorithm

Ideally for a program to show worthwhile improvement from SIMD optimisation, it must be repetitive in its method. An image manipulation program / filter, as was discussed earlier, was considered, however, due to the large amount of additional knowledge

required for this type of application, another algorithm was chosen. There are many areas where looping with a large number of iterations is required and for this research a program to calculate an approximate value of pi was chosen.

When calculating pi there are any number of algorithms that maybe chosen. The series that was used within this project is one of the simplest and least efficient for calculating pi. This inefficiency, however, is desirable in a program such as this that requires a large number of iterations. The series is as follows:

$$1-1/3+1/5-1/7+1/9-1/11 \ldots \approx \pi/4$$

Over 128,000 iterations, the value of pi using this series is still only accurate to 6 decimal places making it an excellent algorithm to use in this project.

In terms of SIMD instructions, this algorithm will test the following:

- STORE and LOAD operations that are performed at the beginning and end of the loop
- Divide
- Add
- Subtract

The algorithms primary loop will only be using the 3 arithmetic operators in the list; however, as these are some of the most common tasks that are asked of SIMD, it still gives a realistic demonstration of each instruction set.

Implementation

In all, the algorithm mentioned above, was implemented three times:

1. A version that uses the CPU alone in a SISD manner. This is to show the performance improvement given by the particular SIMD implementation. This code is architecture independent.
2. A version optimised for Altivec on the PowerPC chip
3. A version optimised for SSE2 on Intel (x86) chips

It is interesting to note that the code for the two optimised versions (Altivec and SSE2), is nearly, instruction-for-instruction, identical. There were only 2 real differences:

1. The names of the SIMD operations being used
2. The data loading / unloading method. These, however, could be made identical if desired as Altivec allows 2 methods of loading data into vectors, only 1 of which is used in SSE.

This unexpected similarity means that comparison of results will be very accurate for the instructions used.

Initial testing on each architecture showed that if the number of iterations was approximately 128,000,000, then execution time would be between 1 and 10 seconds, giving enough variance to make reasonable comparisons. Due to the single precision float limitation of Altivec, accurately calculating pi with a large number of decimal places would be difficult. Whilst this limitation could be overcome, it would require the algorithms for Altivec and SSE to differ, possibly influencing performance. In order to overcome this problem, it was decided that instead of performing the main loop 128,000,000 times, a loop of 128,000 iterations would be performed 1000 times. 128,000 iterations using this series will give a value of pi that is still accurate using single precision floats.

Testbeds

To show the difference between SIMD implementations, testing across as many platforms as possible was required. Not only does this test how Altivec performs against

SSE, but also how other factors such as CPU speed and SIMD revision affect performance.

In all, 8 different configurations were used for testing:

1. Pentium 4 (SSE3) 2.80Ghz on linux (Ubuntu)
2. Pentium 4 (SSE3) 2.80Ghz on OSX (Dev)
3. Pentium 4 (SSE3) 2.80Ghz @ 1.40ghz on linux (Ubuntu)
4. Pentium 4 (SSE3) 2.80Ghz @ 1.40ghz on OSX (Dev)
5. Pentium 4 (SSE2) 2.00ghz on linux (Ubuntu)
6. Quad Xeon (SSE3) 3.10Ghz on linux (Gentoo)
7. Dual G5 (Altivec) 2.7Ghz on OSX (10.4.3)
8. G5 (Altivec) 1.4Ghz on OSX (10.4)

Notes:

- Two versions of linux were used, as the configuration of the Quad Xeon machine could not be altered. Both configurations were using 2.6.x kernels.
- The x86 configurations using OSX were a 'non-official' development build from Apple.
- The compiler used on each platform has a very direct impact on the executables performance. Where possible gcc-4.0 was used however gcc-3.x was used on each of the Ubuntu linux configurations
- Whilst every effort was made to disable auto-vectorisation where possible, the results still show that this has been utilized by the compiler in some instances.

Results

The results shown in this section are all an average of 5 runs on each configuration. This was done to reduce interference from outside processes. It was found, however, that the results did not vary greatly between runs on any operating system.

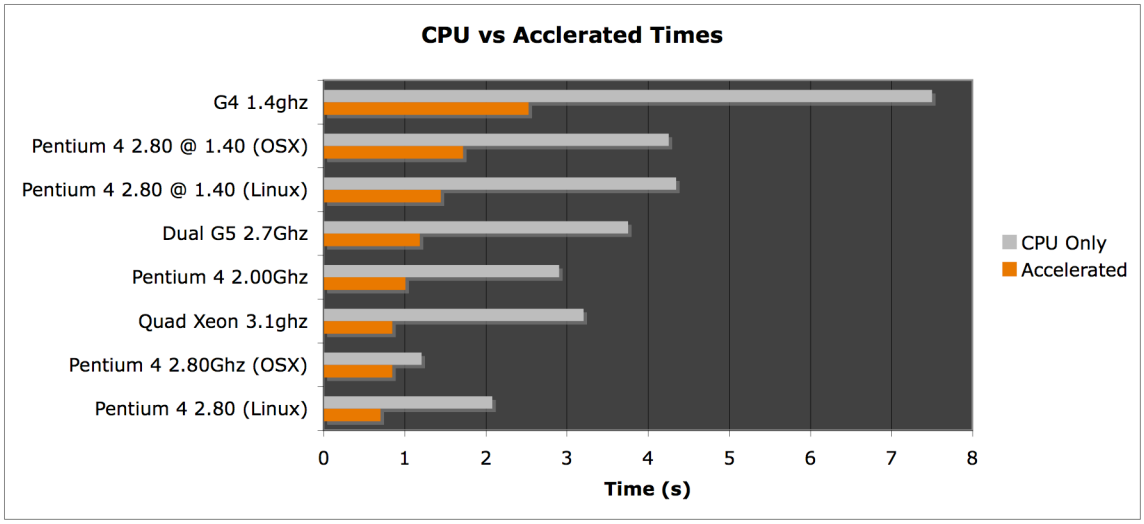

Figure 1 – Combined Results

Figure 1 shows both the raw CPU and optimised results for each of the tested configurations. As expected, a significant performance improvement was made with each SIMD optimised version when compared with the CPU score on the same set-up.

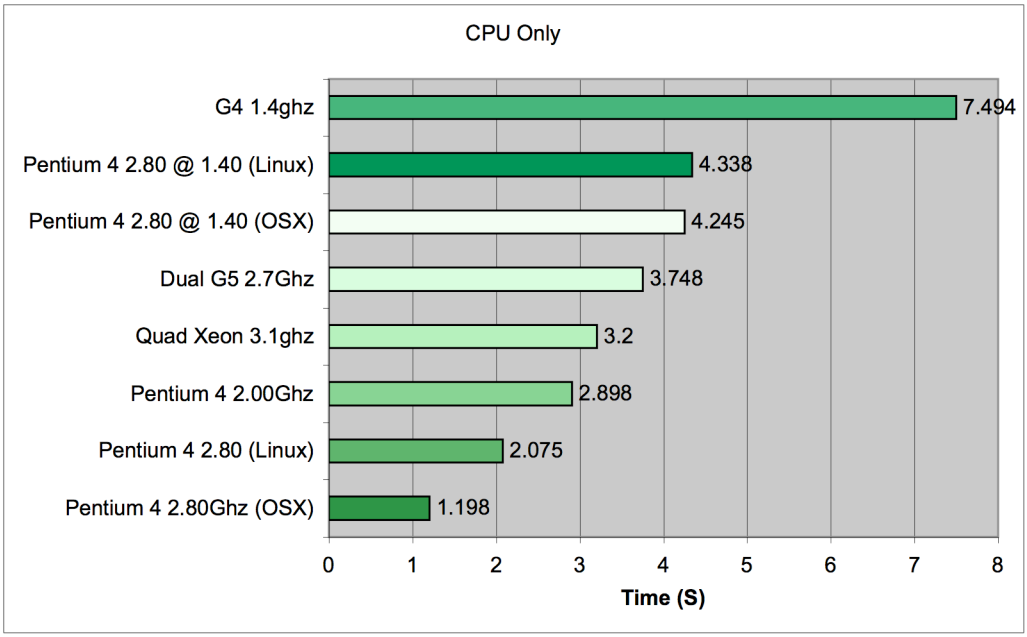

Figure 2 – CPU only results

Figure 2 shows the scores from each configuration using only the CPU. Unsurprisingly, clock speed makes the biggest difference in these results, as can be seen by the slowest 3 results coming from the 3 lowest clock speed configurations. These results also demonstrate the significant difference in performance when the same CPU is running at different clock speeds.

Figure 2, however, also shows 3 results that require further comment:

1. Both of the multi CPU machines (Dual G5 and Quad Xeon) gave results that were significantly slower than expected times for their clock speeds. No certain explanation for this could be found during testing in this project.
2. The PowerPC configurations performed noticeably slower than the x86 set-ups of similar clock speed. This is partly due to a lower clock speed but also because of the Intel chips focus on raw processing speed as opposed to the high throughput in PowerPC.

3. The Pentium 4 2.80ghz performed significantly faster under OSX than under linux. This is most likely due to auto-vectorisation occurring when the program is compiled by gcc-4.0 in the development build of OSX for x86.

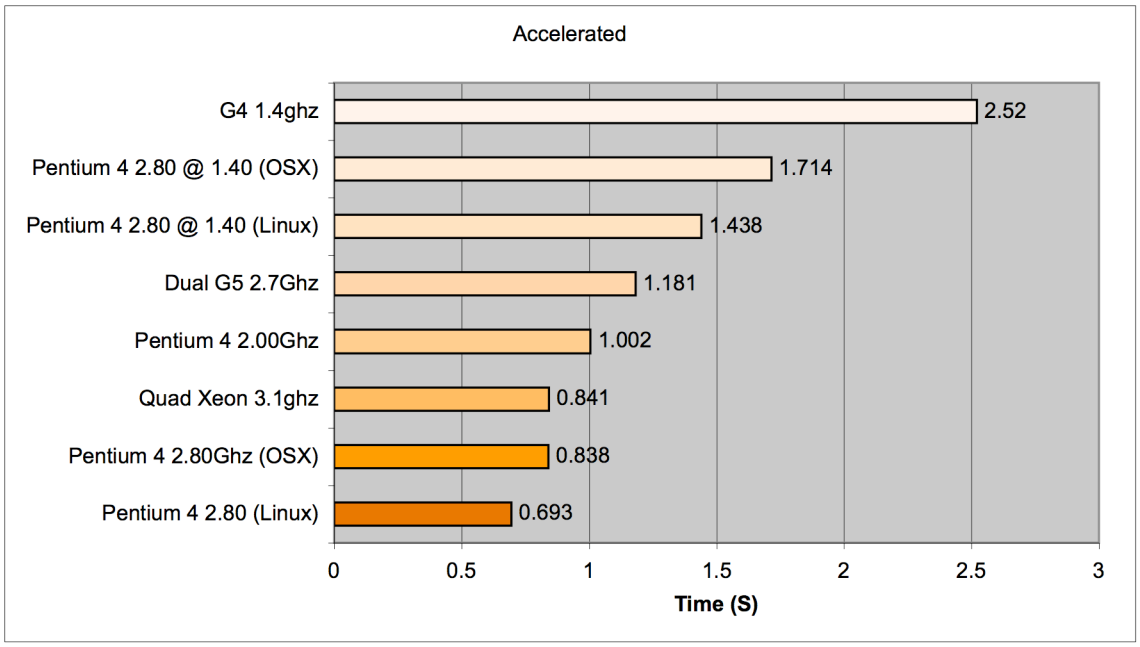

Figure 3 – Accelerated results

Figure 3 shows the results from the SIMD accelerated benchmarking. All times were significantly faster than the raw CPU score on the same configuration showing how SIMD instructions, at least in this 'ideal' scenario, do help improve performance by a large amount. On average, the 2 configurations using Altivec showed a greater increase from SIMD optimisations being ~203% faster compared to the SSE chips being ~175% faster, however, these chips performed poorly when compared to other configurations of a similar clock speed.

In cases where auto-vectorisation did not affect results, the version of GCC being used did not appear to make a noticeable difference.

Figure 3 also shows that chips containing SSE units are typically faster than those with Altivec for any given clock speed. This is particularly true for units with SSE3 capabilities. This does not show that SSE is fundamentally faster than Altivec in every case, however, it means that using the current implementations from Apple (IBM) and Intel and the instructions required by this algorithm, SSE gave significantly better performance.

Finally in the instances where comparison between OSX and linux was available, linux showed small but consistent improvements in performance. Unfortunately time and hardware resource limitations meant that I could not test on a PowerPC based system running linux.

Conclusion

This project aimed to serve as a way of learning to program with SIMD optimisations with a practical comparison of Altivec and SSE being used to demonstrate this. This paper has shown the results of the project in both theoretical and practical terms.

The results of this project show that whilst the Altivec may have a more mature instruction set and better accompanying documentation, its performance is no longer better than SSE. With SSE2 and now SSE3, nearly any program utilising Altivec code can be rewritten in SSE with 80%-100% instruction matching and will most likely see a worthwhile performance increase. This is particularly relevant for Apple developers preparing their applications on the new Intel based Macs.