RESULTS AND DISCUSSION^{*}

Anthony Blake

This work is produced by The Connexions Project and licensed under the Creative Commons Attribution License[†]

In order to test the hypotheses set out in Introduction¹, SFFT was benchmarked alongside FFTW and other libraries on a wide range of machines, as per the methods set out in Benchmark methods². The majority of the data was collected on Linux machines populated with SSE capable Intel microprocessors, with some additional data collected on small set of AVX and ARM NEON machines. The results are divided into three sections: speed, accuracy and setup time, with an additional section detailing a model that predicts SFFT's performance for different configurations. Finally, the chapter concludes by relating the results to other work.

Modelstring	L1d	L2	L3
Intel(R) Pentium(R) 4 CPU 2.80GHz	16	512	-
Intel(R) Pentium(R) D CPU 3.00GHz	16	1024	-
Intel(R) Pentium(R) M processor 1000MHz	32	1024	-
Intel(R) Xeon(TM) CPU 2.40GHz	16	2048	-
Intel(R) Xeon(R) CPU E5335 @ 2.00GHz	32	4096	-
Intel(R) Xeon(R) CPU X5355 @ 2.66GHz	32	8192	-
Intel(R) Xeon(R) CPU E5430 @ 2.66GHz	32	6144	-
Intel(R) Xeon(R) CPU X5560 @ 2.80GHz	32	256	8192
Intel(R) Core(TM)2 CPU 6600 @ 2.40GHz	32	4096	-
Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz	32	4096	-
Intel(R) Core(TM)2 Duo CPU E6850 @ 3.00GHz	32	4096	-
Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz	32	6144	-
Intel(R) Core(TM)2 Duo CPU P8600 @ 2.40GHz	32	3072	-
Intel(R) Core(TM) i5 CPU 660 @ 3.33GHz	32	256	4096
Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz	32	256	8192

Table 1: Linux benchmark machines, listed with the size of each level of cache (in kilobytes)

Table 1 presents a summary of the Linux machines that were used to run benchmarks. The majority of the machines were functioning as either lab workstations or servers in a University environment. The

^{*}Version 1.2: Jul 15, 2012 10:37 pm -0500

 $^{^{\}dagger}\,http://creativecommons.org/licenses/by/3.0/$

 $[\]label{eq:linear} \begin{array}{l} \mbox{1"Introduction" < http://cnx.org/content/m43792/latest/>} \\ \mbox{2"Benchmark Methods" < http://cnx.org/content/m43804/latest/>} \\ \end{array}$

benchmarks took approximately 12 hours to run, and while efforts were made to reduce each machine's load to a minimum, there were still transient system processes, such as log rotations and backups during the night that have introduced noise into the results.

For the Linux benchmarks, both 32-bit and 64-bit statically-linked binaries for SFFT, FFTW 3.3 and SPIRAL were compiled with icc 12.0.5, gcc 4.4.5 and clang 1.1. For the OS X benchmarks, 32-bit and 64-bit binaries for SFFT, FFTW 3.3 and SPIRAL were compiled with icc 12.1.0, llvm-gcc 4.2.1 and clang 3.0. The builds of SFFT and FFTW 3.3.1 for iOS 5 on ARM NEON were compiled with Apple clang 3.0.

Several binary libraries were also benchmarked: Intel IPP 7 and Apple Accelerate. Because these libraries are only available in binary form, they are compared against the icc builds of SFFT, FFTW 3.3 and SPIRAL, because icc generally produced the fastest code.

1 Speed

The speed results are presented in subsections according to the SIMD extensions: SSE, AVX and ARM NEON.



Figure 1: Performance comparison between SFFT and FFTW 3.3 in estimate mode on SSE machines

1.1 SSE



Figure 2: Performance comparison between SFFT and FFTW 3.3 in patient mode on SSE machines



Figure 3: Performance comparison between SFFT and SPIRAL on SSE machines. Although SPIRAL is faster when compiled with clang 1.1, Figure 5 shows that SFFT is faster than SPIRAL when compiled with clang 3.0

Figure 1 summarizes the speed performance of SFFT against FFTW 3.3 running in estimate mode on Linux machines with SSE. Twelve heatmaps are used to present data from different configurations. The three rows in the grid correspond to the three different compilers used, while the four columns correspond to the four different architecture and floating-point precision pairs. Within each heatmap, the rows correspond to different machines, and the columns correspond to different sizes of transform (2¹ through to 2¹⁸). Shades of green indicate that SFFT is faster for a particular point of data, while shades of yellow through to red indicate that FFTW is faster; lighter shades indicate a small difference, while darker shades indicate a bigger difference in performance. The scale for the colour map is computed separately for each of the 12 heatmaps in the grid, so a particular colour in one heatmap is not directly comparable to the same colour in another heatmap; the colours are only meant to indicate differences within each heatmap.

Similarly, Figure 2 compares SFFT to FFTW 3.3 running in patient mode, and Figure 3 compares SFFT to SPIRAL. There are fewer columns in the heatmaps of Figure 3 because SPIRAL only computes single-threaded FFTs for sizes 2^1 through to 2^{13} .

1.1.1 FFTW 3.3 in estimate mode

Figure 1 shows that SFFT is faster than FFTW 3.3 running in estimate mode in almost all cases over a range of Intel x86 machines that implement SSE. The horizontal streaks of yellow-red that can be seen in some heatmaps are outliers and likely caused by transient system processes that were running while SFFT was being benchmarked. Similar streaks appear at the same locations in Figure 5 and Figure 3.

1.1.2 FFTW 3.3 in patient mode



Figure 4: Performance of FFTs on recent Sandy Bridge machines, with x86_64 SSE binaries. Compiler: icc (a) Core i7-2600, single-precision (b) Core i7-2600, double-precision (c) Core i5-2557M, single-precision (d) Core i5-2557M, double-precision

Figure 2 shows that SFFT is faster than FFTW 3.3 running in patient mode in the majority of cases over a range of Intel x86 machines that implement SSE. SFFT was generally slightly slower than fftw3-patient on older machines such as the Pentium 4's and the 1GHz Pentium M, while on the newer machines such as the Sandy Bridge based Core i7-2600 and the Nehalem based Core i5-660, SFFT was clearly faster than FFTW (see Figure 4). This could be explained by the fact that FFTW performs extensive instruction level optimizations, such as scheduling, and that the older processors have smaller instruction and trace caches.

1.1.3 SPIRAL



Figure 5: Performance of clang-compiled x86_64 SSE FFTs on an Intel Core2 Duo P8600 (a) Single-precision, clang 1.1 (b) Double-precision, clang 1.1 (c) Single-precision, clang 3.0 (d) Double-precision, clang 3.0

The last row of Figure 3 shows that SFFT is generally slower than SPIRAL when both libraries are compiled with clang 1.1. However, with more recent releases of clang, which do much more code optimization, the situation is reversed, as shown in Figure 5. In some cases SPIRAL compiled with clang 3.0 is slower than SPIRAL compiled with clang 1.1, while SFFT is generally faster when compiled with clang 3.0. This demonstrates that the speed of automatically tuned SPIRAL code is specific to certain compilers.

SPIRAL's double-precision performance is slightly better than SFFT when compiled with icc or gcc, while SFFT's single-precision code is faster than SPIRAL on recent machines, and of similar speed on older machines.

1.2 AVX

Of the machines that were used for benchmarks, only two supported AVX: the Macbook Air 4,2 with an Intel Core i5-2557M, and a Linux machine with an Intel Core i7-2600. Figure 6 shows that SFFT is clearly faster than FFTW up until about 1024 points, while performance between the two is similar for larger transforms.

Results for Intel IPP are also plotted in Figure 6, but only for the Core i7-2600. IPP did not detect the existence of AVX on the Core i5-2557M, and instead used SSE, as plotted in Figure 4. Apple vDSP does not support AVX, and so SSE vDSP results for the Macbook Air 4,2's Core i5-2557M are also plotted in Figure 4.



Figure 6: Performance of FFTs on recent Sandy Bridge machines, with x86_64 AVX binaries. Compiler: icc (a) Core i7-2600, single-precision (b) Core i7-2600, double-precision (c) Core i5-2557M, single-precision (d) Core i5-2557M, double-precision

1.3 ARM NEON



Figure 7: Performance of single-precision FFTs on ARM NEON devices running iOS. Compiler: Apple clang 3.0 (a) Apple A4 (ARM Cortex-A8) (b) Apple A5 (ARM Cortex-A9)

SFFT and FFTW 3.3.1 were compiled with Apple clang 3.0 and benchmarked on an Apple iPod touch 4G and an Apple iPad 2, which contain the Apple A4 and A5 SoCs respectively. The A4 implements the ARM Cortex-A8, while the A5 implements the ARM Cortex-A9, both of which support ARM NEON.

Figure 7 shows that SFFT is easily faster than FFTW on both devices. This contradicts Frigo and Johnson's claim that the performance of FFTW is portable, and tends to support the idea that it is possible to write fast and portable code without exhaustive searches through the configuration space of all possible FFTs.

A considerable amount of effort was needed to work around several problems that were encountered when targeting ARM NEON with Apple clang 3.0, and many of SFFT's primitive macros for NEON were written in inline assembly code. Among the problems encountered when targeting ARM NEON with Apple clang 3.0:

- 1. There is no way of explicitly specifying memory alignment when using vector intrinsics;
- 2. Fused multiply-add/subtract intrinsics do not currently compile to the correct instructions because of a bug in clang;
- 3. Clang's inline assembly front-end lacks the syntax and semantics to properly address the dual-size aliased vector registers.

The above problems affect all FFT libraries equally, and it seems that portability depends critically on the quality of the machine specific code and macros.

2 Accuracy



Figure 8: Accuracy of FFTs on an Intel Core i7-2600. SFFT, FFTW and SPIRAL were compiled for x86 64 with icc (a) SSE, single-precision (b) SSE, double-precision

The accuracy of each FFT was measured as per the methods in Benchmark methods³. The accuracy of single and double precision FFTs on an Intel Core i7-2600 is plotted in Figure 8, and shows that the relative RMS error for FFTW, SFFT and SPIRAL is within an acceptable range. Graphs for all other machines are similar.

3 Setup time



Figure 9: Setup times of FFTs on an Intel Core i7-2600. SFFT, FFTW and SPIRAL were compiled for x86 64 with icc (a) SSE, single-precision (b) SSE, double-precision

Figure 9 shows that FFTW, in patient mode, requires several orders of magnitude more time to initialize as it searches for a fast FFT configuration. SPIRAL has a very fast setup time, because it is entirely

 $^{^3}$ "Benchmark Methods" <http://cnx.org/content/m43804/latest/>

statically elaborated and needs no dynamic initialization. The setup time for SFFT is comparable to FFTW in estimate mode, though SFFT's setup time begins to increase for transforms larger than 8192 points. This is likely because of repeated calls to the complex exponential function as twiddle factor LUTs are elaborated; no effort was made to optimize this setup code, and it is likely that it would be much faster if the calls to the complex exponential function were optimized.

Graphs for all other machines are similar.

4 Binary size

Compared to other libraries, SFFT produced larger binaries for the benchmarks, because there is currently no optimization performed between transforms contained in the same library. For 64-bit single precision binaries on OS X with AVX, the size of the SFFT benchmark was approximately 2.8 megabytes while the size of the FFTW benchmark was 1.8 megabytes.

5 Predicting performance

For each size of transform on a particular machine, SFFT chooses the fastest configuration from a set of up to eight possible configurations. Small transforms have only one option, which is a fully hard-coded transform, while larger transforms have up to eight, which could include the four-step transform, and several variants of the hard-coded leaf transform, where each variant corresponds to a particular size of leaf sub-transform and size of body sub-transform, and for size-16 leaf sub-transforms, a streaming store variant is included too. The decision of exactly which configuration to use depends on the size of transform, the compiler, and the characteristics of the host machine.

For the benchmarks in this chapter, SFFT used a calibration routine to choose the fastest configuration. The calibration data was collected, along with some data about the machine and the compiler, and used to train a classifier.

The data was processed into instances, with each instance having attributes for the size of the transform and the precision, the size of each level of cache, the architecture and micro-architecture of the machine, the SIMD extensions, the OS, the compiler used, and the CPU frequency. In total there were 3348 instances of data, each of which had 12 attributes.

Weka [9] was used to experiment with several classifiers, and a REPTree classifier with bagging was used to train a model. Using 10-fold cross-validation, the model correctly classified 76.1% of the instances with a weighted average precision of 74.8%, which tends to confirm the existence of a relationship between the characteristics of the machine and the performance of a particular FFT configuration.

The accuracy of the classifier is promising, and it has the potential to replace the calibration code in SFFT. It is highly likely that if the noise in the data was reduced through the use of an isolated benchmarking environment, the accuracy of the classifier would increase. The accuracy would also likely benefit from a larger dataset collected from a larger range of benchmark machines.

6 Split-radix vs. conjugate-pair



Figure 10: Ordinary split-radix versus conjugate-pair split-radix on an Intel Core i5-2557M. SFFT, FFTW and SPIRAL were compiled for x86_64 with icc (a) SSE, single-precision (b) SSE, double-precision

In order to quantify the gain in performance that might be attributable to the use of the conjugate-pair algorithm, SFFT was retrospectively modified to compute the FFT using the ordinary split-radix algorithm as well as the conjugate-pair algorithm. The results of benchmarks between the two algorithms, as well as FFTW and SPIRAL, are plotted in Figure 10.

Unexpectedly, the ordinary split-radix algorithm is faster than the conjugate-pair algorithm for some smaller sizes of transform, but for transforms above a certain size, the conjugate-pair algorithm is faster by a few hundred MFLOPS.

The performance advantage of the ordinary split-radix algorithm for smaller sizes of transforms is likely due to shorter chains of dependent instructions where twiddle factors are loaded and used. Consider that the ordinary split-radix algorithm separately loads two twiddle factors into two registers, and there are no dependencies between these instructions, while the conjugate-pair algorithm must load one twiddle factor and then duplicate it into another register, which does result in dependent instructions. Thus the ordinary split-radix algorithm is faster for smaller transforms where memory bandwidth is not the limiting factor, but when memory bandwidth does become the limiting factor, the conjugate-pair algorithm is faster.

In future, SFFT could exploit the performance advantage of the ordinary split-radix algorithm when computing smaller sizes of transforms.

7 Applications of this work

This section provides an overview of how the techniques presented in this thesis may be applied to the prime-factor algorithm, sparse Fourier transforms, and multi-threaded transforms.

7.1 Prime-factor algorithm

The techniques presented in this work rely on the fact that FFTs operating on signal lengths that are a power-of-two can be factored into smaller power-of-two length components, which are computed in parallel by being evenly divided into a number of SIMD vector registers that are a power-of-two length.

The prime-factor algorithm factors other lengths of FFTs into components that are co-prime in length, and ultimately small prime components, which do not evenly divide into the power-of-two length SIMD registers, except in the special case where a SIMD register contains only one complex element (such is the case with double-precision on SSE machines).

Because the prime components do not evenly divide into power-of-two length SIMD registers, the algorithm level vectorization techniques presented in this work are not directly applicable. In contrast, the auto-vectorization techniques used in SPIRAL [2], [6], [7] are performed at the instruction level, and are applicable to the prime-factor algorithm, but as the results in Figure 4 show, the downside of SPIRAL's lower level approach is that performance for power-of-two transforms scales poorly with the length of the SIMD register.

7.2 Sparse Fourier transforms

The recently published Sparse FFT [4], [3] will benefit from the techniques presented in this work because the inner loops use small DFTs (e.g, 512 point for a certain 256k point sparse FFT), which are currently computed with FFTW. Replacing FFTW with SFFT will almost certainly result in improved performance, because SFFT is faster than both FFTW and Intel IPP for the applicable small sizes of transform on an Intel Core i7-2600 (see Figure 6).

Version 2.0 of the Sparse FFT code is scalar, and would benefit greatly from explicitly describing the computation with SIMD intrinsics. However, a key difference between the sparse Fourier transform and other FFTs is the use of conditional branches on the input signal data. This has performance implications on all machines, but it is worth noting that some machines will be drastically affected by this, such as the ARM Cortex-A8, where the SIMD pipeline is located behind the main pipeline, resulting in fast transfers from the main CPU unit to the SIMD pipeline, but large penalties when SIMD registers or flags are accessed by the main CPU unit.



7.3 Multi-threaded transforms

Figure 11: Speed of multi-threaded four-step algorithm running on an Intel Core i5-2557M with four threads. The algorithm decomposes transforms into smaller single-threaded components, which are computed above with three different implementations. All code was compiled with icc for x86_64 with SSE.

MatrixFFT has recently shown that the four-step algorithm [1], designed to efficiently use hierarchical or external memory on Cray machines in the 1980's, is useful for computing large multi-threaded transforms on modern machines, providing performance far surpassing that of FFTW's multi-threaded performance [8].

The four-step algorithm decomposes a transform of size N into a two-dimensional array of size $n_1 \times n_2$ where $N = n_1 n_2$, and $n_1 = n_2 = \sqrt{N}$ (or close) often obtains the best performance.

The four-steps of the algorithm are:

- 1. Compute n_1 FFTs of length n_2 along the columns of the array;
- 2. Multiply each element of the array with ω_N^{ij} , where i and j are the array coordinates;
- 3. Transpose the array;
- 4. Compute n_2 FFTs of length n_1 along the columns of the array.

Each step can be divided amongst a pool of threads, with a synchronisation barrier between the third and fourth steps. The transforms in steps one and four operate on sequential data, and if they are small enough, they are not subject to bandwidth limitations (and if they are not small enough, they can be further decomposed with the four-step algorithm until they are small enough). The bandwidth bottleneck does not disappear, but it is factored out into the transpose in step three, and because of this, the performance of the small single-threaded 1D transforms used in steps one and four correlate with the overall multi-threaded performance. A simple multi-threaded implementation of the four-step algorithm was benchmarked with SFFT and FFTW transforms, and the results are shown in Figure 11, which tends to confirm that the performance of single-threaded transforms for steps one and four translates to the overall multi-threaded performance when using the four-step algorithm.

8 Similar work

Aside from Bernstein's FFT library, which was designed in the days of scalar microprocessors and has not been updated since 1999, there have been a few other challenges to the automatically adaptive approach of FFTW, but none present concrete results that definitively dismiss the idea. Most recently, Vasilios et al. presented an approach that uses the characteristics of the host machine to choose good FFT parameters at run time [5], but their approach has several issues that render it almost irrelevant. First, the approach uses optimizations that only apply to scalar machines, viz. twiddle factor symmetries are exploited to compress the twiddle LUTs, and arithmetic is avoided when twiddle factors contains zeros or ones. The vast majority of microprocessors, even those found in mobile devices such as phones, feature SIMD extensions, and so an approach that is limited to scalar arithmetic is of little consequence. Second, they benchmark the FFTs in a most unusual way. Rather than repeat a large number of iterations of the FFT, they repeat a large number of iterations of a binary that initializes and then executes only one FFT; such an approach is by no means representative of applications where the performance of the FFT is a concern, and is more a measurement of the initialization time rather than the FFT.

References

- D.H. Bailey. Ffts in external or hierarchical memory. In Proceedings of the 1989 ACM/IEEE conference on Supercomputing, page 2348211;242. ACM, 1989.
- [2] F. Franchetti and M. Puschel. A simd vectorizing compiler for digital signal processing algorithms. In Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM, page 208211;26. IEEE, 2002.
- [3] H. Hassanieh, P. Indyk, D. Katabi, and E. Price. Nearly optimal sparse fourier transform. Arxiv preprint arXiv:1201.2501, 2012.
- [4] H. Hassanieh, P. Indyk, D. Katabi, and E. Price. Simple and practical algorithm for sparse fourier transform. In Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, page 11838211;1194. SIAM, 2012.
- [5] Vasilios I. Kelefouras, George Athanasiou, Nikolaos Alachiotis, Harris E. Michail, Angeliki Kritikakou, and Costas E. Goutis. A methodology for speeding up fast fourier transform focusing on memory architecture utilization. *IEEE Transactions on Signal Processing*, 59(12):6217–6226, 2011.
- [6] S. Kral, F. Franchetti, J. Lorenz, and C. Ueberhuber. Simd vectorization of straight line fft code. Euro-Par 2003 Parallel Processing, page 2518211;260, 2003.
- [7] S. Kral, F. Franchetti, J. Lorenz, C. Ueberhuber, and P. Wurzinger. Fft compiler techniques. In *Compiler Construction*, page 27258211;2725. Springer, 2004.
- [8] J. Klivington R. Crandall and D. Mitchell. Large-scale ffts and convolutions on apple hardware. Technical report, Apple Inc Advanced Computation Group, 2009.
- [9] I.H. Witten and E. Frank. Data Mining: Practical machine learning tools and techniques. Morgan Kaufmann, 2005.