18.337 Project: FFTW and the SiCortex Architecture

Po-Ru Loh

May 19, 2008

1 Introduction

The goals of this project were:

- to study parallel fast Fourier transform (FFT) algorithms in general;
- to understand the performance and limitations of the FFTW package currently provided on SiCortex supercomputers; and
- to investigate novel ways of optimizing FFTs to exploit the peculiarities of SiCortex machines.

To provide a little background, the three current versions of FFTW (as of May 2008) are as follows.

- FFTW 2.1.5 is the latest version of the package that supports distributed-memory parallel transforms using MPI.
- FFTW 3.1.2 is the latest version for serial and multithreaded transforms. It is a complete rewrite of the package, featuring a new API and an improved planner and codelet generator; however, it does not support MPI transforms.
- FFTW 3.2alpha3 is the version of FFTW currently under development. Among other improvements, it provides MPI support and much faster parallel transpose algorithms; however, it still has bugs and a release date is uncertain.

I have structured the bulk of this report to follow my work and findings more or less chronologically. There are many possible directions for future study, some of which I hope to pursue: at present, this project is still very much work in progress!

2 Experiments, Challenges, and Lessons

The starting point of my work was the following (excerpted) email from a SiCortex programmer who had discovered unusual behavior in their FFTW builds:

> Part of my work here at SiCortex is to work with the FFT libraries we provide to our customers .... When I do a comparison of performance of the serial version of FFTW 2.1.5 and 3.2 alpha, FFTW 3.2 alpha is significantly slower.
>
> Both codes are compiled with the same compiler flags. I am running the 64 bit version:

```
CC=scpathcc LD=scld AR=scar RANLIB=scranlib F77=scpathf95
CFLAGS="-g -O3 -OPT:Ofast -mips64" MPILIBS="-lscmpi"
./configure --build=x86_64-pc-linux-gnu
--host=mips64el-gentoo-linux-gnu --enable-fma

FFTW 3.2alpha
bench -opatient -s icf2048x2048
Problem:  icf2048x2048, setup:  3.79 s, time:  4.21 s, "mflops":
109.7

FFTW 2.1.5
Please wait (and dream of faster computers).
SPEED TEST: 2048x2048, FFTW_FORWARD, in place, specific
time for one fft:  2.914490 s (694.868565 ns/point)
"mflops" = 5 (N log2 N) / (t in microseconds) = 158.303319
```

> One thing I should point out is that FFTW 2.1.5 has been modified from its original stock version. It has had some optimizations based on the work by Franchetti and his paper "FFT Algorithms for Multiply-Add Architectures." From my understanding these optimizations have been folded into the genFFT code generator for FFTW 3.*.*
>
> Am I correct in my assumption that FFTW 3.2 should be significantly faster?

While the above results were for serial FFTs (which were not really a focus of my project), fast serial transforms might be an important tool to have in optimizing parallel transforms due to the recursive nature of Cooley-Tukey algorithms. I therefore began by trying to replicate these results on our own SC648 system. Seeing as the tests appeared to have been generated by a pre-existing automated benchmarking routine, I searched for and found the the testing code in the FFTW package.

Unfortunately, I also quickly discovered that the benchmarking code distributed with FFTW was not an independent entity that could be compiled separately and then linked with an FFTW build of choice. Instead, it had its own headers and dependencies embedded in the source tree of the package, making the task of creating a binary difficult to extract from that of building the entire package. This was a bit of a dilemma given that the whole idea was to test the precompiled (and SiCortex-optimized) libraries provided with the machine. In the end I decided to compromise and attempt to download the whole package, build it (with the default configuration options, since I didn't know anything smarter to use), and then hijack the last linking step to relink against the "real" SiCortex version of FFTW.

2.1 Some surprises

My initial tests did not find a discrepancy nearly as big as those reported earlier (for a 2048x2048 complex transform):

Version	Reported	Found
fftw-2.1.5	155 mflops	140 mflops
fftw-3.1.2		120 mflops
fftw-3.2alpha3	110 mflops	130 mflops

Of course, these results were simply using the downloaded version of the libraries without SiCortex tweaks of any sort in code or compiler flags, and also with the gcc compiler versus PathScale.

Strangely, I then tried re-linking `fftw_test` and `bench` using the precompiled fftw libraries that came on the machine and got a substantial slowdown!

Version	Reported	Found	With SC build
fftw-2.1.5	155 mflops	140 mflops	45 mflops
fftw-3.1.2		120 mflops	
fftw-3.2alpha3	110 mflops	130 mflops	90 mflops

One other thing I noticed was that the configure script for FFTW 3 complained about not finding a hardware cycle counter; on the other hand, FFTW 2 didn't produce this warning. Seeing as I had compiled with default flags in both cases, I guessed that only FFTW 3 used a cycle counter and could thus be taking a performance hit. According to the SiCortex docs, however, the MIPS processors in their machines did in fact have cycle counters, leaving me the questions of (a) how to tell FFTW 3 of their existence; and (b) whether or not the SiCortex build of FFTW 3 was using them.

2.2 More serial tests

While waiting for a response from SiCortex, I decided to run a few more tests to gauge the performance of FFTW and understand the peak capabilities of our machine.

First, I was very curious to see how FFTW compared to a "standard" FFT implementation: how substantial was the performance improvement obtained from its self-tuning? I quickly realized that there was no such thing as a "standard" implementation, however: even though most codes available on the web used some variant of Cooley-Tukey, there is an enormous spectrum of possible optimizations, and any given code performs some of these optimizations but foregoes others. (This, apparently, was the observation that engendered the self-tuning of FFTW.) In the end, I simply downloaded the `kube-gustavson` implementation off "Jorg's useful and ugly FFT page" that claimed to be quite fast yet simple: only a little over 500 lines of code in one source and one header.

The following table shows test results from an initial compile, a compile using the optimization flag `-O3`, and a compile using optimizations and linking against the SiCortex optimized math library.

Transform size	Time:	-O3 time:	-O3 -lscm time:
1024x1024 rows	2.040359 sec	0.555099 sec	0.540417 sec
1024x1024 all	4.149302 sec	1.172668 sec	1.143589 sec
2048x2048 rows	8.482921 sec	2.321096 sec	2.260734 sec
2048x2048 all	17.633655 sec	5.325778 sec	5.208607 sec
4096x4096 rows	37.851723 sec	11.917786 sec	11.729611 sec
4096x4096 all	80.187843 sec	28.651167 sec	28.277497 sec

The extent of the speed increase from `-O3` surprised me, albeit perhaps simply because I was too used to thinking in terms of algorithmic speedup versus compiler optimization. The discrepancy between the amounts of time taken to transform the rows and columns is also noteworthy: for small transforms, row time is roughly half the time taken for the entire 2D FFT, but as the size of the transform increases, and in particular once optimizations are used, the column time becomes substantially greater than the row time. This is as expected given the noncontiguous memory access required to transform columns, but it was still nice to have some hard numbers.

Dividing out by $5N \log N$, the "mflops" number for 2048x2048 comes to about 90, showing that FFTW does indeed provide substantial speedup, but not beyond 2x in this case—again, a useful ballpark figure to keep in mind.

As a side note, it also finally occurred to me at this time that performance on the order of 100-200 "mflops" was incredibly slow! Indeed, looking at the benchmark graphs on the benchFFT page, typical modern processors achieved figures in the 1000-10000 "mflop" range. What was happening here?

A quick look at the SiCortex spec sheet provided the answer: 500 MHz MIPS processors simply cannot compete with 2-3 GHz PC processors in a serial computation. This observation was obvious in retrospect, but it underscored the fact that SiCortex machines were really built for parallel. Our real interest in their system was seeing what it might allow us to do not one processor but 500 or 5000 processors.

2.3 Parallel tests

With this in mind, I moved on to testing the capabilities of our SC648 in parallel tasks. An initial question I wanted to answer was what the machine had to offer in terms of relative speeds of communication and computation. Mainly I wanted a clearer picture of where to look for parallel FFT speedup. I ran some tests both on basic communication (and memory) operations and on FFTW, and a couple numbers I got were once again a little surprising.

- Point-to-point communication speed for MPI Isend/Irecv nears 2GB/sec as advertised for message sizes in the MB range. Latency is such that a 32KB message achieves half the max.

- The above speeds seem to be mostly independent of the particular pair of processors which are trying to talk, except that in a few cases—apparently when the processors belong to the same 6-CPU node—top speed is only 1GB/sec. This seems quite counterintuitive and is the exact opposite of what I expected!

  Moreover, I had hoped to see some evidence of the Kautz network used in the interconnect fabric: conceivably there might be room for optimization in the distribution and transfer of data on this network. However, at least based on this test, no processors (aside from those sharing the same node) appeared to be any farther apart than others.

- Upon introducing network traffic by asking 128 processors to simultaneously perform 64 pairwise communications, communication speed drops to around 0.5GB/sec with substantial fluctuations.

- For comparison, the speed of memcpy on a single CPU is as follows:

Size (bytes)	Time (sec)	Rate (MB/sec)
256	0.000000	1152.369380
512	0.000000	1367.888343
1024	0.000001	1509.023977
2048	0.000001	1591.101956
4096	0.000003	1421.082913
8192	0.000006	1436.982715
16384	0.000010	1638.373220
32768	0.000097	337.867886
65536	0.000194	337.250067
131072	0.000413	317.622002
262144	0.001054	248.758781
524288	0.002281	229.862557
1048576	0.004371	239.917118
2097152	0.008626	243.109613

- The table above shows clear fall-offs when the L1 cache (32KB) and L2 cache (256KB) are exceeded. What really surprised me, however, was that according to the previous experiments, sending data to another processor is as fast as memcpy even in the best-case scenario (memcpy staying within L1), and is many times faster once the array being copied no longer fits in cache! This seems very strange: don't the MPI operations also have to leave cache? How can point-to-point copy be so much faster?

2.4 Some answers from a trip to Maynard

A visit to the SiCortex office answered several of the questions raised above. First, part of the mystery behind the slowness of FFTW 3 was revealed: apparently at least one of the builds being tested had been compiled with the `--enable-long-double` flag set, resulting in a huge performance hit. (The long double data type is 10 bytes and could cause problems with efficient memory alignment, aside from being slower to compute with.) After this issue was fixed, performance still lagged FFTW 2 by a factor of nearly 2, however. The missing cycle counter was certainly in part responsible—in fact, without it, FFTW 3 does no timing whatsoever but instead tunes based on a heuristic—but from previous tests I expected the impact to be more on the order of 10-20%, not 2x. Still, it made sense to fix this problem before doing too much more speculation.

A few more of the oddities had I discovered were also explained. The strange relative slowness of on-node communication was apparently due to the fact that whereas the Kautz network connects any two distinct nodes via three disjoint paths, data traveling between processors on a single node can only follow one path. (This is probably an oversimplification and probably not an accurate one due to my own superficial understanding, but in any case, the behavior was known and not an anomaly of my testing code.) Additionally, memcpy was also known to be slower than one would hope: from what I gathered—with the same caveat as above—the MIPS processors only have one address register and do not prefetch, so transfer from one memory location to another takes a hit.

2.5 Follow-up questions and answers

Returning finally to algorithms, one question we all had was which decompositions the FFTW planner typically chooses. Going home and snooping around a bit in the code, I found out that turning on the "verbose" option (`fftw_test -v` for 2.x and `bench -v5`, say, for 3.x) outputs the plan. Based on a handful of test runs on power-of-2 transforms, FFTW 2.1.5 seems to prefer:

- radix 16 and radix 8, for transform lengths up to about 2^{15};
- primarily radix 4 but finishing off with three radix 16 steps, for larger transforms.

There are also occasional radix 2 and 32 steps but these seem to be rare.

In practice, FFTW 2 for 1D power-of-2 transforms seems to just try the various possible radices up to 64—nothing fancier than that. On the other hand, FFTW 3 tries a lot more things, including a radix-$\sqrt{n}$ first step. Finding out what FFTW 3 prefers will have to wait until we have a cycle counter, however: at the moment, the planner information is meaningless.

Running a few more tests on smaller transforms, I also realized that the SiCortex-optimized build for FFTW 2.1.5 was really doing quite well. Although initially we weren't totally satisfied with the numbers we were getting, the transform we were running was fairly large, and in fact on optimal sizes (around 1024), peak performance is over 500 "mflops." Basically, there's a falloff in performance as soon as the L1 cache is exceeded and another one once we start reaching into main memory. All of the graphs on the benchFFT page show these falloffs; indeed, getting a peak "mflops" number slightly better than the clock rate is about the best anyone can do (on non-SIMD processors).

Summarizing, I concluded that FFTW 2 is pretty well-optimized (although the library currently distributed with SiCortex machines is not!) and our focus should really be on FFTW 3. Serial performance of FFTW 2 is good, and as for parallel transforms, the algorithms that FFTW 2 uses are slow and not worth optimizing further. On the other hand, FFTW 3.2alpha already implements several improvements—the first three or four possibilities that came to mind for me were all already there—and despite being in alpha, it's in good enough shape to start playing with once we have a cycle counter.

2.6 A few more words on parallel transforms in FFTW

It is worth pointing out the similarities and differences between the approaches used by FFTW 2 and FFTW 3.2alpha for parallel transforms. Both employ the basic algorithm that alternates between 1D FFT steps and transpose steps (three for 1D transforms and two for 2D transforms, with possible savings if the input or output is allowed to be scrambled). However, on 1D transforms, FFTW 2 chooses the initial radix as close to $\sqrt{n}$ as possible (i.e., viewing the data as a square), whereas FFTW 3.2alpha tries to choose the radix equal to the number of processors if possible.

FFTW 3.2alpha also considers the possibility of skipping local transposes, instead giving the planner the option of deciding whether or not to perform them (to improve memory locality at the cost of the time spent transposing). Additionally, the possibility of doing transposes in stages to reduce the number of pairs of processors that must communicate (at the cost of increased total data movement) is being explored.

2.7 Next Steps

As of a few days ago, I now have a `cycle.h` file to play with thanks to Jud Leonard at SiCortex. Hopefully the new data this provides us from the FFTW 3 planner will give a better indication of which ideas for faster parallel transforms are most promising. I hope to implement and test some possible improvements over the summer.