Optimizing the NPB FT benchmark for multi-core AMD Opteron microprocessors

Stephen Whalen
Cray, Inc.

August 13, 2007

1 Description of FT

1.1 High-level description

FT approximates the solution to the parabolic partial differential equation $u_t = \alpha\nabla^2 u$ by separation of variables [2]. The Class D problem uses a $2048 \times 1024 \times 1024$ discrete Fourier transform (DFT) to approximate u in the transform space, and computes 25 timesteps, each of which is transformed back into physical space [9].

1.2 Algorithm

The MPI parallel implementation requires that the number of processes be a power of two; other process counts could not lead to a well-balanced distribution of work. The processes are arranged in a 1D grid, and the global array is distributed along its last dimension [3]. Assuming p processes, this results in each process holding a $2048 \times 1024 \times (1024/p)$ array.

Conceptually, the distributed DFTs are computed according to a standard algorithm:

1. Compute length-2048 DFTs along the first dimension.
2. Compute length-1024 DFTs along the second dimension.
3. Transpose the first and third dimensions.
4. Compute length-(1024/p) DFTs along the (new) first dimension.

This leaves the transformed data in a transposed layout across the processes.

Backward DFTs are compute by reversing this procedure.

After the forward DFT, the time evolution operator is applied, followed a backward DFT to recover the approximate solution. This procedure (evolution followed by an inverse transform) occurs once per timestep.

1.3 Implementational details

Computing multiple identically-sized serial 1D DFTs at once, rather than each one individually, is a well-known DFT vectorization technique. With proper data layout, computing simultaneous multiple DFTs allows stride-1 access [7]. The NPB2.4-MPI implementation uses this method, computing the serial 1D DFTs in groups whose size is given by the variable `fftblock`. The base code contains no logic to deal with partial blocks, so `fftblock` must divide the size of each dimension of the *u* array. That is, `fftblock` must be a power of two.

Rather than computing DFTs of subarrays of leading dimension `fftblock`, the data is first copied into work arrays. This allows for array padding, as power-of-two leading dimensions could lead to cache thrashing. The work arrays have leading dimension `fftblockpad` $\geq$ `fftblock`, where `fftblockpad` should not be a power of two. The default values are `fftblock` = 16 and `fftblockpad` = 18.

Originally developed for vector supercomputers with interleaved memory, these techniques are directly applicable to modern microprocessors and their cache structures: stride-1 access equates to spatial locality to take advantage of full cache lines, and padded arrays avoid thrashing due to cache associativity.

The base implementation uses the Stockham back-and-forth algorithm [4, 8] for the multiple serial DFTs.

1.4 Profiles

Portions of function-level sampling profiles are shown in Tables 1 and 2. The `fftz2` subroutine is the clear hotspot in both cases.

Samp %	Cum. Samp %	Samp	Imb. Samp	Imb. Samp %	Function
100.0%	100.0%	6487811	--	--	Total
50.1%	50.1%	3248833	1467.98	2.9%	fftz2_
9.9%	59.9%	640567	614.14	5.9%	cffts1_
8.9%	68.9%	579189	1513.17	14.6%	PtlEQPeek
7.5%	76.3%	484264	254.38	3.3%	cffts2_
5.3%	81.7%	346901	375.67	6.6%	transpose2_local_
4.8%	86.5%	309669	601.42	11.2%	evolve_
3.9%	90.3%	251040	127.50	3.2%	transpose2_finish_
2.6%	92.9%	168300	301.31	10.4%	cfftz_
2.2%	95.2%	144761	640.11	22.4%	PtlEQGet
1.5%	96.7%	99550	455.53	23.0%	PtlEQGet_internal

Table 1: CrayPat sampling profile for FT Class D, 64 processes

Samp %	Cum. Samp %	Samp	Imb. Samp	Imb. Samp %	Function
100.0%	100.0%	7163068	--	--	Total
45.3%	45.3%	3242896	483.44	3.7%	fftz2_
13.9%	59.2%	998672	436.94	10.1%	PtlEQPeek
9.0%	68.2%	642862	216.82	8.0%	cffts1_
6.8%	75.0%	487337	237.34	11.1%	cffts2_
4.4%	79.4%	312700	184.52	13.2%	transpose2_local_
4.2%	83.6%	301485	227.32	16.2%	evolve_
3.6%	87.2%	260345	40.03	3.8%	transpose2_finish_
3.5%	90.7%	248259	282.24	22.6%	PtlEQGet
2.4%	93.1%	172737	309.25	31.6%	PtlEQGet_internal
2.3%	95.4%	165864	152.09	19.1%	cfftz_

Table 2: CrayPat sampling profile for FT Class D, 256 processes

2 Optimizations

2.1 Sizing for cache

We certainly expect degraded performance should each block of u spill out of cache. Indeed, this happens with the default `fftblockpad`. In the first stage of the forward DFT, and the last stage of each backward DFT, the blocked subarrays take on their largest size, `fftblockpad` $\times$ 2048. With the default choice of `fftblockpad` = 18, each such subarray extends 576 kilobytes. Since two arrays are required for the back-and-forth algorithm, this choice of `fftblockpad` does indeed cause spill from L2.

Halving the parameters to `fftblock` = 8, `fftblockpad` = 9 keeps the blocks in cache, and results in nearly 30% improvement. Results are shown in Table 3. Note that in Tables 3, 4, 5, and 11, the heading "`fftblockpad` = 9" is meant to imply the setting of `fftblock` = 8 as well.

Tables 4 and 5 show hardware counter data for runs using reference and modified code. In this data, we see that sizing the workspace arrays to fit in L2 has the effect of reducing the number of L2 misses by a remarkable 96%. This results in large reductions in the time spent stalled waiting for the load-store unit, as well as reducing the time that the FPUs are stalled.

MPI processes	**Reference code** (Mop/s/process)	**`fftblockpad` = 9** (Mop/s/process)	**Speedup**
64	361.59	467.72	29.4%
256	327.10	422.50	29.2%

Table 3: Performance results before and after appropriate block size selection

Subroutine `fftz2`		
	Reference code	**`fftblockpad = 9`**
L1 D-cache accesses	149250398564 ops	148195554681 ops
L1 D-cache misses that hit in L2	16727341484 fills	16046991282 fills
L1 D-cache misses that miss in L2	268507845 fills	10702336 fills
D-TLB misses	6063319 misses	2280212 misses
HW FP Ops	198471313626 ops	196918325209 ops
HW FP Ops / User time	944.409 M/sec	1533.563 M/sec
LD & ST per TLB miss	24615.02 refs/miss	64989.43 refs/miss
User time	210.154 secs	128.406 secs
Avg Time FPUs stalled	50.997 secs	20.504 secs
Avg Time LSs stalled	5.837 secs	1.629 secs

Table 4: Hardware counter data for FT Class D, 64 processes, reported as per-process averages.

Subroutine `fftz2`		
	Reference code	**`fftblockpad = 9`**
L1 D-cache accesses	37310402283 ops	37041891556 ops
L1 D-cache misses that hit in L2	4182665584 fills	4011128415 fills
L1 D-cache misses that miss in L2	66335586 fills	2702270 fills
D-TLB misses	1500164 misses	565509 misses
HW FP Ops	49611111727 ops	49229654302 ops
HW FP Ops / User time	947.462 M/sec	1536.249 M/sec
LD & ST per TLB miss	24867.90 refs/miss	65496.53 refs/miss
User time	52.362 secs	32.045 secs
Avg Time FPUs stalled	12.638 secs	5.115 secs
Avg Time LSs stalled	1.458 secs	0.423 secs

Table 5: Hardware counter data for FT Class D, 256 processes, reported as per-process averages.

2.2 FFT libraries

Sizing for cache certainly has an important effect on FT's performance, but one cannot ignore the huge amount of research done over the past decades on the FFT algorithm itself. A thorough study of NPB FT cannot be complete without some attempt to improve upon the reference implementation's Stockham DFT.

To this end, Cray supports multiple DFT libraries that offer Opteron-specific, optimized implementations. The remainder of this section presents results from re-implementing NPB FT using these various libraries. The first two, ACML and FFTW 3.1.1, offer highly-optimized serial DFTs. The latter two, FFTW 2.1.5 and FFTW 3.2-alpha2 (which is not yet Cray supported), offer distributed DFTs.

2.2.1 ACML

The AMD Core Math Library (ACML) offers a large set of mathematical subroutines, including BLAS, LAPACK, FFTs, transcendental functions, and random number generators [1]. Revising NPB FT to use ACML's `zfft1mx` subroutine in place of the reference `fftz2` subroutine was relatively straightforward. The only complication was the necessity to initialize nine separate transforms, as `fftz2` may be called using to up nine different array sizes.

ACML does offer two options for initialization, which involves storing the factorization of the array size, to be used for array decimation, as well as precomputing the twiddle factors [1]. The default initialization method is very fast, using heuristic methods to produce a factorization that is not guaranteed to be an optimal plan. The other option is very much slower, generating many possible plans (though not exhaustive) and timing each to determine a near-optimal factorization. Results for both initialization methods appear in Table 6.

MPI processes	**ACML**	
	default plans	measured plans
64	520.57	514.50
256	448.66	462.39

Table 6: Performance results, in Mop/s/process, after replacing `fftz2` with ACML's `zfft1mx`

2.2.2 FFTW 3.1.1

Another serial FFT package available for XT systems, FFTW 3.1.1 [6], provides tuned DFTs together with four initialization options. Implementing NPB FT with FFTW 3.1.1 is similar to using ACML; nine separate transforms must be initialized, allowing one to replace `fftz2` with calls to `dfftw_execute`.

In order of increasing planning time, and generally increasing performance, FFTW's options are called `FFTW_ESTIMATE`, `FFTW_MEASURE`, `FFTW_PATIENT`, and `FFTW_EXHAUSTIVE`. Results for each method appear in Table 7.

MPI processes	FFTW 3.1.1			
	FFTW_ESTIMATE	FFTW_MEASURE	FFTW_PATIENT	FFTW_EXHAUSTIVE
64	488.13	562.46	571.88	573.59
256	458.36	523.05	530.09	529.88

Table 7: Performance results, in Mop/s/process, using FFTW 3.1.1

2.2.3 FFTW 2.1.5

In addition to the serial transforms presented in subsections 2.2.1 and 2.2.2, Cray supports FFTW 2.1.5, which offers MPI distributed transforms in addition to serial transforms.

Distributed transforms in FFTW 2.1.5 require that global arrays be distributed over their last dimensions, so that each process holds a contiguous portion of the global array [5]. As described in Section 1.2, the reference NPB FT implementation uses the same distribution. As also described in Section 1.2, the reference implementation leaves the data transposed across the processes (with the first and third dimensions interchanged). This saves a global transpose step, in those cases where it is possible to work with transposed data in the transform space.

By default, FFTW 2.1.5 will transpose the data back to the original ordering after the final serial DFT calculations are complete. However, one can request that the last transposition be omitted by using the FFTW_TRANSPOSED_ORDER flag when executing transforms. With this flag, revising NPB FT to use FFTW 2.1.5 was straightforward.

FFTW 2.1.5 offers only FFTW_ESTIMATE and FFTW_MEASURE plan creation. Results for both methods appear in Table 8.

MPI processes	FFTW 2.1.5	
	FFTW_ESTIMATE	FFTW_MEASURE
64	273.78	273.81
256	298.67	300.30

Table 8: Performance results, in Mop/s/process, using FFTW 2.1.5

In spite of FFTW 2.1.5's poor performance on these tests, users looking for a distributed DFT library should not be disheartened. The current alpha release of FFTW 3.2 offers much-improved performance.

2.2.4 FFTW 3.2-alpha2

Cray will support FFTW 3.2 as its distributed DFT package following MIT's official release. Until such time, Cray has built FFTW 3.2-alpha2 for XT systems.

Like FFTW 3.1.1, FFTW 3.2-alpha2 offers four plan creation methods. In Table 9, we list results for all but FFTW_EXHAUSTIVE, as exhaustively planning such large problem sizes would be prohibitively time consuming. This was not an issue with the other libraries described in this paper: for the serial libraries, the DFTs are small enough that planning is very quick; for FFTW 2.1.5, the planning methods are not complex enough to consume large amounts of time. For the reader's interest, the planning times for each library described in this paper are listed in Table 10.

MPI processes	FFTW 3.2alpha2		
	FFTW_ESTIMATE	FFTW_MEASURE	FFTW_PATIENT
64	180.94	591.15	596.41
256	164.16	460.63	460.36

Table 9: Performance results, in Mop/s/process, using FFTW 3.2alpha2

MPI processes	Library	Planning method	Planning time (secs)
64	ACML	default	1.193
		measured	1.253
	FFTW 3.1.1	FFTW_ESTIMATE	1.199
		FFTW_MEASURE	1.196
		FFTW_PATIENT	1.202
		FFTW_EXHAUSTIVE	1.191
	FFTW 2.1.5	FFTW_ESTIMATE	2.527
		FFTW_MEASURE	2.725
	FFTW 3.2alpha2	FFTW_ESTIMATE	2.776
		FFTW_MEASURE	256
		FFTW_PATIENT	4157
256	ACML	default	0.302
		measured	0.355
	FFTW 3.1.1	FFTW_ESTIMATE	0.304
		FFTW_MEASURE	0.301
		FFTW_PATIENT	0.303
		FFTW_EXHAUSTIVE	0.303
	FFTW 2.1.5	FFTW_ESTIMATE	0.682
		FFTW_MEASURE	0.682
	FFTW 3.2alpha2	FFTW_ESTIMATE	0.589
		FFTW_MEASURE	101
		FFTW_PATIENT	1705

Table 10: Time to plan forward and backward DFTs. For ACML and FFTW 3.1.1, time includes planning nine forward and nine backward DFTs.

MPI processes	`fftblockpad = 9`	ACML	FFTW 3.1.1	FFTW 2.1.5	FFTW 3.2alpha2
64	467.72	520.57	573.59	273.81	596.41
256	422.50	462.39	530.09	300.30	460.63

Table 11: Best performance results, in Mop/s/process, for each code revision

3 Conclusions

Given that DFTs are fundamentally important algorithms in a huge variety of computational disciplines, and that NPB FT tests only one type of transform, one must guard against generalizing too much from the results presented in this paper.

Table 11 summarizes these results by listing the best results achieved with each code revision. No single library or planning method is a certain "winner" for NPB FT. One should expect this in general; given several choices of libraries, users who insist on the very best performance would have to try each library for each size DFT used in their codes.

Generally, though, one may conclude that FFTW 3.1.1 shows here as the best of the choices, since FFTW 3.2alpha2 is still pre-release and unsupported. Nevertheless, users looking for quick time-to-solution for distributed DFTs may wish to use FFTW 3.2alpha2 until the official release of FFTW 3.2.

References

[1] Advanced Micro Devices, Inc., Numerical Algorithms Group Ltd. *AMD Core Math Library User Guide*, 2006.

[2] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Report RNR-94-007, NASA Advanced Supercomputing Division, March 1994.

[3] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. RNR Technical Report NAS-95-020, NASA Advanced Supercomputing Division, December 1995.

[4] David H. Bailey. A high-performance FFT algorithm for vector supercomputers. *International Journal of Supercomputer Applications*, 2(1):82–87, 1988.

[5] Matteo Frigo and Steven G. Johnson. *FFTW version 2.1.5 user manual*. Massachusetts Institute of Technology, Cambridge, Massachusetts, November 2003.

[6] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special Issue on Program Generation, Optimization, and Platform Adaptation.

[7] Stefan Goedecker. Rotating a three-dimensional array in an optimal position for vector processing: case study for a three-dimensional fast Fourier transform. *Computer Physics Communications*, 76(3):294–300, 1993.

[8] Paul N. Swarztrauber. FFT algorithms for vector computers. *Parallel Computing*, 1(1):45–63, August 1984.

[9] Rob van der Wijngaart. NAS parallel benchmarks version 2.4. NAS Technical Report NAS-02-007, NASA Advanced Supercomputing Division, October 2002.