Adaptive Computation of Self Sorting In-Place FFTs on Hierarchical Memory Architectures

April 30, 2007

Abstract

Since the publication of the famous Cooley Tukey algorithm for the Fast Fourier Transform (FFT), many algorithms have been introduced for computing DFTs efficiently. While some of the algorithms had lower operation count for certain transform sizes others were just the rescheduling (implementations) adapted to target architectures through optimization of memory access pattern. A naive implementation of the recursive Mixed Radix algorithm would require an additional sorting step to bring the output back "in-order". Various algorithms have been suggested that perform this sorting (digit-reversal) inside the FFT ”butterfly” using the output vector. Performing the computation "in-place", i.e. without a separate output vector or temporary array, can be useful when the input vector needs to be overwritten with the result. It also makes the computation of larger size transforms feasible when the memory is limited. But computing "in-place and in-order" FFT poses a very difficult problem on hierarchical memory architectures where data movement can seriously degrade the performance. In this paper we present the recursive formulation of self sorting in-place FFT algorithm that adapts to the target architecture. When an efficient in-place, in-order execution is not possible, we show how efficient schedules can be constructed that use minimum work-space to perform the computation. To express and construct the FFT schedules, we present the Context Free Grammar that generates a language called

FFT Schedule Specification Language. We conclude by comparing the performance of our in-place in-order FFT implementation with that of other well known FFT libraries. We also present a performance analysis of the performance difference between the cache oblivious execution of out-of-place and in-place FFTs.

1 Introduction

The Fast Fourier Transform (FFT) is one of the most widely used algorithms in many fields of science and engineering, especially in the field of signal processing. Since 1965, various algorithms have been proposed for computing DFTs efficiently. However, the FFT is only a good starting point if an efficient implementation exists for the architecture at hand. Scheduling operations and memory accesses for the FFT for modern platforms, given their complex architectures, is a serious challenge compared to BLAS-like functions. It continues to present serious challenges to compiler writers and high performance library developers for every new generation of computer architectures due to its relatively low ratio of computation per memory access and non-sequential memory access pattern.

FFTW[8, 7], SPIRAL[13, 6] and UHFFT[12, 11, 1, 2] are three current efforts addressing machine optimization of algorithm selection and code optimization for FFTs. SPIRAL is a code generator system that generates optimized codes for specific size transforms. Both UHFFT and FFTW generate highly optimized straight-line FFT code blocks (micro-kernels) called *codelets,* at installation time. These parametrized code blocks adapt to microprocessor architecture and serve as building blocks in the computation of a larger size transform. The final optimization, specific to a FFT problem, is performed at run-time by searching the best schedule (plan) from among an exponential number of factorizations.

Once the factors are determined, the FFT computation can be carried out in a recursive cache oblivious fashion, which derives naturally from the Cooley Tukey Mixed Radix Algorithm [5]. For an input vector $\mathbf{x}$ of size N, where $N = r \times m$, the recursive Cooley Tukey formulation of FFT can be written as:

$$\mathbf{X} = (W_r \otimes I_m)T_m^N(I_r \otimes W_m)\Pi_{N,r}\mathbf{x}$$

where $T_m^N$is a diagonal "twiddle factor" matrix and $\Pi_{N,r}$is a mod $- r$ sort permutation matrix. Although these algorithms reduce the complexity of DFT computation from $O(n^2)$ to $O(n \log n)$, the resulting vector is in "digit-reversed" order, requiring non-trivial amount of work to bring the output back in order. Stockham's Autosort framework[10] performs the sorting inside the FFT "butterfly", thereby avoiding an explicit reordering. Recognizing the matrix multiplication properties, the permutations can be pushed at the start of computation [16], which can be performed within the first rank of butterfly if additional work-space or output vector is available, as illustrated in Figure 1. This is the most commonly used

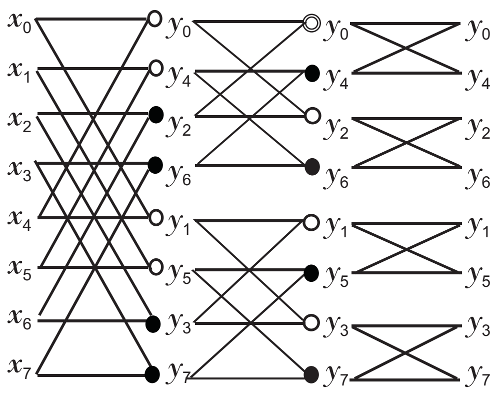

Figure 1: Butterfly showing the execution of Self-sorting out-of-place FFT of size $N = 8$. The transform is stored to output array in "digit-reversed" order in the first rank.

algorithm for the self sorting out-of-place FFT computations in both FFTW and UHFFT. Performing the computation "in-place", i.e. without a separate output vector or temporary array, poses a very difficult problem for self-sorting FFTs. For most transform sizes, data reordering is unavoidable in computing in-place and in-order FFTs due to lack of a separate work-space. For sizes that can be factorized in co-prime numbers, Prime Factor Algorithm (PFA) performs the computation that is both self sorting and in-place [3, 18, 17], without requiring an additional work-space, using special rotated DFT modules (PFA *codelets*). For more prevalent sizes such as powers of two, [14] and later [19, 4] developed the algorithm that was both self sorting and in-place. The algorithm avoids an explicit sorting of the data by performing implicit ordering inside the first $\left\lfloor \frac{\log n}{2} \right\rfloor$ ranks of FFT butterfly computation as shown in Figure 2. However this algorithm works only if the factorization of N is a

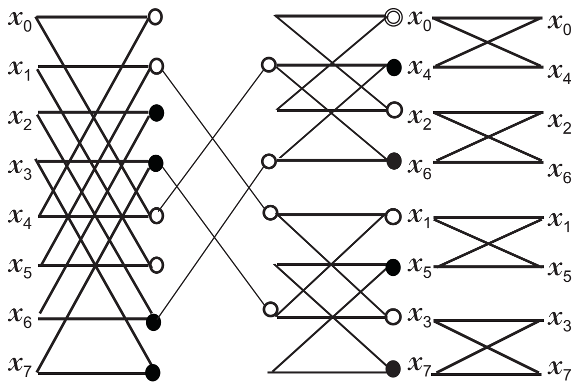

Figure 2: Butterfly showing the execution of Self-sorting in-place FFT of size $N = 8$. The result is ordered in the first $\left\lfloor \frac{\log 8}{2} \right\rfloor = 1$ ranks by performing partial transposed *codelet* transforms on square blocks[19, 4].

“palindrome”, i.e., $N = r \times m \times r$ or $N = r \times r$, which is always possible for powers of 2 sizes. The algorithm was later formulated by [9] in a recursive fashion that is suited to parallel and vector architectures. In this paper, we present the recursive formulation of the algorithm that is suited to hierarchical memory architectures because it maintains both temporal as well as spatial locality. We discuss the efficiency of our implementation by comparing it with an alternate factorization strategy. In fact, many different schedules could be constructed to adapt to different architectures. In order to ensure performance portability and adaptability, we have implemented many alternative algorithms in UHFFT. To that end we have designed a language that is used to construct these schedules at run-time without having to write specific implementations for different FFT sizes.

Expressing the FFT Schedules

Before we discuss the various solutions to the computation of in-place and in-order FFTs, we need to be able to express them in a compact representation. "*Unfortunately, the simplicity and intrinsic beauty of many FFT ideas is buried in research papers that are rampant with vectors of subscripts, multiple summations, and poorly specified recursions. The poor mathematical and algorithmic notation has retarded progress and has led to a literature of duplicated results*"[10]. The most commonly used, "Kronecker product" formulation of FFT is an efficient way to express sparse matrix operations. SPIRAL[13], generates optimized DSP codes using a pseudo-mathematical representation that is based on the Kronecker product formulation. FFTW [8], uses an internal representation of DFT problems using various data structures, i.e., I/O tensors and I/O dimensions etc. Although they offer an efficient design to solving a complex problem, these representations do not provide sufficient abstraction to the mathematical and implementation level details of FFT algorithms. One of the standard ways to visualize a FFT problem is through a signal flow diagram called the "butterfly" representation. This representation has been adopted across the board by various disciplines of science. However, visualizing large FFT problems through the butterfly is not practical. In this paper we present a language called FFT Schedule Specification Language (FSSL), which is generated from a set of context free grammar productions. The grammar provides a direct and compact translation of the FFT butterfly representation and is easy to understand for computer scientists.

The grammar of FSSL is given in Section 2; where we briefly mention the design overview of UHFFT. We discuss our cache oblivious factorization of self-sorting in-place FFT algorithm in Section 3. And in Section 4, we compare the performance results of our implementation in UHFFT with that of FFTW and Intel's MKL library. Both these libraries are known to perform well on hierarchical memory architectures.

2 UHFFT

The UHFFT system comprises of two layers, i.e., the code generator (FFTGEN) and the run-time framework. The code generator generates highly optimized small DFT, straight line "C" code blocks called *codelets* at installation time. These *codelets* are combined together by the run-time framework to solve large FFT problems on Real and Complex data. Block diagram of UHFFT run-time is given in Figure 3.

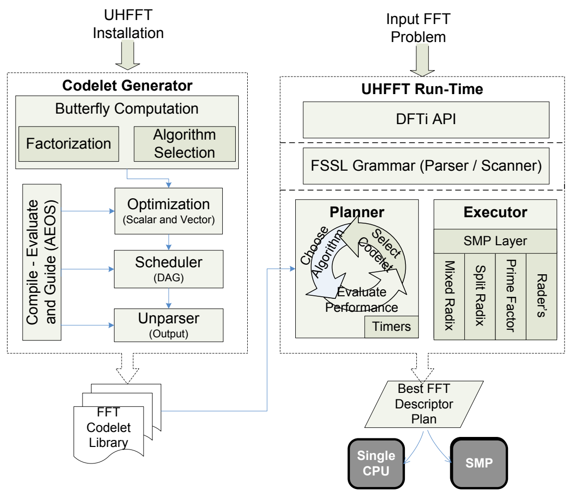

Figure 3: UHFFT2 Block Diagram. *Codelets* specified by user are generated at installation time. At run-time the code blocks are used by planner to compose best FFT computation schedules. UHFFT2 supports Intel's DFT API [15] with some extensions.

Code Generator

The FFT library contains a number of *codelets*, which are generated at installation time. Each *codelet* sequentially computes a part of the transform and overall efficiency of the code depends strongly on the efficiency of these *codelets*. Therefore, it is essential to have a highly optimized set of DFT *codelets* in the library. Apart from the normal DFT *codelets*, we have implemented a few specialized sets of *codelets* that are used in the FFT computation at various stages of the algorithm. Two types of such *codelets* that are relevant to our discussion of self-sorting in-place algorithms are: the “coupled transposed *codelets*” and the “rotated PFA *codelets*”. The code generator adapts to the platform, i.e., compiler and hardware architecture by empirically tuning the *codelets* using iterative compilation and feedback [1].

Planner

Each FFT problem is identified by a DFTi descriptor [15], which describes various characteristics of the problem, i.e., size, precision, placement, input and output data type, number of threads etc. Once the descriptor is submitted, planner selects the best plan, which may be used repeatedly to compute the transforms of same size. Our current implementation supports two strategies for searching the best plan. Both strategies use the *dynamic programming* to search the space of possible factorizations and algorithms, given by a tree as shown in Figure 4. The first approach called *Context Sensitive Empirical Search,* empirically evaluates the sub-plans. To avoid re-evaluation of identical sub-plans, a lookup table is maintained to store their performance. In the second approach, called *Context Free Hybrid Search,* the cost of search is significantly reduced at the expense of quality of plan. In this scheme, the cost of a subplan is estimated by empirically evaluating only the *codelet* that is encountered in a bottom up traversal. In this paper, we use the first approach because it generates good quality plans at reasonable cost of search.

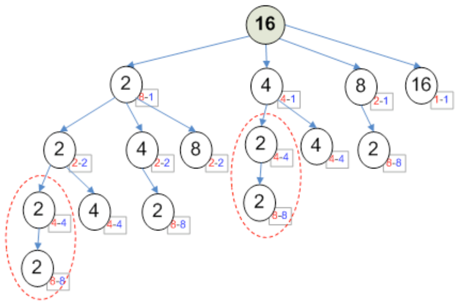

Figure 4: Search Space for Size 16 FFT (out-of-place). There are a total of 8 possible plans shown here. *Codelets* are called with different input and output strides at different levels of tree (recursion) also known as ranks.

FFT Schedule Specification Language (FSSL)

An execution plan determines the *codelets* that will be used for the computation and also the order (schedule) in which they will be executed. A FFT plan is described in concise language using the grammar given in Table 1. It allows different algorithms to be mixed together to generate a high performance execution plan based on properties of the input vector and its factors. Indeed, by implementing a minimal set of rules, adaptive schedules could be constructed that suit different types of architectures.

Table 1: FFT Schedule Specification Language grammar rules.

#	CFG Rules
1-2	ROOT$\longrightarrow$MULTID_FFT | SMPFFT
3-4	MULTID_FFT$\longrightarrow$FFT | [NDFFT , FFT]
5-6	NDFFT$\longrightarrow$NDFFT , FFT | FFT
7	SMPFFT$\longrightarrow$(*mr* $p\mathbb{Z}$ BLOCK , MULTID_FFT) $\mathbb{Z}$
8-9	FFT$\longrightarrow$FFT *mr* MODULE | MODULE
10-11	MODULE$\longrightarrow$CODELET | ORDERED_FFT
12	| [*rader* , FFT] $\mathbb{Z}$
13	ORDERED_FFT$\longrightarrow$(*inplace*$\mathbb{Z}$, FFT)
14	| (*outplace*$\mathbb{Z}$, FFT)
15	| (*pfa*$\mathbb{Z}$, PFAFFT *pfa* ROT_CODELET)
16	CODELET$\longrightarrow n \in$ DFT codelets
17	ROT_CODELET$\longrightarrow n \in$ Rotated DFT codelets
18	BLOCK$\longrightarrow$*b* $\mathbb{Z}$: $\mathbb{Z}$

3 Self Sorting In-Place FFT Implementation

Our implementation of self-sorting in-place FFTs is based on the algorithms given in [19, 4]. These algorithms factorize the given FFT of size N in a palindrome to couple the sorting step with the butterfly computation. However, many alternative factorizations exist for a given transform size that fulfill this requirement. Figure 5 illustrates two factorization schemes for an example of size 48 transform. Even though the two factorizations ultimately use the same factors (*codelets*), their index mapping or data access pattern is quite different. An illustration of the memory blocking resulting from the two schedules is given in Figure 6. Notice that the scheme (a) recurses *outward* by first selecting the largest square factors, while

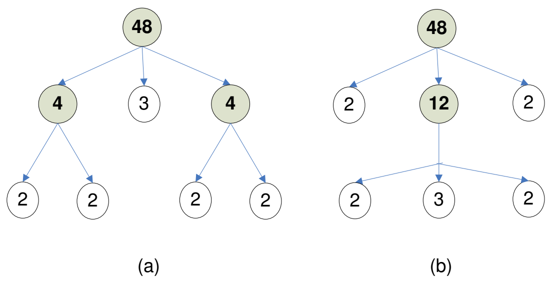

Figure 5: Two factorization schemes that form a palindrome. Factorization scheme (a) (inplace48, (inplace4, 2mr2)mr3mr(inplace4, 2mr2)) and (b) (inplace48, 2mr(inplace12, 2mr3mr2)mr2). Assuming, only two *codelets* have been generated in the library, namely $\{2, 3\}$.

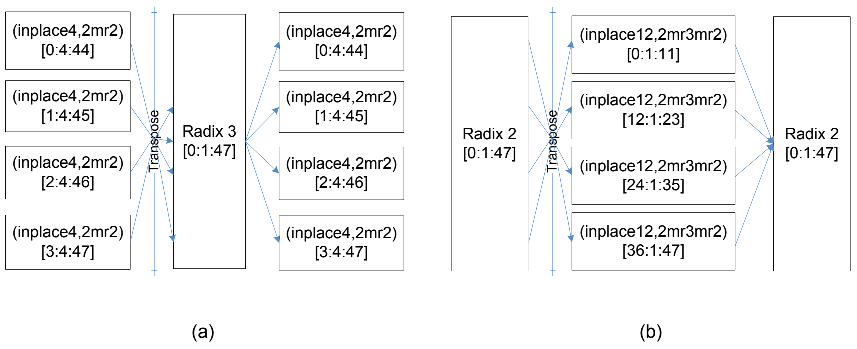

Figure 6: Data Access pattern using the two factorization schemes shown in Figure 5. Each block shows the vector indices accessed in Matlab vector representation.

scheme (b) recurses *inward* by choosing the largest factor to be in the middle of two small factors. The *outward* recursion generates larger blocks that can be executed independently or in large vectors at unit strides. However, in order to maintain spatial locality, the blocks will need to be executed in a breadth first fashion thereby reducing the potential for temporal locality between successive ranks. On the other hand the *inward* recursion allows better blocking that maintains both temporal and spatial locality in a cache oblivious manner. In UHFFT, the planner uses *inward* recursion strategy to factorize and execute the self-sorting in-place algorithm after finding the odd factors to put in the middle rank. Following, we give the simple steps that the planner follows to select the best palindrome factorization of size N:

1. Find the factors with odd exponent to construct the middle rank.
   (a) Factorize size N in powers of prime factors, i.e., $N = \prod_{i=0} p_i^{r_i}$, where all the factors p_i are co-prime and r_i is the exponent.
   (b) Based on the above factorization, divide the size $N = q \times s$ in two sets $q = \prod_{i=0} p_i^{e_i}$ and $s = \prod_{i=0} p_i^{o_i}$, where e_i is even and o_i is either 0 or 1.
   (c) Construct the largest possible PFA plan ρ_{mid} for middle rank of size n_{mid}, from all factors of s and optionally only $p_i^{2\times j}$factors of q.
   (d) If all the factors of s were not used (rotated *codelets* are not generated for all prime factors), then the middle rank will be solved using extra buffers.
   (e) If (c) is true then construct the best out-of-place planρ_{mid} using one of the search schemes.
2. Finally, construct the rest of the in-place plan of size $\frac{N}{n_{mid}}$ by recursively factorizing it inwards:

$$N \quad = \quad m_0 \times n_{mid} \times m_0$$

$$\Rightarrow N = r_0 \times m_1 \times n_{mid} \times m_1 \times r_0$$

3. The search methods implemented in UHFFT can be used to find the best inplace plan:

$$\rho = (\text{inplace}N, r_0\text{mr}r_1\text{mr}\ldots\text{mr }\rho_{mid}\text{mr}\ldots\text{mr}r_1\text{mr}r_0)$$

In the above algorithm we start by finding the odd factor to fit in the middle of palindrome. Since PFA algorithm is both self sorting and in-place, we try to construct the middle rank from co-prime factors. In case, there are remaining factors that do not have even exponent, we will need to use a self-sorting out-of-place algorithm that would use a work-space equal to the size of middle rank. Once the middle rank is factored, rest of the factorization follows recursively in a straightforward manner. To illustrate the algorithm, lets take an example of size $N = 48$. If the library contains rotated *codelet* of size 16, then the largest PFA plan $\rho = (\text{inplace}48, (\text{pfa}48, 3\text{pfa}16))$ of size 48 is constructed which is both self sorting and in-place. Assuming that the rotated *codelet* for size 16 was not generated at installation time. In that case, the plan will use the middle rank of size 12: $\rho = (\text{inplace}48, 2\text{mr}(\text{pfa}12, 3\text{pfa}4)\text{mr}2)$.

In the above discussion, we have omitted the details of twiddle factors multiplication, which is performed after each rank. In UHFFT, we have implemented specialized *codelets* that perform the twiddle factor multiplication inside the *codelets* thereby avoiding the extra loads and stores. The loads and stores are further reduced by implementing "coupled *codelets*"[19] that perform r small DFTs of size r. These square blocks r^2 are then transposed using registers and written back to the input vector. Notice that this kind of optimization is not possible if *outward* recursion is used because for a large size $N = m \times r \times m$, square matrix $m \times m$ will be too large to fit in the registers.

4 Results

We performed the benchmarking of our self-sorting in-place FFT implementation in UHFFT on Itanium 2 machine to compare its performance with FFTW and Intel's Math Kernel Library (MKL). The specification of the experimental setup is given in Table 2. The perfor-

Table 2: Experimental Setup

Feature	Description
Architecture	Itanium 2 1.5 Ghz, Cache: 16K/256K/6M, FP Registers=128 CacheLine: 64B/128B/128B, Associativity: 4/8/12
Compiler	Intel C Compiler icc 9.1, Flags=-O3
Libraries	UHFFT-2.0.1beta, FFTW-3.2alpha, MKL-9.1
Data Sets	Powers of 2, Small Co-prime factor sizes, Mixed

mance numbers, in this paper, are derived from the total execution time of the FFT problem. The performance measure, Million Floating Point Operations Per Second (MFLOPS), is calculated from the execution time and commonly used algorithm complexity of FFT, i.e., $5N \log N$. The results were collected on double precision complex FFT of three sets of sizes ranging from 2 to 16M.

UHFFT vs FFTW vs MKL

UHFFT and FFTW are open source adaptive libraries for computing FFTs efficiently. Intel's MKL is the vendor library that is particularly tuned to Intel's architectures. All the three libraries support both in-place and out-of-place executions of FFT. Since MKL is a vendor library, it wasn't possible for us to find out how the in-place FFTs were being executed internally. Hence, on most occasions we can not provide a detailed analysis of their performance. Figure 7 gives the performance comparison of our implementation of self sorting in-place FFT computation with that of FFTW and MKL for transform sizes that are powers of two. For very small sizes ($2 - 2^5$) the FFT is performed within registers using optimized *codelets*, which is why all the three libraries perform equally well. One of the main differences between our strategy in UHFFT and the other two libraries is that FFTW (and quite possibly MKL) would trade memory for performance; extra buffers can be used for improving the performance of code. On the other hand, UHFFT uses buffers only when an in-place in-order plan does not exist. Notice that for sizes larger than Level 2 cache capacity, i.e., 2^{14}, UHFFT performs better than FFTW. Indeed, for data ($N = 2^{18}$) that do not fit in any level of cache, our implementation performs consistently better be-

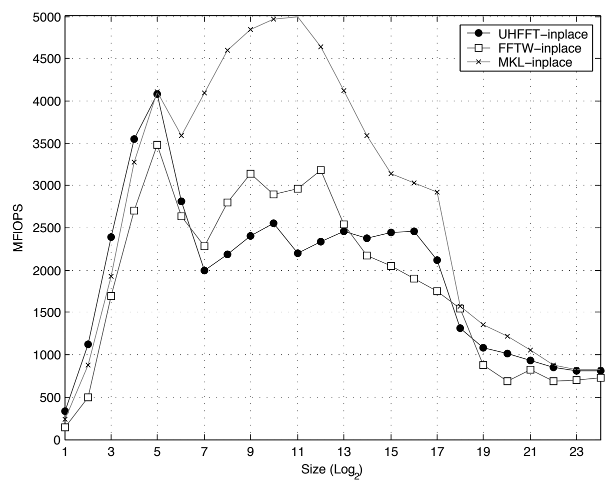

Figure 7: Powers of 2 Sizes. Complex In-place In-order FFT Computation. UHFFT does not use any extra buffers (except the input vector and twiddle factors) in the in-place execution of powers of 2 sizes.

cause of superior cache management. Another advantage that UHFFT has over the other two libraries is its support for fast execution of sizes that have co-prime factors. Prime Factor Algorithms are known to have lower complexity and produce ordered transforms. In Figure 8, we computed the in-place FFT of sizes that have co-prime factors. Both FFTW and MKL, do not support Prime Factor Algorithm and hence they have relatively lower performance on sizes with co-prime factors. In Figure 9, we compared the performance of the three libraries on randomly sampled sizes that have both powers of 2 and prime factors. These results show that UHFFT is equally good at performing in-place self-sorting FFT on non-powers of two sizes, which are less straightforward to factorize in a palindrome.

In-place vs Out-of-place FFT performance

We compared the performance of our in-place implementation of self sorting FFT schedules with that of out-of-place execution. Even though in-place schedules perform many small transposes, overall their performance is quite similar to out-of-place executions as shown in Figures 10 & 11. The reason that there is no drastic degradation of performance in in-place

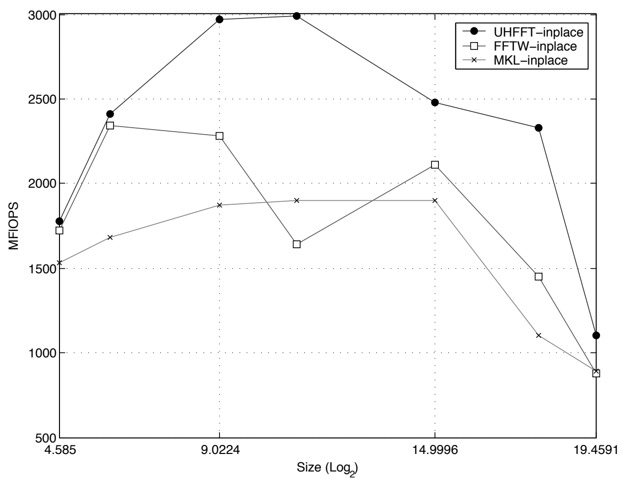

Figure 8: Sizes with small co-prime factors. Complex In-place In-order FFT. UHFFT uses PFA algorithm, which is known to have lower complexity and produces ordered results. FFTW and MKL are not tuned for the sizes that have co-prime factors.

execution is that the transposes are performed inside the *codelets,* using registers only. This avoids the extra cost of loads and stores when the re-ordering is done out of cache. The performance of out-of-place schedules starts falling slightly earlier than in-place schedules around the L3 cache size mark (2^{18}), because of its extra bandwidth requirement (both input and output vector are brought in the cache).

5 Conclusion

Computing self-sorting in-place FFTs poses a challenge when the execution needs to be carried out without use of additional work-space. The re-ordering performed inside the butterfly can seriously degrade the performance on current hierarchical memory architectures if the problem is not carefully factorized. In this paper we presented a recursive cache oblivious formulation of self-sorting in-place FFT algorithm that is suited to hierarchical memory architectures. Despite requiring the re-orderings to be performed in-place, the performance of self sorting in-place FFT was shown to be at par with out-of-place FFT. This is achieved

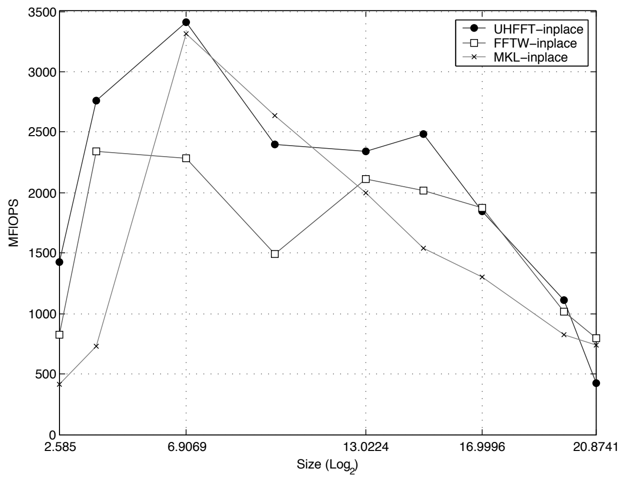

Figure 9: Sizes with prime as well as powers of two factors. Complex In-place In-order FFT Computation. Not all the sizes presented here result in palindrome of factors, which means that small buffers may be required to compute in-place in-order FFT efficiently.

through careful factorization of the problem such that the partial ordering can be performed in registers when the data has already been loaded for butterfly computation; this avoids extra loads and stores. To express the several FFT algorithms and factorizations, we presented a language, FFT Schedule Specification Language (FSSL), that defines the schedules in compact representation. To evaluate and validate the efficacy of our recursive strategy, we compared the performance of our implementation with the in-place in-order FFT performance of FFTW and Intel's MKL on Itanium 2. For Powers of two sizes, UHFFT performs competitively against FFTW, without using any extra buffers. For Non-Powers of two sizes, the performance is much better than that of the other two libraries because UHFFT planner adaptively selects PFA algorithm for sizes that have co-prime factors.

References

[1] Ali, A., Johnsson, L., and Mirkovic, D. Empirical Auto-tuning Code Generator for FFT and Trignometric Transforms. In *ODES: 5th Workshop on Optimizations for*

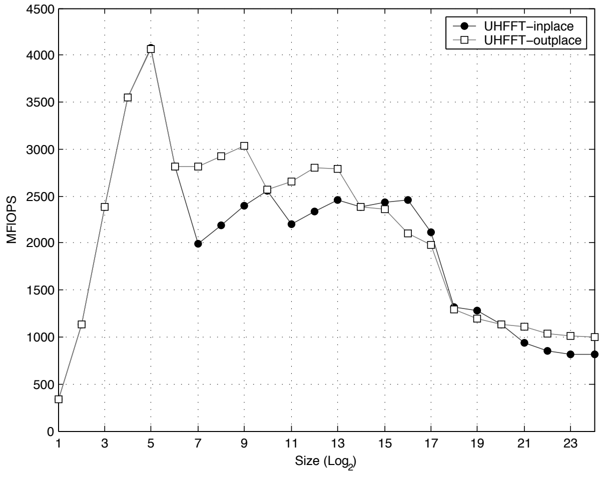

Figure 10: Comparison of out-of-place and in-place FFT performance in UHFFT for powers of 2 sizes. The performance for small size $N \leq 2^6$transforms is identical because both in-place and out-of-place executions use the same *codelet* which performs sorting in the registers before writing the results back to the output vector.

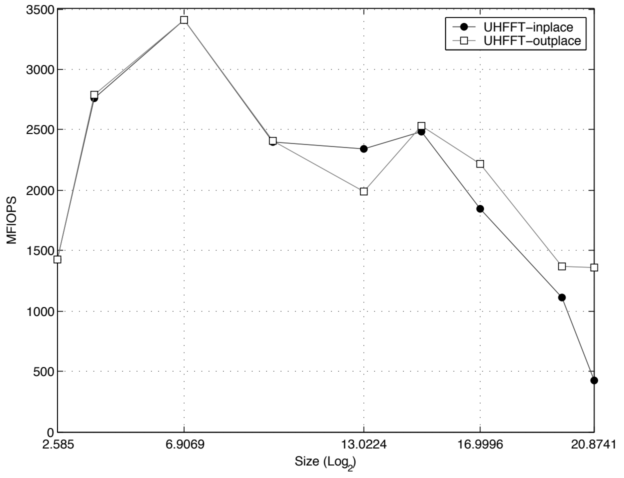

Figure 11: Non Powers of 2 sizes $\{6, 14, 120, 968, 8320, 32760, 131040, 915200, 1921920\}$. For sizes 6 and 14, *codelets* are used directly and for size 120 and 32760, pfa plans are used for both out-of-place and in-place execution, which is why the performance is identical.

DSP and Embedded Systems, in conjunction with International Symposium on Code Generation and Optimization (CGO) (San Jose, CA, March 2007).

[2] Ali, A., Johnsson, L., and Subhlok, J. Scheduling FFT Computation on SMP and Multicore Systems. In *To Appear in International Conference on Supercomputing* (Seattle, WA, June 2007).

[3] Burrus, C. S., and Eschenbacher, P. W. An in-place, in-order prime factor FFT algorithm. *IEEE Transactions on Acoustics, Speech, and Signal Processing 29* (1981), 806–817.

[4] Burrus, C. S., and Johnson, H. W. An in-order, in-place radix-2 FFT. *IEEE Transactions on Acoustics, Speech, and Signal Processing 9* (1984), 473 – 476.

[5] Cooley, J., and Tukey, J. An algorithm for the machine computation of complex fourier series. *Mathematics of Computation 19* (1965), 297–301.

[6] Franchetti, F., Voronenko, Y., and Püschel, M. FFT program generation for shared memory: SMP and multicore. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2006), ACM Press, p. 115.

[7] Frigo, M. A fast Fourier transform compiler. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation* (New York, NY, USA, 1999), ACM Press, pp. 169–180.

[8] Frigo, M., and Johnson, S. G. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2 (2005), 216–231. special issue on "Program Generation, Optimization, and Platform Adaptation".

[9] Hegland, M. A self-sorting in-place fast Fourier transform algorithm suitable for vector and parallel processing. *Numerische Mathematik 68*, 4 (1994), 507–547.

[10] Loan, C. V. *Computational frameworks for the fast Fourier transform.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[11] Mirkovic, D., and Johnsson, S. L. Automatic Performance Tuning in the UHFFT Library. In *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I* (London, UK, 2001), Springer-Verlag, pp. 71–80.

[12] Mirkovic, D., Mahasoom, R., and Johnsson, S. L. An adaptive software library for fast Fourier transforms. In *International Conference on Supercomputing* (2000), pp. 215–224.

[13] Püschel, M., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., Singer, B. W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R. W., and Rizzolo, N. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" 93*, 2 (2005), 232–275.

[14] Singleton, R. C. An algorithm for computing the mixed radix fast Fourier transform. *IEEE Transactions on Audio and Electroacoustics 17* (1969), 93–103.

[15] Tang, P. T. P. DFTI – A New Interface for Fast Fourier Transform Libraries. *ACM Transactions on Mathematical Software 31*, 4 (Dec. 2005), 475–507.

[16] Temperton, C. Self-Sorting Mixed-Radix Fast Fourier Transforms. *Journal of Computational Physics 52* (1983), 1–23.

[17] Temperton, C. Implementation of a Self-Sorting In-Place Prime Factor FFT Algorithm. *Journal of Computational Physics 54* (1985), 283–299.

[18] Temperton, C. A new set of minimum-add small-n rotated DFT modules. *J. Comput. Phys. 75*, 1 (1988), 190–198.

[19] Temperton, C. Self-Sorting In-Place Fast Fourier Transforms. *SIAM Journal on Scientific and Statistical Computing 12*, 4 (1991), 808–823.