An Adaptive Software Library for Fast Fourier Transforms

Dragan Mirković*
Department of Computer Science
University of Houston
Houston, TX 77204
mirkovic@cs.uh.edu

Rishad Mahasoom*
Department of Computer Science
University of Houston
Houston, TX 77204
mahasoom@cs.uh.edu

Lennart Johnsson*
Department of Computer Science
University of Houston
Houston, TX 77204
johnsson@cs.uh.edu

ABSTRACT

In this paper we present an adaptive and portable software library for the fast Fourier transform (FFT). The library consists of a number of composable blocks of code called *codelets*, each computing a part of the transform. The actual FFT algorithm used by the code is determined at run-time by selecting the fastest strategy among all possible strategies, given available codelets, for a given transform size. We also present an efficient automatic method of generating the library modules by using a special–purpose compiler. The code generator is written in C and it generates a library of C *codelets*. The code generator is shown to be flexible and extensible and the entire library can be generated in a matter of seconds. We have evaluated the library for performance on the IBM–SP2, SGI–2000, HP–Exemplar and Intel Pentium systems. We use the results from these evaluations to build performance models for the FFT library on different platforms. The library is shown to be portable, adaptive and efficient.

1. INTRODUCTION

The importance of the fast Fourier transform (FFT) in many applications has provided a strong motivation for development of highly optimized FFT implementations in scientific codes. A considerable research effort has been devoted to this problem over the past forty years. First, the research was focused on the design of algorithms that minimized the number of arithmetic operations. The most important achievement in this direction was the FFT algorithm by Cooley and Tukey [1], which reduced the asymptotic complexity of the DFT from $O(N^2)$ to $O(N \log N)$. Although the fast algorithms for DFT were first described by Gauss in 1805, the publication of [1] was a turning point for the FTT applications. The research that followed has produced a number of important algorithms (split–radix algorithm, prime factor algorithm, Rader's algorithm and Winograd FFT) each of them reducing the number of arithmetic operations by a constant factor. However, a fast algorithm is only a good starting point for an efficient FFT code, since the actual implementation on modern–day (micro)processors has proven to be nontrivial.

*The authors were supported by Air Force Office of Scientific Research (AFOSR) under grant F49620–96–1–0289.

Current state–of–the–art codes adapt themselves to the computer architecture and transform size by using a dynamic construction of the FFT algorithm depending on the size of the transform. The adaptability is accomplished by using a library of composable blocks of code, each computing a part of the transform, and by selecting the optimal combination of these blocks at runtime. The blocks of code, called *codelets*, are highly optimized and usually generated by a special–purpose compiler. An excellent example that uses this approach is the FFTW program [2], developed at MIT.

We have applied a similar approach to the development of an adaptive FFT library. In this paper we describe the optimization procedures for the adaptive FFT library that we have developed, and present a detailed analysis of its performance on a number of different architectures. We use the results of these analyses to build/confirm our performance models for FFTs on different platforms. These performance models are then used to make the selection process of the optimal execution strategy more efficient.

2. MATHEMATICAL BACKGROUND

The Fast Fourier Transform (FFT) is a divide–and–conquer method for quick evaluation of the Discrete Fourier Transform (DFT). In this chapter we give a short list of the algorithms used in the UHFFT library. We refer the reader to [3] and [4] for the more detailed description of the algorithms. In particular, the notation we use here mostly coincides with the notation in [3].

We denote by $\mathbf{C}^N$ the vector space of complex N–vectors with components indexed from zero to $N-1$. The Discrete Fourier Transform (DFT) for $\mathbf{x} \in \mathbf{C}^N$ is a matrix–vector product defined by

$$X_l = \sum_{j=0}^{N-1} \omega_N^{lj} x_j, \qquad (1)$$

where ω_N is an Nth root of unity: $\omega_N = e^{-2\pi i/N}$ and $i = \sqrt{-1}$. In matrix vector terms, the DFT is written as

$$\mathbf{X} = W_N \mathbf{x} \tag{2}$$

where W_N is the DFT matrix. The periodicity of ω_N introduces an intricate structure into W_N, which makes possible the factorization of W_N into a small number of sparse factors. The sparse factorization of W_N is the essence of all fast DFT algorithms. For example, it can be shown that when $N = rq$, W_N can be written as

$$W_N = (W_r \otimes I_q) D_{r,q} (I_r \otimes W_q) \Pi_{N,r}, \tag{3}$$

where $D_{r,q}$is a diagonal **twiddle factor** matrix of the form

$$D_{r,q} = \mathrm{diag}\left(I_q, \Omega_{N,q}, \ldots, \Omega_{N,q}^{r-1}\right),$$

$\Omega_{N,q} = \mathrm{diag}(1, \omega_N, \ldots, \omega_N^{q-1})$ and $\Pi_{N,r}$ is a mod–r sort permutation matrix. If q is not a prime number the above algorithm can be applied recursively. This is the heart of the fast Fourier transform idea, and the algorithm Eq. (3) is the well known Cooley–Tukey mixed–radix splitting algorithm. In this algorithm a non–trivial fraction of the computational work is associated with the construction and the application of the diagonal scaling matrix $D_{r,q}$. The *prime factor FFT* algorithm [5] removes the need for this scaling when r and q are relatively prime, i.e., $gcd(r, q) = 1$. This algorithm is based upon splittings of the form:

$$W_N = P_1(W_r \otimes I_q)(I_r \otimes W_q)P_2 = P_1(W_r \otimes W_q)P_2,$$

where P_1 and P_2 are permutations.

For FFT sizes that are prime, Rader [6] developed an algorithm that involves conversion of a given DFT into a convolution. It uses a number–theoretic permutation of W_N that produces a circulant submatrix of order $N - 1$. This reduces the prime size problem to a non–prime size one for which we may use any other known algorithm. The Rader factorization can be written as

$$W_N = Q_1 \begin{bmatrix} 1 & e^T \\ e & C_{N-1} \end{bmatrix} Q_2,$$

where e is a vector of all ones, Q_1 and Q_2 are permutations, and C_{N-1} is a circulant matrix. The action of the circulant matrix can be obtained efficiently by using the FFT, since W_N diagonalizes C_N

$$W_N C_N W_N^{-1} = \mathrm{diag}(W_N \mathbf{c}),$$

where $\mathbf{c}$ is the first column of C_N.

Standard radix–2 procedures are based upon the fast synthesis of two half–length DFTs. The *split–radix* [7] algorithm is based upon a clever synthesis of one–half length DFT together with two quarter–length DFTs, i.e.,

$$W_N \mathbf{x}(0 : N-1) = \begin{cases} W_{N/2}\mathbf{x}(0{:}2{:}N-1) \\ W_{N/4}\mathbf{x}(1{:}4{:}N-1) \\ W_{N/4}\mathbf{x}(3{:}4{:}N-1). \end{cases} \tag{4}$$

The resulting procedure involves less arithmetic than any of the standard radix–2, radix–4 or radix–8 procedures.

3. OPTIMIZATION STRATEGY

Our FFT library uses an adaptive approach similar to the one used by the FFTW library [2] from MIT. The main idea is to develop a library which can be used over many different platforms. The optimization of the FFT routines in our library is performed on two levels. The first (high) level optimization consists of selecting the optimal factorization of the FFT of a given size, into a number of factors, smaller in size, for which an efficient DFT codelet exists in our library. The optimization on this level is performed during the initialization phase of the procedure, which makes the code adaptive to the architecture it is running on.

The second (low) level optimization involves generating a library of efficient, small size DFT codelets. Since the efficiency of the code depends strongly on the efficiency of the codelets themselves, it is important to have the best possible performance for the codelets to be able to build an efficient library.

3.1 Factorization schemes

Given the parameters of the problem, the initialization routine selects the strategy in terms of execution time on the given architecture. This selection involves two steps.

First, we use a combination of the Mixed–Radix [3] and the Prime Factor Algorithm (PFA) [3] splittings to generate a large number of possible factorizations for a given transform size. Next, we select the fastest factorization in terms of the actual execution time on the given architecture. This approach is time consuming and may not be feasible for all applications. An alternative approach uses the codelet performance data to estimate the cost of each factorization and chooses the best scheme possible. The choice of the approach is left to the user.

3.2 Library of FFT Codelets

The FFT library contains a number of composable blocks of code, called *codelets*, each computing a part of the transform. The 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. We divide the codelet optimization into a number of levels. The first level optimization involves reduction of the number of arithmetic operations for each DFT codelet. The next level of optimization involves the memory hierarchy. In current processor architectures, memory access time is of prime concern for performance. Hence, it is essential to make use of the memory hierarchy in such a way as to minimize the impact of memory systems not capable of delivering data and instructions at a rate needed for full utilization of functional units. Optimizations involving memory accesses are architecture dependent and are performed only once during the installation of the library.

The codelets in our library are generated using a special purpose compiler that we have developed. We used this automatic code generation approach because the actual coding and optimization of the codelets become very tedious and difficult for transform sizes greater than five. For this reason many authors have found it convenient to build the codelets by using different ways of automatic code generation. Our code generator is written in C. It can produce DFT codelets of arbitrary size, direction (forward or inverse), and rotation (for PFA). It first generates an abstraction of the FFT algorithm by using a combination of Rader's algorithm [6], the

Mixed–Radix algorithm [3] and the PFA. The next step is the scheduling of the arithmetic operations such that memory accesses are minimized. We make effective use of temporary variables so that intermediate writes use the cache instead of writing directly to memory. We also use blocking techniques so that data residing in the cache is reused the maximum possible number of times without being written and re–read from main memory.

Finally, the abstract code is unparsed to produce the desired C code. The output of the code–generator is then compiled to produce the executable version of the library. Any ANSI C compliant compiler can be used for this compilation. The strategy to produce codelets that can be compiled with any ANSI C compliant compiler enables the ultimate portability, since there is no need to install any particular compiler on the platform targeted for the UHFFT library installation. However, the strategy does introduce a degree of uncertainty with respect to the performance that will be achieved, and stability of the optimizations being used in generating the codelets. Though we have observed that the output generated by different compilers on different processor architectures vary greatly, the optimizations nevertheless seem to be quite stable in the sense that the choices for code arrangement of a particular codelet that we expect to perform best on a particular architecture in most cases we have studied is independent of the compiler used. As for compiler optimization options, we have noticed that for most compilers and most architectures, the optimization level "02" produces the best performance for most codelets. Since all the codelets are straight line codes without loops, higher level optimizations generally tend to affect the performance of the codelets in a negative way.

Once the executables for the library are ready, we benchmark the codelets to test its performance. These benchmark tests are conducted for various input and output strides of data. The results of these performance tests are then stored in a database that is used later by the execution plan generator algorithm during the initialization phase of an FFT computation. The structure of the library is given in Figure 1.

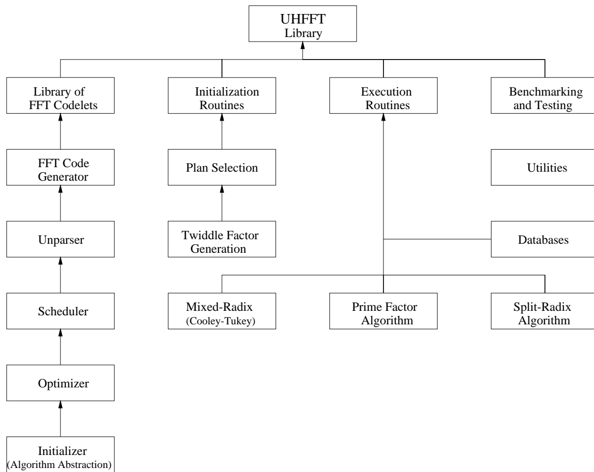


Figure 1: UHFFT Library Organization.

3.3 Execution Plan Generation

Once we have an efficient library of DFT codelets, we can create FFT routines for various sizes using these codelets as building blocks. For a given FFT size we first create an execution plan which determines the codelets that are going to be used for that FFT size and also the order in which the codelets are going to be used. Given the parameters of the problem, the initialization routine attempts to select a strategy that minimizes the execution time on the given architecture.

The basis for generating execution plans are the library of codelets and two databases: the *codelet database* storing information about codelet execution times, and the *transform database* that stores information about the execution times for entire transforms. The codelet database is initialized during installation of the library as a part of the benchmarking routine. The transform database stores the best execution plan for different size transforms. The transform database is initialized for some of the popular FFT sizes during installation (such as power of 2 sizes) and is updated every time a new transform size is executed.

For transform sizes that are not in the database, an execution plan must be created and this can be done in two different ways.

The first method is to empirically find the execution plan that minimizes the execution time by executing all possible plans for the given size, and choose the plan with the best performance. This method ensures that the plan selected will indeed result in the smallest execution time for all choices possible within the UHFFT library, but the time required to find the execution plan may be quite large for large size FFTs. So, unless many transforms of a particular size are needed this method is not practical.

The second method is based on estimating the performance of different execution plans using the information in the codelet database. For each execution plan feasable with the codelets in the library the expected execution time is derived based on the codelets being used in the plan, the number of calls to each codelet, and the codelet performance data in the codelet database. The estimation algorithm also takes into account the input and output strides and transform direction (forward or inverse). It also accounts for the twiddle factor multiplications for each plan as the number of such multiplications depend on the execution plan.

For large transform sizes with many factorizations, the estimation method is considerably faster than the empirical method. The quality of the execution plan based on the estimation approach clearly relies heavily on the assumption that codelet timings can be used to predict transform execution times, and that the memory system will have a comparable impact on all execution plans.

The list of codelets, execution strategy, twiddle factors and other information needed by the application to call the FFT routine are stored in a special structure called the *FftPlan*. Once the execution plan for a given transform size is generated, the application can use the structure to compute any number of FFTs of the given size.

4. PERFORMANCE ANALYSIS

In this section we analyze the performance and efficiency of our FFT library. Since the optimization is performed on two levels, we analyze them separately. We first compare the different algorithms for the FFT codelets and analyze the performance of the codelets on different platforms. Next, we analyze the high level optimization of the execution strategy and compare the performance of different strategies. Before we start the analysis, we first briefly describe the different hardware platforms and environments that we have used to test our library.

4.1 Target Hardware Architectures

We have evaluated the library for performance on the IBM–SP2, SGI–2000, HP–Exemplar and Intel Pentium systems. The SGI Origin 2000 at NCSA has 1528 MIPS R10000 64–bit processors [8] of which 760 operate at 195 MHz and the remaining operate at 250 MHz. We used the 250 MHz processors with the IRIX 6.5.1 operating system for our tests. The SGI R10000 processor supports four instructions per cycle, i.e., two integer and two floating–point instructions plus one load/store per cycle. Thus, peak performance achievable is 500 MFlops (Flop = floating–point operations per second) per processor. This processor has as primary caches a 32 KB two–way set–associative on–chip instruction cache and a 32 KB two–way set–associative, two–way interleaved on–chip data cache with LRU replacement. It also has a 4 MB two–way set–associative L2 secondary cache per CPU. The SGI R10000 has 64 physical registers, each 64 bits wide.

The IBM SP2 at the University of Houston has 64 IBM RS 6000 [9] processors, each running its own copy of the IBM Unix clone AIX 4.3. 56 processors operate at a speed of 120 MHz and 8 processors operate at 135 MHz. We used the 120 MHz processors for the evaluation of our library. These processors have a memory of 128 MB each. Each processor has a data cache of size 128 KB and an instruction cache of size 32 KB. There is no level–two cache. The cache line is 256 Bytes wide while the processor and memory interface widths are 8 32–bit words. The IBM RS 6000 has 32 64–bit floating–point registers.

The IBM RS 6000 can execute up to six instructions per cycle (one branch, one conditional register, two fixed–point, and two floating–point). It has two floating–point units (FPUs), each capable of finishing a fused mult–add operation ($x = a * b + c$) every cycle. Thus, the peak flop rate is 4 times the clock rate.

The HP Exemplar X–Class server installed at the Center for Advanced Computing Research (CACR) is based on the HP PA8200 RISC microprocessor [10] operating at 180 MHz, which due to its pipelined and superscalar architecture is capable of 720 MFlops peak. Each processor has a 1 MB direct–mapped data cache, a 1 MB instruction cache and is running HP–UX, Hewlett–Packard's version of UNIX.

We used University of Houston Intel Pentium–II [11] PCs operating at 400 MHz to test the performance of our library on the Intel family of processors. The Intel Pentium–II is a 32–bit CISC (Complex Instruction Set Computer) microprocessor. The Intel Pentium–II has a 32 KB non–blocking, level–one cache of which 16 KB is reserved as instruction cache and 16 KB is used as data cache. The processor has a dedicated 64–bit cache bus. It also has a 512 KB unified, non–blocking, level–two cache. The speed of the level–two cache scales with the processor core frequency. The Intel Pentium–II has one pipelined FPU for supporting 32–bit and 64–bit arithmetic. The processor has 8 registers, 32 bit wide each. The processors we used had a memory of 128 MB and were using the Linux 2.0.36 operating system. A summary of the characteristics for all processors used for the performance analysis is given in Table 1.

4.2 Analysis of FFT Codelet Algorithms

In this section we present results for the selection of algorithms for the codelets of the UHFFT library. Different algorithms are used for codelets of different sizes. The algorithm resulting in the fewest operations for a given codelet is chosen. Then, we verify that in fact the selected algorithm also result in the fewest instruction cycles when the codelets are compiled with the gnu C compiler version 2.7.23.

In Table 2 we compare the number of floating–point operations for the Mixed–Radix, PFA and Split–Radix algorithms for various transform sizes. For all cases, Rader's algorithm is used to generate codelets of prime size transforms (N=7, 11, 13, 17, 19, 23 etc.). The PFA has fewer arithmetic operations than the Mixed–Radix algorithm for transform sizes that have relatively prime factors (N= 6, 10, 12, 15 etc.). The Split–Radix algorithm results in a reduction of operations for transform sizes that are a power of 2. The reductions can be seen for sizes 16 and above (N = 16, 32, 64 etc.).

Since Rader's algorithm uses transforms of size $N - 1$ to generate prime size transforms of size N, the savings in the number of operations in the codelet of size $N-1$ is reflected in the size N codelets too. Rader's algorithm performs a convolution that involves a forward and inverse transform of size $N - 1$. Hence, the savings in the number of operations in transforms of size $N - 1$ is reflected twice in the codelet for size N transforms. This we clearly see in codelets of size $N = 7, 13, 19$ and 23 for the PFA algorithm and for $N = 17$ for the Split–Radix (SR) algorithm. The algorithms chosen for the UHFFT library are specified in bold font in Table 2.

The codelets generated by the automatic code generator are converted to executable binary code by the compiler for the target platform. The performance of the codelets depend heavily on the output generated by the compiler on different architectures.

In Table 3 we compare the number of cycles required for different codelets when compiled with the *gcc* compiler *version 2.7.2.3* for Intel Pentium–II processors running *Linux 2.0.36*. Several compiler options were tested to get the best result for the codelets. The *-O2 -fomit-frame-pointer -malign-double* optimization option was used. The Performance Counter Library (PCL) [12] was used to evaluate the compiler output for the number of cycles, the number of floating–point instructions and the total number of instructions. The total number of instructions per floating–point instruction remains almost constant ($\approx$ 3) while the total number of instructions per cycle is in the range 1.2 – 1.5.

Processor	SGI R10000	IBM RS 6000	HP PA8200	Intel Pentium-II
Clock Speed	250 MHz	120 MHz	180 MHz	400 MHz
Peak Performance	500 MFlops	480 MFlops	720 MFlops	400 MFlops
Primary Data Cache	32 KB (32 B)	128 KB(256 B)	1 MB	16 KB (32 B)
Secondary Data Cache	4 MB(128 B)	None	None	512 KB
Instruction Cache	32 KB	32 KB	1 MB	16 KB
Cache Associativity	2–way	2–way	1–way	4–way
Operating System	IRIX 6.5.1	AIX 4.3	HP–UX	Linux 2.0.36

Table 1: Processor characteristics.

Transform	UH MR		UH MR + PFA		UH MR+PFA+SR	
Size	Adds	Mults	Adds(diff)	Mults(diff)	Adds(diff)	Mults(diff)
2	**4**	**0**				
3	**12**	**4**				
4	**16**	**0**				
5	**40**	**12**				
6	40	16	**36(4)**	**8(8)**		
7	92	52	**84(8)**	**36(16)**		
8	**52**	**4**				
9	**80**	**40**				
10	108	40	**100(8)**	**24(16)**		
11	236	116	**220(16)**	**84(32)**		
12	104	32	**96(8)**	**16(16)**		
13	232	108	**216(16)**	**76(32)**		
14	224	128	**196(28)**	**72(56)**		
15	196	88	**180(16)**	**56(32)**		
16	148	28			**144(4)**	**24(4)**
17	328	116			**320(8)**	**108(8)**
18	212	112	**196(16)**	**80(32)**		
19	460	292	**428(32)**	**228(64)**		
20	264	96	**240(24)**	**48(48)**		
21	384	232	**336(48)**	**136(96)**		
22	536	272	**484(52)**	**168(104)**		
23	1116	628	**1012(104)**	**420(208)**		
24	274	86	**252(22)**	**44(42)**		
25	**432**	**184**				
32	388	108			**372(16)**	**84(24)**
36	508	248	**464(44)**	**160(88)**		
45	824	436	**760(64)**	**308(128)**		
64	964	332			**912(52)**	**248(84)**

Table 2: Operations count for UHFFT generated codelets.

Transform Size	UH MR			UH MR + PFA			UH MR+PFA+SR		
	Total Instr.	FP Instr.	Total Cycles	Total Instr.	FP Instr.	Total Cycles	Total Instr.	FP Instr.	Total Cycles
2	37	4	29	37	4	29	37	4	29
3	78	16	64	78	16	63	78	16	63
4	88	16	63	88	16	63	88	16	63
5	181	52	132	181	52	132	181	52	132
6	200	56	148	173	44	127	173	44	127
7	413	144	306	365	120	268	365	120	271
8	233	56	165	233	56	165	233	56	165
9	380	120	282	380	120	282	380	120	282
10	472	148	322	406	124	285	406	124	280
11	940	352	672	842	304	596	842	304	595
12	456	136	337	412	112	284	412	112	284
13	950	340	693	853	292	607	853	292	607
14	963	352	698	775	268	601	775	268	601
15	821	284	603	713	236	507	713	236	508
16	623	176	453	623	176	453	607	168	443
17	1243	444	911	1243	444	911	1224	428	905
18	969	324	742	847	276	650	847	276	650
19	1950	752	1478	1734	656	1323	1734	656	1323
20	1076	360	754	919	288	660	919	288	658
21	1630	616	1219	1345	472	1050	1345	472	1050
22	2117	808	1524	1776	652	1300	1776	652	1300
23	4234	1744	9987	3590	1432	4721	3590	1432	4723
24	1146	360	848	1025	296	754	1025	296	754
25	1702	616	1228	1702	616	1226	1702	616	1226
32	1611	496	1182	1611	496	1182	1530	456	1144
36	2194	756	1604	1891	624	1495	1891	624	1495
45	3415	1260	2549	2978	1068	2355	2978	1068	2354
64	3943	1296	9754	3943	1296	9854	3702	1160	7344

Table 3: Operations count for UHFFT generated codelets on Intel Pentium-II.

4.3 Performance of the FFT Codelets

For the performance evaluation of the FFT codelets in the UHFFT library, we have benchmarked the codelets for transforms of size 32 or less for a range of input and output strides for the data. For our benchmarking we chose strides that are powers of 2. Since the sizes of cache lines, cache and memory usually are equal to some power of 2, we expect to catch some of the worst performance behavior this way. Each reported data item is the average of multiple runs, such that the codelets are executed for at least one second. This was done to ensure that errors due to the resolution of the clock are not introduced in the results presented.

We see that for all platforms considered, the performance decreases considerably for large data strides. If two or more data elements required by a particular codelet are mapped to the same physical block in cache, then loading one element results in the expulsion of the other from the cache. This phenomenon known as *cache trashing*occurs most frequently for strides of data that are powers of two because data that are at such strides apart are usually mapped to the same physical blocks in cache depending on the type of cache that is used by the particular architecture. For all architectures, the sharp decrease in performance due to cache trashing occurs when:

$$datapoint_size * stride * \frac{codelet_size}{2} > \frac{cache_size}{Associativity}$$

where $datapoint_size$ is the size of one data element (for complex data with 8 Byte real and imaginary data, each data point is of 16 Bytes), $codelet_size$ is the number of data elements being transformed by the codelet, $cache_size$ is the total size of the cache in Bytes, $stride$ is the data access stride, and $Associativity$ is the type of cache being used by the architecture. We present the performance of codelets as a fraction of the peak performance achievable for each architecture and relate the observed sharp decrease in performance at particular strides to the above model of cache behavior.

4.3.1 The SGI R10000

In this section we present the performance of the UHFFT codelets for power of 2 input and output strides on the SGI R10000 processor. The performance is seen to be symmetric with respect to input and output strides. Thus, reads and writes from cache affect the performance in a similar manner. The SGI R10000 processor has two–way set–associative caches ($Associativity = 2$) for both level–one and level–two caches, i.e., a data point in memory may be mapped to one of two possible physical blocks in cache. The level–one cache is of size 32 KB and the level–two cache is of size 4 MB. No cache trashing occurs for codelet of size 2 (and 3 because a size 3 transform uses size 2 transforms for its computations), since the cache can hold both data elements for a size 2 transform regardless of the stride. For larger codelets, more data needs to be present in the cache at the same time and trashing occurs when a conflict occurs. A sharp drop in performance due to level–one cache trashing for the codelets occurs at the following strides:

$$stride > \frac{32KB}{16 * 2 * \frac{codelet_size}{2}} = \frac{2^{11}}{codelet_size}$$

and due to level–two cache trashing at the following strides:

$$stride > \frac{4MB}{16 * 2 * \frac{codelet_size}{2}} = \frac{2^{18}}{codelet_size}$$

We have illustrated the performance of two codelets as a fraction of the peak achievable performance on the processor in Figure 2. In Figure 3 we present the average performance for each codelet along with the range of performances for each of them. The peak performance is achieved for strides that are smaller than the cache line size (128 Bytes) of the processor.

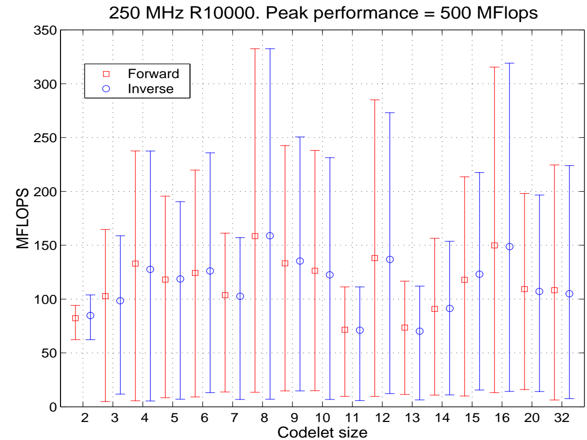


Figure 3: 250 MHz SGI R10000 UHFFT codelet strided performance.

4.3.2 The IBM RS 6000

The performance of two UHFFT codelets on the IBM RS 6000 processor for various input and output strides is illustrated in Figure 4. As for the SGI R10000 the performance is presented as a fraction of the peak achievable performance. The peak processor performance for the IBM RS 6000 processor is 480 MFlops. The IBM RS 6000 processor has only one level of cache and the cache is two–way set–associative ($Associativity = 2$) of size 128 KB. The small cache size (128 KB) of this configuration causes the performance of the codelets to drop drastically at relatively small values of the input or output strides. The performance drop due to cache trashing occurs when

$$stride > \frac{128KB}{16 * 2 * \frac{codelet_size}{2}} = \frac{2^{13}}{codelet_size}.$$

Hence, the performance of codelets drop for smaller and smaller strides as the codelet size increases. The typical codelet performance is shown in Figure 4. In Figure 5 we present the average performance for each codelet along with the range of performances for each of them.

4.3.3 The HP PA8200

As compared to the IBM RS 6000, the larger cache (1 MB) and faster processor (180 MHz) allows the code to perform much better on the HP PA8200. The performance of the codelets is very symmetric with respect to the input and output strides for data. The HP PA8200 processor has only one level of cache and the cache is direct mapped or one–way set–associative ($Associativity = 1$) of size 1 MB. The direct

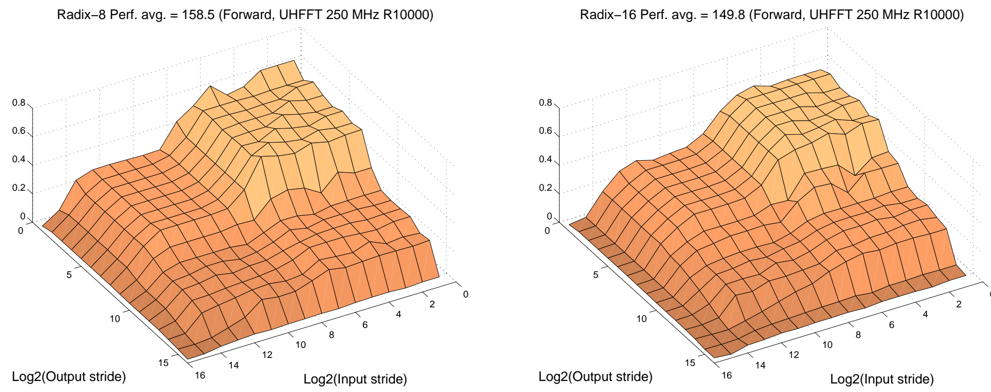


Figure 2: 250 MHz SGI R10000 performance for codelets of size 8 and size 16 (fraction of the peak achivable performance.)

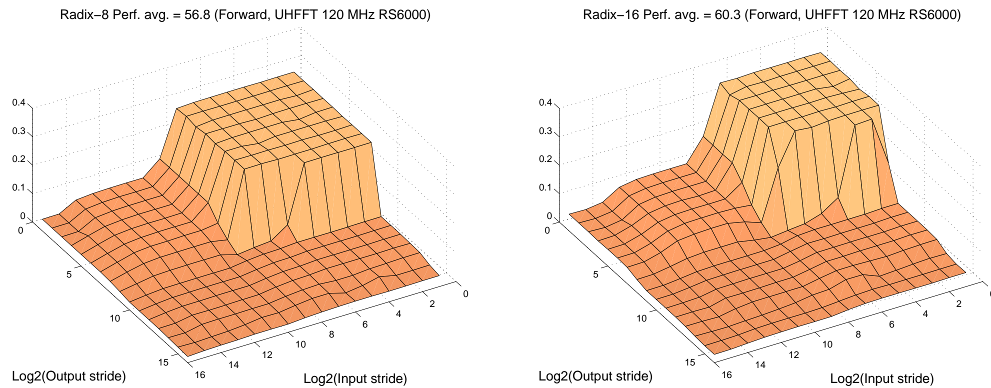


Figure 4: 120 MHz IBM RS 6000 performance for codelets of size 8 and size 16 (fraction of the peak achievable performance.)

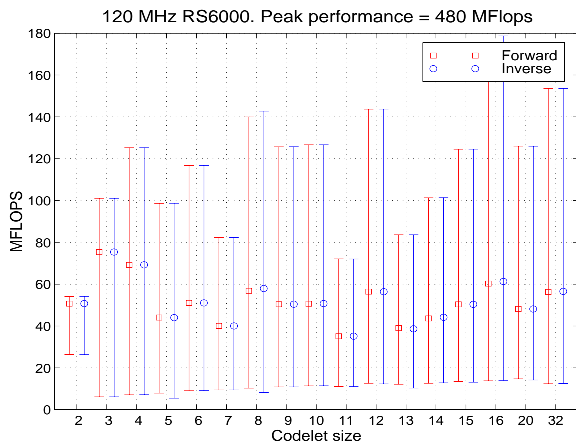


Figure 5: 120 MHz IBM RS 6000 UHFFT codelet strided performance.

mapped cache can hold a particular data element in only one specific block of the cache. Hence, cache trashing occurs for all codelets once the data strides exceed a particular size. The sharp drop in performance due to cache trashing for the codelets occurs at the following strides:

$$stride > \frac{1MB}{16 * 1 * \frac{codelet_size}{2}} = \frac{2^{17}}{codelet_size}$$

The sharp drop in performance can be observed in Figure 6. The advantage of the larger cache can be seen as the performance drop due to cache trashing occurs much later as compared to the IBM RS 6000. Since the performance data for all codelets is very similar, we only present complete codelet performance data for a few of them on this processor. In Figure 7 we present the average performance for each codelet along with the range of performances for each of them.

4.3.4 The Intel Pentium–II

The small data (16 KB) and instruction (16 KB) cache on the Intel Pentium–II processors affect the performance of large codelets (size 16 and 32 for the benchmarked codelets) adversely. The low number of registers (8) also makes the performance of codelets with large number of data points drop drastically. From Figure 9, we see that the codelet of size 4 performs the best on this processor as the codelet and data both fit in the primary instruction and data cache (16K each) respectively. Moreover, the Intel Pentium–II processor has a four–way set–associative cache. Hence, no cache trashing occurs for codelets up to size 4. A sharp drop in

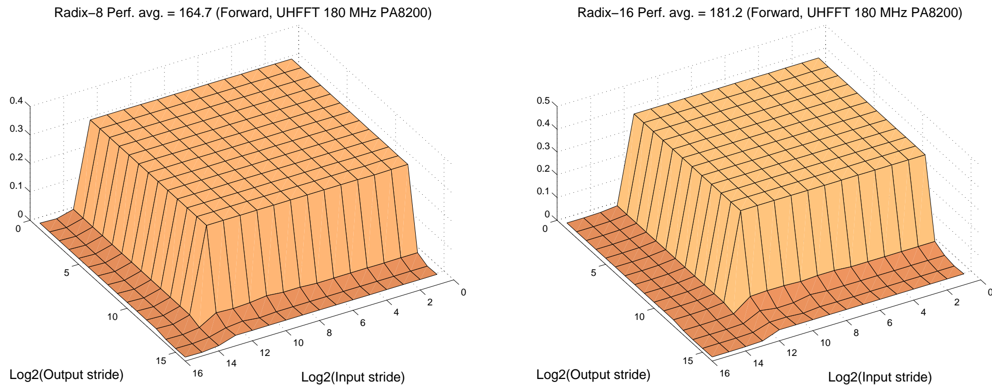


Figure 6: 180 MHz HP PA8200 performance for codelets of size 8 and size 16 (fraction of the peak achievable performance.)

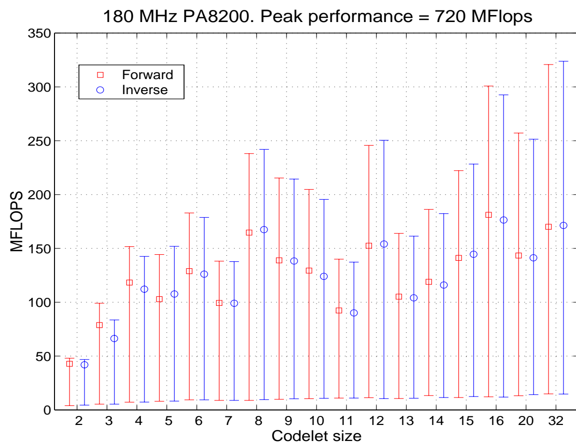


Figure 7: 180 MHz HP PA8200 UHFFT codelet strided performance.

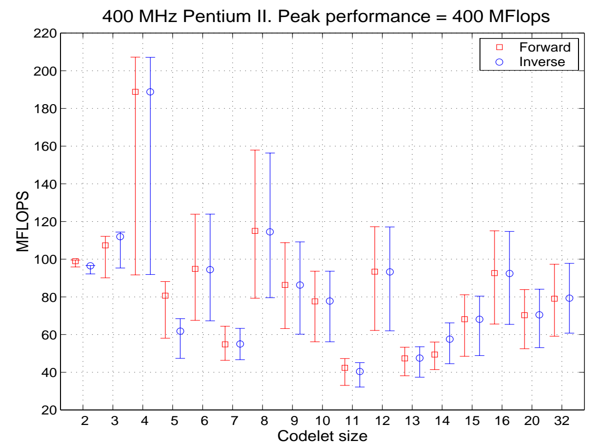


Figure 9: 400 MHz Intel Pentium–II UHFFT codelet strided performance.

performance due to level–one cache trashing for the larger codelets occur at the following strides:

$$stride > \frac{16KB}{16 * 4 * \frac{codelet_size}{2}} = \frac{2^9}{codelet_size}$$

and the performance drop due to level–two cache trashing occurs at strides:

$$stride > \frac{512KB}{16 * 4 * \frac{codelet_size}{2}} = \frac{2^{14}}{codelet_size}$$

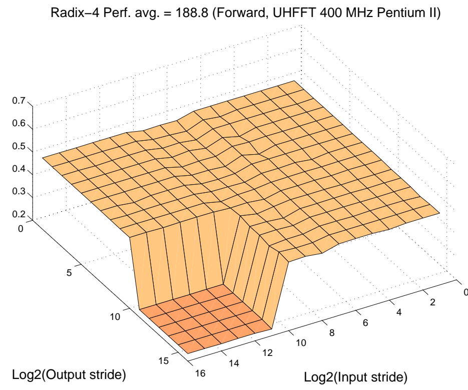
The Intel Pentium–II processor executes reads from memory in a speculative manner. This reduces the number of cache misses for reads as compared to the writes. Writes to cache when missed are directly written to memory in the Intel Pentium–II processor. Hence the performance drop due to large data output strides is greater than that due to large input strides (see Figure 8).

4.4 Analysis of Execution Strategy

In this section we report performance data for different execution plans for the UHFFT. As far as the user of the library is concerned, the performance of the selected execution plan is what is important for the performance of the application that uses the library. The reported execution times include the time for the computation of the transform using the particular execution plan but excludes the time for twiddle factor computation and the time for the generation of the execution plan.

It is impossible to list here the performance of all plans for all possible FFT sizes. In Figures 10, 11, 12, and 13 we illustrate the performance sensitivity for some FFT sizes that are powers of 2, as they are the most commonly used sizes in scientific codes. The performance of the different plans are presented sorted in order of decreasing performance, the *plan index* being the relative order of the plan with respect to performance. Thus, the plan with the smallest execution time is given index 1, the plan with the second smallest execution time is given index 2, etc. Table 4 lists the performance of all plans for an FFT of size 16 for the SGI R10000, while Tables 5, 6 and 7 give the same information for the IBM RS 6000, the HP PA8200 and the Intel Pentium–II respectively.

Figures Figure 10, Figure 11, Figure 12, and Figure 13 show the performance of best plans for selected sizes, and different architectures.



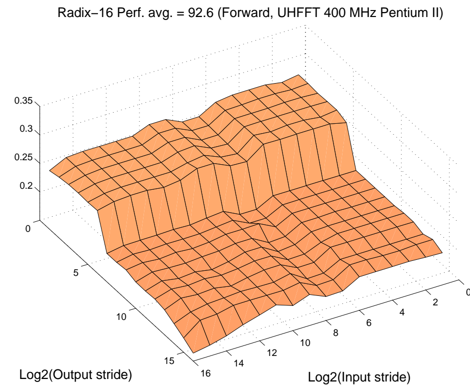


Figure 8: 400 MHz Intel Pentium-II performance for codelets of size 4 and size 16 (fraction of the peak achievable performance.)

Plan index	Plan	Time (sec)	MFlops
1	16	1.493e-06	214.32
2	2 8	4.554e-06	70.27
3	4 4	5.644e-06	56.69
4	8 2	7.210e-06	44.38
5	2 2 4	8.387e-06	38.15
6	2 4 2	1.048e-05	30.53
7	4 2 2	1.099e-05	29.10
8	2 2 2 2	1.371e-05	23.33

Table 4: 250 MHz SGI R10000 execution plan performance for size 16 FFTs.

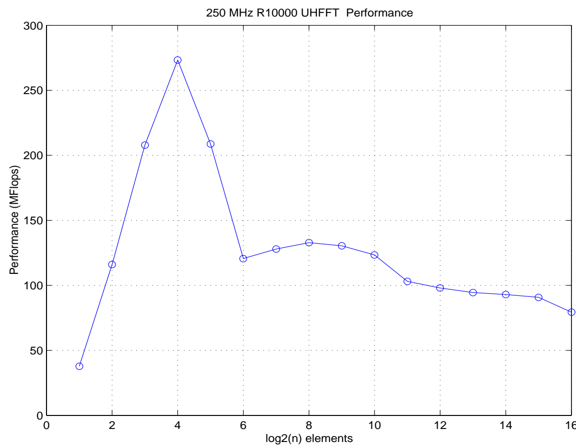


Figure 10: 250 MHz SGI R10000 execution plan performance.

Plan index	Plan	Time (sec)	MFlops
1	16	2.739280e-06	116.82
2	2 8	7.734200e-06	41.37
3	4 4	9.410750e-06	34.00
4	8 2	1.254800e-05	25.50
5	2 2 4	1.349130e-05	23.72
6	2 4 2	1.668680e-05	19.18
7	4 2 2	1.761220e-05	18.17
8	2 2 2 2	2.172130e-05	14.73

Table 5: 120 MHz IBM RS 6000 execution plan performance for size 16 FFTs.

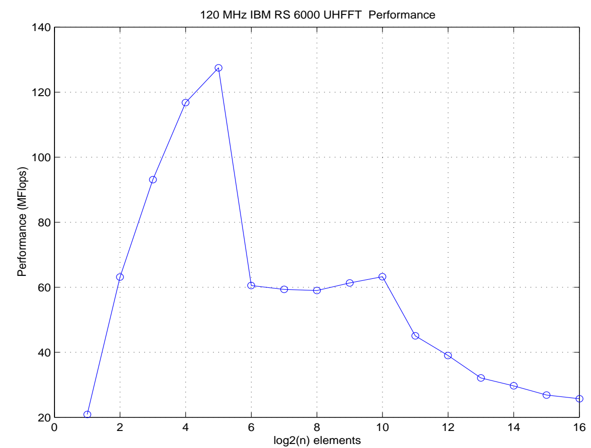


Figure 11: 120 MHz IBM RS 6000 execution plan performance.

Plan index	Plan	Time (sec)	MFlops
1	16	1.874480e-06	170.71
2	2 8	7.932890e-06	40.34
3	4 4	9.535160e-06	33.56
4	8 2	1.222470e-05	26.18
5	2 2 4	1.542790e-05	20.74
6	2 4 2	1.837990e-05	17.41
7	4 2 2	1.954020e-05	16.38
8	2 2 2 2	2.533680e-05	12.63

Table 6: 180 MHz HP PA8200 execution plan performance for size 16 FFTs.

Plan index	Plan	Time (sec)	MFlops
1	2 8	2.879530e-06	111.13
2	16	3.104810e-06	103.07
3	4 4	3.774120e-06	84.79
4	8 2	5.263150e-06	60.80
5	2 2 4	5.283630e-06	60.56
6	2 4 2	6.362200e-06	50.30
7	4 2 2	6.622240e-06	48.32
8	2 2 2 2	8.605230e-06	37.19

Table 7: 400 MHz Intel Pentium–II execution plan performance for size 16 FFTs.

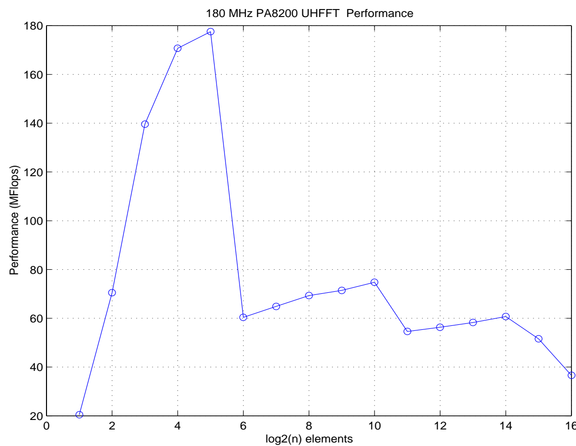


Figure 12: 180 MHz HP PA8200 execution plan performance.

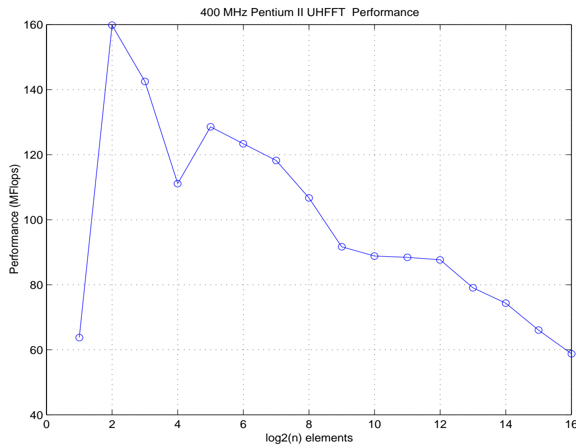


Figure 13: 400 MHz Intel Pentium–II execution plan performance.

5. CONCLUSION AND FUTURE WORK

We have evaluated the UHFFT library on multiple platforms and seen that we achieve good performance on all architectures. The adaptive approach that we have chosen for the library is shown to be an elegant way of achieving both portability and good performance.

Straight line code for DFTs of moderate size prove to be more efficient than constructing them with smaller kernels. For example, a straight line DFT codelet of size 16 is much more efficient than computing the size 16 DFT using smaller codelets (for example, using 4 size 2 codelets), as is shown in Table 5, Table 6 etc. But writing large straight line code is time consuming. Thus, the code generator approach to building the codelets is an efficient way to address the problem.

The overall design of the library is also seen to be flexible and extensible. The ease with which the whole UHFFT library can be regenerated allows us to easily incorporate new codelets and optimization rules to the library.

The UHFFT library is far from complete. We have so far only demonstrated the effectiveness of the adaptive algorithm and our approach to the design of the library. There are still many more optimizations possible. The UHFFT library also needs to be extended further to include real-to-complex, complex-to-real, sine and cosine transforms. Other applications such as convolution also can be included in the library.

We have so far implemented a uniprocessor library. Many applications that use FFT run on parallel systems with data distributed over many processors. Hence, a parallel implementation of the library is essential. The same approach of multi-level optimization can be applied to the parallel implementation too.

6. REFERENCES

[1] J.C. Cooley and J.W. Tukey. An algorithm for the machine computation of complex fourier series. *Math. Comp.*, 19:291–301, 1965.

[2] Matteo Frigo and Steven G. Johnson. The Fastest Fourier Transform in the West. Technical Report MIT-LCS-TR-728, MIT, 1997.

[3] Charles Van Loan. *Computational frameworks for the fast Fourier transform.* Philadelphia:SIAM, 1992.

[4] P. Duhamel and M. Vetterli. Fast Fourier Transforms: A Tutorial Review and a State of the Art. *Signal Processing*, 19:259–299, 1990.

[5] C. Temperton. A Note on Prime Factor FFT Algorithms. *Journal of Computational Physics*, 52:198–204, 1983.

[6] C. M. Rader. Discrete fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56:1107–1108, 1968.

[7] P. Duhamel and H. Hollmann. Split Radix FFT Algorithms. *Electronic Letters*, 20:14–16, 1984.

[8] *MIPS R10000 Microprocessor. Users Manual.* MIPS Technologies, Inc., 1996.

[9] IBM. RS6000 Technical specification. *http://www.rs6000.ibm.com/*, 1999.

[10] *Exemplar Programming Guide.* Hewlett-Packard, Inc., 1997.

[11] *Intel Architecture Optimization. Reference Manual.* Intel Corporation, 1999.

[12] Rudolf Berrendorf and Heinz Ziegler. PCL: The Performance Counter Library. *http://www.fz-juelich.de/zam/PCL/*, 1999.