A Super-Efficient Adaptable Bit-Reversal Algorithm for Multithreaded Architectures

Anne C. Elster and Jan C. Meyer
Department of Computer and Information Science
Norwegian University of Science and Technology
Trondheim, Norway
elster@idi.ntnu.no and janchris@idi.ntnu.no

Abstract

Fast bit-reversal algorithms have been of strong interest for many decades, especially after Cooley and Tukey introduced their FFT implementation in 1965. Many recent algorithms, including FFTW try to avoid the bit-reversal all together by doing in-place algorithms within their FFTs. We therefore motivate our work by showing that for FFTs of up to 65.536 points, a minimally tuned Cooley-Tukey FFT in C using our bit-reversal algorithm performs comparable or better than the default FFTW algorithm.

In this paper, we present an extremely fast linear bit-reversal adapted for modern multithreaded architectures. Our bit-reversal algorithm takes advantage of recursive calls combined with the fact that it only generates pairs of indices for which the corresponding elements need to be exchanges, thereby avoiding any explicit tests. In addition we have implemented an adaptive approach which explores the trade-off between compile time and run-time work load. By generating look-up tables at compile time, our algorithm becomes even faster at run-time. Our results also show that by using more than one thread on tightly coupled architectures, further speed-up can be achieved.

1. Introduction

Bit reversal (or shuffle) algorithms are an essential part of several Fast Fourier Transform (FFT) algorithms, including the original Cooley-Tukey FFT [8], [4]. Other applications include image transposition and generalized sorting of multidimensional arrays. Like the FFT, bit-reversal has been studied extensively in the literature. A nice summary looking at 30 methods for bit-reversing an array on uniprocessors was done by Karp in 1996 [16]. Here, Elster's linear algorithm [1] was shown to be the fastest, beating algorithms proposed by [10], [11], [12], [17], [14], among others.

The bit-reversal algorithm presented in this paper is, like Elster's original algorithm, a linear O(N) algorithm, but based on improvements suggested by Huff and Elster [13] and later implemented by Strandh and Elster [2]. By extending this algorithm with an adaptable scheme, we show that combining our resulting efficient implementations with a fairly straightforward Cooley-Tukey FFT, we can achieve results comparable to or better than the default FFTW [3].

2. Background and Related Work

The naive approach to bit reversal, such as the one used by the FFT algorithm in [18], is O(NlogN), for generating the bit-reversed sequence of N numbers. A typical approach is here to loop from 0 to N-1 and for each iteration compute the bit pattern corresponding to the reversed loop variable.

A slightly better approach is to keep a count of a binary number corresponding to the bit-reversed version of the loop variable, and then for each loop iteration, use binary arithmetic to increment the bit-reversed number. Before swapping the corresponding elements of the array, the algorithm performs a test to determine whether one of the numbers (index and bit-reversed index) is greater than the other. Only in this case are they exchanged, so $\frac{N-\sqrt{N}}{2}$ exchanges are performed. However, all indices and corresponding indices need to be generated and tested. Since reversing the bits of the index is O(logN), these algorithms are still O(NlogN).

Another approach is to compute the bit-reversed index from the normal index, which is also O(NlogN) since each bit of the index has to be examined and put into the bit-reversed index. However, if one uses a pre-computed table, the overall algorithm is still O(NlogN) unless we for any particular value of N, keep a table of size N containing a bit-reversed value for each value of the index itself. There are two problems with such tables. First, the tables have to be computed. If it is computed each time an FFT is called, nothing is gained. If it is instead computed statically, tables for all possible Ns will need to be kept, wasting storage. Note, however, we will be using smaller such tables in our adaptive approach described in more detail later.

2.1. Elster's Linear Bit Reversal Algorithm

Since Elster's algorithm is nice and simple, yet seems to be reinvented several times – even in recent years – less efficiently since its original publication, we include it here.

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

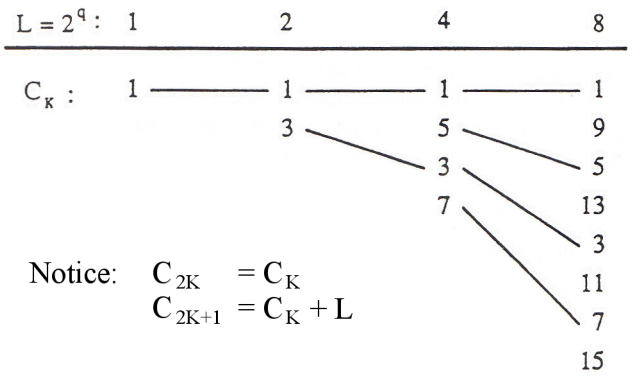

Figure 1. Discerning the Recursive Pattern of the c_ks

Table 1. Factorizing $r_{16}(k)$

	$n = 2^t = 2^4 = 16$			
$(k)_2$	k	$r_{16}(k)$		q
0000	0	$0 = 0 \cdot 2^4$	0000	0
0001	1	$8 = 1 \cdot 2^3$	1000	1
0010	2	$4 = 1 \cdot 2^2$	0100	2
0011	3	$12 = 3 \cdot 2^2$	1100	
0100	4	$2 = 1 \cdot 2^1$	0010	
0101	5	$10 = 5 \cdot 2^1$	1010	3
0110	6	$6 = 3 \cdot 2^1$	0110	
0111	7	$14 = 7 \cdot 2^1$	1110	
1000	8	$1 = 1 \cdot 2^0$	0001	
1001	9	$9 = 9 \cdot 2^0$	1001	
1010	10	$5 = 5 \cdot 2^0$	0101	
1011	11	$13 = 13 \cdot 2^0$	1101	4
1100	12	$3 = 3 \cdot 2^0$	0011	
1101	13	$11 = 11 \cdot 2^0$	1011	
1110	14	$7 = 7 \cdot 2^0$	0111	
1111	15	$15 = 15 \cdot 2^0$	1111	

The trick for going from an O(NlogN) algorithm to a linear algorithm is to view the computations of a bit-reversed index as a *mapping* from one sequence to another, rather than looking at each bit of a bit-reversed number. Elster's algorithm was found by factoring out the bit-reversed number sequence with powers of two. Table 1 shows how this was done for $N = 16 = 2^t = 2^4$. Note that $r_{16}(k) = c_k * 2^{t-q}$ with c_k odd, so the c_ks can be generated recursively as illustrated in Figure 1. Elster's linear sequential algorithm follows by noticing that the 2^{t-q} factors follow a similar recursive pattern. Figure 2 shows an iterative FORTRAN implementation of the resulting algorithm as originally described in [1]. Dr. Fred Gustavson helped Elster with this IBM 3090 encoding.

Having shown that the sequential bit-reversal takes O(N) in an algorithmic sense, Elster showed that the constant related to the linear factor is also quite small. By avoiding multiplications, and only using load, store, shift, and OR-masking, Elster ended up with the fast linear bit-reversal for uniprocessors as seen in Figure 3. Note that this algorithm can be further optimized by generating the first $\frac{n}{2}$ indices,

```
SUBROUTINE BITREV(t,c)
IMPLICIT INTEGER*4 (A-C),(a-c)
DIMENSION c(0:*)
n = 2 ** t
L = 1
c(0) = 0
DO 1 q=0, t-1
    n = n/2
    DO 2 j=0, L-1
        c(L+j) = c(j) + n
2   CONTINUE
  L = L*2
1 CONTINUE
RETURN
END
```

Figure 2. Linear Bit-Reversal in FORTRAN

```
x[0] = 0, x[1] = n/2
FOR L=2 to N-1 STEP 2:
    x[L]   = shift-right ( x[L/2] )
    x[L+1] = x[L] XOR x[1]
END FOR
```

Figure 3. Elster's Bit-Reversal

then the second half by flipping the least significant bits.

Unfortunately, this algorithm does use O(N) space, which on modern architectures can have serious impact on cache performance.

Karp [16] points out that Elster's algorithm is an interesting derivation of the one presented in [15]. This is no surprise given that Elster showed her algorithm to Van Loan and others right after she discovered it in 1987, and Van Loan helped her with the MATLAB style formulation of the algorithm [1]. It was the same year also presented at an IBM seminar at Cornell. Elster's algorithm was explicitly referenced in the main text of an earlier draft of [15], but in the final version she is referenced only as part of an exercise.

As described by Karp[16], Polge[9] proposed an algorithm which did not need extra storage, but it requires more extra computations than standard O(NlogN) algorithms.

Improvements of Elster's original algorithm suggested by Huff and Elster [13] and implemented by Strandh and Elster as described in [2], reduces the use of extra storage space to four registers. This idea is also alluded to by Karp [16].

3. Efficient Recursive Bit-Reversal

Subsection 3.1 describes the Elster-Huff-Strandh bit-reversal algorithm [13], [2], which shuffles N elements in less than $5N$ cycles. Subsection 3.2 presents an overview of its execution, which is the basis for later discussion of our new adaptable implementation.

3.1. The Basic Recursive Algorithm

The Elster-Huff-Strandh algorithm works by storing a pair of values which are maintained as bit-reversed counterparts

throughout a run. This is done by performing each operation on both values, using mirrored bitmasks to reverse their endianness relative to each other. The masks address two bits at a time, beginning with the MSB/LSB pair and shifting inwards towards the middle. The choice on how to apply the masks at each successive pair of intermediate values depends on the values they may attain by assigning the remaining bits in the middle. If the values are equal up to the remaining bits in the center, their comparative values may not yet be determined. If they differ, the greater/smaller relation between their values will hold for any assignment of the remaining middle bits, as bits of greater significance have already been assigned.

Let the mutually reversed pair of values be denoted V and V'. With the understanding that bit masks only address two specified bits, and that their relative shifted positions are determined by the step at which a mask is applied, we have 4 possible bit masks: $(0\cdots0,0\cdots0)$, $(1\cdots1,1\cdots1)$, $(1\cdots0,0\cdots1)$, and $(0\cdots1,1\cdots0)$.

Since the application of one mask to one value uniquely determines which must apply to the other, (V,V') can undergo 4 possible modifications at each step of the algorithm. The simplest case is when an already equal pair is assigned equal masks, maintaining the equality of the values. In this case, only $(0\cdots0,0\cdots0)$ and $(1\cdots1,1\cdots1)$ are appropriate pairs of masks. As an example, choosing alternating such steps for a pair of 6-bit values produces intermediate values $(V_0,V_0') = (0.....0,0....0)$ from applying $(0\cdots0,0\cdots0)$, $(V_1,V_1') = (01..10,01..10)$ from applying $(1\cdots1,1\cdots1)$, and $(V_2,V_2') = (010010,010010)$ from applying $(0\cdots0,0\cdots0)$. Maintaining equality at each step yields duplicate palindromes, which are not interesting.

The intermediate values of these cases are of greater interest, however, as each equal pair gives a starting point for assigning unequal mirror pairs to the middle bits. When inequality is introduced, all 4 mask pairs become relevant to subsequent steps. Note that applying both of the two unequal mask pairs to a pair of equal intermediate values corresponds to how identical pairs can be generated in opposite order. As an example, selecting each of them as the final step of the 6-bit example above produces $(011010,010110)$ and $(010110,011010)$, respectively. Generating both is redundant work, so we will restrict the number of applicable mask pairs to 3 for those steps which change an intermediate pair $V_i = V_i'$ into $V_{i+1} \neq V_{i+1}'$. We arbitrarily choose $(1\cdots0,0\cdots1)$, giving $V_{i+1} > V_{i+1}'$.

These considerations imply a simple recursive statement of the algorithm for an n-bit reversal in pseudocode, given in Figure 4. As the pseudocode and above discussion indicate, the algorithm demands that generated bit patterns are of even length. Mirroring the added middle bit in an odd-length pattern is a matter of finding the even-length pairs of length n-1, and duplicating them with infixes 0 and 1 in the center. This can be achieved by a constant-time extension

```
Input : n  (even) bit pattern length
generate_pairs ( 0, 0, 0, 1<<n, 1 )
FUNCTION generate_pairs ( V, V', step, l, r )
 // l is leftmost bit of mask , r is rightmost,
 // step tracks the length of the execution path
 IF ( step != n/2 ) THEN
   IF ( V == V' ) THEN
     // Intermediate values are equal still
     // (0...0, 0...0) step
     generate_pairs(V,V',step+1,l>>1,r<<1)
     // (1...1, 1...1) step
     generate_pairs(V+l+r,V'+l+r,step+1,l>>1,r<<1)
     // (1...0, 0...1) step
     generate_pairs(V+l,V'+r,step+1,l>>1,r<<1)
   ELSE
     // Intermediate values already differ
     // (0...0, 0...0) step
     generate_pairs(V,V',step+1,l>>1,r<<1)
     // (1...1, 1...1) step
     generate_pairs(V+l+r,V'+l+r,step+1,l>>1,r<<1)
     // (1...0, 0...1) step
     generate_pairs(V+l,V'+r,step+1,l>>1,r<<1)
     // (0...1, 1...0) step
     generate_pairs(V+r,V'+l,step+1,l>>1,r<<1)
   END IF
 ELSE
   IF ( V != V' ) THEN
     // Completed V and V' in n/2 steps
     output ( V, V' )
   END IF
 END IF
END FUNCTION
```

Figure 4. Recursive Statement of the Basic Algorithm

of the work involved in applying final results. Treating this otherwise trivial case can only make the presented algorithm less readable, so it will not be discussed further here.

3.2. Recursive Computation Tree

The pseudocode in Figure 4 proceeds by an exhaustive depth-first backtracking trace of a tree where the interesting pairs are generated at the leaf nodes. The structure of this tree features a set of nodes of degree 3 at the upper levels, corresponding to the two steps which produce equal intermediate values, and the one step which introduces inequality. Subtrees below the former cases retain the same structure, while subtrees below the latter have constant degree 4, with one step for each possible combination of masks. An example of the structure for a 4-bit reversal is given in Figure 5. Note that for the sake of simplicity, the pseudocode implies that traversal proceeds to the bottom even in the case where equality is retained throughout, filtering the output where the final results are equal. These cases are marked with parenthesized pairs in Figure 5. Letting the traversal proceed this far is redundant work, which can easily be omitted by splitting the two phases of recursion into separate, dedicated functions, and halting the recursion of the equal case one stage before completion.

Additionally, straightforward recursive implementation

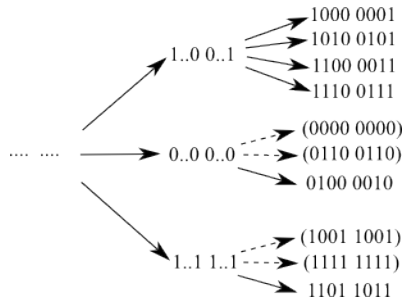

Figure 5. 4-bit Reversal Computation Tree

utilizes the run-time stack to store the state of computation for backtracking; as pointed out by Strandh and Elster[2], this state information can be retained in global variables by reversing each mask operation after call return instead. This can eliminate the need for establishing a stack frame for every function call, reducing the workload per step to a small number of shift, add and jump operations, and avoiding the memory traffic cost potentially caused by recursion. It also creates a sequential dependency between function calls, making it difficult to parallelize such an implementation.

4. Application to the Fast Fourier Transform on Modern Architectures

A common use of bit-reversal permutations is as a pre- or post-processing step to the Cooley-Tukey Fast Fourier Transform [8] [4], which computes its results in digit-reversed order. The algorithm achieves its favorable asymptotic time complexity by expressing an N-point transform value in terms of two $\frac{N}{2}$-point transforms (odd and even terms), suggesting a computation tree which may be approached in both depth-first or breadth-first manners. The depth-first approach favored by the highly efficient FFTW library [3] lends itself to recursive statement, and has the side-effect of implicitly reordering output during the computation as each partial transform is completed before proceeding. The FFTE library [6] also orders output implicitly, applying autosorting FFT on small partial transforms. Textbook implementations, on the other hand, often favor a breadth-first approach [7], which results in the need for a separate reordering phase which restores the output to natural order.

The existence of efficient FFT libraries which do not require a bit-reversal phase [3] [6] makes it appropriate to question whether bit-reversal algorithms have any significant practical mandate. This section argues that they do, by describing a prototype breadth-first FFT implementation with bit-reversal, measuring its performance, and outlining its suitability for multi-core processors.

4.1. CT-FFT Implementation

Our tested implementation was initially written as an in-place 1-dimensional breadth-first FFT, similar to the FORTRAN routine given in [4]. This program is a triply nested loop, ordering the computation by the size of partial transforms, next by sequence of applicable twiddle factors, with an innermost loop generating the indices of pairs and applying butterfly operations.

Initial runs using the `-O2` optimization flag of `gcc` were verified to produce correct results. They also indicated that the cost of complex multiplications in the innermost butterfly operations was reduced by compiler optimizations when multiplying real and imaginary components separately. Further improvements were made by separating the initial 2-point transform iteration into a single pass, thereby reducing the expression for each pair of values to additions.

In order to break sequential dependencies and improve locality, memory usage was extended to include workspace equal to the transform size N. This permits the butterfly operation to store the even and odd partial transforms sequentially in separate arrays, unifying them with a single sequential copy operation between stages. This uses an extra $\frac{N}{2}$ space, and reorders values throughout the computation such that the inputs to each butterfly in a phase are stored consecutively. With this improvement in memory locality, it is beneficial to reorder the innermost loops to one sequential pass over the intermediate values instead of ordering it by generating applicable twiddle factors on the fly. The remaining $\frac{N}{2}$ workspace was used to precompute a lookup table of these factors, as half of them are used in more than one stage of the computation. Finally, the bit-reversing stage included by Cooley et. al.[4] was replaced with the optimized Elster's algorithm described in Subsection 3.1.

It should be noted that none of the FFT optimizations are particularly novel: the benefits of substituting simpler arithmetic operations are all but universally recognized, and the locality improvement from reordering intermediate values is mentioned by Frigo and Johnson[3]. The purpose of describing our test program in detail is to illustrate that its simplicity would permit most programmers to create a similar implementation with little effort.

4.2. Test Results and Discussion

Timings for even powers of 2 from 2^8 through 2^{18} are given in Figure 6. Reported results were collected from test runs on AMD Athlon64 and Intel Core 2 architectures. Measurements are median values from 11 batches of 25-run averages. Sample size was chosen for quick execution, while producing repeatable results in the presence of OS jitter and minimal background load. Testing was done using the ESTIMATE planner of FFTW[3], and regular native C99

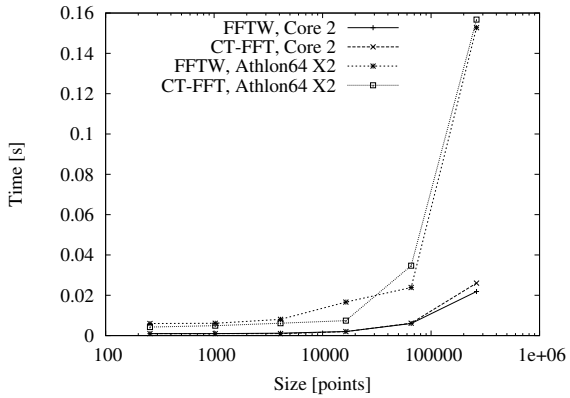

Figure 6. Even Power-of-Two FFT Timings on Athlon64 and Core 2

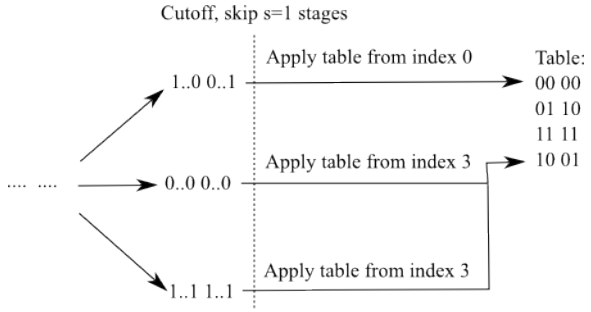

Figure 7. 4-bit Reversal Computation Tree, Last Stage Tabulated

complex arithmetic in our CT-FFT implementation. ESTIMATE mode causes FFTW to plan based on the number of operations, as opposed to tuning wrt. timings from a specific processor. The optimizations of our implementation are similarly generic, relying only on the principle of locality. While benchmarking the full potential of FFTW requires its PATIENT planning mode and multithreaded execution, such a comparison would also require similar tuning capabilities in our prototype, leading to a significant development effort beyond the scope of this work. Our results are therefore not intended to show peak achievable FFT performance on the target platforms, but merely to argue that bit-reversing breadth-first FFT approaches may still have applications.

The most notable feature of Figure 6 is the transition which occurs in the last two measurements ($N = 2^{16}$, $N = 2^{18}$), where the relative performances of our implementation and the FFTW on Athlon64 trade places in favor of FFTW. This can be explained by the transition between transform sizes which fit in the 256KB level 2 cache of the test processor. Comparing this with the Core 2 results, the test processor featured a 6MB L2 cache, in which all test cases fit comfortably. The resulting performance figures remain relatively close to each other on this architecture.

It is not surprising that the memory wall has a significant impact on an algorithm featuring a separate bit-reversal phase. The algorithm outlined in subsection 3.1 indicates a negligible arithmetic intensity (a handful of shifts and additions per mutually reverse pair), while the resulting memory traffic when executing a shuffle is bound to display poor locality: ordering the swap operations by the bits of one word will create an access pattern which alternates between upper and lower halves of the address space of the other.

What is interesting to note is that the performance gap between computation and memory access is sufficiently closed already at the L2 cache level that this effect becomes insignificant, even lending a slight performance edge to our simple prototype implementation on the Athlon64. This observation is of particular importance to multi-core processor design, as L2 cache size has displayed considerable growth in recent hardware generations, and contemporary multi-core architectures feature large L2 caches shared between multiple cores. This suggests that high-performance parallel FFT solvers for in-cache transforms may benefit from optimized, threaded bit-reversal.

5. Parallel Implementation on Multi-Core

Subsection 5.1 describes how a tradeoff between lookup table size and run-time computation forms a performance parameter which we target for parameterized adaptation[5]. Subsection 5.2 describes how this strategy is further exploited in our parallel implementation.

5.1. Parameterized Adaptation

The simplest strategy for implementing an N-element shuffle is through using a lookup table. As the bit-reversed pairs for a given pattern length are immutable, the pairs can be pregenerated once and for all using even the trivial algorithm which counts from 0 to $N-1$ and reversing each number bit by bit. When such a computation completes, its result can be stored in an array, reducing the work of sorting in bit-reversed order to a sequence of memory reads. Note that terminating recursion with s stages remaining means that the computation may be cut short along a path in the tree both with and without equality being preserved at the last intermediate stage. This means that the lookup table must contain tuples for equal pairs also, but taking care to order the lookup table with greater-than tuples collected at one end allows table access to proceed sequentially. Figure 7 shows this for a 4-bit reversal, with 1 stage tabulated.

The most obvious drawback of this strategy lies in the size of the table required. The table size grows linearly with input size, effectively doubling the memory requirement of the

array to be sorted. In addition to the potentially prohibitive storage requirement for large lookup tables, the relative costs of replacing a number of computational operations with memory transfers are not obvious with the non-uniform access times introduced by hierarchically organized memory systems. As witnessed in Figure 6, the cost of accessing a cached memory element can significantly reduce the impact of a program's memory access pattern at best, but caching both a large lookup table and an equally large array of data to be shuffled will either cause memory traffic, or reduce the size of admissible problem sizes by at least $\frac{1}{2}$.

Noting that the efficiency of a lookup table is strongly tied to its size, we expect a given processor to provide performance improvements in return for using lookup tables only up to a certain table size, determined by the balance of costs between its cache/memory access and computational operations. As these properties vary even between revisions of a single processor model, it is desirable to parameterize programs with respect to lookup table size, to simplify performance tuning for a specific machine. The idea tested in this work is to combine moderately sized tables with the recursively defined algorithm, exploiting the fact that the middle bits of intermediate pairs are undefined at the upper levels of the computation tree. Any number of middle bits can be precomputed into a set of bit masks, and recursion can end when it reaches the first pair of bits for which a mask has been computed. This replaces the lower levels of the recursion tree with a linear loop accessing a precomputed set, giving some control over the balance between table lookups and computation. Our expectation is that this will allow table size to be empirically adjusted to a processor's optimal point for table size vs. performance benefit.

5.2. Thread-level Parallelism

A side effect of tabulating a set of pregenerated masks is that it suggests a simple approach for thread-level parallelism. The table of masks which applies to the interior of the patterns is accessed in a read-only manner when it has been generated, which lets it be shared between cores without contention. This suggests a parallelization approach where the beginning and end bits of patterns can be generated on the fly, and cutting off recursion, reducing the remaining work to the sequential masking of all pregenerated table entries for the middle bits. The following subsections report performance figures collected from an implementation which is parameterized to perform this cutoff for a table of bit patterns of length $2s$, with s indicating how many levels of recursion have been cut short as in Figure 7. Single-thread and 2-thread versions are tested. The single-thread version alters its execution to an iterative application of all masks at the cutoff level. The 2-thread version features a master thread to generate the outer bits, dispatching the remaining work to a workpool where it is picked up by another thread which applies the tabulated values. The master thread joins the workpool when all pattern exteriors have been generated.

6. Performance Measurements

6.1. Experimental Methodology

Our tests were run in batches of 100 consecutive executions, and median timing values were collected to filter the offset from occasional background load interference. Time is reported in terms of clock cycles per shuffled element, and is normalized by a factor $\frac{1}{N}$, in order to present the relatively large range of problem sizes on comparable terms. The program performed a full bit-reversal permutation of a complex number array, like the input to the FFT program in Figure 6. The parameter space is explored from array sizes 2^8 through 2^{20}, with cutoff values from $s = 0$ through $s = \frac{N}{2} - 1$, and single vs. two-thread versions are covered. The omission of the cutoff value $s = \frac{N}{2}$ is made for the sake of presentation clarity. Our results already show that the table approach has reached diminishing returns by $s = \frac{N}{2} - 1$, so the inclusion of the final possible table size only obscured figures by requiring graphs with larger scale.

As the Core 2 test processor actually featured 4 cores, preliminary tests were performed using 4 threads, but performance results did not differ noticeably from the 2-thread version. This is probably a reflection of the fact that the L2 caches of the processor are shared between pairs of cores.

6.2. Results and Discussion

Our results are presented in Figures 8, 9, 10, and 11.

The first thing to notice in the presented results is that they verify our expectation of an optimal table size for various choices of s, which marks a turning point for where the precomputed table strategy yields performance benefits. This effect is most clearly visible in the single-thread results (Figures 8 and 10), but it is present also in the 2-thread version. To highlight this point, the 18-bit reversals for both 2-thread test sets are plotted together in Figure 12. The range of s is cut down in this figure to show the area where the optimal value is found. Figure 12 also makes it visible that the optimal table size varies not only with problem size, but also with the test processor. This justifies that code which admits a range of table sizes permits processor-specific tuning which can be automated.

Another notable result is that while cache size limitations result in very pronounced performance gaps in the single-thread results of Figures 8 and 10, the 2-thread results in Figure 9 and 11 do not display this sensitivity. This can be attributed to the fact that in the single thread implementation, execution will stall every time a memory request must be satisfied. This can account for substantial overhead due to the somewhat erratic access pattern of the swap operations.

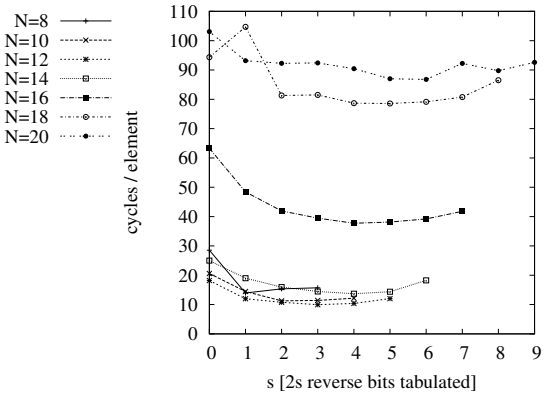

Figure 8. Single Thread Performances on Athlon64

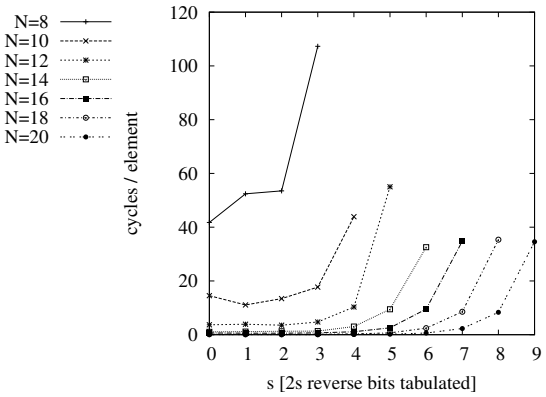

Figure 9. 2-Thread Performances on Athlon64

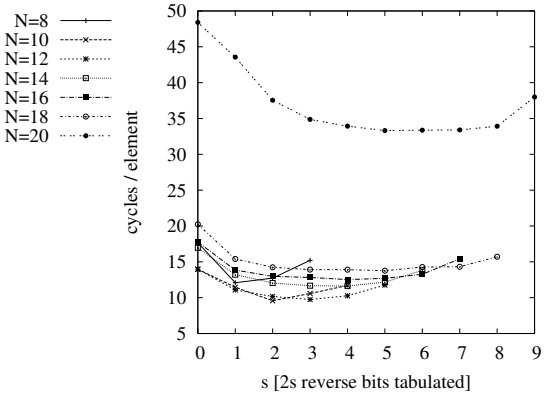

Figure 10. Single Thread Performances on Core 2

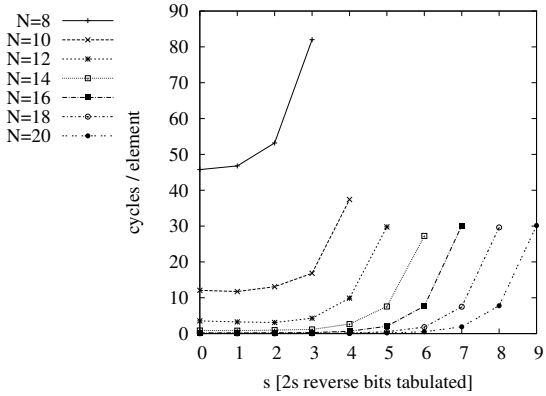

Figure 11. 2-Thread Performances on Core 2

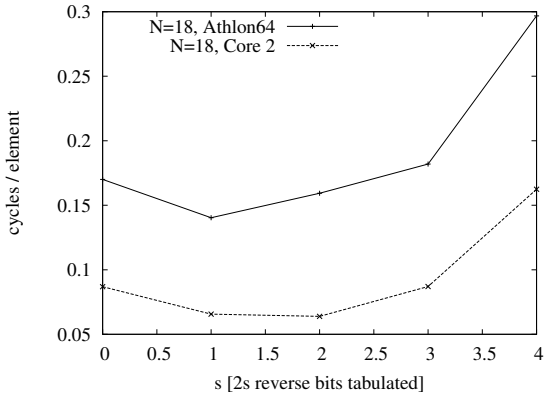

Figure 12. 2-Thread Performances for N=18, both architectures

The threaded version will not suffer from this to the same extent, since further partial patterns can be generated independently of the completion of each swap.

Figure 12 also shows that although the optimal table size for the compared processors differs between them, both cases provide the best tradeoff in exchange for reasonably small tables. This suggests that modern processors with large caches space can use them for storing lines containing the exchanged values themselves. This means that a separate bit-reversal phase may fill the cache before the computation of the transform, compensating in some measure for the time spent extending the overall computation with an additional phase. Establishing the magnitude of this effect would require profiling work beyond the scope of this paper, but it is a relevant consideration for future experiments.

Finally, our results indicate that the division of labor

between two threads substantially speeds up the bit-reversal algorithm in tightly coupled scenarios. While the parallel scalability of the approach is likely to remain limited, this still suggests that its application to in-cache Fourier transforms is an interesting direction for future research, suggesting a threaded implementation for shared-cache multicore processors as a natural first step.

7. Conclusions

On modern architectures, a diverse range of performance parameters can be found even between different processor models of a single architectural family. This creates a growing need for software to be adaptable to a range of hardware configurations, in order to provide portable performance.

Our adaptable bit-reversal implementation finds an optimal tradeoff between precalculated results and run-time computation on modern multithreaded architectures. This is motivated by the observation that a breadth-first FFT with a separate bit-reversal phase can provide competitive performance results for in-cache transform sizes.

The ideal table size for our test processors turns out to be small, which indicates that the strategy is well suited for coupling with further exploration of FFT optimizations on an appropriate scale. Our implementation is also suited for parallel implementation, displaying favorable speedup when distributed across 2 tightly coupled cores. We expect similar speedup for 3 or more cores, as long as the cores share cache.

8. Future Work

A natural extension to this work is to examine the relative costs of different FFT implementations for cacheable problem sizes more thoroughly. The coarse tests which justify a more sophisticated bit-reversal implementation in this work indicate that the relative rates of computation and communication on the chip-local level presents a different performance tradeoff to parallel implementations than larger problems, making it interesting to explore the limits of various algorithms on this level.

Finally, we note that finding the ideal table size for a given processor model is easily quantified, and would make a suitable candidate for an automated implementation which adapts the algorithm to the processor it is being generated for at compile time. One extension of our lookup table method is that recursion may also be cut off by calling a pregenerated subroutine with bit masks stored in instruction operands. In the terminology of the AEOS approach described by Whaley et. al.[5], this would correspond to *source code adaptation*, as opposed to the *parameterized adaptation* described here. Such an approach would utilize instruction cache, which may lead to different results. An optimal combination of these two methods is not obvious, and therefore presents an interesting parameter space to explore.

References

[1] A. C. Elster, *Fast Bit-Reversal Algorithms*, IEEE International Conf. on Acoustics, Speech, and Signal Processing 1989 (ICASSP'89), Vol. 2, pp. 1099–1102, May 1989.

[2] R. Strandh and A. C. Elster, *A Very Efficient Linear-time Logarithmic-Space Bit Reversal Algorithm*, Center for Numerical Analysis, Tech. rep. no CNA-288, The University of Texas at Austin, Oct. 1998

[3] M. Frigo and S. G. Johnson, *The Design and Implementation of FFTW3*, Proc. IEEE, Vol. 93, No. 2, pp. 216–231, Feb. 2005

[4] J. W. Cooley and P. A. Lewis and P. D. Welch, *The Fast Fourier Transform and Its Applications*, IEEE Transactions on Education, Vol. 12, No. 1, Mar. 1969

[5] R. C. Whaley and A. Petitet and J. Dongarra, *Automated Empirical Optimization of Software and the ATLAS Project*, Parallel Computing, Vol. 27, 2001

[6] D. Takahashi, *A Blocking Algorithm for Parallel 1-D FFT on Shared-Memory Parallel Computers*, in LNCS 2367, J. Fagerholm et. al., Ed. Springer-Verlag, 2002, pp. 380–389

[7] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, Addison-Wesley, ISBN 0-201-60078-1, 1992

[8] J. W. Cooley and J. W Tukey, *An Algorithm for the Machine Calculation of Complex Fourier Series*, Math. of Comput., Vol. 90, pp 297-301, Apr. 1965.

[9] R. J. Polge et al., *Fast Computation Algorithms for Bit Reversal*, IEEE Trans. on Computers, Vol C-23, No. 1, Jan. 1974.

[10] Gold and C. M. Rader. *Digital Signal Processing*, McHraw-Hill, 1969 (reprinted 1983).

[11] D. M. W. Evans, *An Improved Digit-Reversal Permutation Algorithm for the Fast Fourier and Hartley Transforms*, IEEE Transactions on ASSP,Vol. 35, pp 1105-1125, Aug. 1987.

[12] C. S. Burrus, *Unscrambling for Fast DFT Algorithms*, IEEE Transactions on ASSP, Vol 36, pp. 1086–1087, Jul. 1988.

[13] R. Huff, *Bit Reversal in Linear Time*, Course presentation notes, Computer Science Dept., Cornell University, 1990. http://www.idi.ntnu.no/~elster/doc/huff-notes.pdf

[14] J. Jeong and W. J. Williams, *A Fast Recursive Bit-Reversal Algorithm*, IEEE International Conf. on Acoustics, Speech and Signal Processing, Apr. 1990.

[15] C. F. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM Philadelphia, PA., 1992.

[16] A. H Karp, *Bit-Reversal on Uniprocessors*, SIAM Review, Vol. 38, No. 1, pp. 1–26, 1996.

[17] J. J. Rodriguez, *An Improved Bit-Reversal Algorithm for the Fast Fourier Transform*, IEEE International Conf. on Acoustics, Speech, and Signal Processing 1988 (ICASSP'88), pp. 1407–1410.

[18] W. H. Press. and B. P. Flannery and S. A Teukolsky and W. T. Vetterling, *Numerical Recipes in C*, Second Edition, Cambridge University Press, 1992.