A FASTER FFT IN THE MID-WEST

BY

ALEXANDER JIH-HING YEE

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Professor Marc Snir

ABSTRACT

FFT implementations today generally fall into two categories: Library generators (such as FFTW and Spiral) and specialized FFTs (such as prime95). Specialized FFTs have the obvious limitation of being specialized. However they are hand-tuned and generally offer superior performance. Library generators are generic and easier to port. But their performance is generally suboptimal.

We describe in this paper an FFT library that was built while paying special attention to locality. The library achieves significantly better performance than FFTW, for long vectors. Unlike FFTW or Spiral, the recursive decomposition of the FFT is not created by a library generator; it is created by macro expansion that has a few selectable parameters. This provides an interface that can be more easily modified by users.

To my parents, for their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CPU	Central Processing Unit
GPU	Graphics Processing Unit
FFT	Fast Fourier Transform
DFT	Discrete Fourer Transform
DIT	Decimation-in-Time
DIF	Decimation-in-Frequency
DCT	Discrete Cosine Transform
FFTW	The ”Fastest Fourier Transform in the West” Library.
ESSL	Engineering and Scientific Subroutine Library
MKL	Intel Math Kernel Library
GIMPS	Great Internet Mersenne Prime Search
SIMD	Single Instruction, Multiple Data
SSE	Streaming SIMD Extensions
AVX	Advanced Vector Extensions
FMA	Fused-Multiply Add
ISA	Instruction Set Architecture
FPU	Floating-Point Unit
GFlops	Giga-Floating-Point Operations per Second

CHAPTER 1

INTRODUCTION

Fourier Transforms (FT) are core to many scientific applications; they consume a significant fraction of the cycles dedicated to such applications. Therefore, there have been significant efforts to develop efficient Fast Fourier Transform (FFT) libraries. Vendors, such as Intel or IBM provide FFT libraries hand-tuned to their platforms, as part of their scientific library offerings [1, 2]; more efficient, domain-specific libraries have been developed for applications such as infinite precision integer arithmetic [3]. More recently, library generators have been used to develop FFT libraries tuned to particular sizes and to particular platforms [4, 5].

Specialized FFT libraries have the obvious limitation of being specialized. Hand-tuned code is hard to port and maintain, hence vendor libraries often lag in performance. While library generators could be expected to generate well-tuned FFT libraries, they do often produce code with poor performance, for reasons we discuss in the next section.

A library generator, such as FFTW, has two components: A component that generates highly tuned, platform-specific "codelets" – specialized straight-line FFT codes for small input sizes (`genfft`); and a component that recursively decompose a large FFT into smaller ones, using the codelets as the base of the recursion (`planner`). As we shall show, FFTW is efficient for small size FFTs, but inefficient at larger sizes: It produces codelets of good quality but bad plans. The reason seems to be that library generators have a hard time taking into account locality, for reasons we detail later. However, locality is key to performance for large FFTs.

We describe, in this paper, an FFT library that achieves significantly better performance than FFTW, for large FFTs. The library is designed by carefully hand-tuning small FFTs and building, recursively, larger FTTs. The recursion is done by macro expansion, driven by a few carefully selected control parameters. In the work presented here, the values of these parameters were hand-tuned.

Our work shows that there still is significant scope for improving the performance of FFT libraries. It shows the efficiency of new techniques for improving the locality of FFT codes. It is debatable whether these new techniques can be incorporated in current FFT library generators.

CHAPTER 2

FFT LIBRARIES

Because of the importance of Fourier Transforms, significant efforts have been devoted to the optimization of FFT libraries. Traditionally, such libraries have been developed by vendors and carefully hand-tuned to a specific architectures; the Intel Math Kernel Library (MKL) [1] and the IBM Engineering and Scientific Subroutine Library (ESSL) [2] are two such examples. There is little public information on these libraries: They seem to be written in low-level languages (C and/or assembly) and to be updated with each major architectural change.

2.1 Library Generators

More recently, several groups have developed FFT *library generators* to generate automatically libraries that are optimized for a particular processor architecture and for a particular FFT size.

The FFT generators use algebraic identities to transform an FFT formulation into an equivalent one that is more efficient (e.g., has fewer operations). This algebraic formulation is then "compiled" into source code. An auto-tuner can, in addition, use measurements from actual runs to guide the transformations. FFTW [4] and Spiral [5] are two visible examples of this approach. FFT transforms generated by such systems are widely used.

Libraries generated by FFTW perform better than MKL or ESSL in some cases, worse in other cases. The code generated by FFT library generators is often far from optimal, for several reasons:

2.1.1 Hard to Express Computation Costs

In earlier days, operation count was the dominant factor in performance. Therefore minimizing the number of operations (adds/subtracts/multiplies) would produce the fastest FFTs. Since it is easy to determine the operation count of algebraic expressions, it was easy to predict performance for a specific approach. Thus producing an optimal performance FFT consisted little more than a branch-and-bound search of different algebraic reductions.

Today, performance is much more difficult to predict. This is due to the ever-increasing complexity of modern hardware. In particular, computation costs are no longer *additive*: The total cost of an operation is no longer the sum of the individual costs of all the sub-operations. If the sub-operations overlap, then the total cost may be lower. Conversely, if there is overhead for switching between sub-operations, the total cost may be higher.

Instruction Level Complexity At the instruction-level, processor features such as superscalar and out-of-order execution make estimating computation costs nearly impossible. Instructions can be executed in parallel subject to dependencies and resource availability. The result is that a set of operations can run at drastically different speeds depending on just the order of the instructions.

Brent's theorem[6] can be used to bound the run-time to:

$$max(\frac{Work}{Throughput}, Crit.Path) < RunTime < \frac{Work}{Throughput} + Crit.Path$$

However, this bounds is too large to be useful for auto-tuning. It's worth noting that FFT butterflies have dependency graphs of logarithmic critical-paths and linear work parallelism. So theoretically the run time should be close to $\frac{Work}{Throughput}$. However, due to limited register size and complications with memory associativity, large butterflies are not feasible on current hardware.

Therefore FFTs are stuck in the "middle-ground", therefore Brent's theorem remains of little use.

Communication Costs Communication costs such as cache and memory accesses can be difficult to model in modern hardware.

Due to the high latencies of cache misses, modern hardware is typically designed to overlap them as much as possible. This by itself is a very large non-additive effect that is difficult to model without detailed knowledge of the target (often proprietary) hardware. To make matters worse, effects such as data-alignment and cache associativity can cause unexpected performance anomalies that may be dependent on run-time environments.

As with instruction complexity, upper-bounds on performance can be obtained for communication costs in much the same way. But the gap between actual performance and theoretical performance is typically even wider and more volatile than for instructions complexity.

Auto-tuning Since static modeling of performance is largely futile due to the reasons discussed above, modern libraries turn to auto-tuning. Instead of using static models, actual performance measurements are made of different decompositions of the task and the best is chosen.

However, the complexity of current architectures result into an exponentially large search space, and a badly formed integer optimization problem that is highly non-linear, has many local minima and has narrow local optima [7]. Exhaustive search is not feasible, and search heuristics can often be ineffective.

2.1.2 Hard to Express Transformations

Some code transformations that have significant impact on performance are hard to express as mathematical transformations in a formal system.

A short list of examples include:

- Cache Optimizations: blocking, padding schemes to improve efficiency and break alignment
- High-Radix Reductions: Formula size is exponential to the depth of a reduction.

Both of such optimizations/transformations play a major part in performance. But without a way to feasibly express them in mathematical transformations, they cannot be easily applied to the current generation of formula generators that take formulas as input.

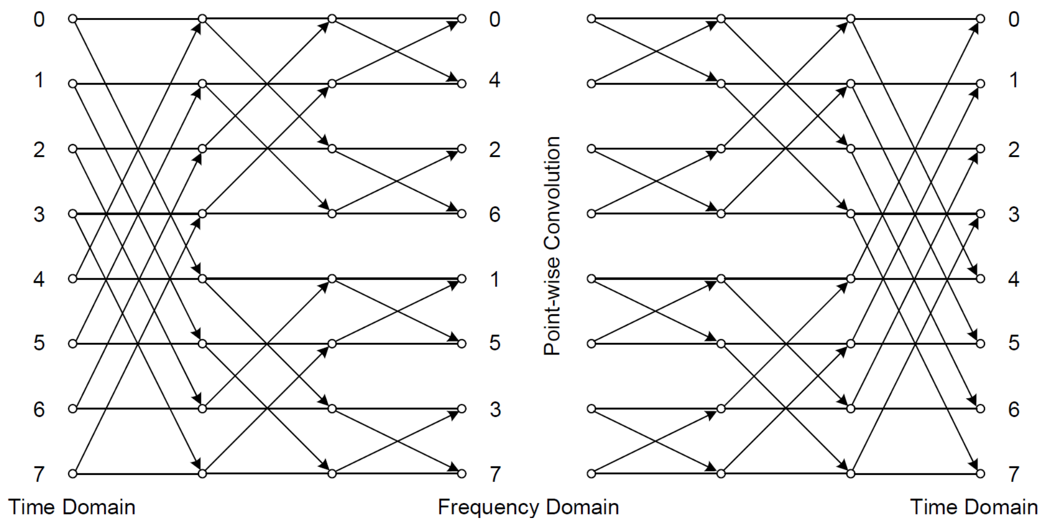


Figure 2.1: Since convolution is oblivious to the order of the frequency domain, it can be implemented without bit-reversals.

2.1.3 Compiler and Architectural Limitations

Algebraic transformations can result in very long straight-line code. Many compilers have a hard time compiling machine generated code . Furthermore, the generated code can overflow the instruction cache – yet another performance factor that is hard to express in a mathematical framework.

2.2 Specialized FFTs

Improved FFT performance can be achieved by taking advantage of specific properties of the application domain.

For example, complex FFTs can also be specialized and optimized when inputs are real numbers, or when many inputs are known to be zero (e.g., objects in 2D or 3D that do not fill a full square or cube).

2.2.1 General Performance Optimizations

In addition to specialization, specialized FFTs often gain just as much (if not more) speedup from hand-optimizations. In many cases, hand-optimized C or assembly from a capable programmer will easily outperform a modern compiler compiling unoptimized source code. Furthermore, it is known that compilers are poor at optimizing machine generated source code — which is a problem that the developers of FFTW and Spiral admit themselves [5].

2.2.2 Bit-Reversed Transforms

FFT algorithms based on Cooley-Tukey [8] require that either the input or the output be in bit-reversed order (the entry with index $a_1 \dots a_n$ is stored at location $a_n \dots a_1$). In order to obtain an in-order output, a bit-reverse copy is needed to shuffle the data at some part of the algorithm.

However, bit-reversal copies are not trivial and can consume a significant fraction of the transform time. This allows for an opportunity for specialization — since in many cases, the FFT output is used for a convolution, and the order of the entries is immaterial. One can perform the convolution in bit-reversed order, and input the result to the backward FFT in bit-reversed order, so that the final result is correctly ordered. No expensive bit-reversal copy is ever needed.

FFT libraries do not normally provide this functionality. We show later that out-of-order FFTs are significantly faster than regular FFTs. Infinite precision packages often use this optimization to quickly multiply large numbers. The Prime95 [3, 9] from the Great Internet Mersenne Prime Search (GIMPS) code is one such example [10].

But for applications that require in-order FFT output, the bit-reverse copy is unavoidable. Typical implementations for in-order FFTs are quite straightforward: A standard bit-reversed FFT followed by a bit-reverse copy. Some implementations do tricks like merging the bit-reversed copy in the FFT computation to reduce the cost. But fundamentally, the bit-reversal still needs to be done at some point.

The trouble with bit reversals is that bit-reversing an index is expensive without hardware support. Even worse, is that bit-reversed data-access exhibits extremely poor memory access.

A common naïve implementation is to bit-reverse an index and then proceed to access the memory. However, this runs into the following problems:

- There will likely be a near 100% cache miss rate if the data does not fit into cache.

- For every cache line that is brought in, only a single word is used. The rest is wasted.
- The cost of bit-reversing an integer is non-trivial. On x86 (which lacks hardware support), bit-reversing an integer requires more than 20 instructions.

Better implementations involve in-place sorts that resemble some of the standard $O(nlog(n))$ sorting algorithms. The best bit reversal implementations are $O(n)$, but require very hardware specific cache line and prefetching tricks to avoid the cache misses that are encountered in the naïve implementation. Our implementation is based on this latter approach and will be discussed later.

In most cases, this data reordering is a non-trivial task that can often be more expensive than the FFT computation itself! So it is useful to consider the possibility of completely omitting the data-reordering step for applications that oblivious to the order of either the time or frequency domain.

These are called *bit-reversed* FFTs, or in a more generalized case, *out-of-order* FFTs[1].

[1]We will use the term *out-of-order* instead of *bit-reversed* due to the fact that there exists even faster FFT algorithms that produce worse-than-bit-reversed time or frequency domains. Such algorithms will be discussed later.

CHAPTER 3

OUR IMPLEMENTATION

Our FFT library, like many others, is built by implementing and carefully tuning small FFTs, then implementing larger FFTs recursively, using these building blocks. For the basic building blocks, we pay attention to CPU effects: register allocation, instruction scheduling, etc. For the recursive decomposition, we pay attention to locality issues. The goal is to ensure one does not loose the efficiency of the basic building blocks. As for FFTW, the base building blocks are carefully hand-tuned. Our implementation has fully inlined base case sizes up to 64-point FFTs - all of which are optimized using micro-optimization techniques that we will discuss later. The recursive decomposition is implemented by a parameterized code. An efficient FFT is obtained by properly selecting the parameter values. This can be done manually or, as for FFTW, through an empirical search. The relatively small number of parameters ensures the search is feasible. Rather than using a program generator, we use macro expansion. This results in a code that is easier to understand and to maintain. Therefore, the fully automated configuration search performed by systems such as FFTW and Spiral can be replaced by a human-guided search that is more flexible and can often achieve better performance than a fully automated system.

The final step is to build whatever wrappers are needed to implement the specific functionality that are needed. This includes the bit-reversals and SIMD packing code.

3.1 Generalized Bailey's 4-step Algorithm

On modern systems, memory bandwidth is usually the limiting factor for performance of FFTs. So it is important to consider the amount of memory access that is needed for a particular FFT algorithm. In the classic

FFT Size	Datasize	Reduction Radix	Traversals	Memory Access
2^{30}	16 GB	2	12	384 GB
2^{30}	16 GB	4	7	224 GB
2^{30}	16 GB	8	5	160 GB
2^{30}	16 GB	2^{15} (Bailey's 4-step)	2	60 GB

Table 3.1: Dataset traversals and memory access for various reduction sizes assuming fully associative 8 MB cache.

approaches of using a fixed radix reduction, the entire dataset is traversed once at each radix until the transform sizes fit into cache. Assuming perfect cache, the number of dataset traversals is:

$$traversals = \lceil log_{radix}(\frac{datasize}{cachesize})\rceil + 1$$

Where the +1 is the pass needed to perform the sub-transforms that fit into cache. If the writes are streamed and do not require reading from memory, then the amount of memory access becomes:

$$bytes = 2 * datasize * \lceil log_{radix}(\frac{datasize}{cachesize})\rceil + 1$$

For FFTs that are significantly larger than the cache, many traversals will be needed. For larger radices, fewer traversals will be needed. When taken to the extreme, this becomes equivalent to Bailey's 4-step algorithm.

3.1.1 Bailey's 4-step Algorithm

This 4-step algorithm by Bailey [11] is a cache-oblivious FFT algorithm that performs an FFT of length $N = n * m$ as the following 4 steps:

1. Perform n FFTs of size m.
2. Multiply by twiddle factors.
3. Transpose the dataset.
4. Perform m FFTs of size n.

In the case of the DIF (Decimation-in-Frequency) transforms, steps 1 and 2 together form a radix reduction of size m and step 4 is the sub-transforms.

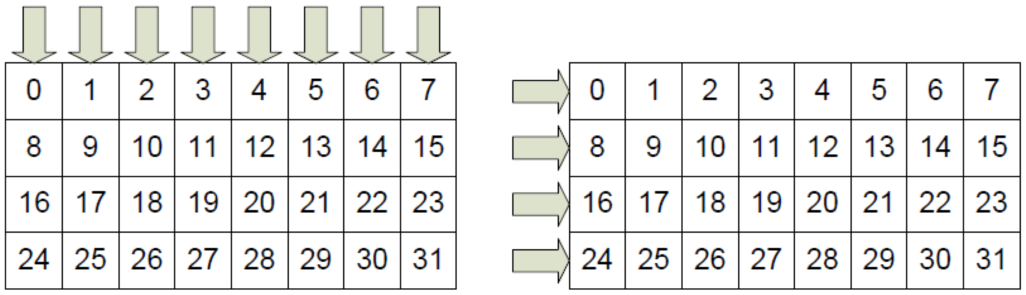


Figure 3.1: 4-Step FFT Algorithm: FFTs on columns, then on rows.

In our implementation, steps 1 and 2 are combined into a single pass. The FFTs in step 1 are done with strided memory access and the FFTs in step 4 are sequential. This eliminates the need for the transpose.

Therefore, when both an FFT of size n and an FFT of size m can be done entirely in cache, Baileys 4-step algorithm requires only 2 passes over the dataset. Therefore, n and m are often chosen to be close to $\sqrt{N}$.

Figure 3.1 illustrates our in-place implementation of Bailey's 4-step algorithm. FFTs are first performed on the columns. (the radix-reduction pass) Then FFTs are performed on the rows. (the recursive sub-transforms)

3.1.2 Generalized Approach

In our implementation, the 4-step algorithm is applied recursively to the FFTs in steps 1 and 4. This serves two purposes: To efficiently handle the case when a $\sqrt{N}$ size FFT cannot be done in cache and to optimize for the different levels of cache.

For situations where a $\sqrt{N}$ size FFT cannot be done in cache, we choose m to be the largest size such that a size m FFT fits into cache. This at least allows step 1 to be done with only 1 traversal. Step 4, while requiring more than 1 traversal, is handled recursively using the 4-step algorithm.

Recursive use of the 4-step algorithm is cache-oblivious optimal. It can be also used to optimize for the different levels of cache. Therefore, our implementation is fully recursive for both step 1 and step 4 FFTs — where a radix-selector is used to choose the values n and m. This radix-selection function is tunable and can be customized for different target machines.

In our implementation, the recursions for the step 1 transforms are terminated when the transform lengths drop to 2 or 4. The recursions for the step

4 transforms are terminated when they drop down to 2 or 4x of the block size which will be discussed in the next section.

3.2 Blocking/Data Padding

In our implementation, the step 1 FFTs the Bailey's 4-step algorithm are heavily strided and suffer from conflict cache misses and poor usage of cache lines. To solve this issue, combine the 4-step algorithm with data blocking and padding. Without blocking and padding, our code is much slower than the fixed radix approach.

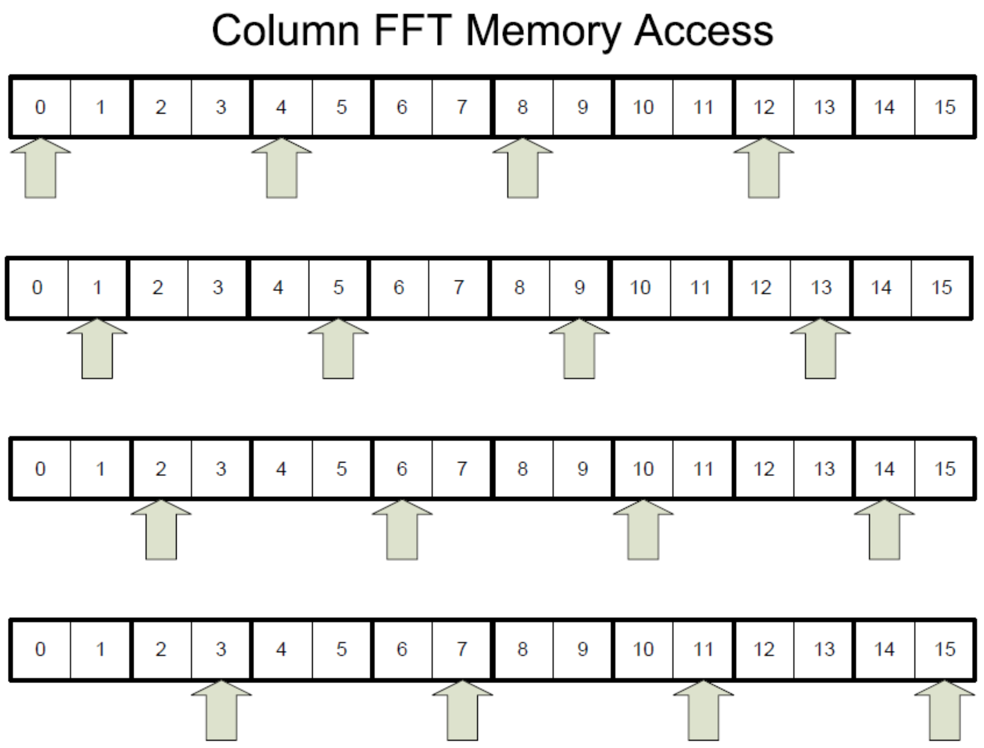
3.2.1 Data Blocking

To achieve CPU efficiency, points are grouped into blocks of b consecutive points. This block size b is a tunable parameter, usually with a value of 32 or 64. The entire FFT algorithm is then reformulated in terms of a shorter FFT acting on vectors. For example, a 2^{30} point FFT with a 16-byte point (two doubles) can be reformulated as a 2^{24} point FFT with a 1024-byte point.

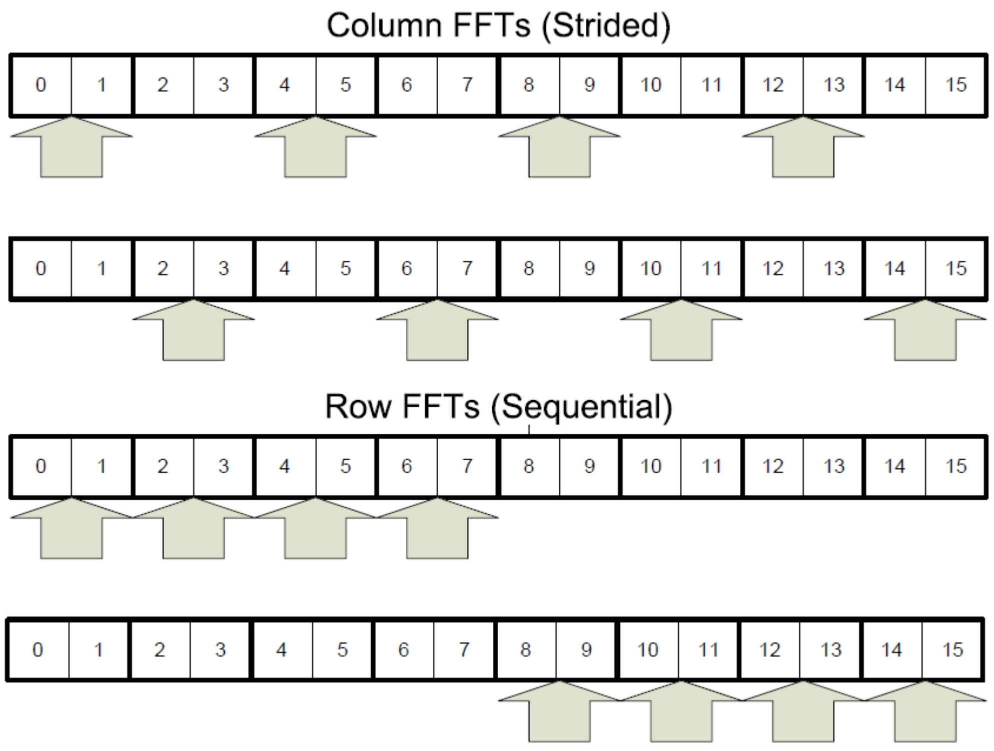
The generalized 4-step method is then used to compute this reformulated FFT. Due to the blocking, all accesses to memory are in chunks of the block size as opposed to 16 bytes. This maximizes the use of cache lines and improves the usefulness of the hardware prefetcher.

Our code supports FFT sizes that are a product of a small odd number and a large power-of-two. In our implementation, the odd factor is "absorbed" into the block size thus preserving the reformulated FFT as a power-of-two length. For an FFT of size $3*2^{27}$, the block size is chosen to be 48 which gives the reformulated FFT a length of 2^{23}. By keeping the reformulated FFT a power-of-two length, this allows nearly all the source code to be reused.

The drawback of blocking is that it reduces the size of the largest radix reduction that can be done. For a 2^{30} point FFT, the maximum radix reduction is 2^{15}. When blocked with a size of 64, the reformulated FFT has length 2^{24} and the maximum radix reduction reduces to 2^{12}. Furthermore, the amount of memory needed to perform the FFTs in step 1 of the 4-step algorithm increases, which decreases the maximum radix reduction that can be done without spilling cache. The result is that even moderately sized



(a) Without blocking, strided access makes horrible use of cache and SIMD. The performance is far worse than the small fixed-radix algorithms.



(b) Blocking maximizes SIMD and CPU efficiency at the cost of a larger working size.

Figure 3.2: Straight-forward vs. Blocked approaches to in-place Bailey's 4-step algorithm.

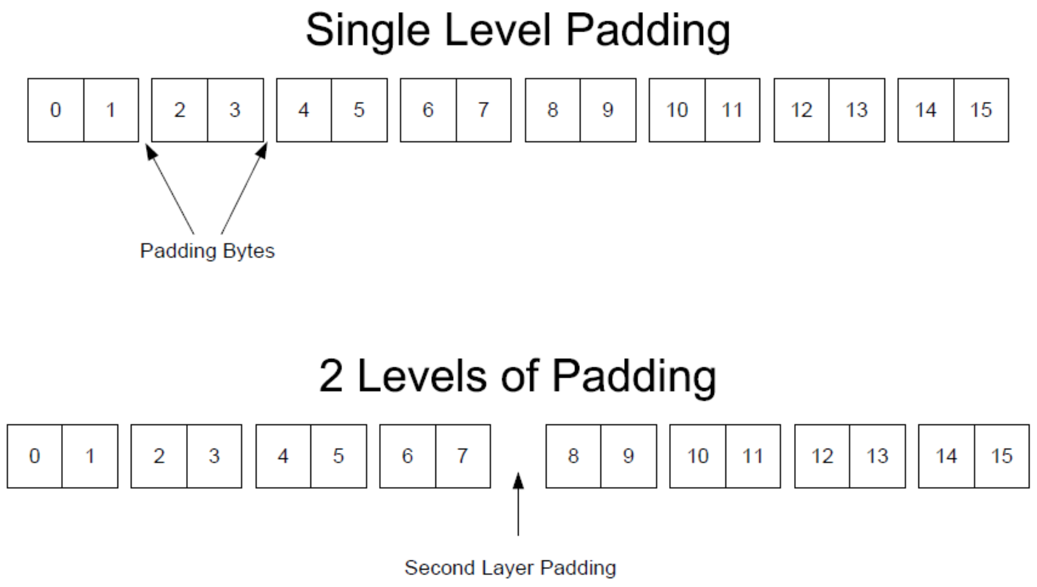


Figure 3.3: Multiple levels of padding are needed to break conflict cache misses for very large FFTs.

FFTs cannot be done in 2 traversals. However, 3 traversals usually suffices to handle nearly all FFTs of realistic size.

3.2.2 Data Padding

To reduce the number of conflict cache misses due to associative cache, we use padding. This is implemented on the reformulated FFT with block-granularity.

The padding that we implement is done by inserting n bytes after every m blocks. Assume an 32 Kbyte 4-way associative cache with 64 byte cache lines. Addresses that are congruent modulo 2^{13} map to the same associativity set. Now we wish to perform a double-precision complex FFT of size 2^{12} using radix 8. Accesses in the first step will have a stride of 2^{13} bytes (or 2^9 points). The 8 points needed to compute one "butterfly" will all fall in the same (4-entry) associativity set, causing multiple cache misses.

To solve this issue, we can insert 64 bytes after every 512 points (or 2^{13} bytes).

For extremely large FFTs, a single layer of padding is insufficient. Using the same example, but with an FFT of size 2^{25}, accesses in step 1 have a stride of 2^{22} points, or $2^{13} * (2^{13} + 64)$ bytes, using the same padding. This stride is divisible by 2^{13}, therefore the 8 points in a butterfly will still fall into the same associativity set. To mitigate this, we insert a second level of padding after every 2^{22} points (or 2^{26} bytes).

In some extreme cases, 3 or more levels of padding are needed. Naturally, this destroys the regular layout of the FFT so the blocks need to be addressed by index with an address generator to convert the index to a pointer.

Given the index of a block, its address is computed as:

$$\begin{gathered} block\ address = base\ address + index * (bytes\ per\ block) \\ +n_0 * \frac{index}{m_0} + n_1 * \frac{index}{m_1} + ... + n_{L-1} * \frac{index}{m_{L-1}} \end{gathered}$$

Where L is the number of levels of padding. To keep the cost of address generation low, we restrict all m_i values to a power of two so that the divisions reduce to shifts. Furthermore, all n_i values needs to be a multiple of the SIMD vector size to maintain data alignment. The cost of generating an address is paid every time a block needs to be accessed. If this method is applied directly to an FFT without blocking, the cost is paid every time a point is accessed and would surely dominate the total computational cost. Therefore, blocking a sufficiently large number of points becomes necessary for this padding scheme to be feasible.

3.3 Fast Bit-Reversal

Using the aforementioned methods we implemented an efficient bit-reversed FFT. In order to produce the full in-order FFT we implemented a fast out-of-place bit-reversal copy. Our implementation uses a single pass over the dataset, and thus is $O(n)$. To make this efficient, we will use a combination of blocking and cache-aligning.

The fundamental problem with a single-pass bit-reversal is that either the reads or the writes are non-sequential, leading to bad spatial locality. The solution is to reorder the accesses so that both reads and writes access full cache lines, by blocking both reads and writes in a consistent way.

Consider the set $S_{a_1 \ldots a_k}$ of 2^{2j} points with indices $x_1 \ldots x_j a_1 \ldots a_k y_1 \ldots y_j$, for fixed values $a_1, \ldots, a_k$. The bit-reversal permutation maps this set onto itself. The set can be bit-reversed by reading 2^j blocks of consisting of 2^j

adjacent points, and writing 2^j blocks of 2^j points. Both read and writes will move full cache lines, provided that

- 2^j points occupy one or multiple cache lines;
- 2^{2j} points fit in cache; and
- The array of points is cache line aligned

Figure 3.4 shows how a 16-point bit-reversal can be performed using 2-way blocking. At each step, two strides of memory are read and written. Furthermore, all memory accesses are in 2-point chunks.

In our implementation, we choose 2^j to be the number of points in one cache line, i.e., 4, for complex double precision points and 64 byte cache lines. Therefore, our bit-reversal implementation uses 4-way strided access with 4-point chunks. This forces all memory accesses to be exactly one cache-line wide. Combining this with proper alignment ensures that all memory accesses are full cache-lines.

The final part is that we choose to iterate the reads sequentially, and the writes non-sequentially. This allows for 4 sequential read streams which works well with the hardware prefetcher. The writes are left non-sequential, but are done using SSE2 streaming writes. Since all the writes are aligned to the cache-line and are in cache-line chunks, no read-modify-write traffic is generated and all writes go directly to memory.

3.4 Vector Instructions

Most current processors have vector units that can achieve a higher floating-point throughput than their scale units. An FFT lends itself very well to vectorization. Since the butterflies in a radix reduction are all completely independent. Indeed, vectorization is already extensively used in libraries such as FFTW and Spiral. Compilers today are very poor at automatic vectorization, so it must be done manually, to get the best performance. We support vector instructions for both x86 and Power7. On x86 we support SSE3 and AVX instructions. On Power7 we support AltiVec. Support for FMA instructions on x86 is slated for the future when the appropriate hardware becomes available.

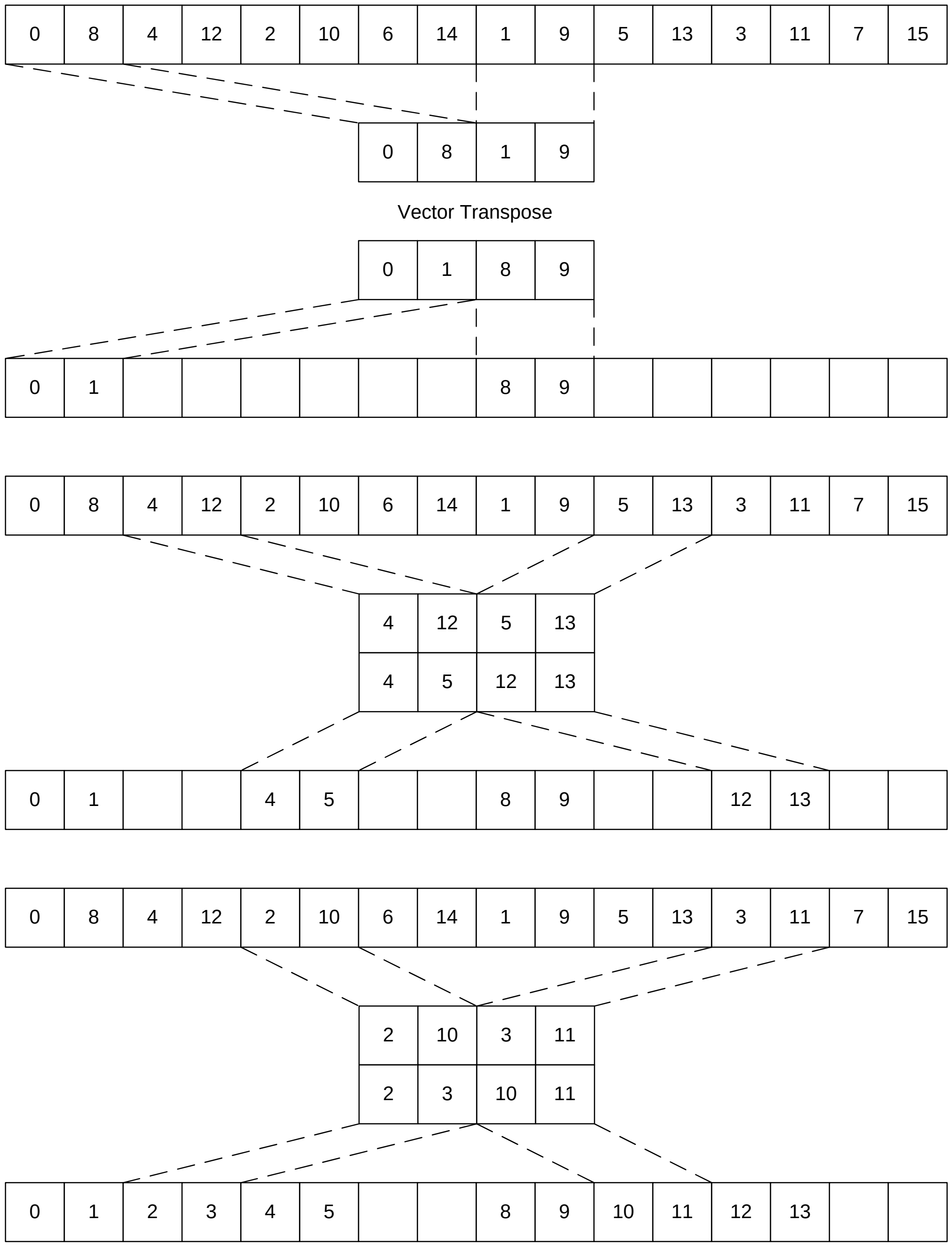


Figure 3.4: Fast Bit-reversal

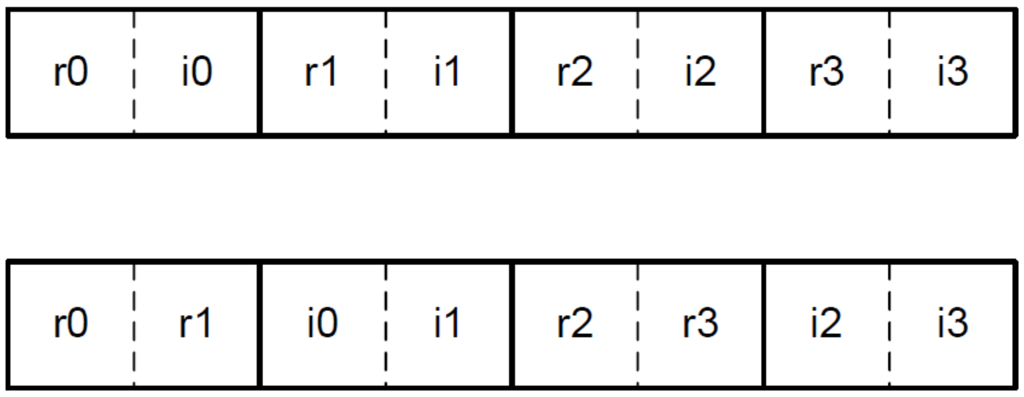


Figure 3.5: Data Packing: Array-of-Structs (top) vs. Struct-of-Arrays (bottom)

3.4.1 Array-of-Structs vs. Struct-of-Arrays

Our approach to vectorizing the FFT is slightly different than the standard approach. The usual packing of data is that real and imaginary parts of the same point are stored together in adjacent memory locations and thus packed together into the same vector, in an *array-of-struct* layout. Although this works well with additions and subtractions, it complicates multiplications due to need to cross multiply across vector elements. This results in extra instructions to perform data shuffling and negation. The SSE3 instruction set for x86-64 attempts to mitigate this issue by introducing the *addsubpd* instruction that eliminates the negation, but it does nothing to reduce the data shuffling.

To get around this issue, our implementation uses a *struct-of-arrays* packing where the real and imaginary parts are stored in different vectors. Using this packing, nearly all data shuffles can be eliminated since all arithmetic is completely vertical (i.e., involves aligned operands in two distinct vectors). There are no cross element operations until the final steps in the FFT where there are data dependencies between adjacent points. The drawback of this method is that the added register pressure from having to perform 2 butterflies at the same time (4 for AVX). On x86 this packing results in very little speedup due the additional register spills. On x64, there are plenty of registers, so the new packing results in 30% speedup.

Since Power7 has even more registers than x64, we did not attempt a *array-of-structs* implementation for it. So we cannot say how much faster *struct-of-arrays* is.

3.4.2 Portability Considerations: Type-Generic Macros

The use of SIMD instructions is not portable: Different architectures have different instruction sets (x86 and Power7) and different generations of the same architecture have different vector sizes (SSE3 and AVX). We handle these differences without generating different source code versions; instead we we implement the inner-most loops of our FFT using polymorphic macros. These macros work completely independently of the vector size, the names of the intrinsics, and the types of instructions that are needed. An example is shown in figure 3.6. A typical polymorphic macro takes parameters such as:

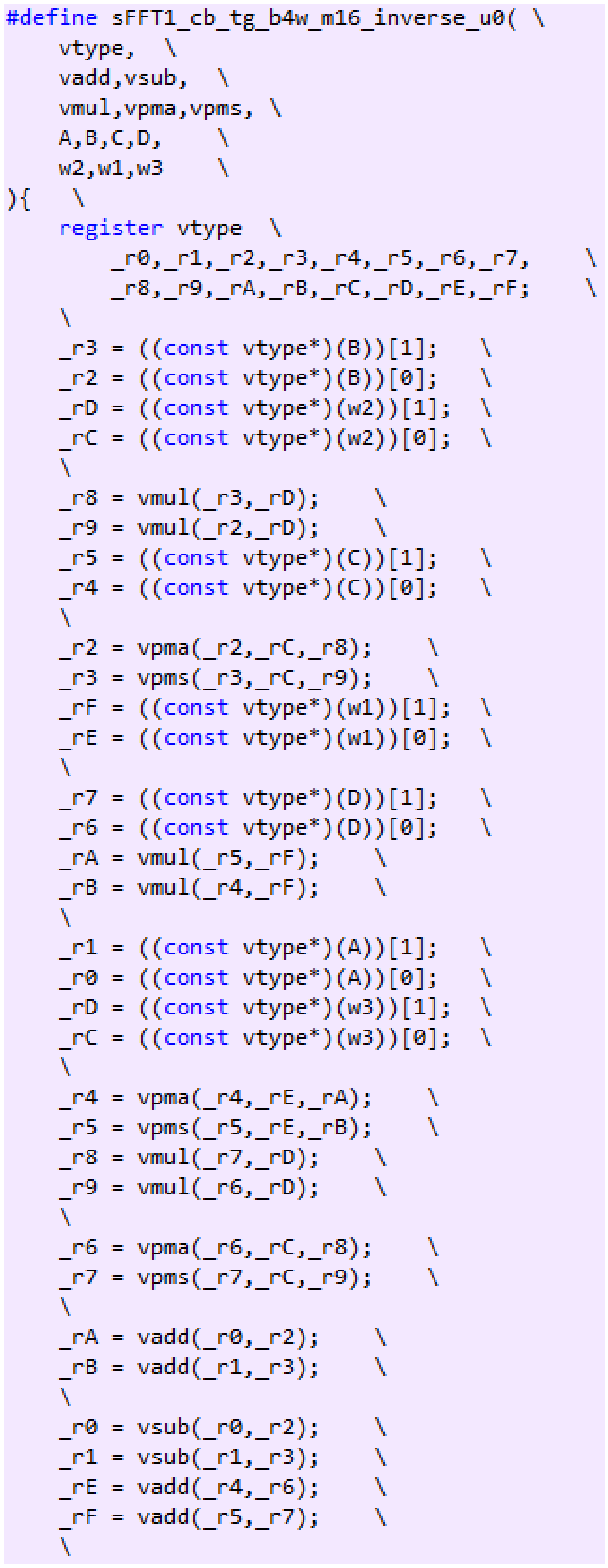
- The datatype. (ex. __m128d)
- Add, Subtract, and Multiply intrinsics. (ex. _mm_add_pd)
- Fused Multiply-Add intrinsics. (ex. _mm256_macc_pd)
- Source and destination operands.

These macros are used by passing the appropriate parameters from the target ISA. When no such instruction exists, it is made by combining the necessary instructions as is the case with FMA instructions on all x86 processors up to and including AVX.

Such macros are used for loops such as the radix 2 and radix 4 butterflies. They are all fully optimized using the other micro-optimizations that we will discuss later. Furthermore, multiple versions of the same macro are usually implemented and will differ in aspects such as:

- The order of the instructions.
- The number of temporary variables used.
- The number of iterations unrolled.

For each ISA, a common header file contains preprocessor definitions that specify which macros to use as well as mapping the names of the appropriate SIMD intrinsics to the ones used by the macros. An observation is



```
#define sFFT1_cb_tg_b4w_m16_inverse_u0( \
    vtype,  \
    vadd,vsub,  \
    vmul,vpma,vpms, \
    A,B,C,D,    \
    w2,w1,w3    \
){   \
    register vtype  \
        _r0,_r1,_r2,_r3,_r4,_r5,_r6,_r7,    \
        _r8,_r9,_rA,_rB,_rC,_rD,_rE,_rF;    \
    \
    _r3 = ((const vtype*)(B))[1];   \
    _r2 = ((const vtype*)(B))[0];   \
    _rD = ((const vtype*)(w2))[1];  \
    _rC = ((const vtype*)(w2))[0];  \
    \
    _r8 = vmul(_r3,_rD);    \
    _r9 = vmul(_r2,_rD);    \
    _r5 = ((const vtype*)(C))[1];   \
    _r4 = ((const vtype*)(C))[0];   \
    \
    _r2 = vpma(_r2,_rC,_r8);    \
    _r3 = vpms(_r3,_rC,_r9);    \
    _rF = ((const vtype*)(w1))[1];  \
    _rE = ((const vtype*)(w1))[0];  \
    \
    _r7 = ((const vtype*)(D))[1];   \
    _r6 = ((const vtype*)(D))[0];   \
    _rA = vmul(_r5,_rF);    \
    _rB = vmul(_r4,_rF);    \
    \
    _r1 = ((const vtype*)(A))[1];   \
    _r0 = ((const vtype*)(A))[0];   \
    _rD = ((const vtype*)(w3))[1];  \
    _rC = ((const vtype*)(w3))[0];  \
    \
    _r4 = vpma(_r4,_rE,_rA);    \
    _r5 = vpms(_r5,_rE,_rB);    \
    _r8 = vmul(_r7,_rD);    \
    _r9 = vmul(_r6,_rD);    \
    \
    _r6 = vpma(_r6,_rC,_r8);    \
    _r7 = vpms(_r7,_rC,_r9);    \
    \
    _rA = vadd(_r0,_r2);    \
    _rB = vadd(_r1,_r3);    \
    \
    _r0 = vsub(_r0,_r2);    \
    _r1 = vsub(_r1,_r3);    \
    _rE = vadd(_r4,_r6);    \
    _rF = vadd(_r5,_r7);    \
    \
```

Figure 3.6: Polymorphic Macros: By substituting the relevant types and intrinsics, this radix-4 butterfly macro can be used on all architectures.

that this method works well due to the Struct of Arrays packing. With this packing, nearly all SIMD arithmetic is vertical, and it is generally safe to assume that vector ISAs will support vertical instructions for all the common operations. With the Array of Structs packing, operations such as data-shuffling and negation would need to be used, and support for these operations are less standard across different ISAs. For example, x86 SSE3 supports the *addsubpd* instruction which Power7 does not have and Power7 supports a byte-granularity permute whereas x86 SSE only has a 32-bit granularity shuffle.

Porting our FFT to a new ISA is fairly simple. Since our FFT is completely recursive, the only non-trivial work that is needed is to code the base-cases where cross-element arithmetic is needed. This usually consumes about 90% of the man hours needed for the port. Once completed, the rest of the FFT usually falls into place and simply works. The only work left is to tune it by running a series of benchmarks and picking out the best set of parameters.

Since most of the FFT code is completely reused for all ISAs, we are able to support new ISAs in a timely manner. When the Intel Sandy bridge processors (featuring the 256-bit AVX instruction set) were released in January 2011, we were able to produce a fully working and optimized FFT utilizing AVX by early February - within 2 weeks of when we acquired the hardware. By comparison, FFTW did not support AVX until June. We were also able to add support for AltiVec in under 1 week from start to finish due to its similarity to SSE2.

3.5 Micro-Optimizations: Loop Unrolling

In our implementation, we use loop unrolling to gain additional speedup. Although compilers can perform loop unrolling, they tend to be too conservative with larger loops for fear of bloating the code size. We have found that we can always do better with manual unrolling.

Our method of loop-unrolling follows a three-step process.

3.5.1 Step 1: Write Sequential Loop

The first step is to write a single iteration of the loop using as few variables as possible. This process usually involves carefully designing the code to minimize the number of variables and to reuse as many temporary variables as possible. This step may introduce false dependencies into the code. So there may be a trade-off between fewer variables and fewer dependencies. However, modern processors have out-of-order execution which will break false dependencies via register renaming. As a result, it is almost always better to favor fewer variables.

3.5.2 Step 2: Unroll the Loop

The second step is to unroll the loop to as many iterations as possible while keeping the total number of variables less than the number of logical registers in the ISA. This ensures that there are no unexpected register spills that could hurt performance.

To allow for the instruction interleaving in step 3, the variables for the different iterations are all given new and distinct names. In many cases, temporary variables can be shared by different iterations. This sometimes allows the loop to be unrolled by an additional iteration.

Most loops are unrolled to 2 or 4 iterations so that they divide evenly into the trip counts which have many factors of two. This eliminates the need for cleanup code which increases code size and complicates branch prediction. Some compilers, such as the Intel Compiler, will (unnecessarily) compute the trip count of a loop at run-time for unknown reasons. When the amount of unrolling is not a power-of-two, this results in an integer division. Although the integer division is usually optimized as an invariant multiply, the multiply is double-width and still adds significant overhead to the startup of the loop.

Since our code is written in C, we actually have no direct control of register allocation. Modern compilers also tend to ignore the register keyword. However, we have found that compilers will rarely introduce register spills when the code satisfies the following conditions:

- The total number of register-sized variables is less than the number of logical registers in the ISA.

- Every C statement in the function can be implemented without additional registers. (Examples: $a = a + b$, $a = a * b + c$. Counterexamples: $a = b * c + a$, function calls)

- There are no common sub-expressions. This includes multiple loads from the same address.

Modern compilers generally use a graph coloring algorithm for register allocation. When these 3 conditions hold, the resulting graph is colorable. Although only the first condition is necessary for the graph to be colorable, the latter two conditions are needed to prevent the compiler from generating additional variables. The final issue is whether the compiler will be able to find the solution to the resulting colorable graph as this problem is known to be NP-complete. From our experience, the compilers we have tested are almost always able to find the solution without spills — even for very large sizes ($>$ 1000 instructions).

3.5.3 Step 3: Schedule the Loop

The final step is to schedule the instructions in the unrolled loop. Since the optimal scheduling is highly variable even for different processors with the same ISA, we leave much of this task to the compiler. Optimal instruction scheduling is NP-complete, and compilers can produce very poor code for large basic blocks. This is especially the case when all good solutions require a large number of instructions to be moved far from their initial positions. Our solution is to manually interleave the independent iterations. Since steps 1 and 2 almost guarantee that there are no spills regardless of the instruction ordering, there is no penalty for simultaneously running as many independent dependency chains as possible. Therefore, the optimal scheduling is usually very close to an interleaved order of all the independent iterations. Manual interleaving moves the instructions to better starting positions, thereby "helping" the compiler find a near-optimal scheduling even in the largest of basic blocks. For some very large loops, this has been observed to produce a speedup of more than 2x.

Overall, our method of avoiding register spills and interleaving instructions only works well on RISC architectures. Complications arise for instructions that are bounded to specific registers such as some x86 integer instructions as well as the x87 FPU. Fortunately, our FFT code uses only the x86 SIMD registers, which are nearly RISC-like. For small loops and loop with long dependency chains, loop-unrolling can provide more than 2 – 3x speedup. However, for our FFT code, we only gain about $10-20\%$ overall on x64. This is because the bulk of the computation is spent performing radix 4 butterflies, which are large and already have plenty of instruction level parallelism.

With the Struct-of-Arrays packing, radix 4 butterflies need more than 8 registers. This makes them difficult to unroll with only 16 SIMD registers. By comparison, we gain about 50% on Power 7 due to the larger number of registers and longer instruction latencies. Reverting to Array-of-Struct packing cuts the register count in half, but the overhead of the resulting data shuffles far outweighs any speedup gained from extra loop-unrolling.

3.6 Micro-Optimizations: Function Inlining

The final micro-optimization that we use is function inlining. To reduce the overhead of recursion, we inline our base cases to very large sizes. In our implementation, we terminate the FFT recursion at 32 and 64 points. Each of these is implemented as a single function call with completely straight-line code and no loops. As a result, these functions can be very large: 1600 instructions for 64-points using SSE3 instructions.

Due to the sizes of these, we do not actually write them entirely by hand. Instead, they are implemented as a combination of small macros which are put together without loops. For example, we first implement a 4-point FFT in a macro. By combining this macro with the polymorphic macros for radix-2 and radix-4 butterflies, we can easily build the 32-point and 64-point FFTs using very little code and without loops or function calls. By nesting macros, this method can be easily extended to build arbitrarily large FFTs that are fully inlined. However, we stop at 64 points because larger sizes will run the risk of overrunning the instruction cache.

The other places where we use inlining are the radix reductions and the address generators for data padding. Radix reductions are typically implemented as a loop inside a function which is called from the FFT recursion. We implement both the radix conversions and address generation entirely using macros so that they are inlined. This ensures that the only functions calls in our FFT recursion are the recursive calls themselves all other function calls are completely eliminated by means of inlining.

CHAPTER 4

ISSUES

Along with the numerous low-level and high-level optimizations that we have implemented in our FFT, there were also other optimizations and techniques that did not prove as successful.

4.1 Explicit Cache Regions

A major problem with the FFT is the need for either data transposes or strided memory accesses. In our implementation, we chose to use strided memory accesses. But as mentioned before, it suffers heavily from cache conflicts - which we attempted to solve with padding.

However, padding only solves one of two problems caused by strided access. It eliminates the conflict miss latency within each butterfly, but there is a second more subtle issue which is not solved. We will call this the *reduced effective cache size.*

Suppose we have a directly mapped cache of 32k bytes. Now suppose a function has memory access localized to two 16k blocks. Since the two 16k blocks fit into the 32k cache, every access should be a cache hit. However, now suppose that the 16k blocks have base address offset by 32k. The result is that both blocks will map into the same cache locations. Since only half the cache is usable, we call this 16k of usable cache the*reduced effective cache size.* This example is a worst case, and in general, only a partial overlap will occur. Higher associativity helps, but it only pushes the problem to when there are more blocks than the cache associativity.

This type of strided access is unavoidable in a transpose-less implementation of Baileys 4-step algorithm. In order for step 1 of the algorithm to be done with only one traversal over memory, all the step 1 butterflies need to fit entirely into cache. But since they are strided, the *effective* cache size is

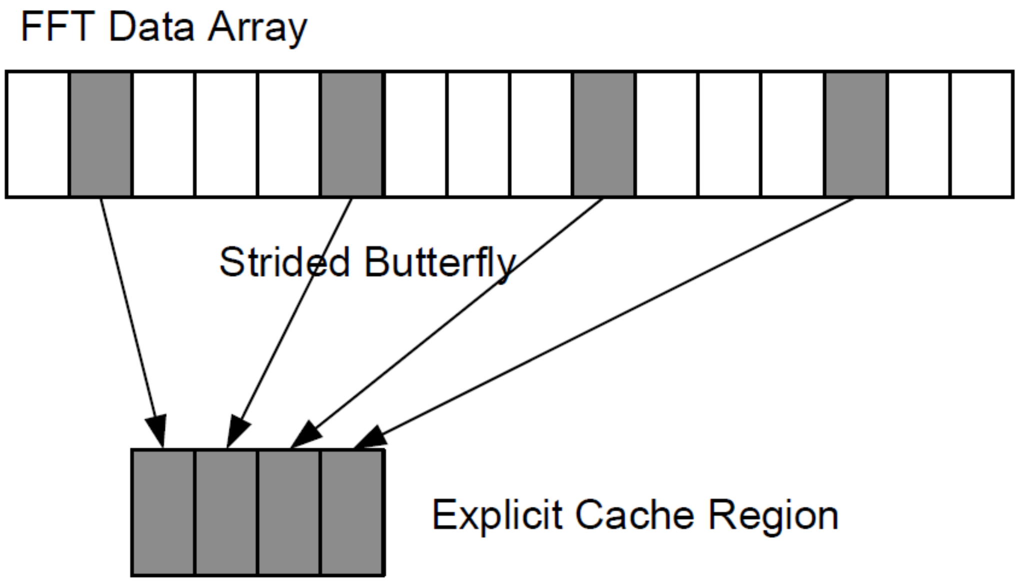


Figure 4.1: Strided data is copied to a local cache before any work is performed. This eliminates conflict cache misses since the data becomes sequential.

typically much smaller than the *physical* cache size. Padding does not help as it can only shift the base addresses by a small fixed amount.

The solution that we tried is to declare a specific region of memory to be *explicit cache*. The idea is to copy data from memory into a region of memory that is assumed to be in cache. Then all the computational work is done within this cache region. If the cache region is contiguous and signficantly smaller than the cache then we can almost guarantee that it will be kept in the actual processor cache.

However, after implementing this approach and benchmarking the results, we found that the overhead of the copies far exceeded the benefit of eliminating the in-place strided access. We tried two different approaches:

- Straight-forward memcpy().
- Lazy copy. (with and without streaming load/store)

In our initial implementation, we used a straight-forward copy of the data from memory into the cache region. Every time a step-1 FFT is called, all the data is copied from memory into the explicit cache buffer. The computation is performed and the data is copied back.

The performance of this 3-phase approach was roughly 20% slower than the baseline (blocking+padding). Commenting out the data copies reduced the run-time to 70% of the baseline - thus confirming that the data movement is indeed the bottleneck.

Our first attempt to solve this issue was to try a lazy-copy approach. The problem with the 3-phase approach is that there is no possible overlap of memory access and computation. Using a lazy approach where the memory is pulled from memory only when needed, we can take advantage of prefetching to achieve this overlap. Thus we were able to squeeze out about 10% performance gain.

The second optimization we did was to use SSE streaming writes. But there was no noticable speedup. We decided to stop our efforts here.

While the explicit cache method has worked for other algorithms, the FFT seems to have too little computation to benefit from it. The data locality in FFT computation is too brief to get a net gain from moving it to faster memory.

CHAPTER 5

EVALUATION

To evaluate the performance of our FFT, we benchmarked both our in-order and bit-reversed FFTs against FFTW version 3.3 and the specialized FFT found in y-cruncher v0.5.5.

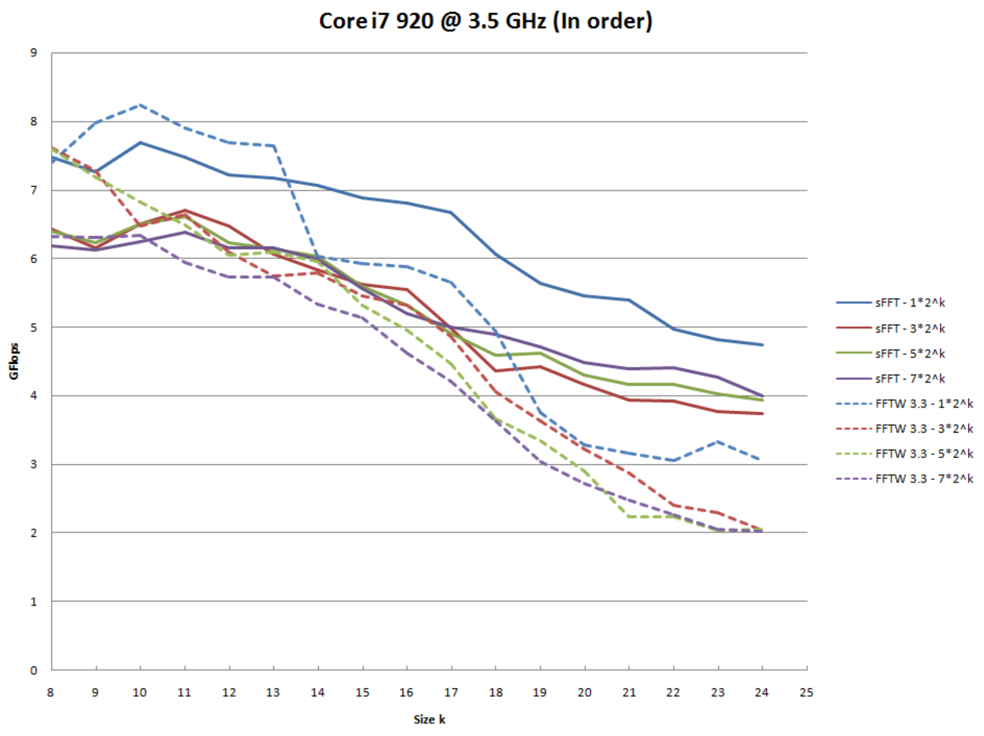
Figures 5.1, 5.2, and 5.3 show the performance of our FFT against FFTW (a library generator) and y-cruncher (a specialized FFT). Performance is measured in GFlops. (higher is better)

For each processor, we note the processor's per-core GFlops peak as well as the FFT theoretical peak. The processor's per-core GFlops the maximum throughput of the processor's capability, while the FFT theoretical peak is the maximum throughput that can possibily be obtained with perfect utilization of the processor's execution units.

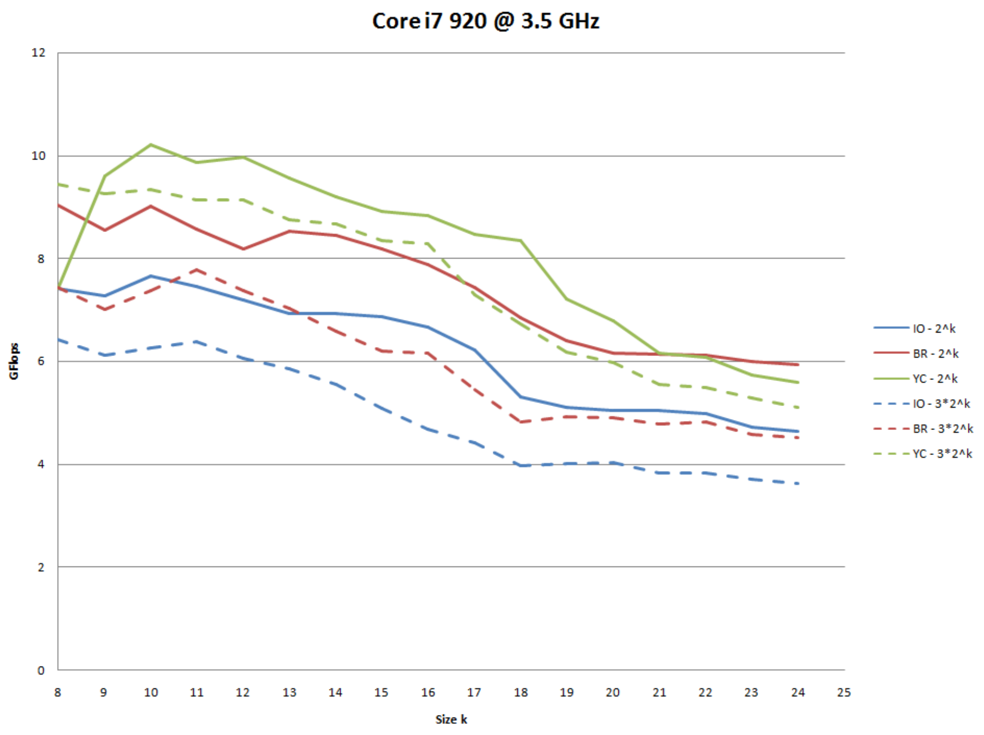
The difference is due the fact that the processors we tested have seperate addition and multiplication units. Each of them can sustain 1 add/subtract and 1 multiply instruction per cycle. However, the FFT does not have this 1-to-1 ratio. There are more additions and subtractions than multiplications. Therefore we adjusted the theoretical peak performance to reflect this. For this we used the radix-4 FFT which has a roughly 11-to-6 ratio of addition/subtractions to multiplications.

5.1 Our FFT vs. FFTW

The first observation is that for the larger sizes, our in-order FFT code significantly out-performs FFTW. (often by more than a factor of two) This can be attributed to our optimizations for cache and memory. Although we beat FFTW for all of the larger transform sizes, we note that FFTW is especially slow for the non-power-of-two sizes. This suggests that FFTW's handling of non-power-of-two sizes is very inefficient.

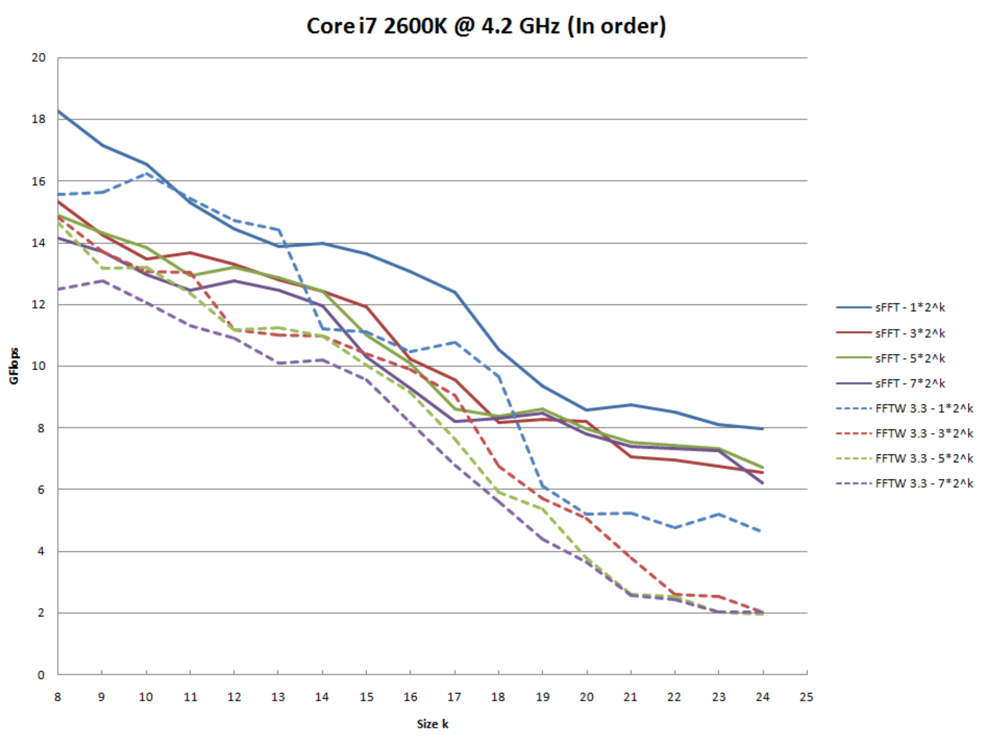


(a) In-order FFT vs. FFTW

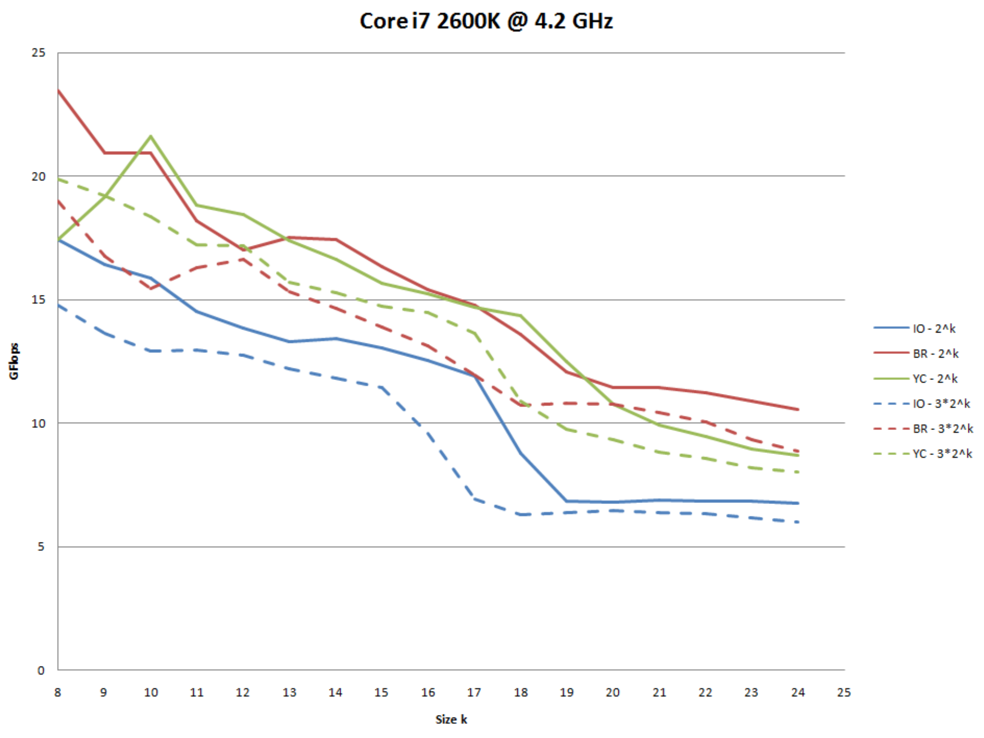


(b) In-order vs. Bit-reversed vs. y-cruncher v0.5.5

Figure 5.1: Benchmarks on Intel Core i7 920 @ 3.5 GHz (Processor Peak: 14 GFlops, FFT Theoretical Peak: 10.82)

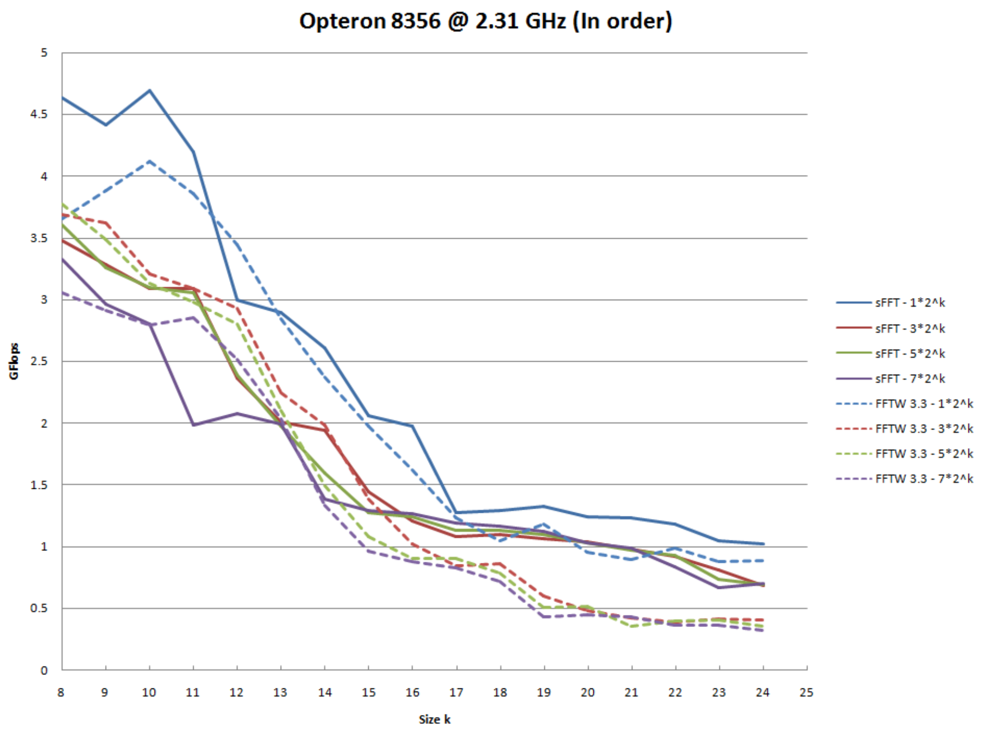


(a) In-order FFT vs. FFTW

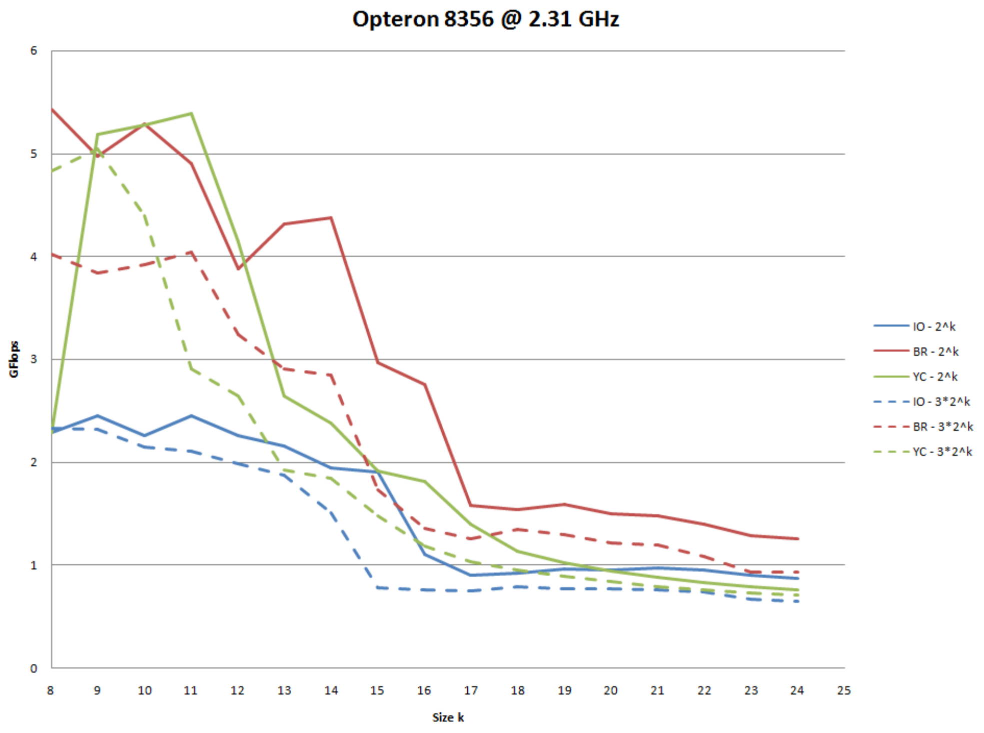


(b) In-order vs. Bit-reversed vs. y-cruncher v0.5.5

Figure 5.2: Benchmarks on Intel Core i7 2600k @ 4.2 GHz (Processor Peak: 33.6 GFlops, FFT Theoretical Peak: 25.96)



(a) In-order FFT vs. FFTW



(b) In-order vs. Bit-reversed vs. y-cruncher v0.5.5

Figure 5.3: Benchmarks on AMD Opteron 8356 @ 2.31 GHz (Processor Peak: 9.24 GFlops, FFT Theoretical Peak: 7.14)

For the smaller sizes, our in-order FFT is roughly on par with FFTW.

5.2 In-order vs. Bit-reversed vs. Specialized

Next we compare the performance of our in-order FFT and our bit-reversed FFT against the specialized FFT used in the y-cruncher multi-threaded Pi-program[12].

Overall, the results show that (as expected) the bit-reversed FFT is always significantly faster than the in-order. This gives a glimpse of how expensive the bit-reverse copy is. The interesting comparison is with y-cruncher's FFT. y-cruncher uses a worse-than-bit-reversed FFT that halves the number of twiddle factor loads. We will discuss this optimization later.

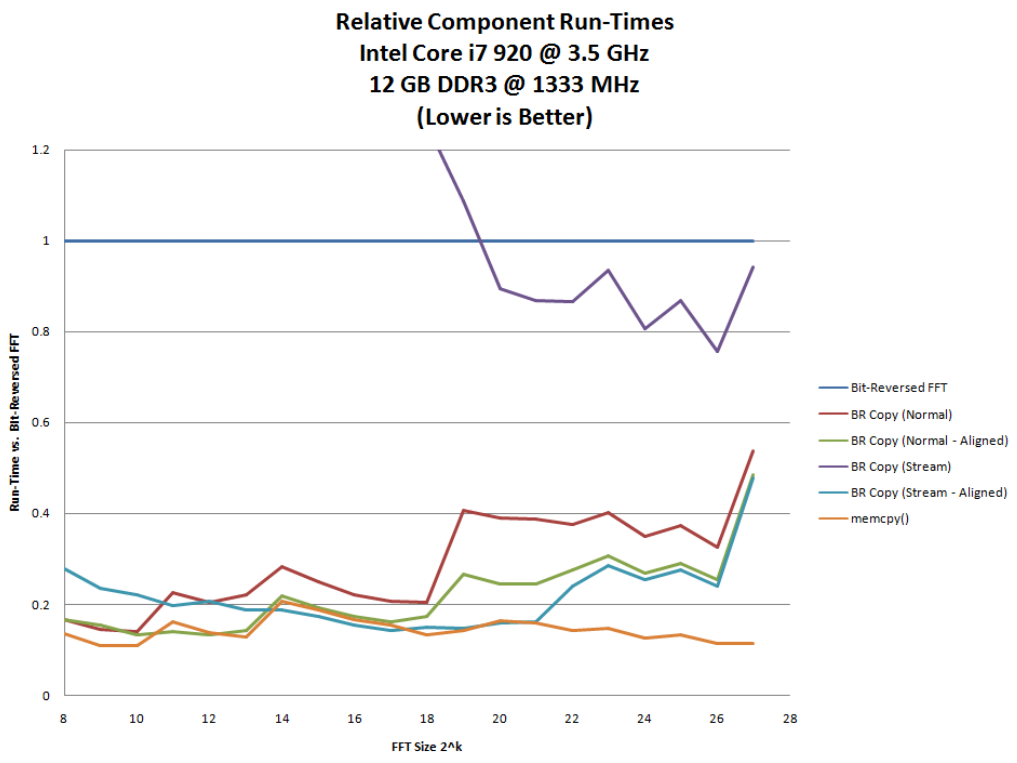
We first note that on all three processors, our bit-reversed FFT beats y-cruncher on the largest FFT sizes. This is because y-cruncher uses a flat radix-4 FFT implementation and does not use the 4-step algorithm. Therefore it suffers on the largest FFT sizes. Nevertheless, this performance is sufficient because y-cruncher does not use large FFTs.

For the smaller sizes, the results are quite mixed. On the Core i7 920 (Nehalem), y-cruncher beats our bit-reversed FFT by a noticable margin. On the Core i7 2600K (Sandy Bridge) and the Opteron 8356, our bit-reversed FFT is about neck-and-neck with y-cruncher.

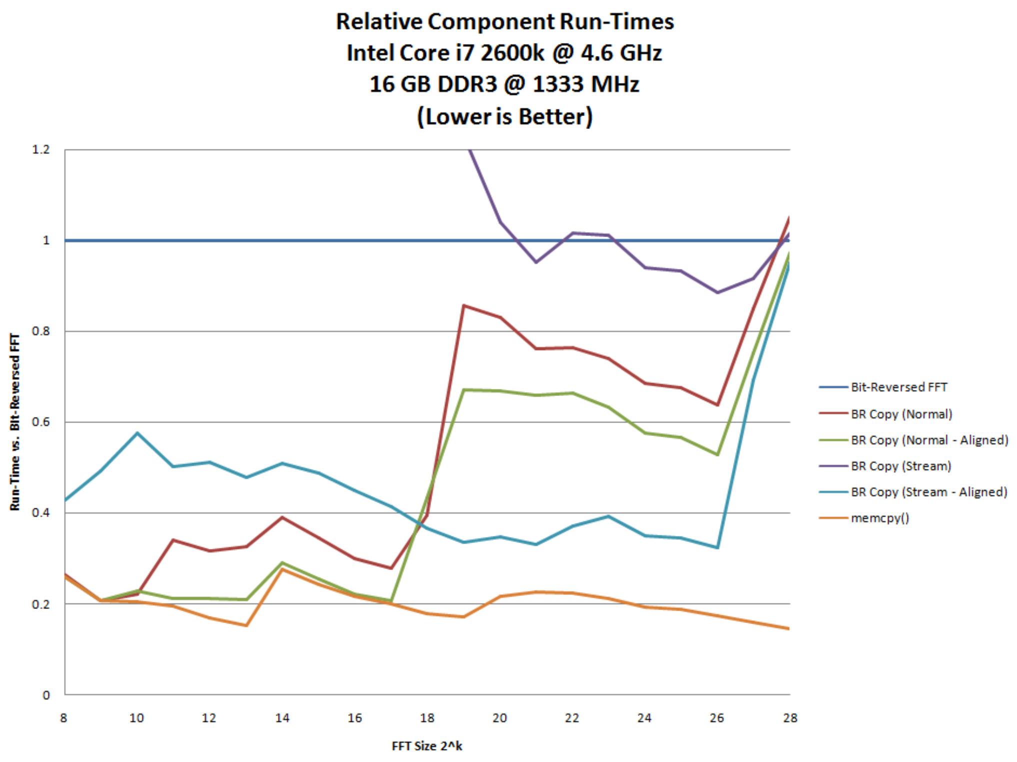
Figure 5.4 show the run-times of the various different bit-reverse copy implementations relative to the bit-reversed FFT. The run-time for a *memcpy*() of the entire dataset is included as a reference point and as an upper-bound on performance since neither a bit-reversed copy nor a bit-reversed FFT should be faster than a *memcpy*(). *Aligned* indicates that the dataset is aligned to the cacheline, and *stream* indicates whether or not streaming stores were used.

Due to the nature of our implementation of the bit-reversed copy, its performance extremely sensitive to both the data alignment and whether or more streaming stores are used. In the best case, the bit-reversed copy is nearly as fast as a *memcpy*() — thus indicating near optimal performance. However, in the worst cases, it can be much slower than the bit-reversed FFT itself.

Overall, when the datasize fits into CPU cache, it is best to use normal



(a) Bit-reversal performance on Core i7 920 @ 3.5 GHz.



(b) Bit-reversal performance on Core i7 2600k @ 4.6 GHz.

Figure 5.4: Comparison of various bit-reverse copy implementations relative to our bit-reversed FFT.

stores. When the datasize does not fit into cache, it is best to use streaming stores. The cross-over point is very sharp in most cases, so optimal performance for the bit-reversed copy requires a very carefully tuned threshold for switching from normal to streaming stores.

In all cases, the data-alignment is crucial. The bit-reversed copy is horribly slow when the data is not aligned to the cache-line. Furthermore, our implementation requires that the dataset be aligned to the SIMD vector to run at all. Because of this sensitivity, our in-order FFT requires cache-line alignment to achieve any significant gain in performance over FFTW.

CHAPTER 6

FUTURE DIRECTIONS

6.1 Generalizations

Although we focus our efforts in optimizing the one-dimensional FFT, much of the methods we use are fully applicable to other transforms.

6.1.1 Higher-Dimension FFTs

A close observation of higher dimension FFTs reveals that any N-dimensional FFT has an identical data dependency graph as a 1-dimensional FFT of the same size. Therefore, any N-D FFT exhibits an identical memory access pattern as a 1-D FFT of the same size. The only difference is the twiddle factors that are used. This means that any N-dimensional FFT can be implemented in an identical manner as a 1-D FFT using modified twiddle factors. The implication of this is that all FFTs, regardless of dimension, can be implemented using 2 traversals (assuming the cache is large enough).

For multi-node clusters and supercomputers, it allows any N-D FFT to be performed using only one all-to-all communication transpose. This is of interest for the case of multi-node 3D-FFTs.

Large multi-node 3-D FFTs are typically implemented in one of two ways:

- 2D decomposition: 1D FFT $\rightarrow$ transpose $\rightarrow$ 1D FFT $\rightarrow$ transpose $\rightarrow$ 1D FFT
- Slab decomposition: 1D FFT $\rightarrow$ transpose $\rightarrow$ 2D FFT

The 2D decomposition is the most natural approach because each dimension is handled separately, while the slab decomposition is more efficient because it only has 1 transpose.

3D FFT Alg. (size: N^3)	# of Transposes	Max. Scalability (nodes)
2D Decomposition	2	N^2
Slab Decomposition	1	N
1D + Baileys 4-step (2-pass alg.)	1	$\sqrt{N^3}$
1D + Bailey (3-pass alg.)	2	N^2
1D + Bailey (4-pass alg.)	3	$N^{\frac{9}{4}}$
1D + Min. Radix Cooley-Tukey	$\Omega(N^3)^1$	$\frac{N^3}{smallest\ prime\ factor\ of\ N}$

Table 6.1: Transpose counts and scalability limits of different implementations of a multi-node 3D FFT

In both cases scalability is limited by the size of the dimensions. For an N^3 3D FFT, the 2D decomposition will fail to scale beyond N^2 nodes while the slab decomposition will not scale past N nodes. This could cap performance on larger supercomputers with many nodes.

By reformulating the 3D FFT as a 1D FFT with modified twiddle factors, we can apply Baileys 4-step algorithm with radix $\sqrt{N^3}$ on the resulting 1D FFT with N^3 points. This allows the 3D FFT to be scaled up to $\sqrt{N^3}$ nodes while still maintaining only 1 transpose. Further scaling is possible at the cost of more transposes, but current and near-future supercomputers with realistically sized 3D FFTs are unlikely to benefit from using more than $\sqrt{N^3}$ nodes due to the cost of the all-to-all transposes.

6.1.2 Other Transforms

The methods we use are by no means restricted to floating-point FFTs and are largely applicable to other types of DFTs. For example, the methods we have discussed are mostly applicable to the Discrete Cosine Transform (DCT) for compression. All our methods except for type-generic macros are fully applicable to the Number-Theoretic Transform (NTT) for integer convolution and multi-precision arithmetic.

6.1.3 Micro-Optimization Techniques

Much of the micro-optimizations that we have used are easily generalized to a wide variety of other applications. In particular, function inlining and loop-unrolling are universally applicable techniques used by both compilers and high performance software developers. Our approach can enhance these by avoiding register spills and improving instruction scheduling. Many highly vectorizable applications can utilize type-generic macros to generate efficient and portable code for all SIMD architectures regardless of the vector size and syntax.

6.2 Other Optimizations

There are a number of optimizations that could be applied to further improve the performance of our FFT:

6.2.1 Saving Twiddle Factor Loads

The djbfft FFT library [13] uses a subtle trick to reduce the number of twiddle factor loads by up to a factor of two at the cost of further scrambling the order of the output. (to worse than bit-reversed)

Bernstein observed that for a split-radix butterfly, one can change the ω^3 twiddle factor to ω^{-1}. The result is still a correct FFT output, but with the order scrambled.

Now observe that $\omega^{-1} = \bar{\omega}$ because $\|\omega\| = 1$ for twiddle factors. This means that applying this trick to a split-radix butterfly will change the twiddle factors from ω and ω^3 to ω and $\bar{\omega}$. Explicitly, it is $\omega = a + bi$ and $\omega = a - bi$.

Since a and b are used in both twiddle factors, they can each be loaded once and used twice. By comparison, the conventional approach of $\omega = a + bi$ and $\omega^3 = c + di$ requires double the number of twiddle factor loads. No additional arithmetic operations are needed since the negation can be optimized out via sign-propagation.

While eliminating half of the twiddle factor loads may not be significant in itself, it allows the twiddle factor table to be halved in size since there is no longer a need to cache the ω^3 factors. So not only does it reduce menory usage, but it also reduces the demand on memory bandwidth — which is increasingly the bottleneck in modern systems.

This "Conjugate Twiddle" method generalizes to other butterflies. It can be applied to the standard radix-4 butterfly in the same way as the split-

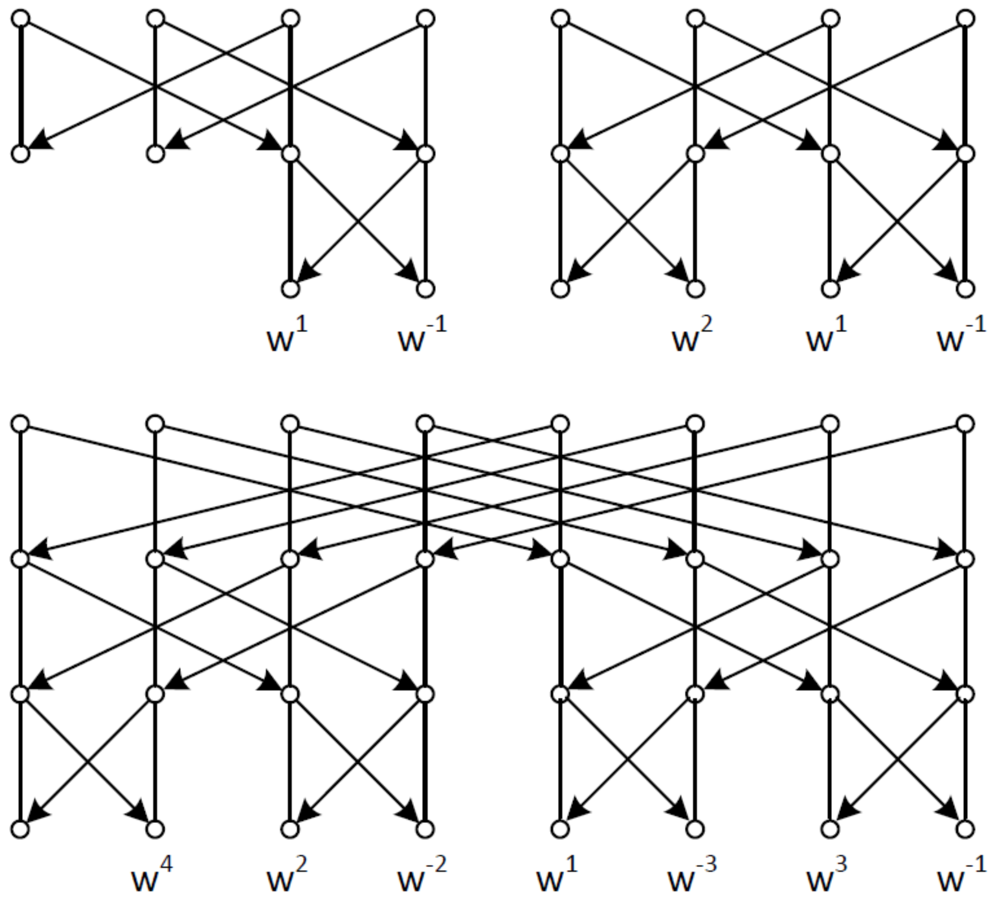


Figure 6.1: Conjugated twiddle factors for the standard split-radix 2/4, radix-4, and radix-8 butterflies.

radix butterfly which Bernstein originally used. For the standard radix-4 butterfly, the twiddle factors change from w^2, w, and w^3 to w^2, w, and w^{-1} with a 33% reduction in twiddle factor loads.

Furthermore, it can be generalized to higher radices as well as higher order split-radices. In general, the exponent of any twiddle factor can be "flipped" using the following relation:

$$old\ twiddle - new\ twiddle = radix\ of\ butterfly$$

For a radix-8 butterfly, the w^7 twiddle can be flipped to w^{-1} thereby reusing the same load as the w^1 twiddle. Similarly, the w^5 twiddle can be flipped to w^{-3} to reuse the same load as w^3. For a radix N butterfly, $N/2 - 1$ twiddle factors can be conjugated and reused. Split-radix butterflies allow exactly half the twiddle factors to be conjugated — thereby each twiddle factor load is used exactly twice.

Currently we do not use this optimization because the order of the output is no longer "mappable" and is dependent on the order in which the FFT radix reductions are applied. For example, performing a conjugate twiddle FFT using (radix-4 $\rightarrow$ radix-8) will produce a different output order from (radix-8 $\rightarrow$ radix-4)! Inverting such an FFT requires that the same sequence of radix reductions be performed in reverse. Furthermore, Baileys 4-step algorithm is also partially incompatible with conjugated twiddles.

Some uses of the FFT will tolerate any order of the output as long as it is possible to efficiently determine where a point is. This is the case for the bit-reversed FFT, but not for the conjugate twiddle FFT since the radices are chosen dynamically.

As a result, conjugate twiddle FFTs are only useful for applications that strictly use FFTs for convolution. Most libraries give little attention to bit-reversed FFTs let alone conjugate twiddle FFTs. Therefore, conjugate twiddle FFTs have mostly remained only of academic interest. Nevertheless, implementations do exist and are used by some highly specialized applications.

[1]$\Omega(n)$ is the big Omega function that counts the number of prime factors in n with multiplicity.

6.2.2 Fused Multiply-Add Algorithm

Most modern architectures have hardware support for Fused Multiply-Add (FMA) operations. Examples include PowerPC, Sparc, ARM, and GPUs. x86/64 is one of the last high-performance processors to support FMA. AMD's Bulldozer line was the first to support it in Q4 2011 followed by Intel's Haswell in Q2 2013. Therefore there is a growing interest to develop FFT algorithms for FMA architectures.

The computational complexity of FFTs had been conventionally measured by counting additions/subtractions and multiplications separately. However, in FMA architectures, a single Fused Multiply-Add operation has the same cost as a single multiply, thereby giving the extra addition or subtraction for free.

The basic operation in complex FFT is the multiplication of two complex numbers. A complex multiplication requires 4 multiplications and two addition/subtractions. Therefore, it would seem that the use of FMAs can reduce arithmetic complexity by 1/3, at best (from 6 to 4). However, [14] has shown that by "scaling" the FFT, it is possible to expose more multiplications to be fused into FMAs. Although this approach increases the total number of additions and multiplications, the total operation count is reduced to less than the classic approach.

This scaled FFT algorithm will be given more attention in the future when FMA is finally supported by x86/64.

CHAPTER 7

CONCLUSION

We have shown in this paper several techniques for improving the locality of FFT code, and for performing various micro-optimizations.

It is conceivable that FFT library generators, such as FFTW and Spiral, could incorporate the optimizations discussed in this paper; indeed, such an outcome would be highly desirable, as it would replace the manual search for a good execution plan by an automated one. However, we suspect that such a goal will require some significant changes in these library generators. Many of the transformations we discussed do not have any obvious representation in a tensor calculus, such as used by Spiral. Furthermore, as we argued in Section 2, execution time is not additive: The time $T(A;B))$ to execute codelet A, followed by codelet B, need not be equal to the sum $T(A)+T(B)$ of the times it takes to execute each codelet separately. This is so because of cache effects: As a result of the preceding execution of A, B may incur a smaller number of cold misses then it would incur otherwise. Therefore, $T(A;B)$ cannot be determined solely as a function of $T(A)$ and $T(B)$: As a result, the dynamic programming approach used by the FFTW planner becomes much less effective: One either need to use very large codelets, so that the effect of cold cache misses can be ignored; or use a much more complex characterization of codelets that includes initial cache state. As locality becomes the dominant performance factor, performance tuning becomes harder, since locality is not a "local" property of codelets, but a correlation between codelets.

REFERENCES

[1] "Intel math kernel library reference manual," http://software.intel.com/sites/products/documentation/hpc/mkl/ mklman/mklman.pdf.

[2] IBM, "Essl guide and reference," http://publib.boulder.ibm.com/epubs/ pdf/a2322682.pdf.

[3] G. Woltman, "Prime95 - great internet mersenne prime search," http://www.mersenne.org/freesoft/, 2011.

[4] M. Frigo and S. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[5] M. Pusche, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolol, "SPIRAL: Code generation for DSP transforms," in *Special Issue on Program Generation, Optimization, and Platform Adaptation*, 1990.

[6] R. P. Brent, "The parallel evaluation of general arithmetic expressions." *Journal of the Association for Computing Machinery*, vol. 21, no. 2, 1974.

[7] P. Balaprakash, S. M. Wild, and B. Norris, ""spapt": Search problems in automatic performance tuning," Argonne National Lab, Tech. Rep. ANL/MCS-P1872-0411, April 2011.

[8] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," in *Math. Comput.*, 1965.

[9] R. Crandall and B. Fagin, "Discrete weighted transforms and large-integer arithmetic," *Mathematics of Computation*, vol. 62, no. 205, pp. 305–324, 1994.

[10] G. Ziegler, "The great prime number record races," *Notices of the AMS*, vol. 51, no. 4, pp. 414–416, 2004.

[11] D. H. Bailey, "FFTs in external or hierarchical memory," in *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 1990.

[12] A. Yee, "y-cruncher - multi-threaded pi program," http://www.numberworld.org/y-cruncher/, 2010.

[13] D. J. Bernstein, "Frequently asked questions," http://cr.yp.to/djbfft/faq.html, 2010.

[14] E. N. Linzer and E. Feig, "Implementation of efficient FFT algorithms on fused multiply-add architectures," in *IEEE Transactions on Signal Processing*, 1993.