Streaming FFT*

Anthony Blake

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 3.0†

This chapter describes SFFT: a high-performance FFT library for SIMD microprocessors that is, in many cases, faster than the state of the art FFT libraries reviewed in Existing libraries.

Implementation details described some simple implementations of the FFT and concluded with an analysis of the performance bottlenecks. The implementations presented in this chapter are designed to improve spatial locality, and utilize larger straight line blocks of code at the leaves, corresponding to sub-transforms of sizes 8 through to 64, in order to reduce latency and stack overheads.

In distinct contrast to the simple FFT programs of Chapter 3, this chapter employs meta-programming. Rather than describe FFT programs, we describe programs that *statically elaborate* the FFT into a DAG of nodes representing the computation, apply some optimizing transformations to the graph, and then generate code. Many other auto-vectorization techniques, such as those employed by SPIRAL, operate at the instruction level [2], but the techniques presented in this chapter vectorize blocks of computation at the algorithm level of abstraction, thus enabling some of the algorithms structure to be utilized.

Three types of implementation are described in this chapter, and the performance of each depends on the parameters of the transform to be computed and the characteristics of the underlying machine. For a given machine and FFT to be computed (which has parameters such as length and precision), the fastest configuration is selected from among a small set of up to eight possible FFT configurations – a much smaller space compared to FFTW's exhaustive search of all possible FFTs. The fastest configuration is easily selected by timing each of the possible options, but it is shown in Results and discussion that it is also possible to use machine learning to build a classifier that will predict the fastest based on attributes such as the size of the cache.

SFFT comprises three types of conjugate-pair implementation, which are:

1. Fully hard-coded FFTs;
2. Four-step FFTs with hard-coded sub-transforms;
3. FFTs with hard-coded leaves.

1 Fully hard-coded

Statically elaborating a DAG that represents a depth-first recursive FFT is much like computing a depth-first recursive FFT: instead of performing computation at the leaves of the recursion and where smaller DFTs are combined into one, a node representing the computation is appended to the end of a list, and the list of nodes, i.e., a topological ordering of the DAG, is later translated into a program that can be compiled and executed.

Emitting code with a vector length of 1 (i.e., scalar code or vector code where only one complex element fits in a vector register) is relatively simple and is described in "Vector length 1" (Section 1.1: Vector length

*Version 1.3: Jul 15, 2012 11:04 pm -0500
†http://creativecommons.org/licenses/by/3.0/

1). For vector lengths above 1, vectorizing the topological ordering of nodes poses some subtle challenges, and these details are described in "Other vector lengths" (Section 1.2: Other vector lengths). The fully hard-coded FFTs described in this section are generally only practical for smaller sizes of transforms, typically where $N \leq 128$, however these techniques are expanded in later sections to scale the performance to larger sizes.

1.1 Vector length 1

A VL of 1 implies that the computation is essentially scalar, and only one complex element can fit in a vector register. An example of such a scenario is when using interleaved double-precision floating-point arithmetic on an SSE2 machine: one 128-bit XMM register is used to store two 64-bit floats that represent the real and imaginary parts of a complex number.

When $VL = 1$, the process of generating a program for a hard-coded FFT is as follows:

1. Elaborate a topological ordering of nodes, where each node represents either a computation at the leaves of the transform, or a computation in the body of the transform (i.e., where smaller sub-transforms are combined into a larger transform);
2. Write the program header to output, including a list of variables that correspond to registers used by the nodes;
3. Traverse the list of nodes in order, and for each node, emit a statement that performs the computation represented by the given node. If a node is the last node to use a variable, a statement storing the variable to its corresponding location in memory is also emitted;
4. Write the program footer to output.

1.1.1 Elaborate

p. **??** is a function, written in C++, that performs the first task in the process. As mentioned earlier, elaborating a topological ordering of nodes with a depth-first recursive structure is much like actually computing an FFT with a depth-first recursive program (cf. Listing 3 in Appendix 2). Table 1 lists the nodes contained in the list '`ns`' after elaborating a size-8 transform by invoking `elaborate(8, 0, 0, 0)`.

```
        ~~CSplitRadix::elaborate(int~N,~int~ioffset,~int~offset,~int~stride)~{
~~~~if(N~>~4)~{
~~~~~~elaborate(N/2,~ioffset,~offset,~stride+1);
~~~~~~if(N/4~>=~4)~{
~~~~~~~~elaborate(N/4,~ioffset+(1≪stride),~offset+(N/2),~stride+2);
~~~~~~~~elaborate(N/4,~ioffset-(1≪stride),~offset+(3*N/4),~stride+2);
~~~~~~}else{
~~~~~~~~CNodeLoad~*n~=~new~CNodeLoad(this,~4,~ioffset,~stride,~0);
~~~~~~~~ns.push_back(assign_leaf_registers(n));
~~~~~~}
~~~~~~for(int~k=0;k<N/4;k++)~{
~~~~~~~~CNodeBfly~*n~=~new~CNodeBfly(this,~4,~k,~stride);
~~~~~~~~ns.push_back(assign_body_registers(n,k,N);
~~~~~~}
~~~~}else~if(N==4)~{
~~~~~~CNodeLoad~*n~=~new~CNodeLoad(this,~4,~ioffset,~stride,~1);
~~~~~~ns.push_back(assign_leaf_registers(n));
~~~~}else~if(N==2)~{
~~~~~~CNodeLoad~*n~=~new~CNodeLoad(this,~2,~ioffset,~stride,~1);
~~~~~~ns.push_back(assign_leaf_registers(n));
~~~~}
~~}
```

Listing 1: Elaborate function for hard-coded conjugate-pair FFT

A transform is divided into sub-transforms with recursive calls at lines 4, 6 and 7, until the base cases of size 2 or size 4 are reached at the leaves of the elaboration. As well as the size-2 and size-4 base cases, which are handled at lines 20-21 and 17-18 (respectively), there is a special case where two size-2 base cases are handled in parallel at lines 9-10. This special case of handling two size-2 base cases as a larger size-4 node ensures that larger transforms are composed of nodes that are homogeneous in size – this is of little utility when emitting $VL = 1$ code, but it is exploited in "Other vector lengths" (Section 1.2: Other vector lengths) where the topological ordering of nodes is vectorized. The second row of Table 1 is just such a special case, since two size-2 leaf nodes are being computed, and thus the size is listed as 2(x2).

The `elaborate` function modifies the class member variable '`ns`' at lines 10, 14, 18 and 21, where it appends a new node to the back of the list. After the function returns, the `ns` list represents a topological ordering of the computation with `CNodeLoad` and `CNodeBfly` nodes. The nodes of type `CNodeLoad` represent leaf computations: these computations load elements from the input array and perform a small amount of leaf computation, leaving the result in a set of registers. The `CNodeBfly` nodes represent computations in the body of the transform: these use a twiddle factor to perform a butterfly computation on a vector of registers, leaving the result in the same registers.

Type	Size	Addresses	Registers	Twiddle
`CNodeLoad`	4	{0,4,2,6}	{0,1,2,3}	
`CNodeLoad`	2(x2)	{1,5,7,3}	{4,5,6,7}	
`CNodeBfly`	4		{0,2,4,6}	ω_8^0
`CNodeBfly`	4		{1,3,5,7}	ω_8^1

Table 1: VL-1 size-8 conjugate-pair transform nodes

The constructor for a `CNodeLoad` object computes input array addresses for the load operations using the input array offset (`ioffset`), the input array `stride`, the size of the node (the nodes instantiated at lines 9 and 17 are size-4, and the node instantiated at line 20 is size-2) and a final parameter that is non-zero if the node is a single node (the nodes instantiated at lines 17 and 20 are single nodes, while the node instantiated at line 9 is composed of two size-2 nodes).

As the newly instantiated `CNodeLoad` objects are appended to the back of `ns` at lines 10, 14 and 21, the `assign_leaf_registers` function assigns registers to the outputs of each instance. Registers are identified with integers beginning at zero, and when each register is created it is assigned an identifier from an auto-incrementing counter ($R_{counter}$). This function also maintains a map of registers to node pointers, referred to as `rmap`, where the node for a given register is the last node to reference that register.

The constructor for a `CNodeBfly` object uses `k` and `stride` to compute a twiddle factor for the new instance of a butterfly computation node. When the new instance of `CNodeBfly` is appended to the end of `ns` at line 14, the `assign_body_registers` function assigns registers R_i to a node of size $N_{\texttt{node}}$ with the following logic:

$$R_i = R_{\texttt{counter}} - N + k + i \times \frac{N}{4} \tag{1}$$

where $i = 0, \cdots, N_{\texttt{node}} - 1$ and $R_{\texttt{counter}}$ is the auto-incrementing register counter. The `assign_body_registers` functions also updates the map of registers to node pointers by setting `rmap` $[R_i]$ to point to the new instance of `CNodeBfly`.

1.1.2 Emitting code

```
        ~~void~sfft_dcf8_hc(sfft_plan_t~*p,~const~void~*vin,~void~*vout)~{
~~~~const~SFFT_D~*in~=~vin;
~~~~SFFT_D~*out~=~vout;
~~~~SFFT_R~r0,r1,r2,r3,r4,r5,r6,r7;
~
~~~~L_4(in+0,in+8,in+4,in+12,&r0,&r1,&r2,&r3);
~~~~L_2(in+2,in+10,in+14,in+6,&r4,&r5,&r6,&r7);
~~~~K_0(&r0,&r2,&r4,&r6);
~~~~S_4(r0,r2,r4,r6,out+0,out+4,out+8,out+12);
~~~~K_N(VLIT2(0.7071,0.7071),VLIT2(0.7071,-0.7071),&r1,&r3,&r5,&r7);
~~~~S_4(r1,r3,r5,r7,out+2,out+6,out+10,out+14);
~~}
```

Listing 2: Hard-coded VL-1 size-8 FFT

Given a list of nodes, it is a simple process to emit C code that can be compiled to actually compute the transform.

The example in p. ?? would be emitted from the list of four nodes in Table 1. Lines 1–4 are emitted from a function that generates a header, and line 13 is emitted from a function that generates a footer. Lines 6–11 are generated based on the list of nodes.

p. ?? contains references to several types, functions and macros that use upper-case identifiers – these are *primitive* functions or types that have been predefined as inline functions or macros. A benefit of using primitives in this way is that the details specific to numerical representation and the underlying machine have been abstracted away; thus, the same function can be compiled for a variety of types and machines by simply including a different header file with different primitives. p. ??, for example, could be compiled for double-precision arithmetic on an SSE2 machine by including `sse_double.h`, or it could be compiled with much slower scalar arithmetic by including `scalar.h`. The same code can even be used, without modification, to compute forward and backwards transforms, by using C preprocessor directives to conditionally alter the macros.

In order to accommodate mixed numerical representations, the signature of the outermost function references data with void pointers. In the case of the double-precision example in p. ??, `SFFT_D` would be defined to be `double` in the appropriate header file, and the void pointers are then cast to `SFFT_D` pointers.

The size-8 transform in Table 1 uses 8 registers, and thus a declaration of 8 registers of type `SFFT_R` has been emitted at line 4 in p. ??. In the case of double-precision arithmetic on a SSE2 machine, `SFFT_R` is defined as `__m128d` in `sse_double.h`.

The first two rows of Table 1 correspond to lines 6 and 7 of p. ??, respectively. The `L_4` primitive is used to compute the size-4 leaf node in the first row of the table. The second row is a load/leaf node of size 2(x2), indicating two size-2 nodes in parallel, which is computed with the `L_2` primitive. The input addresses in the table are the addresses of complex words, while the addresses in the generated code refer to the real and imaginary parts of a complex word, and thus the addresses from Table 1 are multiplied by a factor of 2 to obtain the addresses in p. ??.

The final two `CNodeBfly` nodes of Table 1 correspond to the `K_0` and `K_N` sub-transform (a.k.a. butterfly) primitives at lines 8 and 10, respectively. Because the node in the third row of Table 1 has a twiddle factor of ω_8^0 (i.e., unity), the computation requires no multiplication, and the `K_0` primitive is used for this special case. The `K_N` primitive at line 10 does require a twiddle factor, which is passed to `K_N` as two vector literals that represent the twiddle factor in unpacked form.[1]Fast interleaved complex multiplication describes how interleaved complex multiplication is faster if one operand is pre-unpacked.

After each node is processed, the registers that have been used by it are checked in a map (`rmap`) that maps each register to the last node to have used that register. If the current node is the last node to have used a register, the register is stored to memory. In the case of the transform in p. ??, four registers are stored with an instance of the `S_4` primitive at lines 9 and 11. In contrast to the load operations at the leaves, which are decimated-in-time and thus effectively pseudo-random memory accesses, the store operations are to linear regions of memory, the addresses of which can be determined from each register's integer identifier. The store address offset for data in register R_i is simply $i \times 2 \times VL$.

1.2 Other vector lengths

If $VL > 1$, the list of nodes that results from the `elaborate` function in p. ?? is vectorized. Broadly speaking, `CNodeLoad` objects that operate on adjacent memory locations are collected together and computed in parallel. After each such computation, each position in a vector register contains an element that belongs to a different node. Transposes are then used to transform sets of vector registers such that each register contains elements from one node. Finally, the `CNodeBfly` objects can be easily computed in parallel, as they were with VL-1 because the elements in each vector register correspond to one node.

[1]For the purposes of brevity, the precision has been truncated to only a few decimal places.

1.2.1 Overview

Table 2 lists the nodes that represent a VL-1 size-16 transform. A VL of 2 implies that each vector register contains 2 complex words, and load operations on each of the 4 addresses in the first row of Table 2 will also load the complex words in the adjacent memory locations. Note that the complex words that would be incidentally loaded in the upper half of the VL-2 registers are the complex words that the third `CNodeLoad` object at row 5 would have loaded. This is exploited to load and compute the first and third `CNodeLoad` objects in parallel.

Type	Size	Addresses	Registers	Twiddle
`CNodeLoad`	4	{0,8,4,12}	{0,1,2,3}	
`CNodeLoad`	2(x2)	{2,10,14,6}	{4,5,6,7}	
`CNodeBfly`	4		{0,2,4,6}	ω_{16}^{0}
`CNodeBfly`	4		{1,3,5,7}	ω_{16}^{2}
`CNodeLoad`	4	{1,9,5,13}	{8,9,10,11}	
`CNodeLoad`	4	{15,7,3,11}	{12,13,14,15}	
`CNodeBfly`	4		{0,4,8,12}	ω_{16}^{0}
`CNodeBfly`	4		{1,5,9,13}	ω_{16}^{1}
`CNodeBfly`	4		{2,6,10,14}	ω_{16}^{2}
`CNodeBfly`	4		{3,7,11,15}	ω_{16}^{3}

Table 2: VL-1 size-16 conjugate-pair transform nodes

Type	Sizes	Addresses	Registers	Twiddles
Load	{4,4}	{{0,1},{8,9},{4,5},{12,13}}	{{0,1},{2,3},{8,9},{10,11}}	
Load	{2(x2),4}	{{2,3},{10,11},{14,15},{6,7}}	{{4,5},{6,7},{14,15},{12,13}}	
Bfly	{4,4}		{{0,1},{2,3},{4,5},{6,7}}	{ ω_{16}^{0}, ω_{16}^{2} }
Bfly	{4,4}		{{0,1},{4,5},{8,9},{12,13}}	{ ω_{16}^{0}, ω_{16}^{1} }
Bfly	{4,4}		{{2,3},{6,7},{10,11},{14,15}}	{ ω_{16}^{2}, ω_{16}^{3} }

Table 3: VL-2 size-16 conjugate-pair transform nodes

The second `CNodeLoad` object computes two size-2 leaf transforms in parallel, while the last `CNodeLoad` object computes a size-4 leaf transform. Because the size-4 transform is composed of two size-2 transforms, and memory addresses of the fourth `CNodeLoad` are adjacent (although permuted), some of the computation can be computed in parallel.

If the `CNodeLoad` objects at rows 1 and 5 are computed in parallel, the output will be four VL-2 registers: {{0,8}, {1,9}, {2,10}, {3,11}} – i.e., the first register contains what would have been register 0 in the lower half, and what would have been register 8 in the top half etc. Similarly, computing rows 2 and 6 in parallel would yield four VL-2 registers: {{4,14}, {5,15}, {6,12}, {7,13}} – note the permutation of the upper halves in this case. These registers are transposed to {{0,1}, {2,3}, {8,9}, {10,11}} and {{4,5}, {6,7}, {14,15}, {12,13}}, as in row 1 and 2 of Table 3.

With the transposed VL-2 registers, it is now possible to compute `CNodeBfly` nodes in parallel. For example, rows 2 and 3 of Table 2 can be computed in parallel on four VL-2 registers represented by {{0,1}, {2,3}, {4,5}, {6,7}}, as in row 3 of Table 3.

1.2.2 Implementation

p. ?? is a C++ implementation of the `vectorize_loads` function. This function modifies a topological ordering of nodes (the class member variable `ns`) and uses two other functions: `find_parallel_loads`, which searches forward from the current node to find another `CNodeLoad` that shares adjacent memory addresses; and `merge_loads(a,b)`, which adds the addresses, registers and type of `b` to `a`. Type introspection is used at lines 7 and 36 (and in other Listings), to differentiate between the two types of object.

p. ?? is a C++ implementation of the `vectorize_ks` function. For each `CNodeBfly` node, the function searches forward for another `CNodeBfly` that does not have a register dependence. Once found, the registers of the latter node are added to the former node, and the latter node erased. Finally, at line 19, the registers of the vectorized `CNodeBfly` node are merged using a perfect shuffle, which is then recursively applied on each half of the list. The effect is a merge that works for any power of 2 vector length.

```
        ~~void~CSplitRadix::vectorize_ks()~{
~~~~vector<CNodeHardCoded~*>::iterator~i;
~~~~for(i=ns.begin();~i~!=~ns.end();++i)~{
~~~~~~if(!(*i)->type().compare(``blockbfly''))~{
~~~~~~~~vector<CNodeHardCoded~*>::iterator~j~=~i+1,~pj~=~i;
~~~~~~~~int~count~=~1;
~~~~~~~~while(j~!=~ns.end()~&&~count~<~VL)~{
~~~~~~~~~~if(!(*j)->type().compare(``blockbfly'')~&&~!register_dependence(*i,~*j))~{
~~~~~~~~~~~~(*i)->rs.insert(~(*i)->rs.end(),~(*j)->rs.begin(),~(*j)->rs.end());
~~~~~~~~~~~~ns.erase(j);
~~~~~~~~~~~~count++;
~~~~~~~~~~~~j~=~pj+1;
~~~~~~~~~~}else~{
~~~~~~~~~~~~pj~=~j;~++j;
~~~~~~~~~~}
~~~~~~~~}
~~~~~~~~(*i)->merge_rs();
~~~~~~}
~~~~}
~~}
```

Listing 3: Body node vectorization

```
        ~~CNodeLoad~*
~~CSplitRadix::find_parallel_load(vector<CNodeHardCoded*>::iterator~i){
~~~~CNodeLoad~*b~=~(CNodeLoad~*)(*i);
~~~~for(int~k=0;k<((N>2)?4:2);k++)~{
~~~~~~vector<CNodeHardCoded~*>::iterator~j~=~i+1;
~~~~~~while(j~!=~ns.end())~{
~~~~~~~~if(!(*j)->type().compare(``blockload''))~{
~~~~~~~~~~CNodeLoad~*b2~=~(CNodeLoad~*)(*j);
~~~~~~~~~~if(b2->iaddrs[k]~>~b->iaddrs[0]~&&~b2->iaddrs[k]~<~b->iaddrs[0]+VL)~{
~~~~~~~~~~~~ns.erase(j);
~~~~~~~~~~~~return~b2;
~~~~~~~~~~}
~~~~~~~~~~++j;
~~~~~~~~}
~~~~~~}
~~~~}
~~~~return~NULL;
~~}
~~void~CSplitRadix::merge_loads(CNodeLoad~*b1,~CNodeLoad~*b2)~{
~~~~for(int~i=0;i<b1->size;i++)~{
~~~~~~for(int~j=0;j<b2->iaddrs.size();j++)~{
~~~~~~~~if(b2->iaddrs[j]~>~b1->iaddrs[i]~&&~b2->iaddrs[j]~<~b1->iaddrs[i]+VL)~{
~~~~~~~~~~b1->iaddrs.push_back(b2->iaddrs[j]);
~~~~~~~~~~b1->rs.push_back(b2->rs[j]);
~~~~~~~~~~if(rmap[b2->rs[j]]~==~b2)~rmap[b2->rs[j]]~=~b1;
~~~~~~~~}
~~~~~~}
~~~~}
~~~~b1->types.push_back(b2->types[0]);
~~}
~~void~CSplitRadix::vectorize_loads()~{
~~~~vector<CNodeHardCoded~*>::iterator~i;
~~~~for(i=ns.begin();~i~!=~ns.end();++i)~{
~~~~~~if(!(*i)->type().compare(``blockload''))~{
~~~~~~~~while(CNodeLoad~*b2~=~find_parallel_load(i))
~~~~~~~~~~merge_loads((CNodeLoad~*)(*i),~b2);
~~~~~~}
~~~~}
~~}
```

Listing 4: Leaf node vectorization

If `vectorize_loads` and `vectorize_ks` are invoked with $VL = 2$ on the topological ordering of nodes in Table 2, the result is the vectorized node list shown in Table 3. As in "Emitting code" (Section 1.1.2: Emitting code), emitting code is a fairly simple process, and p. **??** is the code emitted from the node list in . There are only a few differences to note about the emitted code when $VL > 1$.

1. The register identifiers in line 4 of p. ?? consist of a list of two integers delimited with an underscore. The integers listed in each register's name are the VL-1 registers that were subsumed to create the larger register (cf. VL-1 code in p. ??);
2. The leaf primitives (lines 6 and 7 in p. ??) have a list of underscore delimited integers in the name, where each integer corresponds to the type of sub-transform to be computed on that position in the vector registers. For example, the `L_4_4` primitive is named to indicate a size-4 leaf operation on the lower and upper halves of the vector registers, while the `L_2_4` primitive performs two size-2 leaf operations on the lower half of the registers and a size-4 leaf operation on the upper halves;
3. The body node primitives (`K_N`) and store primitives (`S_4`) are unchanged because they perform the same operation on each element of the vector registers. This is as a result of the register transposes that were previously performed on the outputs of the leaf primitives.

```
        void~sfft_fcf16_hc(sfft_plan_t~*p,~const~void~*vin,~void~*vout)~{
~~const~SFFT_D~*in~=~vin;
~~SFFT_D~*out~=~vout;
~~SFFT_R~r0_1,r2_3,r4_5,r6_7,r8_9,r10_11,r12_13,r14_15;
~
~~L_4_4(in+0,in+16,in+8,in+24,&r0_1,&r2_3,&r8_9,&r10_11);
~~L_2_4(in+4,in+20,in+28,in+12,&r4_5,&r6_7,&r14_15,&r12_13);
~~K_N(VLIT4(0.7071,0.7071,1,1),
~~~~~~VLIT4(0.7071,-0.7071,0,-0),
~~~~~~&r0_1,&r2_3,&r4_5,&r6_7);
~~K_N(VLIT4(0.9239,0.9239,1,1),
~~~~~~VLIT4(0.3827,-0.3827,0,-0),
~~~~~~&r0_1,&r4_5,&r8_9,&r12_13);
~~S_4(r0_1,r4_5,r8_9,r12_13,out+0,out+8,out+16,out+24);
~~K_N(VLIT4(0.3827,0.3827,0.7071,0.7071),
~~~~~~VLIT4(0.9239,-0.9239,0.7071,-0.7071),
~~~~~~&r2_3,&r6_7,&r10_11,&r14_15);
~~S_4(r2_3,r6_7,r10_11,r14_15,out+4,out+12,out+20,out+28);
}
```

Listing 5: Hard-coded VL-2 size-16 FFT

1.2.3 Scalability

So far, hard-coded transforms of vector length 1 and 2 have been presented. On Intel machines, VL-1 can be used to compute double-precision transforms with SSE2, while VL-2 can be used to compute double-precision transforms with AVX *and* single-precision transforms with SSE. The method of vectorization presented in this chapter scales above VL-2, and has been successfully used to compute VL-4 single-precision transforms with AVX.

The leaf primitives were coded by hand in all cases; VL-1 required `L_2` and `L_4`, while VL-2 required `L_2_2`, `L_2_4`, `L_4_2` and `L_4_4`. In the case of VL-4, not all permutations of possible leaf primitive were required – only 11 out of 16 were needed for the transforms that were generated.

It is an easy exercise to code the leaf primitives for $VL \leq 4$ by hand, but for future machines that might feature vector lengths larger than 4, the leaf primitives could be automatically generated (in fact, "Other vector lengths" (Section 3.5: Other vector lengths) is concerned with automatic generation of leaf sub-transforms at another level of scale).

1.2.4 Constraints

For a transform of size N and leaf node size of S ($S = 4$ in the examples in this chapter), the following constraint must be satisfied:

$$N/VL \geq S \tag{2}$$

If this constraint is not satisfied, the size of either VL or S must be reduced. In practice, VL and S are small relative to the size of most transforms, and thus these corner cases typically only occur for very small sized transforms. Such an example is a size-2 transform when $VL = 2$ and $S = 4$, where in this case the transform is too small to be computed with SIMD operations and should be computed with scalar arithmetic instead.

1.3 Performance

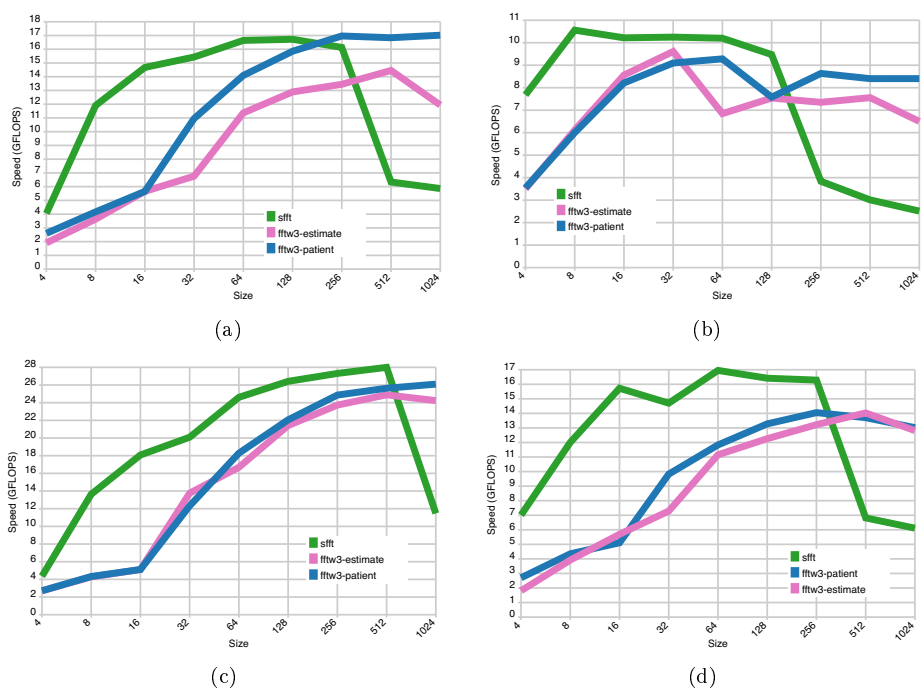

Figure 1: Performance of hard-coded FFTs on a Macbook Air 4,2. (a) Single-precision, SSE (VL-2) (b) Double-precision, SSE (VL-1) (c) Single-precision, AVX (VL-4) (d) Double-precision, AVX (VL-2)

Figure 1 shows the results of a benchmark for transforms of size 4 through to 1024 running on a Macbook Air 4,2. The speed of FFTW 3.3 running in estimate and patient modes is also shown for comparison.

FFTW running in patient mode evaluates a huge configuration space of parameters, while the hard-coded FFT required no calibration.

A variety of vector lengths are represented, and the hard-coded FFTs have good performance while $N/VL \leq 128$. After this point, performance drops off and other techniques should be used. The following sections use the hard-coded FFT as a foundation for scaling to larger sizes of transforms.

2 Hard-coded four-step

This section presents an implementation of the four-step algorithm [1] that leverages hard-coded sub-transforms to compute larger transforms. The implementation uses an implicit memory transpose (along with vector register transposes) and scales particularly well with VL. In contrast to the fully hard-coded implementation in the previous section, the four-step implementation requires no new leaf primitives as VL increases, i.e., the code is much the same when $VL > 1$ as it is when $VL = 1$.

2.1 The four-step algorithm

A transform of size N is decomposed into a two-dimensional array of size $n_1 \times n_2$ where $N = n_1 n_2$. Selecting $n_1 = n_2 = \sqrt{N}$ (or close) often obtains the best performance results [1]. When either of the factors is larger than the other, it is the larger of the two factors that will determine performance, because the larger factor effectively brings the memory wall closer. The four steps of the algorithm are:

1. Compute n_1 FFTs of length n_2 along the columns of the array;
2. Multiply each element of the array with ω_N^{ij}, where i and j are the array coordinates;
3. Transpose the array;
4. Compute n_2 FFTs of length n_1 along the columns of the array.

For this out-of-place implementation, steps 2 and 3 are performed as part of step 1. Step 1 reads data from the input array and computes the FFTs, but before storing the data in the final pass, it is multiplied by the twiddle factors from step 2. After this, the data is stored to *rows* in the output array, and thus the transpose of step 3 is performed implicitly. Step 4 is then computed as usual: FFTs are computed along the columns of the output array.

This method of computing the four-step algorithm in two steps requires only minor modifications in order to support multiple vector lengths: with $VL > 1$, multiple columns are read and computed in parallel without modification of the code, but before storing multiple columns of data to rows, a register transpose is required.

2.2 Vector length 1

When $VL = 1$, three hard-coded FFTs are elaborated.

1. FFT of length n_2 with stride $n_1 \times 2$ for the first column of step 1;
2. FFT of length n_2 with stride $n_1 \times 2$ and twiddle multiplications on outputs – for all other columns of step 1;
3. FFT of length n_1 with stride $n_2 \times 2$ for columns in step 4.

In order to generate the code for the four-step sub-transforms, some minor modifications are made to the fully hard-coded code generator that was presented in the previous section.

The first FFT is used to handle the first column of step 1, where there are no twiddle factor multiplications because one of the array coordinates for step 2 is zero, and thus ω_N^0 is unity. This FFT may be elaborated as in "Vector length 1" (Section 1.1: Vector length 1) with the addition of a stride factor for the input address calculation. The second FFT is elaborated as per the first FFT, but with the addition of twiddle factor multiplications on each register prior to the store operations. The third FFT is elaborated as per the first FFT, but with strided input *and* output addresses.

```
        const~SFFT_D~__attribute__~((aligned(32)))~*LUT;
const~SFFT_D~*pLUT;
void~sfft_dcf64_fs_x1_0(sfft_plan_t~*p,~const~void~*vin,~void~*vout){
~~const~SFFT_D~*in~=~vin;
~~SFFT_D~*out~=~vout;
~~SFFT_R~r0,r1,r2,r3,r4,r5,r6,r7;
~~L_4(in+0,in+64,in+32,in+96,&r0,&r1,&r2,&r3);
~~L_2(in+16,in+80,in+112,in+48,&r4,&r5,&r6,&r7);
~~K_0(&r0,&r2,&r4,&r6);
~~S_4(r0,r2,r4,r6,out+0,out+4,out+8,out+12);
~~K_N(VLIT2(0.7071,0.7071),VLIT2(0.7071,-0.7071),&r1,&r3,&r5,&r7);
~~S_4(r1,r3,r5,r7,out+2,out+6,out+10,out+14);
}
void~sfft_dcf64_fs_x1_n(sfft_plan_t~*p,~const~void~*vin,~void~*vout){
~~const~SFFT_D~*in~=~vin;
~~SFFT_D~*out~=~vout;
~~SFFT_R~r0,r1,r2,r3,r4,r5,r6,r7;
~~L_4(in+0,in+64,in+32,in+96,&r0,&r1,&r2,&r3);
~~L_2(in+16,in+80,in+112,in+48,&r4,&r5,&r6,&r7);
~~K_0(&r0,&r2,&r4,&r6);
~~r2~=~MUL(r2,LOAD(pLUT+4),LOAD(pLUT+6));
~~r4~=~MUL(r4,LOAD(pLUT+12),LOAD(pLUT+14));
~~r6~=~MUL(r6,LOAD(pLUT+20),LOAD(pLUT+22));
~~S_4(r0,r2,r4,r6,out+0,out+4,out+8,out+12);
~~K_N(VLIT2(0.7071,0.7071),VLIT2(0.7071,-0.7071),&r1,&r3,&r5,&r7);
~~r1~=~MUL(r1,LOAD(pLUT+0),LOAD(pLUT+2));
~~r3~=~MUL(r3,LOAD(pLUT+8),LOAD(pLUT+10));
~~r5~=~MUL(r5,LOAD(pLUT+16),LOAD(pLUT+18));
~~r7~=~MUL(r7,LOAD(pLUT+24),LOAD(pLUT+26));
~~S_4(r1,r3,r5,r7,out+2,out+6,out+10,out+14);
~~pLUT~+=~28;
}
void~sfft_dcf64_fs_x2(sfft_plan_t~*p,~const~void~*vin,~void~*vout){
~~const~SFFT_D~*in~=~vin;
~~SFFT_D~*out~=~vout;
~~SFFT_R~r0,r1,r2,r3,r4,r5,r6,r7;
~~L_4(in+0,in+64,in+32,in+96,&r0,&r1,&r2,&r3);
~~L_2(in+16,in+80,in+112,in+48,&r4,&r5,&r6,&r7);
~~K_0(&r0,&r2,&r4,&r6);
~~S_4(r0,r2,r4,r6,out+0,out+32,out+64,out+96);
~~K_N(VLIT2(0.7071,0.7071),VLIT2(0.7071,-0.7071),&r1,&r3,&r5,&r7);
~~S_4(r1,r3,r5,r7,out+16,out+48,out+80,out+112);
}
void~sfft_dcf64_fs(sfft_plan_t~*p,~const~void~*vin,~void~*vout)~{
~~const~SFFT_D~*in~=~vin;
~~SFFT_D~*out~=~vout;
~~pLUT~=~~LUT;
~~int~i;
~~sfft_dcf64_fs_x1_0(p,~in,~out);
~~for(i=1;i<8;i++)~sfft_dcf64_fs_x1_n(p,~in+(i*2),~out+(i*16));
~~for(i=0;i<8;i++)~sfft_dcf64_fs_x2(p,~out+(i*2),~out+(i*2));
}
```


Listing 6: Hard-coded four-step VL-1 size-64 FFT

2.2.1 Example

p. **??** is a VL-1 size-64 hard-coded four-step FFT. Before it can be used, an initialization procedure (not shown) allocates and populates the LUT at line 1 with the twiddle factors that are required for the step 2 multiplications. Line 44 shows the main function that executes the first sub-transform on the first column (line 49), and the second sub-transform on all remaining columns (line 50). Finally, the sub-transforms corresponding to step 4 of the four-step algorithm are executed on all columns in line 51.

The twiddle factor multiplication that corresponds to step 2 of the four-step algorithm takes place in lines 21-23 and lines 26-29. The first register is not multiplied with a twiddle factor because the first row of twiddle factors are ω_N^0 (i.e., unity). The other registers are multiplied with two registers loaded from the LUT, which are the unpacked real and imaginary parts (see Fast interleaved complex multiplication for details about unpacked complex multiplication).

2.3 Other vector lengths

```
        const~SFFT_D~__attribute__~((aligned(32)))~*LUT;
const~SFFT_D~*pLUT;
void~sfft_fcf64_fs_x1(sfft_plan_t~*p,~const~void~*vin,~void~*vout)~{
~~const~SFFT_D~*in~=~vin;
~~SFFT_D~*out~=~vout;
~~SFFT_R~r0,r1,r2,r3,r4,r5,r6,r7;
~~L_4(in+0,in+64,in+32,in+96,&r0,&r1,&r2,&r3);
~~L_2(in+16,in+80,in+112,in+48,&r4,&r5,&r6,&r7);
~~K_0(&r0,&r2,&r4,&r6);
~~K_N(VLIT4(0.7071,0.7071,0.7071,0.7071),
~~~~~~VLIT4(0.7071,-0.7071,0.7071,-0.7071),&r1,&r3,&r5,&r7);
~~r1~=~MUL(r1,LOAD(pLUT+0),LOAD(pLUT+4));
~~TX2(r0,r1);
~~r2~=~MUL(r2,LOAD(pLUT+8),LOAD(pLUT+12));
~~r3~=~MUL(r3,LOAD(pLUT+16),LOAD(pLUT+20));
~~TX2(r2,r3);
~~r4~=~MUL(r4,LOAD(pLUT+24),LOAD(pLUT+28));
~~r5~=~MUL(r5,LOAD(pLUT+32),LOAD(pLUT+36));
~~TX2(r4,r5);
~~r6~=~MUL(r6,LOAD(pLUT+40),LOAD(pLUT+44));
~~r7~=~MUL(r7,LOAD(pLUT+48),LOAD(pLUT+52));
~~TX2(r6,r7);
~~S_4(r0,r2,r4,r6,out+0,out+4,out+8,out+12);
~~S_4(r1,r3,r5,r7,out+16,out+20,out+24,out+28);
~~pLUT~+=~56;
}
void~sfft_fcf64_fs_x2(sfft_plan_t~*p,~const~void~*vin,~void~*vout)~{
~~const~SFFT_D~*in~=~vin;
~~SFFT_D~*out~=~vout;
~~SFFT_R~r0,r1,r2,r3,r4,r5,r6,r7;
~~L_4(in+0,in+64,in+32,in+96,&r0,&r1,&r2,&r3);
~~L_2(in+16,in+80,in+112,in+48,&r4,&r5,&r6,&r7);
~~K_0(&r0,&r2,&r4,&r6);
~~K_N(VLIT4(0.7071,0.7071,0.7071,0.7071),
~~~~~~VLIT4(0.7071,-0.7071,0.7071,-0.7071),&r1,&r3,&r5,&r7);
~~S_4(r0,r2,r4,r6,out+0,out+32,out+64,out+96);
~~S_4(r1,r3,r5,r7,out+16,out+48,out+80,out+112);
}
void~sfft_fcf64_fs(sfft_plan_t~*p,~const~void~*vin,~void~*vout)~{
~~const~SFFT_D~*in~=~vin;
~~SFFT_D~*out~=~vout;
~~pLUT~=~~LUT;
~~int~i;
~~for(i=0;i<4;i++)~sfft_fcf64_fs_x1(p,~in+(i*4),~out+(i*32));
~~for(i=0;i<4;i++)~sfft_fcf64_fs_x2(p,~out+(i*4),~out+(i*4));
}
```

Listing 7: Hard-coded four-step VL-2 size-64 FFT

For $VL > 1$, the FFTs along the columns are computed in parallel. Thus, in step 1, n_1/VL FFTs are computed along the columns of the array with stride $= 2 \times VL$, and in step 4, n_2/VL FFTs are computed along the columns with stride $= 2 \times VL$.

An implication of computing the first column in parallel with other columns is that the first column is now multiplied by unity twiddle factors, and thus only two sub-transforms are used instead of three.

The only other difference when $VL > 1$ is that the registers need to be transposed before storing columns to rows (the implicit transpose that corresponds to step 3). To accomplish this when generating code, $n = VL$ store operations are latched before the transpose and store code is emitted.

2.3.1 Example

p. **??** implements a VL-2 size-64 hard-coded four-step FFT. The main function (line 39) computes 8 FFTs along the columns for step 1 at line 44, and 8 FFTs along the columns for step 4 at line 45. There are only 4 iterations of the loop in each case because two sub-transforms are computed in parallel with each invocation of the sub-transform function.

In the function corresponding to the sub-transforms of step 1 (line 3), two store operations are latched (lines 23 and 24) before emitting code, which includes the preceding transposes (the `TX2` operations) and twiddle factor multiplications (lines 13–22).

2.4 Performance

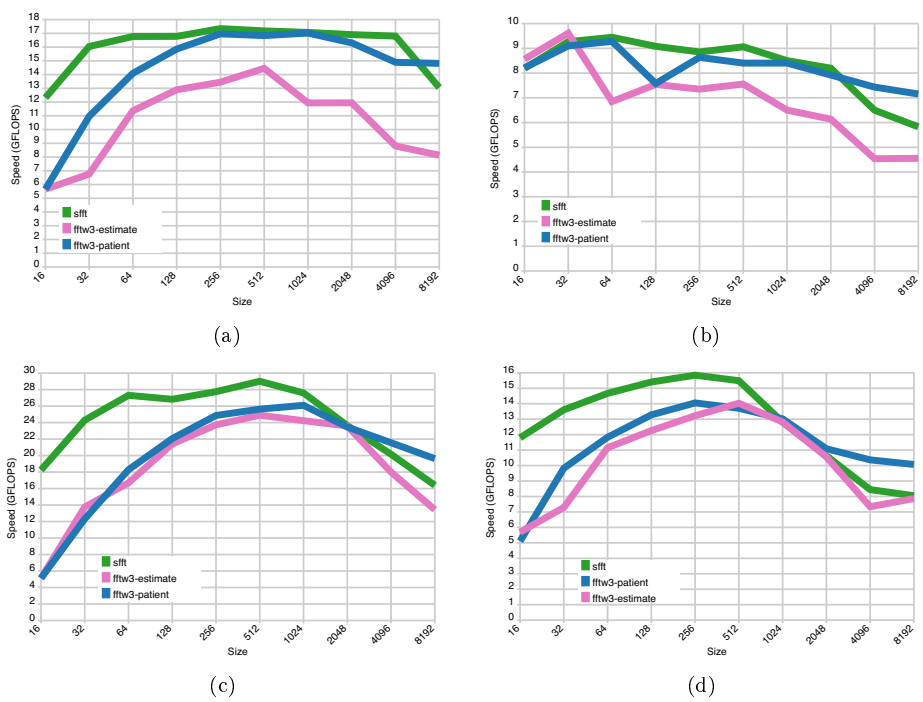

Figure 2: Performance of hard-coded four-step FFTs on a Macbook Air 4,2. (a) Single-precision, SSE (VL-2) (b) Double-precision, SSE (VL-1) (c) Single-precision, AVX (VL-4) (d) Double-precision, AVX (VL-2)

Figure 2 shows the results of a benchmark for transforms of size 16 through to 8192 running on a Macbook Air 4,2. The speed of FFTW 3.3 running in estimate and patient modes is also shown for contrast.

The results show that the performance of the four-step algorithm improves as the length of the vector increases, but, as was the case with the hard-coded FFTs in "Fully hard-coded" (Section 1: Fully hard-coded), the performance of the hard-coded four-step FFTs is limited to a certain range of transform size.

3 Hard-coded leaves

The performance of the fully hard-coded transforms presented in "Fully hard-coded" (Section 1: Fully hard-coded) only scales while $N/VL \leq 128$. This section presents techniques that are similar to those found in the fully hard-coded transforms, but applied at another level of scale in order to scale performance to larger sizes.

3.1 Vector length 1

The fully hard-coded transforms in "Fully hard-coded" (Section 1: Fully hard-coded) used two primitives at the leaves: a size-4 sub-transform (`L_4`) and a double size-2 sub-transform (`L_2`). These sub-transforms loaded four elements of data from the input array, performed a small amount of computation, and stored the four results to the output array.

Performance is scaled to larger transforms by using larger sub-transforms at the leaves of the computation. These are automatically generated using fully hard-coded transforms, and thus the size of the leaf computations is easily parametrized, which is just as well, because the optimal leaf size is dependent on the size of the transform, the compiler, and the target machine.

The process of elaborating a topological ordering of nodes representing a hard-coded leaf transform of size N with leaf sub-transforms of size N_{leaf} is as follows:

1. Elaborate a size N_{leaf} sub-transform;
2. Elaborate a two size $N_{leaf}/2$ sub-transforms as one sub-transform;
3. Elaborate the main transform using the sub-transforms from steps 1 and 2 as the leaves of the computation.

The node lists for steps 1 and 2 are elaborated using the fully hard-coded `elaborate` function from p. **??**, but because the leaf sub-transform in step 2 is actually two sub-transforms of size $N_{leaf}/2$, the `elaborate` function is invoked twice with different offset parameters:

1. elaborate($N_{leaf}/2$, 0, 0, 1) ;
2. elaborate($N_{leaf}/2$, -1, $N_{leaf}/2$, 1) ;

The code corresponding to steps 1 and 2 is emitted slightly differently than was the case with the fully hard-coded transforms. Instead of hard coding the input array indices, the indices are themselves loaded from an array that is precomputed when the transform is initialized.

The node list corresponding to the main transform in step 3 is elaborated as in the function in p. **??**, but with some minor change. First, the recursion terminates with leaf nodes of size N_{leaf}. Second, because the loops in the body of the sub-transform will be at least $2 \times N_{leaf}$ iterations, the loop for the body sub-transforms (line 12 of p. **??**) is not statically unrolled. Instead only one node is added to the list of nodes, and the loop is computed dynamically.

```
        void~sfft_dcf64_hcl16_4_e(offset_t~*is,const~SFFT_D~*in,SFFT_D~*out){
~~SFFT_R~r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15;
~~L_4(in+is[0],in+is[1],in+is[2],in+is[3],&r0,&r1,&r2,&r3);
~~L_2(in+is[4],in+is[5],in+is[6],in+is[7],&r4,&r5,&r6,&r7);
~~K_0(&r0,&r2,&r4,&r6);
~~K_N(VLIT2(0.7071,0.7071),VLIT2(0.7071,-0.7071),&r1,&r3,&r5,&r7);
~~L_4(in+is[8],in+is[9],in+is[10],in+is[11],&r8,&r9,&r10,&r11);
~~L_4(in+is[12],in+is[13],in+is[14],in+is[15],&r12,&r13,&r14,&r15);
~~K_0(&r0,&r4,&r8,&r12);
~~S_4(r0,r4,r8,r12,out+0,out+8,out+16,out+24);
~~K_N(VLIT2(0.9239,0.9239),VLIT2(0.3827,-0.3827),&r1,&r5,&r9,&r13);
~~S_4(r1,r5,r9,r13,out+2,out+10,out+18,out+26);
~~K_N(VLIT2(0.7071,0.7071),VLIT2(0.7071,-0.7071),&r2,&r6,&r10,&r14);
~~S_4(r2,r6,r10,r14,out+4,out+12,out+20,out+28);
~~K_N(VLIT2(0.3827,0.3827),VLIT2(0.9239,-0.9239),&r3,&r7,&r11,&r15);
~~S_4(r3,r7,r11,r15,out+6,out+14,out+22,out+30);
}
void~sfft_dcf64_hcl16_4_o(offset_t~*is,const~SFFT_D~*in,SFFT_D~*out){
~~SFFT_R~r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15;
~~L_4(in+is[0],in+is[1],in+is[2],in+is[3],&r0,&r1,&r2,&r3);
~~L_2(in+is[4],in+is[5],in+is[6],in+is[7],&r4,&r5,&r6,&r7);
~~K_0(&r0,&r2,&r4,&r6);
~~S_4(r0,r2,r4,r6,out+0,out+4,out+8,out+12);
~~K_N(VLIT2(0.7071,0.7071),VLIT2(0.7071,-0.7071),&r1,&r3,&r5,&r7);
~~S_4(r1,r3,r5,r7,out+2,out+6,out+10,out+14);
~~L_4(in+is[8],in+is[9],in+is[10],in+is[11],&r8,&r9,&r10,&r11);
~~L_2(in+is[12],in+is[13],in+is[14],in+is[15],&r12,&r13,&r14,&r15);
~~K_0(&r8,&r10,&r12,&r14);
~~S_4(r8,r10,r12,r14,out+16,out+20,out+24,out+28);
~~K_N(VLIT2(0.7071,0.7071),VLIT2(0.7071,-0.7071),&r9,&r11,&r13,&r15);
~~S_4(r9,r11,r13,r15,out+18,out+22,out+26,out+30);
}
void~sfft_dcf64_hcl16_4_X_4(SFFT_D~*data,~int~N,~SFFT_D~*LUT){
~~X_4(data,~N,~LUT);
}
void~sfft_dcf64_hcl16_4(sfft_plan_t~*p,~const~void~*vin,~void~*vout){
~~const~SFFT_D~*in~=~vin;
~~SFFT_D~*out~=~vout;
~~p->is~=~p->is_base;
~~sfft_dcf64_hcl16_4_e(p->is,in,out+0);
~~p->is~+=~16;~~sfft_dcf64_hcl16_4_o(p->is,in,out+32);
~~sfft_dcf64_hcl16_4_X_4(out+0,32,p->ws[0]);
~~p->is~+=~16;~~sfft_dcf64_hcl16_4_e(p->is,in,out+64);
~~p->is~+=~16;~~sfft_dcf64_hcl16_4_e(p->is,in,out+96);
~~sfft_dcf64_hcl16_4_X_4(out+0,64,p->ws[1]);
}
```

Listing 8: Hard-coded VL-1 size-64 FFT with size-16 leaves

3.1.1 Example

p. ?? is a size-64 hard-coded leaf transform with size-16 leaves. The first function (lines 1–17) is a size-16 leaf sub-transform, while the second (lines 18–32) consists of two size-8 leaf sub-transforms in parallel. The main function (lines 36–46) invokes four leaf sub-transforms (lines 40, 41, 43 and 44), and two loops of body sub-transforms (lines 42 and 45).

The first parameter to the leaf functions (see lines 1 and 18) is a pointer into an array of precomputed indices for the input data array. At lines 41 and 43–44, the array is incremented before subsequent calls to the leaf functions, and at line 39 the pointer is reset to the base of the array so that the transform can be used repeatedly.

The function used for the body sub-transforms (lines 33–35) is a wrapper for a primitive that computes a radix-2/4 butterfly. The last parameter to this function is a pointer to a precomputed LUT of twiddle factors for a sub-transform of size N (the second parameter).

3.2 Improving memory locality in the leaves

Size	Input array addresses
16	{0, 64, 32, 96, 16, 80, 112, 48, 8, 72, 40, 104, 120, 56, 24, 88}
8(x2)	{4, 68, 36, 100, 20, 84, 116, 52, 124, 60, 28, 92, 12, 76, 108, 44}
16	{2, 66, 34, 98, 18, 82, 114, 50, 10, 74, 42, 106, 122, 58, 26, 90}
16	{126, 62, 30, 94, 14, 78, 110, 46, 6, 70, 38, 102, 118, 54, 22, 86}
16	{1, 65, 33, 97, 17, 81, 113, 49, 9, 73, 41, 105, 121, 57, 25, 89}
8(x2)	{5, 69, 37, 101, 21, 85, 117, 53, 125, 61, 29, 93, 13, 77, 109, 45}
16	{127, 63, 31, 95, 15, 79, 111, 47, 7, 71, 39, 103, 119, 55, 23, 87}
8(x2)	{3, 67, 35, 99, 19, 83, 115, 51, 123, 59, 27, 91, 11, 75, 107, 43}

Table 4: Size-16 leaf nodes in VL-1 size-128 hard-coded leaf FFT

Table 4 lists the addresses of data loaded by each of the size-16 leaf nodes in a size-128 transform. It is difficult to improve the locality of accesses within a leaf sub-transform (doing so would require the use of expensive transposes), but the order of the leaf sub-transforms can be changed to yield better locality between sub-transforms.

Size	Input array addresses
16	{0, 64, 32, 96, 16, 80, 112, 48, 8, 72, 40, 104, 120, 56, 24, 88}
16	{1, 65, 33, 97, 17, 81, 113, 49, 9, 73, 41, 105, 121, 57, 25, 89}
16	{2, 66, 34, 98, 18, 82, 114, 50, 10, 74, 42, 106, 122, 58, 26, 90}
8(x2)	{3, 67, 35, 99, 19, 83, 115, 51, 123, 59, 27, 91, 11, 75, 107, 43}
8(x2)	{4, 68, 36, 100, 20, 84, 116, 52, 124, 60, 28, 92, 12, 76, 108, 44}
8(x2)	{5, 69, 37, 101, 21, 85, 117, 53, 125, 61, 29, 93, 13, 77, 109, 45}
16	{126, 62, 30, 94, 14, 78, 110, 46, 6, 70, 38, 102, 118, 54, 22, 86}
16	{127, 63, 31, 95, 15, 79, 111, 47, 7, 71, 39, 103, 119, 55, 23, 87}

Table 5: Sorted size-16 leaf nodes in VL-1 size-128 hard-coded leaf FFT

Table 5 is the list of nodes from Table 4 after the rows have been sorted according to the minimum address in each row. There are now three distinct groups in the list: the first three sub-transforms of size-16, the second three sub-transforms of 2x size-8, and the final two sub-transforms of size-16. The memory accesses are now linear between consecutive sub-transforms, though the second and third groups operate on a permuted ordering of the addresses.

The pattern exhibited by Table 5 can be exploited to access the data stored in the input array with better locality, as Figures Figure 3 and Figure 4 show. Figure 3 depicts the memory access pattern of an FFT with size-16 hard-coded leaves, while Figure 4 depicts the same FFT with *sorted* hard-coded leaves.

To compute the FFT with sorted leaves, the leaf sub-transforms and the body sub-transforms are split into two separate lists, and the entire list of leaf sub-transforms is computed before any of the body sub-transforms. There is, however, a cost associated with this re-arrangement: each leaf sub-transform's offset into the output array is not easy to compute because the offsets are now essentially decimated-in-frequency, and thus they are now pre-computed. Overall, the trade-off is justified because the output memory accesses within each leaf sub-transform are still linear.

Figure 3: Memory access pattern of the straight line blocks of code in a VL-1 size-128 hard-coded leaf FFT

Figure 4: Memory access pattern of the straight line blocks of code in a VL-1 size-128 hard-coded leaf FFT after leaf node sorting

The leaf transforms can be computed in three loops. The first and third loops compute size-N_{leaf} sub-transforms, while the second loop computes size-$N_{leaf}/2$ sub-transforms. The size of the three loops i_0, i_1 and i_2 are:

$$i_0 = \lfloor \frac{N}{3 \times N_{leaf}} \rfloor + 1 \tag{3}$$

$$i_1 = \lfloor \frac{N}{3 \times N_{leaf}} \rfloor + \lfloor \left(\frac{N}{N_{leaf}} \ mod\ 3 \right) \times \frac{1}{2} \rfloor \tag{4}$$

and

$$i_2 = \lfloor \frac{N}{3 \times N_{leaf}} \rfloor \tag{5}$$

The transform can now be elaborated *without* leaf nodes, and the code for the three loops emitted in the place of calls to individual leaf sub-transforms.

3.2.1 Example

p. **??** is the main function for the FFT that corresponds to the leaf node list in Table 5. The first and third loops invoke size-16 sub-transforms at lines 8 and 16, and the second loop invokes 2x size-8 sub-transforms at line 12. Following the leaf sub-transforms, the body sub-transforms are called at lines 19-23.

```
        void~sfft_dcf128_shl16_4(sfft_plan_t~*p,const~void~*vin,void~*vout){
~~const~SFFT_D~*in~=~vin;
~~SFFT_D~*out~=~vout;
~~offset_t~*is~=~p->is_base;
~~offset_t~*offsets~=~p->offsets_base;
~~int~i;
~~for(i=3;i>0;--i)~{
~~~~sfft_dcf128_shl16_4_e(is,~in,~out+offsets[0]);
~~~~is~+=~16;~offsets~+=~1;
~~}
~~for(i=3;i>0;--i)~{
~~~~sfft_dcf128_shl16_4_o(is,~in,~out+offsets[0]);
~~~~is~+=~16;~offsets~+=~1;
~~}
~~for(i=2;i>0;--i)~{
~~~~sfft_dcf128_shl16_4_e(is,~in,~out+offsets[0]);
~~~~is~+=~16;~offsets~+=~1;
~~}
~~sfft_dcf128_shl16_4_X_4(out+0,~32,~p->ws[0]);
~~sfft_dcf128_shl16_4_X_4(out+0,~64,~p->ws[1]);
~~sfft_dcf128_shl16_4_X_4(out+128,~32,~p->ws[0]);
~~sfft_dcf128_shl16_4_X_4(out+192,~32,~p->ws[0]);
~~sfft_dcf128_shl16_4_X_4(out+0,~128,~p->ws[2]);
}
```

Listing 9: Hard-coded VL-1 size-128 FFT with size-16 leaves (sub-transforms omitted)

3.2.2 Scalability

In terms of code size, computing the leaf sub-transforms with three loops is economical. As the size of the transform grows, the code size attributed to the leaf sub-transforms remains constant. However, as the size of the transform begins to grow large (e.g., $\geq 65,536$), the instructions required for the body sub-transform calls (lines 19-23 in p. **??**) begins to dominate the overall program size. "Optimizing the hierarchical structure" (Section 3.4: Optimizing the hierarchical structure) describes a method for compressing the code size of the body sub-transform calls while maintaining performance.

Because the input array references between consecutive leaves are now linear, and like types of leaf sub-transforms are grouped together, it is now possible to compute several leaf sub-transforms in parallel, which is fully described in "Other vector lengths" (Section 3.5: Other vector lengths).

3.3 Body sub-transform radix

The radix of the body sub-transforms can be increased in order to reduce the number of passes over the data and make better use of the cache. In practice, the body sub-transform radix is limited by the associativity of the cache as the size of the transform increases. If the radix is greater than the associativity of the nearest level of cache in which a sub-transform cannot fit, there will be cache misses for every iteration of the sub-transform's loop, resulting in severely degraded performance.

All Intel SIMD microprocessors since the Netburst micro-architecture have had at least 8-way associativity in all levels of cache, and thus increasing the radix from 4 to 8 is a sensible decision when targeting Intel machines.

Just as the split-radix 2/4 algorithm requires two different types of leaf sub-transforms, a split-radix 2/8 algorithm would require three, which increases the complexity of statically elaborating and generating code. There is an alternative that does not require implementing three types of leaf sub-transform: where a size-N body sub-transform divides into a size $N/2$ body sub-transform and two size $N/4$ sub-transforms, the size N and size $N/2$ sub-transforms may be collected together and computed as a size-8 sub-transform. Thus the transform is computed with two types of leaf sub-transform and two types of body sub-transform, instead of three types of leaf sub-transform and one type of body sub-transform, as with the standard split-radix 2/8 algorithm.

For the size-128 tranform in p. **??**, either the sub-transform at line 19 can be subsumed into the sub-transform at line 20, or the sub-transform at line 20 can be subsumed into the sub-transform at line 23 – but not both. The latter choice is better because it involves larger transforms.

```
        CBody~*CHardCodedLeaf::find_subsumable_sub_transform(
~~~~~~~vector<CNode~*>::reverse_iterator~i)~{
~~CBody~*first~=~(CBody~*)(*i);~~i++;
~~while(i~!=~bs.rend())~{
~~~~if(!((*i)->type().compare("body")))~{
~~~~~~CBody~*second~=~(CBody~*)(*i);
~~~~~~if(first->N~==~second->N*2~&&~first->offset~==~second->offset){
~~~~~~~~bs.erase((++i).base());
~~~~~~~~return~second;
~~~~~~}
~~~~}
~~~~++i;
~~}
~~return~NULL;
}
void~CHardCodedLeaf::increase_body_radix(void)~{
~~vector<CNode~*>::reverse_iterator~ri;
~~for(ri=bs.rbegin();~ri!=bs.rend();~++ri)~{
~~~~if(!((*ri)->type().compare("body")))~{
~~~~~~CBody~*n1~=~(CBody~*)(*ri);
~~~~~~CBody~*n2~=~find_subsumable_sub_transform(ri);
~~~~~~if(n2)~n1->size~*=~2;
~~~~}
~~}
}
```

Listing 10: Doubling the radix of body sub-transforms

The code in p. **??** iterates in reverse over a list of sub-transforms and doubles the radix of the body sub-transforms. Because the list may include multiple types, type introspection at lines 6 and 20 filters out all types that are not body sub-transforms. For each body sub-transform, the `increase_body_radix` function

searches upwards through the list for a subsumable body sub-transform (using `find_subsumable_sub_transform`) and if a match is found, the smaller sub-transform is removed from the list, and the size of the larger sub-transform is doubled.

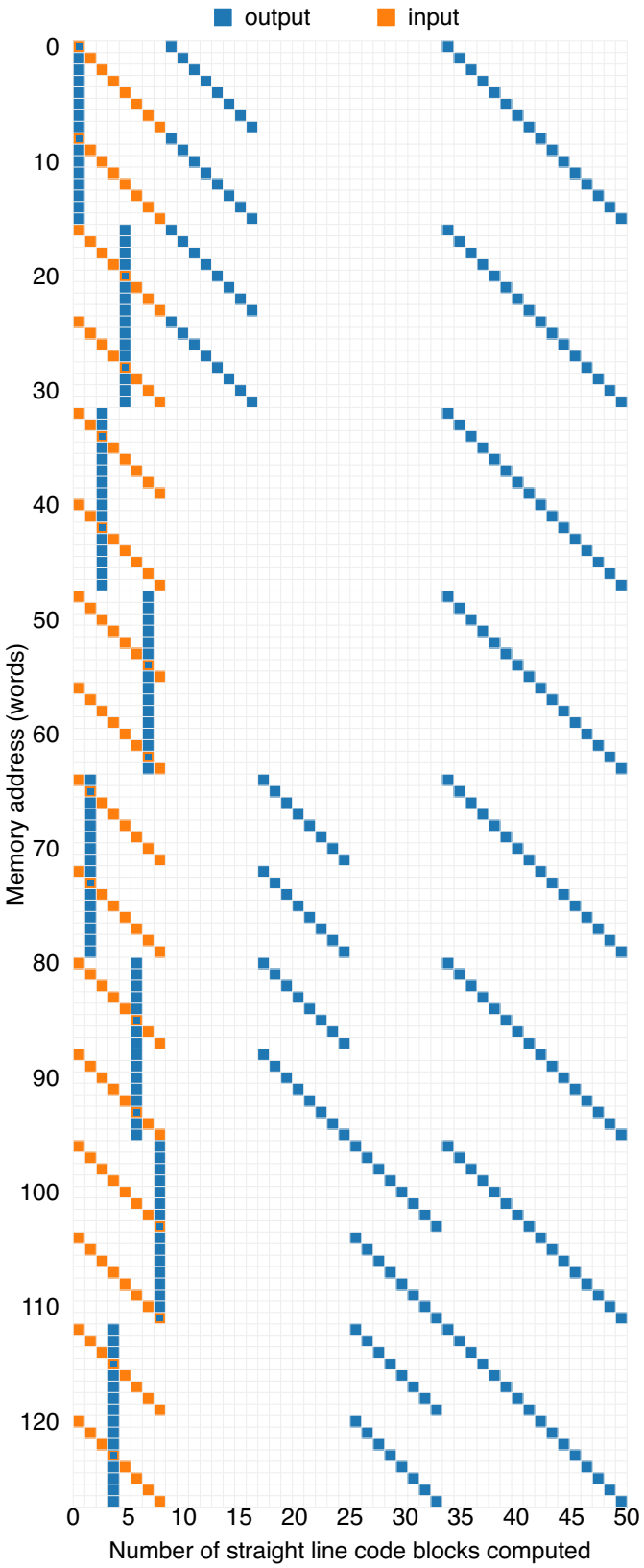

Figure 5: Memory access pattern of the straight line blocks of code in a VL-1 size-128 hard-coded leaf FFT with sorted radix-2/4 and size-8 body sub-transforms

Figure 5 depicts the memory access patterns of a size-128 transform where the outermost body sub-transform has subsumed a smaller sub-transform to become a size-8 sub-transform. The columns from 33 onwards show the sub-transform accessing eight elements in the output data array (cf. Figure 4, which shows the memory access patterns of the same transform prior to doubling the radix of the outer sub-transform).

3.4 Optimizing the hierarchical structure

The largest transform that has been considered so far is size-128. As it stands, the hard-coded leaf approach begins to generate code of unwieldy proportions as the size of the transform tends towards tens of thousands or hundreds of thousands of points. This is due to the lists of statically elaborated body sub-transform calls, e.g., a size-262,144 transform contains a lengthy list of 7279 such calls.

While long lists of statically elaborated calls are one extreme, the other is to compute the body sub-transforms with a recursive program. The former option degrades performance for larger transforms, while the latter option curbs performance for smaller transforms. A compromise is to somehow compress blocks of statically elaborated sub-transform calls.

The approach presented here extracts the hierarchical structure from the sequence of body sub-transforms and emits a set of functions that are neither too small (as in the case of a recursive program) nor too large (as is the case with full static elaboration). This is accomplished by adapting the Sequitur algorithm [3], which builds a grammar of rules from a sequence of symbols, and enforces two basic constraints:

1. no pair of adjacent symbols (referred to as a digram) appears more than once in the grammar;
2. every rule is used more than once.

The resulting grammar is an efficient hierarchical representation of the original sequence. Additional constraints can be imposed to limit the maximum or minimum size of each rule, which enable the size of the resulting functions to be tuned to be not too small and not too large.

To build the grammar, each body sub-transform is represented by a symbol consisting of the size and offset of the sub-transform. The radix is discarded, because it can be inferred from the size. Here are several other details relevant to this particular application of Sequitur:

- A digram of two sub-transforms is deemed to match another digram when the size of each sub-transform matches the size of the other digram's respective sub-transform *and* the relative offsets between sub-transforms within each digram match;
- Sub-transform offsets are maintained to be always relative to the base of the containing rule – when a rule is constructed, the offsets of the symbols within that rule are adjusted to be relative to the base of the new rule, and when a rule is subsumed (due to violation of constraint 2: every rule must be used more than once), the offsets are recomputed to be relative to the subsuming rule.

```
        void~sfft_dcf8192_shl16_8_4(sfft_plan_t~*p,~SFFT_D~*out)~{
~~X_4(out+0,~32,~p->ws[0]);
~~X_4(out+128,~32,~p->ws[0]);
~~X_4(out+192,~32,~p->ws[0]);
~~X_8(out+0,~128,~p->ws[2]);
}
void~sfft_dcf8192_shl16_8_5(sfft_plan_t~*p,~SFFT_D~*out)~{
~~X_8(out+0,~64,~p->ws[1]);
~~X_8(out+128,~64,~p->ws[1]);
}
void~sfft_dcf8192_shl16_8_9(sfft_plan_t~*p,~SFFT_D~*out)~{
~~X_8(out+0,~64,~p->ws[1]);
~~X_4(out+128,~32,~p->ws[0]);
~~X_4(out+192,~32,~p->ws[0]);
~~sfft_dcf8192_shl16_8_5(p,~out+256);
~~X_8(out+0,~256,~p->ws[3]);
}
void~sfft_dcf8192_shl16_8_13(sfft_plan_t~*p,~SFFT_D~*out)~{
~~sfft_dcf8192_shl16_8_4(p,~out+0);
~~sfft_dcf8192_shl16_8_5(p,~out+256);
~~sfft_dcf8192_shl16_8_4(p,~out+512);
~~sfft_dcf8192_shl16_8_4(p,~out+768);
~~X_8(out+0,~512,~p->ws[4]);
}
void~sfft_dcf8192_shl16_8_14(sfft_plan_t~*p,~SFFT_D~*out)~{
~~sfft_dcf8192_shl16_8_9(p,~out+0);
~~sfft_dcf8192_shl16_8_9(p,~out+512);
}
void~sfft_dcf8192_shl16_8_18(sfft_plan_t~*p,~SFFT_D~*out)~{
~~sfft_dcf8192_shl16_8_9(p,~out+0);
~~sfft_dcf8192_shl16_8_4(p,~out+512);
~~sfft_dcf8192_shl16_8_4(p,~out+768);
~~sfft_dcf8192_shl16_8_14(p,~out+1024);
~~X_8(out+0,~1024,~p->ws[5]);
}
void~sfft_dcf8192_shl16_8_22(sfft_plan_t~*p,~SFFT_D~*out)~{
~~sfft_dcf8192_shl16_8_13(p,~out+0);
~~sfft_dcf8192_shl16_8_14(p,~out+1024);
~~sfft_dcf8192_shl16_8_13(p,~out+2048);
~~sfft_dcf8192_shl16_8_13(p,~out+3072);
~~X_8(out+0,~2048,~p->ws[6]);
}
void~sfft_dcf8192_shl16_8_1(sfft_plan_t~*p,~SFFT_D~*out)~{
~~sfft_dcf8192_shl16_8_22(p,~out+0);
~~sfft_dcf8192_shl16_8_18(p,~out+4096);
~~sfft_dcf8192_shl16_8_18(p,~out+6144);
~~sfft_dcf8192_shl16_8_22(p,~out+8192);
~~sfft_dcf8192_shl16_8_22(p,~out+12288);
~~X_8(out+0,~8192,~p->ws[8]);
}
```

Listing 11: Optimized body sub-transforms for size-8192 FFT

3.4.1 Example

A size-8192 hard-coded leaf FFT requires 229 calls to radix-2/4 and size-8 body sub-transforms. After optimizing the sequence of calls with Sequitur, the compact set of functions shown in p. **??** replaces a sequence of 229 calls.

Compared to the full list of statically elaborated calls, the optimized set of functions requires less code space while achieving better performance; and compared to a recursive program, the optimized set of function calls is faster (due to lower call and stack overhead) while trading off an acceptably small amount of code space.

3.4.2 Scalability

The technique presented in this section has been verified for transforms ranging in size from 2^6 through to 2^{25} (32 mega) points. The technique works well up until sizes of about 2^{18} points, but for larger transforms the elaboration and compile times begin to exceed 1 second or so, and the code size again begins to grow large. For transforms larger than 2^{18} points, a recursive program can be used until leaves of size 2^{18} points are reached, at which point the technique presented in this section is used.

3.5 Other vector lengths

The method of vectorizing the hard-coded leaf FFT is similar to that of the hard-coded FFT in "Other vector lengths" (Section 1.2: Other vector lengths); the only difference here is the level of scale.

The hard-coded FFT was vectorized by collecting together primitive leaf operations that loaded data from adjacent memory locations. The hard-coded leaf FFT has already been sorted such that consecutive leaf sub-transforms load data from adjacent memory locations (see "Improving memory locality in the leaves" (Section 3.2: Improving memory locality in the leaves)), so the task is easier in this case – at least in one respect.

Size	Input array addresses
16	{0, 64, 32, 96, 16, 80, 112, 48, 8, 72, 40, 104, 120, 56, 24, 88}
16	{1, 65, 33, 97, 17, 81, 113, 49, 9, 73, 41, 105, 121, 57, 25, 89}
16	{2, 66, 34, 98, 18, 82, 114, 50, 10, 74, 42, 106, 122, 58, 26, 90}
8(x2)	{3, 67, 35, 99, 19, 83, 115, 51, 123, 59, 27, 91, 11, 75, 107, 43}
8(x2)	{4, 68, 36, 100, 20, 84, 116, 52, 124, 60, 28, 92, 12, 76, 108, 44}
8(x2)	{5, 69, 37, 101, 21, 85, 117, 53, 125, 61, 29, 93, 13, 77, 109, 45}
16	{126, 62, 30, 94, 14, 78, 110, 46, 6, 70, 38, 102, 118, 54, 22, 86}
16	{127, 63, 31, 95, 15, 79, 111, 47, 7, 71, 39, 103, 119, 55, 23, 87}

Table 6: Sorted size-16 leaf nodes in size-128 hard-coded leaf FFT, grouped for VL-2

Table 6 shows the sorted size-16 leaf sub-transforms for a size-128 transform with the rows divided into VL-2 groups. Because each group of two leaf sub-transforms loads data from adjacent memory locations, the group of sub-transforms can be loaded in parallel with vector memory operations, and all (or some) of the computation done in parallel. The first, third and fourth groups in Table 6 contain leaf nodes of the same size/type; these are the easiest vector leaf sub-transforms to compute, as described in "Homogeneous leaf sub-transform vectors" (Section 3.5.1: Homogeneous leaf sub-transform vectors). The second group of rows contains leaf sub-transforms of differing size/type, and computing these sub-transforms is covered separately in "Heterogeneous leaf sub-transform vectors" (Section 3.5.2: Heterogeneous leaf sub-transform vectors).

3.5.1 Homogeneous leaf sub-transform vectors

```
        void
sfft_fcf128_shl16_8_ee(offset_t~*is,const~SFFT_D~*in,SFFT_D~*out){
~~SFFT_R~r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15;
~~L_4(in+is[0],in+is[1],in+is[2],in+is[3],&r0,&r1,&r2,&r3);
~~L_2(in+is[4],in+is[5],in+is[6],in+is[7],&r4,&r5,&r6,&r7);
~~K_0(&r0,&r2,&r4,&r6);
~~K_N(VLIT4(0.7071,0.7071,0.7071,0.7071),
~~~~~~VLIT4(0.7071,-0.7071,0.7071,-0.7071),
~~~~~~&r1,&r3,&r5,&r7);
~~L_4(in+is[8],in+is[9],in+is[10],in+is[11],&r8,&r9,&r10,&r11);
~~L_4(in+is[12],in+is[13],in+is[14],in+is[15],&r12,&r13,&r14,&r15);
~~K_0(&r0,&r4,&r8,&r12);
~~K_N(VLIT4(0.9239,0.9239,0.9239,0.9239),
~~~~~~VLIT4(0.3827,-0.3827,0.3827,-0.3827),
~~~~~~&r1,&r5,&r9,&r13);
~~TX2(&r0,&r1);~TX2(&r4,&r5);~TX2(&r8,&r9);~TX2(&r12,&r13);
~~S_4(r0,r4,r8,r12,out0+0,out0+8,out0+16,out0+24);
~~S_4(r1,r5,r9,r13,out1+0,out1+8,out1+16,out1+24);
~~K_N(VLIT4(0.7071,0.7071,0.7071,0.7071),
~~~~~~VLIT4(0.7071,-0.7071,0.7071,-0.7071),
~~~~~~&r2,&r6,&r10,&r14);
~~K_N(VLIT4(0.3827,0.3827,0.3827,0.3827),
~~~~~~VLIT4(0.9239,-0.9239,0.9239,-0.9239),
~~~~~~&r3,&r7,&r11,&r15);
~~TX2(&r2,&r3);~TX2(&r6,&r7);~TX2(&r10,&r11);~TX2(&r14,&r15);
~~S_4(r2,r6,r10,r14,out0+4,out0+12,out0+20,out0+28);
~~S_4(r3,r7,r11,r15,out1+4,out1+12,out1+20,out1+28);
}
```

Listing 12: Homogeneous size-16 leaf sub-transform for VL-2 size-128 hard-coded leaf FFT

The vector leaf sub-transforms of a single size/type are handled in the same way as a VL-1 sub-transform, with one difference: the vector registers must be transposed before the data is stored to memory in the output array. In the example shown in p. **??**, the transposes take place at lines 16 and 25.

Prior to the store operations, each position of the vector register (each position being a whole complex word) contains an element belonging to each of the leaf sub-transforms composing the vectorized sub-transform. Because each leaf sub-transform is stored sequentially to different locations in memory with aligned vector store operations, sets of registers are transposed such that each vector register contains elements from only one leaf sub-transform.

3.5.2 Heterogeneous leaf sub-transform vectors

```
        void
sfft_fcf128_shl16_8_eo(offset_t~*is,const~SFFT_D~*in,SFFT_D~*out){
~~SFFT_R~r0_1,r2_3,r4_5,r6_7,r8_9,r10_11,r12_13,r14_15,
r16_17,r18_19,r20_21,r22_23,r24_25,r26_27,r28_29,r30_31;
~~L_4_4(in+is[0],in+is[1],in+is[2],in+is[3],
~~~~~~&r0_1,&r2_3,&r16_17,&r18_19);
~~L_2_2(in+is[4],in+is[5],in+is[6],in+is[7],
~~~~~~&r4_5,&r6_7,&r20_21,&r22_23);
~~K_N(VLIT4(0.7071,0.7071,1,1),VLIT4(0.7071,-0.7071,0,-0),
~~~~~~&r0_1,&r2_3,&r4_5,&r6_7);
~~L_4_2(in+is[8],in+is[9],in+is[10],in+is[11],
~~~~~~~~&r8_9,&r10_11,&r28_29,&r30_31);
~~L_4_4(in+is[12],in+is[13],in+is[14],in+is[15],
~~~~~~~~&r12_13,&r14_15,&r24_25,&r26_27);
~~K_N(VLIT4(0.9239,0.9239,1,1),VLIT4(0.3827,-0.3827,0,-0),
~~~~~~&r0_1,&r4_5,&r8_9,&r12_13);
~~S_4(r0_1,r4_5,r8_9,r12_13,out0+0,out0+8,out0+16,out0+24);
~~K_N(VLIT4(0.3827,0.3827,0.7071,0.7071),
~~~~~~VLIT4(0.9239,-0.9239,0.7071,-0.7071),
~~~~~~&r2_3,&r6_7,&r10_11,&r14_15);
~~S_4(r2_3,r6_7,r10_11,r14_15,out0+4,out0+12,out0+20,out0+28);
~~K_N(VLIT4(0.7071,0.7071,1,1),VLIT4(0.7071,-0.7071,0,-0),
~~~~~~&r16_17,&r18_19,&r20_21,&r22_23);
~~S_4(r16_17,r18_19,r20_21,r22_23,out1+0,out1+4,out1+8,out1+12);
~~K_N(VLIT4(0.7071,0.7071,1,1),VLIT4(0.7071,-0.7071,0,-0),
~~~~~~&r24_25,&r26_27,&r28_29,&r30_31);
~~S_4(r24_25,r26_27,r28_29,r30_31,out1+16,out1+20,out1+24,out1+28);
}
```

Listing 13: Heterogeneous size-16 leaf sub-transform for VL-2 size-128 hard-coded leaf FFT

In the case of a vector comprising heterogeneous leaf sub-transforms, the data is transposed into separate sub-transforms following the primitive leaf operations. The remainder of the computation is carried out separately for each leaf sub-transform in the vector, and no further transposes are required.

When elaborating and generating code for VL-2 transforms, there are only two heterogeneous leaf sub-transforms that might be required, but for other vector lengths the combinations are more complex. During the elaboration process, each unique combination that is encountered in the sorted list of leaf sub-transforms is elaborated into a function with repeated calls to the `elaborate` function, as was done in "Vector length 1" (Section 3.1: Vector length 1) in order to elaborate a sub-transform composed of two size $N_{leaf}/2$ sub-transforms.

p. **??** is an example of a heterogeneous size-16 VL-2 leaf sub-transform, where one size-16 leaf sub-transform is loaded into the lower halves of the vector registers, and the data from another leaf sub-transform composed of two size-8 sub-transforms is loaded into the upper halves. The primitive leaf operations at lines 5, 7, 11 and 13 transpose each sub-transform's data into separate vector registers, and the remainder of the

computation is performed on each sub-transform separately. The size-16 sub-transform is stored to sequential locations in memory at lines 17 and 21, while the sub-transform composed of two size-8 leaf sub-transforms is stored to memory at lines 24 and 27.

3.6 Streaming stores

Some machines support streaming store or non-temporal store instructions; these instructions are used to store data to locations that do not have temporal locality, and thus the cache can be bypassed. The hard-coded leaf FFT described in the previous sections splits the computation into a pass of leaf sub-transforms and several passes of body sub-transforms. For large transforms where the size of the data exceeds the outermost level of cache, the non-temporal store instructions can be used in the leaf sub-transforms to bypass the cache when storing data to the output array; this can greatly improve performance by keeping other data in cache. The Intel SSE and AVX vector extensions both support streaming stores.

3.7 Performance

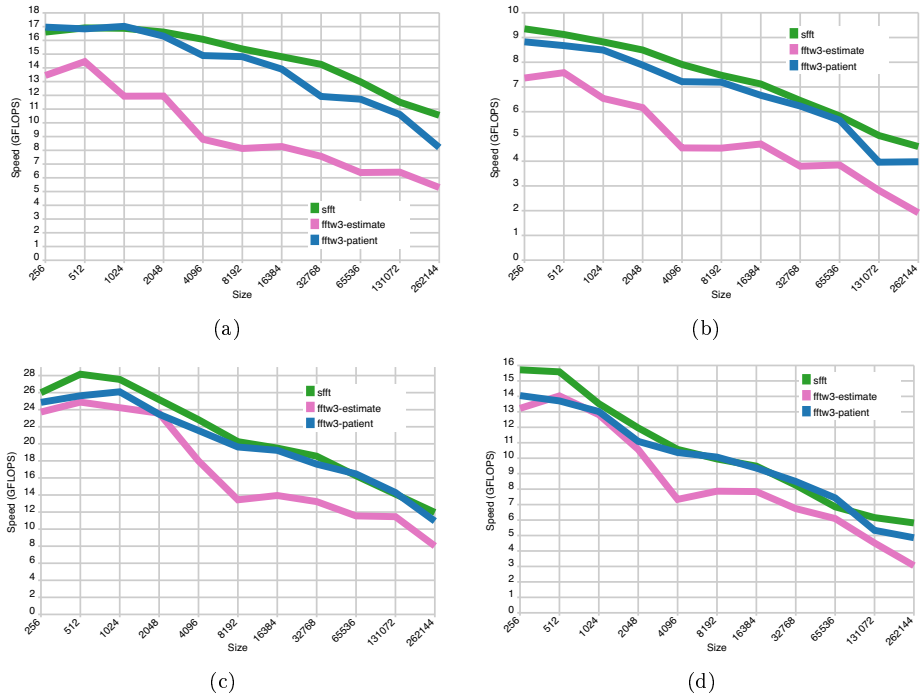

Figure 6: Performance of hard-coded leaf FFTs on a Macbook Air 4,2. (a) Single-precision, SSE (VL-2) (b) Double-precision, SSE (VL-1) (c) Single-precision, AVX (VL-4) (d) Double-precision, AVX (VL-2)

Figure 6 shows the results of a benchmark for transforms of size 256 through to 262,144 running on a Macbook Air 4,2. The speed of FFTW 3.3 running in estimate and patient modes is also shown for comparison.

For each size of transform, precision and vector length (i.e., either SSE or AVX), several configurations of hard-coded leaf FFT were generated: three configurations of leaf size (16, 32 and 64), and if the transform was larger than 32,768, an additional transform with size-16 leaves and streaming store instructions was also

generated. Before running the benchmark, the library was calibrated and the fastest configuration selected (details of the calibration are described in "Calibration" (Section 4.1.3: Calibration)).

For most sizes of transform, precision and vector length, SFFT is faster than FFTW running in patient mode. For the transforms with memory requirements that are approximately at the limits of the cache, FFTW running in patient mode is sometimes marginally faster than SFFT. Once the transforms exceed the size of the cache, SFFT is again the fastest.

It is important to note that FFTW running in patient mode evaluates a huge configuration space of parameters (and thus takes a long time to calibrate), while SFFT has, in this case, only evaluated either three or four configurations per transform.

4 In practice

SFFT is not itself an FFT library; the name refers to the elaboration program that reads a configuration file and generates the code for an FFT library. The code for the FFT library is then built as any other library would be.

4.1 Organization

As well as the generated code, there is infrastructure code which is common to all libraries generated by SFFT. This can be broadly categorized into three parts: initialization, dispatch and calibration.

4.1.1 Initialization

Before an application can compute an FFT with SFFT, it must initialize a plan for the specific size, precision and direction of FFT. The library may have several FFTs and configurations that can compute the requested FFT, and it chooses the fastest option by timing each of the candidate configurations, which is at most 8 for any size of transform – a very small space compared to FFTW's exhaustive search of all possible FFT algorithms and configurations. Results and discussion describes an alternative to calibration, where machine learning is used with data collected from benchmarks to build a model that predicts performance.

After determining which implementation and parameters will be used, the initialization code allocates memory and populates any lookup tables that may be required. Before returning the plan to the application, a function pointer in the plan is updated to point to the FFT that has just been initialized.

4.1.2 Dispatch

Applications do not invoke any of the FFTs within SFFT directly. Rather they invoke a dispatch function on an initialized plan, which in turn transfers control to the correct FFT code within SFFT. The use of a dispatch function is purely a matter of convenience, so that users only need to deal with a few simple functions.

4.1.3 Calibration

SFFT contains calibration code to measure the performance of the possible configurations of FFT on the target machine, which is at most 8 for each size of transform. Following calibration, the timing data is written to a file, which is then used by SFFT to select the fastest possible FFT for a given problem running on that machine.

4.2 Usage

SFFT is used much like other FFT libraries:

1. A plan for an FFT is initialized;
2. Using the plan, an FFT is computed (this step may be repeated many times);

3. The plan is destroyed.

The plan is initialized for a given size, precision and direction of transform, and may then be executed any number of times on any data. Any number of plans can be simultaneously created and used.

```
        ~~int~n~=~1024;
~~double~complex~__attribute__~((aligned(32)))~*input,~*output;
~~input~=~_mm_malloc(n~*~sizeof(double~complex),~32);
~~output~=~_mm_malloc(n~*~sizeof(double~complex),~32);
~
~~for(i=0;i<n;i++)~input[i]~=~i;
~
~~sfft_plan_t~*p~=~sfft_init(i,~SFFT_FORWARD|SFFT_DOUBLE|SFFT_AVX);
~
~~if(p)~{
~
~~~~sfft_execute(p,~input,~output);
~~~~for(i=0;i<n;i++)
~~~~~~printf("%d~%f~%f\n",~i,~creal(output[i]),~cimag(output[i]));
~~~~sfft_free(p);
~
~~}else{
~~~~printf("Plan~unsupported\n");
~~}
```

Listing 14: SFFT example usage

In p. **??**, a size-1024 transform is computed on double-precision data with AVX enabled. In lines 2-4, the input and output arrays are allocated with 32 byte alignment, as is required for aligned AVX memory operations. The plan is initialized at line 8, used to compute an FFT at line 12 (provided the requested plan is supported), and finally freed at line 20.

4.3 Other optimizations

In addition to generating a general-purpose library that can be calibrated for a machine and application at runtime, there are several situations where the SFFT library can be specially optimized:

1. If the machine and application are fixed, a one time calibration can be performed and an optimized library containing only the fastest transforms specific to the application and machine is generated;
2. If the application is fixed, an optimized library containing only the transforms specific to the application is generated (and the library is calibrated the first time it is used on each machine);
3. If the machine is fixed, an optimized library containing only the transforms specific to the machine is generated (and an application can use any transform without calibration).

References

[1] D.H. Bailey. Ffts in external or hierarchical memory. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, page 2348211;242. ACM, 1989.

[2] J. Lorenz, S. Kral, F. Franchetti, and C.W. Ueberhuber. Vectorization techniques for the blue gene/l double fpu. *IBM Journal of Research and Development*, 49(2.3):4378211;446, 2005.

[3] C.G. Nevill-Manning and I.H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:678211;82, 1997.