FFTE on SVE: SPIRAL-Generated Kernels

Daisuke Takahashi
Center for Computational Sciences
University of Tsukuba
Tsukuba, Ibaraki, Japan
daisuke@cs.tsukuba.ac.jp

Franz Franchetti
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA, USA
franzf@andrew.cmu.edu

ABSTRACT

In this paper we propose an implementation of the fast Fourier transform (FFT) targeting the ARM Scalable Vector Extension (SVE). We performed automatic vectorization via a compiler and an explicit vectorization through code generation by SPIRAL for FFT kernels, and compared the performance. We show that the explicit vectorization of SPIRAL generated code improves performance significantly. Performance results of FFTs on RIKEN's Fugaku processor simulator are reported. With the ARM compiler SPIRAL-generated FFT kernels written in SVE intrinsic are up to 3.16 times faster than FFT kernels of FFTE written in Fortran and up to 5.62 times faster than SPIRAL-generated FFT kernels written in C.

CCS CONCEPTS

• **Mathematics of computing → Mathematical software performance**; • **Software and its engineering → Source code generation**.

KEYWORDS

FFT, ARM SVE, SPIRAL, vectorization

ACM Reference Format:
Daisuke Takahashi and Franz Franchetti. 2020. FFTE on SVE: SPIRAL-Generated Kernels. In *International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia2020), January 15–17, 2020, Fukuoka, Japan.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3368474.3368488

1 INTRODUCTION

The fast Fourier transform (FFT) [6] is widely used in science and engineering. Nowadays a number of processors have short vector SIMD instructions. These instructions provide substantial speedup for digital signal processing applications.

Recently, ARM introduced the Scalable Vector Extension (SVE) [20] as a vector extension to the Armv8 architecture. Fujitsu's A64FX [29] is the first processor implementing the Armv8-A SVE architecture, which is used in the supercomputer Fugaku.

Related work. Efficient FFT implementations with short vector SIMD instructions have also been investigated thoroughly [8, 14, 18, 19, 22]. SPIRAL [8, 18, 19] supports automatic SIMD vectorization of FFTs for a wide range of SIMD vector ISAs including the Intel SSE, AVX and AVX-512 instruction sets. FFTW [14] supports the Intel AVX-512 instructions [15]. To best of our knowledge, an implementation of the FFT on ARM SVE has not yet been reported.

FFTE [24] is a Fortran subroutine library for computing the FFT in one or more dimensions. It includes real, complex, mixed-radix, and parallel transforms. In this paper, we compare the performance of the original FFT kernels of FFTE to FFTE with FFT kernels generated by SPIRAL.

Synopsis. The remainder of this paper is organized as follows. Section 2 explains the ARM scalable vector extension. Section 3 describes the SPIRAL code generator. Section 4 presents a vectorization of FFT kernels. The performance results are then presented in Section 5. Finally, Section 6 presents some concluding remarks.

2 ARM SCALABLE VECTOR EXTENSION

The ARM Scalable Vector Extension (SVE) [20] is a vector extension for the Armv8 architectures. It allows implementations to choose a vector register length between 128 and 2,048 bits and supports a Vector Length Agnostic (VLA) programming model. The actual SIMD width depends on the processor that implements SVE, and almost all instructions are provided as masked instructions by via predicate to realize VLA operations. SVE introduces 32 scalable vector registers (Z0 to Z31) and 16 scalable predicate registers (P0 to P15).

ARM defines the ARM C language extensions (ACLE) [1] to target SVE in the C/C++ language. ACLE is supported by the ARM C/C++ compiler and Fujitsu C compiler. Using `daxpy` as an example, ARM explains the outline of using ACLE in computing kernels.

`DAXPY` is one of the BLAS (Basic Linear Algebra Subroutines) subroutines that calculates "constant a times a vector x plus a vector y" in double-precision. The number of elements in input vectors x and y is n. A C code implementation of a unit-stride `DAXPY` is shown in Listing 1.

Listing 1: A C code implementation of daxpy [2]

```
1 void daxpy_1_1(int64_t n, double da, double *dx,
2                double *dy)
3 {
4   for (int64_t i = 0; i < n; ++i) {
5     dy[i] = dx[i] * da + dy[i];
6   }
7 }
```

An ACLE implementation of `DAXPY` is shown in Listing 2.

HPCAsia2020, January 15–17, 2020, Fukuoka, Japan

ACM ISBN 978-1-4503-7236-7/20/01…$15.00
https://doi.org/10.1145/3368474.3368488

Listing 2: An ACLE implementation of daxpy [2]

```
1  void daxpy_1_1(int64_t n, double da, double *dx,
2                 double *dy)
3  {
4    int64_t i = 0;
5    svbool_t pg = svwhilelt_b64(i, n);
6    do {
7      svfloat64_t dx_vec = svld1(pg, &dx[i]);
8      svfloat64_t dy_vec = svld1(pg, &dy[i]);
9      svst1(pg, &dy[i],
10           svmla_x(pg, dy_vec, dx_vec, da));
11     i += svcntd();
12     pg = svwhilelt_b64(i, n);
13   } while (svptest_any(svptrue_b64(), pg));
14 }
```

The following steps explain this example:

Line 5: The active elements are selected with the `svwhilelt_b64(i, n)` function and saved in the predicate register `pg`. The predicate register `pg` is specified for vector instructions to mask the result of the vector operation.

Line 7: Load one vector register from the start address specified in the second argument using `svld1` function and save it in `svfloat64_t dx_vec`.

Line 8: Load one vector register from the start address specified in the second argument using `svld1` function and save it in `svfloat64_t dy_vec`.

Lines 9–10: Perform a floating-point multiply-add operation `da * dx_vec + dy_vec` using `svmla_x` function, and store the result in array `&dy[i]` using `svst1` function.

Line 11: `svcntd()` is the value obtained by dividing the SIMD width of the SVE processor by 64 bits.

Line 12: The active elements are selected with the `svwhilelt_b64(i, n)` function and save in the predicate register `pg`.

Line 13: Determine whether there is at least one active element in predicate register `pg` and continues the loop until all elements have been computed.

3 SPIRAL CODE GENERATOR

Overview. The Spiral system [8, 19] is a program and library generation/synthesis and autotuning system that translates rule-encoded high-level specifications of mathematical algorithms into highly optimized/library-grade implementations for a large set of computational kernels and platforms. The system is available as open source under a permissible (BSD style) license at `www.spiral.net`. It has been developed over the last 20 years. Spiral formalizes a selection of computational kernels from the signal and image processing domain, software-defined radio, numerical solution of partial differential equations, graph algorithms, and robotic vehicle control, among others. The historical core of Spiral is the discrete Fourier transform and it's fast algorithms, the FFTs [19]. Spiral targets platforms spanning from mobile devices, to desktop and server multicore processors, and to large high performance and supercomputing systems, and it has demonstrated performance comparable to expertly hand tuned code across a variety of kernels and diversity of platforms. To maximize portability and to leverage the work of backend compiler developers, when producing software Spiral usually targets vendor compilers or widely available compilers like the GNU C compiler.

Vector instructions in Spiral. A particular focus of Spiral is SIMD vector instructions and multicore processors [7]. SIMD vectorization in Spiral has a long history across a wide range of platforms [9, 10, 13]. In the space of supercomputing, Spiral was used in a winning HPC Challenge submission [12] that run a Spiral-generated Global FFT benchmark on 128k cores, and Spiral is currently being retargeted for the Summit supercomputer in the context of FFTX [11]. Spiral was also ported to the K computer.

Formal framework. Spiral represents the DFT and FFT algorithms using the Kronecker product formalism summarized in [16, 26, 27].

The DFT of n input samples $x_0, \ldots, x_{n-1}$ is defined in summation form as

$$y_k = \sum_{0 \le \ell < n} \omega_n^{k\ell} x_\ell, \quad 0 \le k < n, \tag{1}$$

with $\omega_n = \exp(-2\pi j/n)$. Stacking the x_ℓ and y_k into vectors x and y yields the equivalent form of a matrix-vector product:

$$y = \mathrm{DFT}_n\, x, \quad \mathrm{DFT}_n = [\omega_n^{k\ell}]_{0 \le k,\ell < n}. \tag{2}$$

We use a *point-free* notation where we drop x and y and simply think of the matrix DFT_n as the transform, implicitly assuming that it is multiplied to x. Fast algorithms are now expressed as factorizations of DFT_n using the following formalism.

We denote with I_n the $n \times n$ identity matrix, and the *butterfly matrix* with

$$\mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{3}$$

The *Kronecker product* of matrices A and B is defined as

$$A \otimes B = [a_{k,\ell} B], \quad \text{for } A = [a_{k,\ell}].$$

It replaces every entry $a_{k,\ell}$ of A by the matrix $a_{k,\ell} B$. Most important are the cases where A or B is the identity. The *stride permutation* matrix L^{mn}_m permutes the elements of the input vector as $in + j \mapsto jm + i$, $0 \le i < m$, $0 \le j < n$. If the vector x is viewed as an $n \times m$ matrix, stored in row-major order, then L^{mn}_m performs a transposition of this matrix.

The matrix formalism can be used to express FFTs as factorizations of the matrix DFT_n in (2). As an example, the recursive general-radix decimation in time (DIT) Cooley-Tukey FFT for $n = km$ is

$$\mathrm{DFT}_{km} = (\mathrm{DFT}_k \otimes I_m) T^n_m (I_k \otimes \mathrm{DFT}_m) L^n_k. \tag{4}$$

Here, T^n_m is a diagonal matrix containing the *twiddle factors*.

Simple transposition using that $\mathrm{DFT}_n^\top = \mathrm{DFT}_n$ and applying rules from [16] yields three more variants of the Cooley-Tukey FFT algorithm,

$$\mathrm{DFT}_{km} = L^n_m (I_k \otimes \mathrm{DFT}_m) T^n_m (\mathrm{DFT}_k \otimes I_m) \tag{5}$$

$$\mathrm{DFT}_{km} = (\mathrm{DFT}_k \otimes I_m) T^n_m L^n_k (\mathrm{DFT}_m \otimes I_k) \tag{6}$$

$$\mathrm{DFT}_{km} = L^n_k (I_m \otimes \mathrm{DFT}_k) L^n_m T^n_m (I_k \otimes \mathrm{DFT}_m) L^n_k, \tag{7}$$

which correspond to the decimation in frequency (DIF) Cooley-Tukey FFT (5), the Four Step algorithm (6), and the Six Step algorithm (7).

Matrix formula	Matlab pseudo code
$y = (A_n B_n)x$	`t[0:1:n-1] = B(x[0:1:n-1]);` `y[0:1:n-1] = A(t[0:1:n-1]);`
$y = (I_m \otimes A_n)x$	`for (i=0; i<m; i++)` `  y[i*n:1:i*n+n-1] = A(x[i*n:1:i*n+n-1]);`
$y = (A_m \otimes I_n)x$	`for (i=0; i<n; i++)` `  y[i:n:i+m*n-n] = A(x[i:n:i+m*n-n]);`
$y = D_n x$	`for (i=0; i<n; i++)` `  y[i] = Dn[i]*x[i];`

$y = L^{mn}_m x$	`for (i=0; i<m; i++)`
	`  for (j=0; j<n; j++)`
	`  y[i+m*j] = x[n*i+j];`

Table 1: From matrix formulas to code. The subscript of A, B specifies the (square) matrix size. `x[b:s:e]` denotes the sub-vector of x starting at `b`, ending at `e`, extracted at stride `s`. The diagonal elements of D are stored in an array with the same name.

The prime-factor and Rader FFT, respectively, can be written as the following structured factorizations of the DFT matrix [27].

$$\mathrm{DFT}_n = V_n^{-1}(\mathrm{DFT}_k \otimes I_m)(I_k \otimes \mathrm{DFT}_m)V_n, \quad (8)$$

$$\mathrm{DFT}_n = W_n^{-1}(I_1 \oplus \mathrm{DFT}_{p-1})E_n(I_1 \oplus \mathrm{DFT}_{p-1})W_n, \quad (9)$$

Here, V, W are the permutation matrices and E is "almost" diagonal with 2 additional off-diagonal entries.

Code generation and tuning. Spiral explores the recursive expansions of an DFT of size n with breakdown rules (4)–(9) to derive algorithm variants to implement FFTs of size n. The choice of breakdown can be driven by models, heuristics, or performance feedback (automatic performance tuning). Table 1 shows how to translate matrix formulas into basic sequential loop code. For small input sizes ≤ 16 as used in this paper matrix formulas are often implemented as fully unrolled code blocks. In this case array scalarization and, to a lesser extent, algebraic optimizations and scheduling are used to achieve best performance and can be completely automated [19, 28]. For example, a DFT_8 kernel is implemented in a few tens of lines of code.

4 VECTORIZATION OF FFT KERNELS

Listing 3: Example of a vectorizable radix-2 FFT kernel [23]

```
1       SUBROUTINE FFT(A,B,W,M,L)
2       COMPLEX*16 A(M,L,*),B(M,2,*),W(*)
3       COMPLEX*16 C0,C1
4       DO J=1,L
5         DO I=1,M
6           C0=A(I,J,1)
7           C1=A(I,J,2)
8           B(I,1,J)=C0+C1
9           B(I,2,J)=W(J)*(C0-C1)
10        END DO
11      END DO
12      RETURN
13      END
```

Listing 4: First stage of a vectorizable radix-2 FFT kernel [23]

```
1       SUBROUTINE FFT1ST(A,B,W,L)
2       COMPLEX*16 A(L,*),B(2,*),W(*)
3       COMPLEX*16 C0,C1
4       DO J=1,L
5         C0=A(J,1)
6         C1=A(J,2)
7         B(1,J)=C0+C1
8         B(2,J)=W(J)*(C0-C1)
9       END DO
10      RETURN
11      END
```

Listing 5: Middle stage of a vectorizable radix-2 FFT kernel

```
1       SUBROUTINE FFTMDL(A,B,M,L)
2       COMPLEX*16 A(M,L,*),B(M,2,*)
3       COMPLEX*16 C0,C1
4       DO I=1,M
5         C0=A(I,J,1)
6         C1=A(I,J,2)
7         B(I,1,J)=C0+C1
8         B(I,2,J)=C0-C1
9       END DO
10      RETURN
11      END
```

Vectorization approach. An example of a vectorizable radix-2 FFT kernel is shown in Listing 3. In the program shown in Listing 3, arrays `A` and `B` are the input array and the output array, respectively. The twiddle factors [4] are stored in array `W`. Twiddle factors are stored continuously in memory, based on the method described in [3]. The problem size n corresponds to `M`×`L`×2.

In the radix-2 FFT kernel the innermost loop length varies from 1 to $n/2$ for n-point FFTs during the $\log_2 n$ stages of the algorithm. For the first stage of the radix-2 FFT kernel in Listing 3, the innermost loop length is 1. In this case, the double-nested loop can be collapsed into a single-nested loop to expand the innermost loop length, as shown in Listing 4. When vectorizing the program in Listing 4, it is necessary to load the value of `W(J)` using the gather instruction and to store the result in `B(1,J)` and `B(2,J)` using the scatter instruction.

In the program shown in Listing 3 for `J=1` the value of the twiddle factor `W(J)` is $1 + 0j$, where $j = \sqrt{-1}$. Therefore, the complex multiplication can be omitted in the calculation of `W(J)*(C0-C1)` on line 9 of Listing 3. This program is shown in Listing 5.

We use the Stockham autosort FFT algorithm [5] in FFTE. The Stockham autosort FFT algorithm requires a scratch array of the same size as the input array but does not require a digit-reversal permutation. Table 2 shows the real inner-loop operations for radix-2, 3, 4, 5, 6, 8, 10, 12, and 16 double-precision complex FFT kernels. The higher radices are more efficient in terms of both memory and floating-point operations. For example, in view of the Byte/Flop ratio the radix-16 FFT is preferable to the radix-2, 4, and 8 FFTs [21]. Higher radix FFTs require more floating-point registers to hold intermediate results, and since the A64FX processor has 32 512-bit registers higher radix kernels are a good choice. A power-of-two point FFT (more than or equal to 64-point FFT) can

Table 2: Real inner-loop operations for radix-2, 3, 4, 5, 6, 8, 10, 12, and 16 double-precision complex FFT kernels based on the Stockham FFT

	Radix-2	Radix-3	Radix-4	Radix-5	Radix-6	Radix-8	Radix-10	Radix-12	Radix-16
Loads	4	6	8	10	12	16	20	24	32
Stores	4	6	8	10	12	16	20	24	32
Multiplications	4	12	12	28	28	32	60	60	84
Additions	6	16	22	40	46	66	102	118	174
Byte/Flop ratio	6.400	3.429	3.765	2.353	2.595	2.612	1.975	2.157	1.984

be performed by a combination of radix-8 and radix-16 steps containing at most three radix-8 steps [25]. In other words, power-of-two FFTs can be performed as a length $n = 2^p = 8^q 16^r$ ($p \geq 6,\ 0 \leq q \leq 3,\ r \geq 0$).

Coding high radix FFT kernels by hand is complicated. Since FFTE [24] is coded by hand, only radix-2, 3, 4, 5, and 8 FFT kernels are implemented. In contrast, SPIRAL has the advantage that it can automatically generate arbitrary radix FFT kernels. Therefore we generate radix-2, 3, 4, 5, 6, 8, 10, 12, and 16 FFT kernels as initial set via Spiral.

Vectorization with Spiral. We now discuss how we use Spiral to generate FFTE SVE kernels. The approach follows the following steps:

- Use Spiral's algorithms and scalar code generation engine to generate scalar DFT kernels and twiddle factor multiplications.
- Replace the scalar operations with SVE vector operations to promote scalar code to vector code.
- Use the SVE `while` idiom to tile and vectorize the most efficient loop.

This approach follows the basic idea of SIMD vectorization in Spiral [10], but with one major difference: so far Spiral SIMD code was specialized for a vector length known at *code generation time*. The generate SVE code is not specialized for any vector length, and thus can be run for any current and future SVE-supported vector length. This renders a number of standard methods Spiral uses for efficient code generation inapplicable.

As shown in Listing 3–5, FFTE code has one or 2 nested loops around the kernel. In the code generation approach for SVE we favor vectorizing the loop leading to unit stride access, and only vectorize non-unit-stride loops if no other loop is available. Further, we specialize code for iteration 0 of the outer loop where applicable to remove superfluous multiplications by 1. This is achieved automatically via copy propagation, strength reduction, common subexpression elimination and algebraic simplification in the Spiral basic block compiler.

Listing 6: Spiral script to generate FFTE kernels for SVE

```
# Radix n kernel for FFTE, targeting ARM SVE
LoadImport(packages.ffte);
opts := FFTE_SVEopts;

opts.breakdownRules.DFT := [
    DFT_Base, DFT_CT, DFT_CT_Mincost, DFT_Rader,
    DFT_PD, DFT_GoodThomas, DFT_SplitRadix ];

# DFT configuration
n := 2;
vlen := TInt;
kk := -1;
name := When(kk=-1, "dft", "idft")::
        StringInt(n)::"_sve";

# variables
t := var.fresh_t("t", TInt);
j := var.fresh_t("j", TInt);
k := var.fresh_t("k", TInt);
l := var.fresh_t("l", TInt);
m := var.fresh_t("m", TInt);

# temp arrays
tmp1 := var.fresh_t("R", TArray(TVectSVE, 2*n));
tmp2 := var.fresh_t("S", TArray(TVectSVE, 2*n));
tmp3 := var.fresh_t("T", TArray(TVectSVE, 2*n));

# gather
gi := _i -> VGath_L(1, vlen, 2*(k+j*m+_i*l*m));
g := RulesSums(SumsSPL(
         VStack(List([0..n-1], gi)), opts));
gc := Compile(
        opts.codegen(g, tmp1, X, opts), opts);

# scatter
si := _i -> VScat_L(1, vlen, 2*(k+n*j*m + _i*m));
ss := SumsSPL(HStack(List([0..n-1], si)), opts);
s := RulesSums(SubstTopDown(ss,
    @(1, ScatAcc), e->Scat(@(1).val.func)));
sc := Compile(
    opts.codegen(s, Y, tmp3, opts), opts);

# generate DFT kernel
dft := RC(DFT(n, kk));
rts := AllRuleTrees(dft, opts);
opcnts := List(rts, r ->
    [ Length(Collect(CodeRuleTree(r, opts),
      @(1, [add, sub, mul], e->e.t=TReal))), r]);
rt := Minimum(opcnts)[2];
dfts := SumsRuleTree(rt, opts);
dftc := BlockUnroll(opts.codegen(
    dfts, tmp2, tmp1, opts), opts);

# twiddles as lookup table
i := var.fresh_t("i", 2*n);
twt := var.fresh_t("TW", TPtr(TReal));
twf := Lambda(i, cond(eq(i, 0), 1, eq(i, 1), 0,
    nth(twt, 2*((n-1)*j) + 2 * idiv(i, 2) +
        (imod(i, 2) - 2))));
```

```
60 tws := RCDiag(twf);
61 twc :=BlockUnroll(unroll_cmd(
62     opts.codegen(tws, tmp3, tmp2, opts)), opts);
63
64 # stitch code fragments together
65 cl := func(TVoid, "transform",
66     [Y, X, twt, j, k, l, m, opts.pg],
67     decl([tmp1, tmp2, tmp3],
68         chain(gc, dftc, twc, sc)));
69 cc := Compile(cl, opts);
70
71 # final code
72 cl := func(TVoid, "transform", [Y, X, twt, l, m],
73     decl(c.cmd.vars::c.free()::[opts.pg],
74         loopn(j, l,
75             sve_loopn(k, m, c.cmd.cmd
76             )
77         )
78     )
79 );
80
81 # print Radix N FFTE kernel as function
82 PrintCode(name::"_jk", cl, opts);
83 PrintTo(name::"_jk"::".c",
84     PrintCode(name, cl, opts));
```

Generator scripts. We now show a (simplified) Spiral script to generate FFTE kernels (see Listing 6) and discuss the details. The full scripts are available at the Spiral web page `www.spiral.net` and as part of the experimental FFTE for SVE distribution.

Lines 2–3: The prologue of the script. Loads the FFTE package and sets up the options record.

Lines 5–7: The configuration for FFT algorithms to be used in the kernel. The script uses the 2-point FFT (`DFT_Base`), the general Cooley-Tukey FFT algorithm (`DFT_CT`), a special variant that treats the first iterations separately to remove multiplications by 1 (`DFT_Mincost`), the *partial diagonalization* approach (`DFT_PD`), the Good-Thomas prime factor algorithm (`DFT_GoodThomas`), Rader's algorithm for prime sizes (`DFT_Rader`), and the Split Radix algorithm (`DFT_SplitRadix`).

Lines 10–14: This sets up the DFT kernel size and function name. `n` can range up to 16 or 32, and be a prime or composite number or a 2-power.

Lines 16–26: This sets up temporary variables and temporary arrays for the computation to connect the various components together. Since the entire kernel will be unrolled eventually these temporary arrays will be scalarized.

Lines 28-33: This section generates the data gathering code as function of the outer loops. It generates a matrix *stack* of gather operations that separate real and imaginary values into two vectors and invokes the compiler and code generator to produce the corresponding code snippet. The complexity stems from the unknown vector length that is not natively supported by the Spiral code generation system.

Lines 35–41: This section generates the final data scattering. As with the gather code, complexity stems from the unknown vector length. This code snippet invokes the basic block compiler and rewriting system to compactly encode complicated code transformations. A particular issue on the scatter side is the distinction between accumulating and writing into target locations, which require special treatment in Spiral.

Lines 43–52: This section generates the actual DFT kernel. It enumerates all DFT algorithms known to Spiral for the given size, generates an internal code representation for the kernels (called `icode`), and picks the algorithm with minimum operations count. It then compiles this kernel down to code while using the proper temporary variables to stitch the kernels together with the twiddles and gather/scatter operations.

Lines 54–62: In this section we generate the loads into FFTE's twiddle table as function of the outer loops. The twiddle table is represented as Spiral*lambda finction* that is evaluated at the locations of use. Fresh symbols are created to enable the construction of the lambda function. The symbol `RCDiag` is a complex diagonal matrix in real arithmetic, parameterized by a lambda function. This object is the key SPL object to capture FFTE's twiddle factor layout. `nth` abstracts array accesses while `cond` abstracts the conditional assignment `?:`.

Lines 64–69: This section stitches together the internal code representation fragments for the gather operation, the DFT kernel, the twiddle scaling and the final scatter operation. It then invokes the basic block compiler on the merged code to run another pass of basic block optimization that cleans up the boundaries between the code fragments. The result is the final kernel code as function of the (yet undefined) outer loops.

Lines 71–79: Now the outer loops are wrapped around the inner kernel to complete the full Stockham stage as required by FFTE. In this case the outer loop is a normal C loop while the inner loop is a SVE vetorized loop that eventually will be unparsed (pretty-printed) using the standard `while` construct suggested for SVE vectorized loops.

Lines 81–83: The final step in the script is to print out the generate code in properly named C files.

Special cases. Listing 6 shows the general flow of Spiral script to generate FFTE kernels. As mentioned above, variants are needed to address special cases when loops collapse in the Stockham algorithm or to handle the first iteration that results in trivial twiddle factors. Minor modifications to Listing 6 are sufficient to generate all code variants: 1) the gather operations and twiddle operations need to be changed to support strided access, and 2) some loop variables need to be parially evaluated to 0, and their bouds need to be partially evaluated to 1.

A Spiral-generated radix-2 FFT kernel equivalent to Listing 3 is shown in Listing 7.

Listing 7: Spiral-generated radix-2 FFT kernel

```
1 #include <include/omega64.h>
2 #include <arm_sve.h>
3
4 void dft2_sve_jk(float64_t *Y, float64_t *X,
5                  float64_t *TW1, int l1, int m1) {
```

```
  svfloat64x2_t s49, s52, svex2_3, svex2_4;
  int a104, a105;
  float64_t a106, a107;
  svbool_t pg1;
  svfloat64_t s50, s51, s53, s54, s55, s56;
  for (int j1 = 0; j1 < l1; j1++) {
    int k1 = 0;
    pg1 = svwhilelt_b64(k1, m1);
    do {
      a104 = k1 + j1 * m1;
      s49 = svld2_f64(pg1, X + 2 * a104);
      s50 = s49.v0;
      s51 = s49.v1;
      s52 = svld2_f64(pg1,
            X + 2 * (a104 + l1 * m1));
      s53 = s52.v0;
      s54 = s52.v1;
      s55 = svsub_f64_x(pg1, s50, s53);
      s56 = svsub_f64_x(pg1, s51, s54);
      a105 = ((2)*(j1));
      a106 = TW1[a105];
      a107 = TW1[a105 + 1];
      svex2_3.v0 = svadd_f64_x(pg1, s50, s53);
      svex2_3.v1 = svadd_f64_x(pg1, s51, s54);
      svst2_f64(pg1, Y + 2 * (k1 + 2 * j * m1),
                svex2_3);
      svex2_4.v0 = svnmls_n_f64_x(pg1,
        svmul_n_f64_x(pg1, s56, a107), s55, a106);
      svex2_4.v1 = svmla_n_f64_x(pg1,
      svmul_n_f64_x(pg1, s56, a106), s55, a107);
      svst2_f64(pg1, Y + 2 * (k1 + (2 * j1 * m1)
                + m1), svex2_4);
      k1 += svcntd();
      pg1 = svwhilelt_b64(k1, m1);
    } while(svptest_any(svptrue_b64(), pg1));
  }
}
```

First stage of a Spiral-generated radix-2 FFT kernel equivalent to Listing 4 is shown in Listing 8.

Listing 8: First stage of a Spiral-generated radix-2 FFT kernel

```
#include <include/omega64.h>
#include <arm_sve.h>

void dft2_sve_j(float64_t *Y, float64_t *X,
                float64_t *TW1, int l1) {
  svfloat64x2_t s53, s56;
  float64_t *a94;
  int a93;
  svbool_t pg1;
  svfloat64_t a95, a96, s54, s55, s57, s58, s59,
              s60, s61, s62, s63, s64;
  int j1 = 0;
  pg1 = svwhilelt_b64(j1, l1);
  do {
    a93 = 2 * j1;
    s53 = svld2_f64(pg1, X + a93);
    s54 = s53.v0;
    s55 = s53.v1;
    s56 = svld2_f64(pg1, X + 2 * (j1 + l1));
    s57 = s56.v0;
    s58 = s56.v1;
    s59 = svadd_f64_x(pg1, s54, s57);
    s60 = svadd_f64_x(pg1, s55, s58);
    s61 = svsub_f64_x(pg1, s54, s57);
    s62 = svsub_f64_x(pg1, s55, s58);
    a94 = TW1 + a93;
    a95 = svld1_gather_s64offset_f64(pg1, a94,
          svindex_s64(0, 16));
    a96 = svld1_gather_s64offset_f64(pg1, a94 + 1,
          svindex_s64(0, 16));
    s63 = svnmls_f64_x(pg1,
          svmul_f64_x(pg1, a96, s62), a95, s61);
    s64 = svmla_f64_x(pg1,
          svmul_f64_x(pg1, a95, s62), a96, s61);
    svst1_scatter_s64offset_f64(pg1, Y + 4 * j1,
          svindex_s64(0, 32), s59);
    svst1_scatter_s64offset_f64(pg1, 1 + Y
          + 4 * j1, svindex_s64(0, 32), s60);
    svst1_scatter_s64offset_f64(pg1, 2 + Y
          + 4 * j1, svindex_s64(0, 32), s63);
    svst1_scatter_s64offset_f64(pg1, 3 + Y
          + 4 * j1, svindex_s64(0, 32), s64);
    j1 += svcntd();
    pg1 = svwhilelt_b64(j1, l1);
  } while(svptest_any(svptrue_b64(), pg1));
}
```

Middle stage of a Spiral-generated radix-2 FFT kernel equivalent to Listing 5 is shown in Listing 9.

Listing 9: Middle stage of a Spiral-generated radix-2 FFT kernel

```
#include <include/omega64.h>
#include <arm_sve.h>

void dft2_sve_k(float64_t *Y, float64_t *X,
                int m1) {
  svfloat64x2_t s43, s46, svex2_3, svex2_4;
  svbool_t pg1;
  svfloat64_t s44, s45, s47, s48;
  int k1 = 0;
  pg1 = svwhilelt_b64(k1, m1);
  do {
    s43 = svld2_f64(pg1, X + 2 * k1);
    s44 = s43.v0;
    s45 = s43.v1;
    s46 = svld2_f64(pg1, X + 2 * (k1 + m1));
    s47 = s46.v0;
    s48 = s46.v1;
    svex2_3.v0 = svadd_f64_x(pg1, s44, s47);
    svex2_3.v1 = svadd_f64_x(pg1, s45, s48);
    svst2_f64(pg1, Y + 2 * k1, svex2_3);
    svex2_4.v0 = svsub_f64_x(pg1, s44, s47);
    svex2_4.v1 = svsub_f64_x(pg1, s45, s48);
    svst2_f64(pg1, Y + 2 * (k1 + m1), svex2_4);
    k1 += svcntd();
    pg1 = svwhilelt_b64(k1, m1);
  } while(svptest_any(svptrue_b64(), pg1));
}
```

5 PERFORMANCE RESULTS

To evaluate the vectorized FFT kernels we evaluated size of the form $n = 2^p 3^q 5^r$, where p, q, and r were varied. We compared the performance of automatically vectorized FFT kernels written in Fortran (FFTE 6.0) [24], explicit vectorized FFT kernels written in SVE intrinsic generated by SPIRAL, and automatic vectorized FFT kernels written C generated by SPIRAL.

At the time of writing this paper, the A64FX processor was still under development, and we were not able to use the actual machine. Therefore, the performance was evaluated using the RIKEN Fugaku processor simulator [17]. The specifications of the A64FX chip are shown in Table 3. The A64FX processor has 48 cores, however, we evaluated our approach on one core and one thread to focus on vectorization.

The execution times obtained from 10 executions of complex forward FFTs were averaged. The FFTs were performed on double-precision complex data and the table for twiddle factors was precomputed. The problem sizes n were in the range of 64 to 65,536, which fit into the L2 cache of the processor. Normalized cycle times (the number of cycles required for execution divided by $5n \log_2 n$) were compared across the various implementations for n-point FFTs. The smaller these normalized values, the higher the performance.

We used the Fujitsu C/C++/Fortran compiler Version 4.0.0, and the Arm C/C++/Fortran Compiler version 19.1. We set the compiler options as `-Kfast,restp=all` and `-Kfast` for the Fujitsu C/C++ and Fujitsu Fortran compilers, respectively. The compiler option `-Kfast` specifies the optimizations for high speed execution on the target machine. The compiler option `-Krestp` specifies to perform the optimization of restricted pointers assuming that the `restrict` qualifier is specified for all pointers. The compiler options used were specified as `-Ofast -march=armv8-a+sve` for the ARM C/C++/Fortran Compiler. The compiler option `-Ofast` specifies the optimizations for high speed execution on the target machine. The compiler option `-march=armv8-a+sve` targets Armv8 application architecture profile with SVE. The original FFTE program is written in Fortran. SPIRAL-generated FFT kernels are written in C.

The software used for the evaluation including the compiler is still under development and its performance may be different when the supercomputer Fugaku starts its operation. The result of this simulator [17] is just an estimate, and it does not guarantee the performance of the supercomputer Fugaku at the start of its operation.

Figures 1 and 2 show the performance of power-of-two FFTs with Fujitsu compiler and ARM compiler, respectively. Whether using the Fujitsu C compiler or ARM C compiler, SPIRAL (SVE intrinsic) is faster than SPIRAL (automatic vectorization). When Fujitsu C compiler is used, SPIRAL (SVE intrinsic) is approximately 6.33 times faster than SPIRAL (automatic vectorization) for $n = 256$. Furthermore, when ARM C compiler is used, the SPIRAL (SVE intrinsic) is approximately 5.62 times faster than the SPIRAL (automatic vectorization) for $n = 32,768$. From these results, it can be seen that explicit vectorization using intrinsic by SPIRAL is effective.

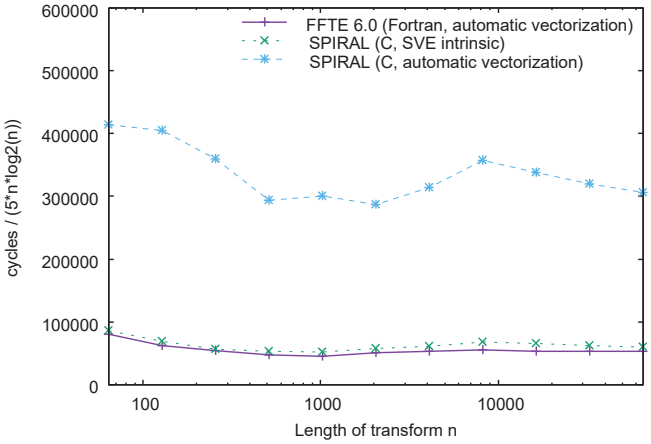

Figure 1: Performance of power-of-two FFTs with Fujitsu compiler

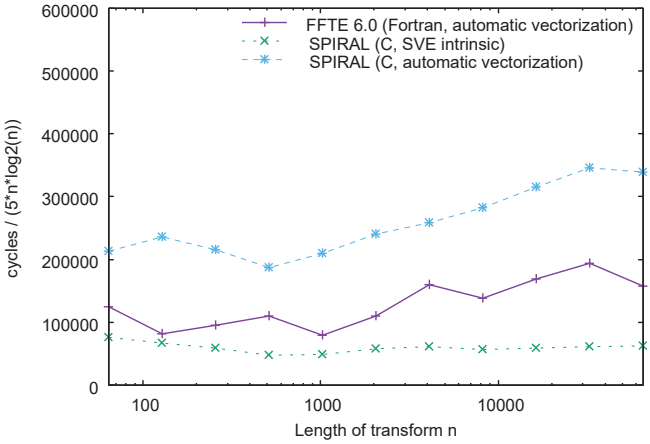

Figure 2: Performance of power-of-two FFTs with Arm compiler

With Fujitsu C and Fortran compilers, the FFTE (automatic vectorization) is slightly faster than the SPIRAL (SVE intrinsic). On the other hand, with ARM compiler, the SPIRAL (SVE intrinsic) is faster than the FFTE (automatic vectorization). Especially, for $n = 32,768$, the SPIRAL (SVE intrinsic) is approximately 3.16 times faster than FFTE (automatic vectorization).

The performance of the FFTE compiled with the Fujitsu Fortran compiler is higher than that compiled with the ARM Fortran compiler. On the other hand, the performance of the SPIRAL (both SVE intrinsic and automatic vectorization) compiled with the ARM C compiler is higher than that compiled with the Fujitsu C compiler. From these results we conclude that the optimization of the Fujitsu Fortran compiler worked more effectively for the FFTE than

Table 3: Specifications of the A64FX Chip [29]

Architecture	Armv8.2-A (AArch64 only)
	SVE 512-bit wide SIMD
Number of cores	48 computing cores + 4 assistant cores
Theoretical peak performance	Over 2.7 TFlops (double precision)
L1 Data Cache	64KiB, 4way
L2 Cache	8MiB, 16way
Memory capacity	32 GiB (HBM2)
Memory bandwidth	1024 GB/s

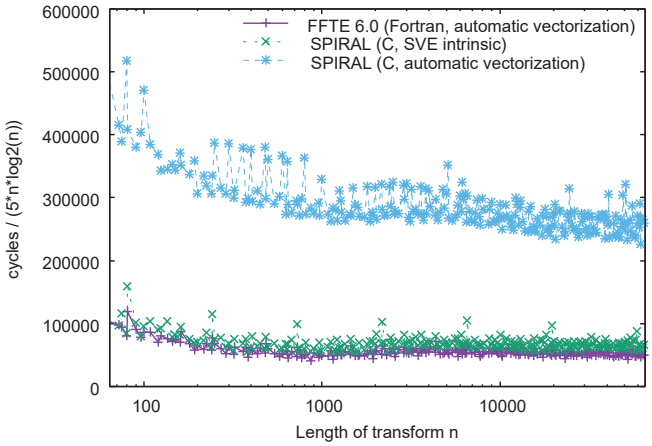

Figure 3: Performance of non-power-of-two FFTs with Fujitsu compiler

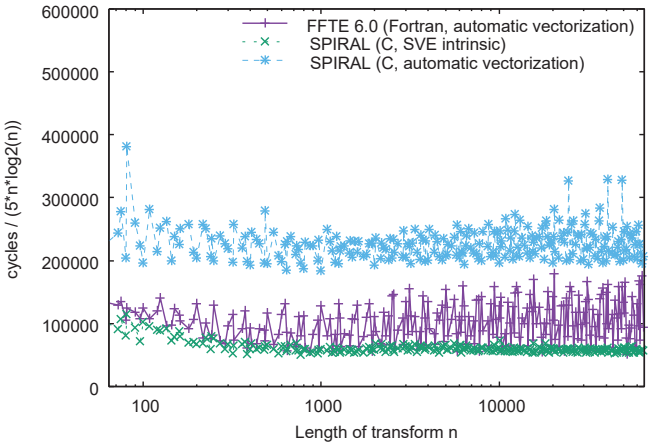

Figure 4: Performance of non-power-of-two FFTs with Arm compiler

the ARM Fortran compiler. Furthermore, the optimization of the ARM C compiler worked more effectively for the SPIRAL than the Fujitsu C compiler.

Figures 3 and 4 show the performance of non-power-of-two FFTs with Fujitsu compiler and Arm compiler, respectively. It can be seen from Figures 1–4 that the performance results of non-power-of-two FFTs have the same tendency as the performance results of power-of-two FFTs.

6 CONCLUSION

In this paper, we proposed the implementation of FFT on the ARM SVE. We performed automatic vectorization by the compiler and explicit vectorization by SPIRAL for FFT kernels and compared the performance. With the ARM compiler, Spiral-generated FFT kernels written with SVE intrinsic are up to 3.05 times faster than FFT kernels of FFTE written in Fortran, and up to 5.36 times faster than Spiral-generated FFT kernels written in C. From these results, it can be seen that explicit vectorization using ARM SVE intrinsic by SPIRAL is effective.

It is expected that the compiler optimization capability and automatic vectorization capability for ARM SVE will be further improved. In addition, we plan to evaluate the actual machine equipped with the A64FX processor, which is the first processor of the Armv8-A SVE architecture.

ACKNOWLEDGMENTS

A part of this research is supported by RIKEN joint research "Post-K parallel programming environment and network research."

REFERENCES

[1] ARM Limited 2019. *ARM C Language Extensions for SVE, Version 00bet2.* ARM Limited. https://static.docs.arm.com/100987/0000/acle_sve_100987_0000_01_en.pdf
[2] ARM Limited 2019. *ARM Compiler Scalable Vector Extension User Guide, Version 6.12.* ARM Limited.
[3] David H. Bailey. 1987. A High-Performance Fast Fourier Transform Algorithm for the Cray-2. *J. Supercomput.* 1 (1987), 43–60.
[4] E. Oran Brigham. 1988. *The Fast Fourier Transform and Its Applications.* Prentice-Hall, Upper Saddle River.
[5] W. T. Cochran, J. W. Cooley, D. L. Favin, H. D. Helms, R. A. Kaenel, W. W. Lang, G. C. Maling, D. E. Nelson, C. M. Rader, and P. D. Welch. 1967. What is the Fast Fourier Transform? *IEEE Trans. Audio and Electroacoust.* 15 (1967), 45–55.
[6] James W. Cooley and John W. Tukey. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.* 19 (1965), 297–301.
[7] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. 2009. Operator Language: A Program Generation Framework for Fast Kernels. In *IFIP Working Conference on Domain Specific Languages (DSL WC).*
[8] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M. Veras, Daniele G. Spampinato, Jeremy R. Johnson, Markus Püschel, James C. Hoe, and

José M. F. Moura. 2018. SPIRAL: Extreme Performance Portability. *Proc. IEEE* 106 (2018), 1935–1968.

[9] Franz Franchetti and Markus Püschel. 2002. A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms. In *Intl. Parallel and Distributed Processing Symposium (IPDPS)*. 20–26.

[10] Franz Franchetti and Markus Püschel. 2003. Short Vector Code Generation for the Discrete Fourier Transform. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE, 10–pp.

[11] Franz Franchetti, Daniele G. Spampinato, Anuva Kulkarni, Thom Popovici, Tze-Meng Low, M. Franusich, A. Canning, P. McCorquodale, B. Van Straalen, and P. Colella. 2018. FFTX and SpectralPack: A First Look. In *IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*.

[12] Franz Franchetti, Yevgen Voronenko, and Gheorghe Almasi. 2012. Automatic Generation of the HPC Challenge's Global FFT Benchmark for BlueGene/P. In *High Performance Computing for Computational Science (VECPAR)*.

[13] Franz Franchetti, Yevgen Voronenko, Peter A. Milder, Srinivas Chellappa, Marek Telgarsky, Hao Shen, Paolo D'Alberto, Frédéric de Mesmay, James C. Hoe, José M. F. Moura, and Markus Püschel. 2008. Domain-Specific Library Generation for Parallel Software and Hardware Platforms. In *NSF Next Generation Software Program workshop (NSFNGS)*.

[14] Matteo Frigo and Steven G. Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93 (2005), 216–231.

[15] Intel Corporation. 2018. *Intel Architecture Instruction Set Extensions and Future Features Programming Reference*.

[16] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. 1990. A Methodology for Designing, Modifying, and Implementing FFT Algorithms on Various Architectures. *Circuits Systems Signal Processing* 9 (1990), 449–500.

[17] Yuetsu Kodama, Tetsuya Odajima, Akira Asato, and Mitsuhisa Sato. 2019. Evaluation of the RIKEN Post-K Processor Simulator. *Computing Research Repository (CoRR)* abs/1904.06451 (2019), 1–6. http://arxiv.org/

[18] Daniel S. McFarlin, Volodymyr Arbatov, Franz Franchetti, and Markus Püschel. 2011. Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction sets. In *Proc. 25th International Conference on Supercomputing (ICS'11)*. 265–274.

[19] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"* 93, 2 (2005), 232– 275.

[20] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37 (2017), 26–39.

[21] Daisuke Takahashi. 2003. A Radix-16 FFT Algorithm Suitable for Multiply-Add Instruction Based on Goedecker Method. In *Proc. 2003 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2003)*, Vol. 2. 665–668.

[22] Daisuke Takahashi. 2007. An Implementation of Parallel 1-D FFT Using SSE3 Instructions on Dual-Core Processors. In *Proc. 8th International Workshop on State of the Art in Scientific Computing (PARA 2006) (Lecture Notes in Computer Science)*, Vol. 4699. Springer-Verlag, 1178–1187.

[23] Daisuke Takahashi. 2012. An Implementation of Parallel 2-D FFT Using Intel AVX Instructions on Multi-core Processors. In *Proc. 12th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2012), Part II (Lecture Notes in Computer Science)*, Vol. 7440. Springer-Verlag, 197–205.

[24] Daisuke Takahashi. 2014. *FFTE: A Fast Fourier Transform Package*. http://www.ffte.jp/

[25] Daisuke Takahashi. 2017. An Implementation of Parallel 1-D Real FFT on Intel Xeon Phi Processors. In *Proc. 17th International Conference on Computational Science and Its Applications (ICCSA 2017), Part I (Lecture Notes in Computer Science)*, Vol. 10404. Springer International Publishing, 401–410.

[26] R. Tolimieri, M. An, and C. Lu. 1997. *Algorithms for discrete Fourier transforms and convolution* (2nd ed.). Springer.

[27] C. Van Loan. 1992. *Computational Framework of the Fast Fourier Transform*. SIAM.

[28] J. Xiong, J. Johnson, R. Johnson, and D. Padua. 2001. SPL: A Language and Compiler for DSP Algorithms. In *Proc. Programming Language Design and Implementation (PLDI)*. 298–308.

[29] Toshio Yoshida. 2018. Fujitsu High Performance CPU for the Post-K Computer. In *Proc. 2018 IEEE Hot Chips 30 Symposium*.