Adaptation of A64 Scalable Vector Extension for Spiral

Naruya Kitai[1] Daisuke Takahasi[2] Franz Franchetti[3] Takahiro Katagiri[1]
Satoshi Ohshima[1] Toru Nagai[1]

Abstract: In this paper, we propose an adaptation of the A64 Scalable Vector Extension for SPIRAL to generate discrete Fourier transform (DFT) implementations. The performance of our method is evaluated, using the Supercomputer "Flow" at Nagoya University. The A64 scalable vector extension applied DFT codes are up to 1.98-times faster than scalar DFT codes and up to 3.63-times higher in terms of the SIMD instruction rate.

Keywords: SPIRAL, Arm SVE

1. Introduction

Computer architectures are becoming increasingly diverse, with configurations varying greatly depending on the environment, such as the number of CPU cores, memory access speed, cache structure, and presence or absence of a graphics processing unit (GPU). To achieve a high performance, software tuning is important for numerical software; however, optimizing the software for each computing environment requires specialized knowledge of the hardware as well as significant time and effort. In addition, the performance of software tuned for one specific environment may be degraded in another. In this case, the software must be tuned again when the environment is migrated. Furthermore, optimization by widely used compilers is often insufficient for software developers.

Software tuning (AT) for numerical computations [1] is a technology that automates the tuning performance to improve the software performance. By using AT technology, software optimized for one computer environment is expected to perform equally well in a different computer environment. This is known as the performance portability of the software. For this reason, many numerical software packages with AT functions have been developed [2][3]. Furthermore, AT frameworks have been proposed for creating numerical software with AT functions, including OpenTuner [4], Xevolver [5], and FIBER [6].

One effort to achieve high-performance software portability is automatic code generation through the use of an AT technique. Automatic code generation is applied to computational kernels based on the mathematical background to provide implementations of numerical algorithms for various computing environments and programming languages.

SPIRAL [7][8] is a program generation system that automatically generates optimized implementations of mathematical algorithms, including linear transforms such as a discrete Fourier transform (DFT) and a discrete cosine transform (DCT). In this paper, we propose a method for applying the A64 Scalable Vector Extension (SVE) [12][13] to DFT generation in SPIRAL. SPIRAL supports code generation, to which SIMD instructions are explicitly applied. For instance, Intel SSE, Intel AVE, and Arm NEON instruction sets can be used. SPIRAL also supports the generation of FFTE kernels using SVE [9]. However, we need another way to support general codes using Arm SVE.

The remainder of this paper is organized as follows. Section 2 describes the SPIRAL code generator. Section 3 details the Arm SVE. Section 4 presents the adaptation of the Arm SVE to DFT generation in SPIRAL. Section 5 presents the results. Finally, some concluding remarks are provided in Section 6.

2. Spiral Code Generator

2.1 Overview

In this chapter, we describe the SPIRAL code generation system and DFT, which are the focus of this research.

SPIRAL is a system that automatically generates implementations of mathematical algorithms. The code can be optimized to suit the computing environment. It has been under development at Carnegie Mellon University for more than 20 years. SPIRAL is still being developed in terms of the portability of numerical software. It provides high-performance implementations for a wide range of hardware, including embedded systems, HPC, GPUs, and FPGAs. Although creating a high-performance library optimized for a computational environment requires a great deal of effort by engineers with a high level of expertise, the SPIRAL approach can reduce this effort.

In addition to the choice of algorithm, SPIRAL optimizes the program by specifying optional parameters that define what kind of optimizations to apply, such as SIMD instructions and the degree of loop unrolling. The specified parameters were applied sequentially to the code generation procedure. Then, the parameters were reflected in the final generated code. The generated program is output as a file containing functions that implement the mathematical algorithm. The generated program can then be compiled and used similarly to a normal program file.

2.2 Representation of DFT in Spiral

DFT is a Fourier transform of a sampled signal in both the time and frequency domains and is widely used in signal processing and other applications. The DFT for a sample of n points $x_0, x_1, \dots, x_{n-1}$ is defined as follows:

$$y_k = \sum_{l=0}^{n-1} \omega_n^{kl} x_l \qquad (0 \le k < n), \tag{1}$$

1 Nagoya University, Nagoya, Aichi 464-0814, Japan
2 University of Tsukuba, Tsukuba, Ibaraki 305-8577, Japan
3 Carnegie Mellon University, Pittsburgh, PA 15213, USA

ⓒ2021 Information Processing Society of Japan

with

$$\omega_n = e^{\frac{-2\pi j}{n}}. \quad (2)$$

In addition, a fast Fourier transform (FFT), which can perform the same transform as a DFT with fewer operations and a faster speed, can be used [10]. In practice, a DFT is almost always implemented using an FFT.

In SPIRAL, the DFT for a vector of length n is defined as follows:

$$DFT_n: \mathbb{C}^n \rightarrow \mathbb{C}^n;\ x \mapsto \left[\omega_n^{ij}\right]_{i,j} x. \quad (3)$$

Using (3), (1) can be expressed in the form of a vector matrix product of vectors x and y:

$$y = DFT_n x. \quad (4)$$

The FFT algorithm can be obtained by decomposing Equation (4). For example, the decimation-in-time Cooley-Tukey FFT algorithm is expressed as

$$DFT_n \rightarrow (DFT_k \otimes I_m) T_m^n (I_k \otimes DFT_m) L_k^n, \quad n = km \quad (5)$$

where each operator and matrix are as defined below. The Kronecker product of matrices A and B is

$$A \otimes B = [a_{k,l}B], \quad \text{for } A = [a_{k,l}]. \quad (6)$$

The identity matrix is

$$I_n = [i_{k,l}], \quad i_{k,l} = \begin{cases} 1\ (k = l) \\ 0\ (k \neq l) \end{cases}. \quad (7)$$

The Twiddle matrix is

$$T_m^n = [t_{k,l}], \quad t_{k,l} = \begin{cases} \omega_m^k\ (k = l) \\ 0\ \ (k \neq l) \end{cases}. \quad (8)$$

Finally, the stride permutation matrix is

$$L_k^n = [l_{i,j}],$$
$$l_{i,j} = \begin{cases} 1\ (k(i \bmod m) + \lfloor j/k \rfloor = ni + j) \\ 0\ \ otherwise \end{cases}. \quad (9)$$

In (5), DFT_n is decomposed into an expression containing a smaller DFT_k and DFT_m. By applying (5) recursively, the DFT is decomposed into smaller and smaller units.

Finally, we need a rule that corresponds to the smallest unit, DFT_2, which is defined as follows:

$$DFT_2 \rightarrow F_2, \quad F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \quad (10)$$

This corresponds to the butterfly operation in FFT. If a radix other than two is involved, the smallest unit generated using an algorithm such as Rader's FFT algorithm is applied.

In addition to (5), the following decompositions can be applied [11]:

The decimation-in-frequency Cooley-Tukey FFT is

$$DFT_n \rightarrow L_m^n (I_k \otimes DFT_m) T_m^n (DFT_k \otimes I_m),\ n = km. \quad (11)$$

The four-step FFT is

$$DFT_n \rightarrow (DFT_k \otimes I_m) T_m^n L_k^n (DFT_m \otimes I_k),\ n = km. \quad (12)$$

Finally, the six-step FFT is

$$DFT_n \rightarrow L_k^n (I_m \otimes DFT_k) L_m^n T_m^n (I_k \otimes DFT_m) L_k^n,\ n = km. \quad (13)$$

2.3 Process of Code Generation

In this section, we consider the case of generating DFT_4 using SPIRAL as an example to explain the usage and processing flow of the SPIRAL code generation. Fig. 1 shows an example input for a user to generate code using SPIRAL, and Fig. 2 shows an overview of the internal code generation process of SPIRAL when the input in Fig. 1 is given.

```
1  opts := SpiralDefaults;
2  transform := DFT(4);
3  ruletre := RandomRuleTree(transform, opts);
4  icode := CodeRuleTree(ruletree, opts);
5  PrintCode("DFT4", icode, opts);
```

Figure 1: Example input for generating DFT_4 using SPIRAL

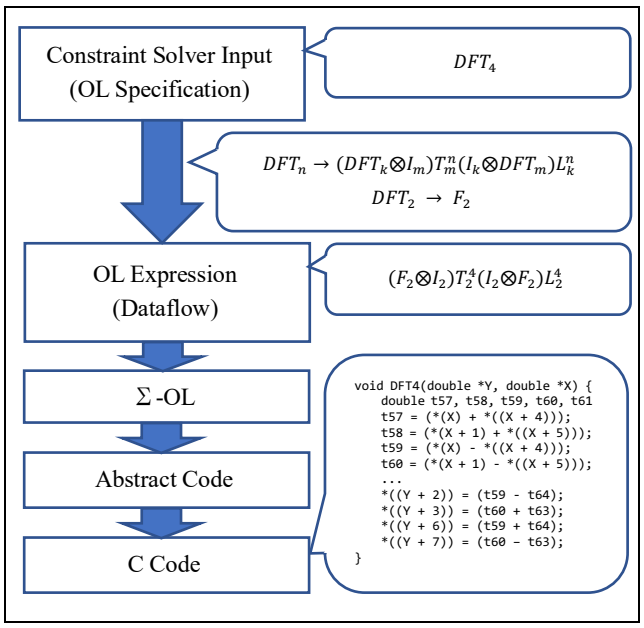

Figure 2: Internal process of code generation in SPIRAL

The first step is to specify the transform for which the code is to be generated. These notations are called the operator language (OL). In Fig. 1, the variable opts is assigned as *SpiralDefaults*, which specifies the parameters to be used for code generation. This information was used to determine the optimization of the target transform. The dataflow is then determined by expanding the specified OL with breakdown rules, as shown in (5) and (10). Because the data flow has a significant impact on the performance of the generated code, the application pattern of the breakdown rules is important. *RandomRuleTree()* is a function for randomly selecting the dataflow. However, it does not always select the optimal dataflow. To determine the optimal value, an automatic dataflow search using the performance measurements is provided. The dataflow will eventually become a C program through the formation of loop structures (Σ-OL) and abstract code.

ⓒ2021 Information Processing Society of Japan

3. Arm Scalable Vector Extension

The Arm scalable vector extension is a SIMD extension for Armv8-A architectures [14] and is being developed for use in HPCs. Arm has a SIMD extension, NEON [15]; however, SVE is not a successor to NEON, but a completely new vector instruction.

In traditional SIMD architectures such as NEON and Intel's Advanced Vector Extensions (AVX) [16], vector registers are defined with a fixed length. In an SVE, however, only the maximum length of the vector registers is defined, allowing licensees to develop implementations with arbitrary vector lengths. Specifically, the licensee can choose a vector length of up to 2048 bits. Because of the variable vector length, software implementations using an SVE should be written in a new programming style called Vector Length Agnostic (VLA) programming [17]. VLA is a programming model that does not have a fixed vector length, and almost all SVE instructions are implemented by hiding the vector length through predicates to realize VLA operations. As examples of VLA programming, Figs. 3 and 4 show programs that demonstrate the same behavior with and without SVE instructions, respectively. The Arm C Language Extension (ACLE) [18] provides an interface to the SVE for the C/C++ language, as shown in Fig. 4.

```
1  int64_t i = 0;
2  do {
3      z[i] = x[i] + y[i];
4      i++;
5  } while(i<N);
```

Figure 3: Example of a program without SVE instructions

```
 1  int64_t i = 0;
 2  svbool_t pg = svwhilelt_b64(i, N);
 3  do {
 4      svfloat64_t x_sve = svld1(pg, &x[i]);
 5      svfloat64_t y_sve = svld1(pg, &y[i]);
 6      svfloat64_t z_sve = svadd_x(pg, x_sve, y_sve);
 7      svst1(pg, &z[i], z_sve);
 8      i += svcntd();
 9      pg = svwhilelt_b64(i, N);
10  } while(svptest_any(svptrue_b64(), pg));
```

Figure 4: Example of a program with SVE instructions

In the above example, the process is to add the values of arrays x and y together and store them in array z. The followings explain how the program shown in Fig. 4 works, where *svbool_t* and *svfloat64_t* are vector types and store the number of elements corresponding to the vector length of the processor.

Line 2: In the predicate type variable (pg), valid elements are assigned true values, and invalid elements are assigned false values. In addition, svwhilelt_b64() is a function that controls the loop iteration and determines whether the process has deviated from the loop range. For elements that deviate from the range, a false value is assigned to the predicate such that the elements will not be executed through the functions.

Line 4-5: Load the values from the arrays into the vectors.

Line 6: Add vectors to each other and assign them to a vector.

Line 7: Store the values from a vector into an array.

Line 8: Add the number of elements stored in the vector to the loop variable.

Line 9: Update the predicate variable to determine the loop continuation.

Line 10: If all values of the predicate are false, the loop terminates. If the predicate contains a true value, there are still elements to be processed.

4. Adaptation of the Arm SVE for Spiral

SPIRAL supports a code generation by applying SIMD instructions with fixed vector lengths, such as Intel's AVX and Arm NEON, and many functions for an efficient code generation assume fixed-length vectors. However, the vector length of the SVE is variable, and these functions therefore cannot be used. Thus, in this study, we developed a script to convert the scalar C code into an efficient SVE code, which tentatively supports the generation of the SVE code.

Figs. 5 and 6 show the flow of the code generation by applying AVX and SVE, respectively. Fig. 6 applies SIMD instructions using standard SPIRAL functions, and Fig. 5 applies the SVE conversion script (SVE converter) implemented in this study.

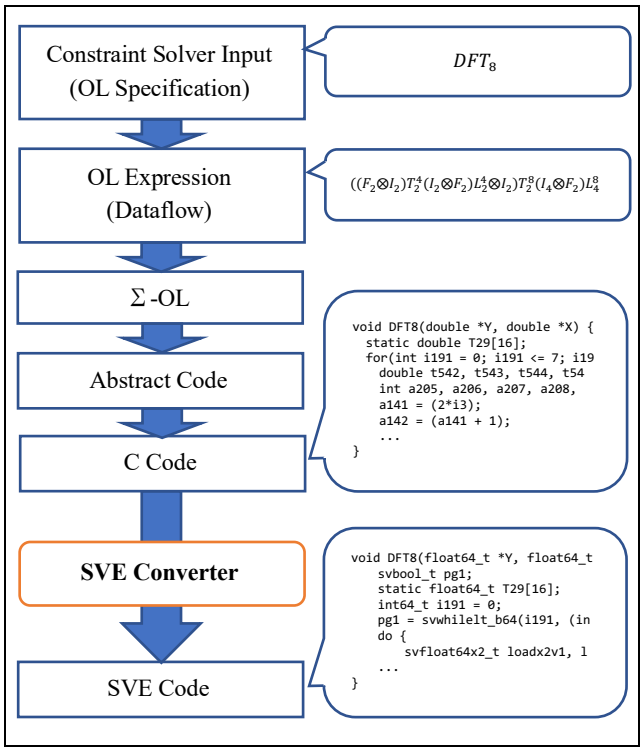

Figure 5: Flow of code generation applying SVE

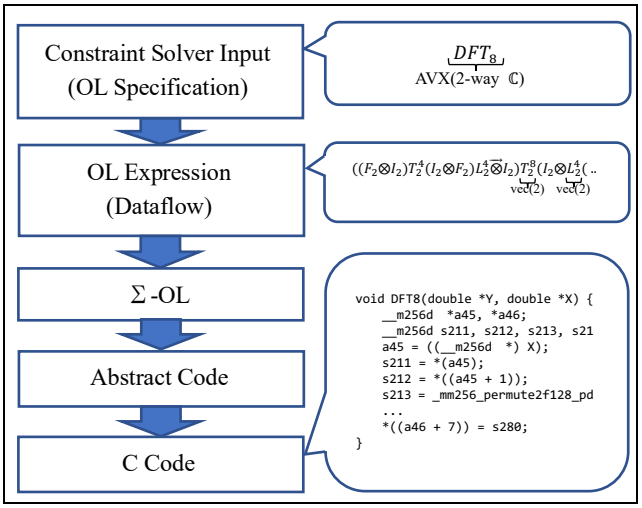

Figure 6: Flow of code generation applying AVX

The SVE conversion script can also be applied to automatic dataflow searches. By applying the script, the dataflow optimized for the SVE-applied program can be selected.

The following is a summary of the SVE conversion script.

(1) Set the innermost loop as the loop to be vectorized.
The loop structure is used to convert to the SVE code. In the case of nested loops, the innermost loop is parameterized. The structure of the loop is rewritten as shown in Fig. 4, and the predicate is set. If the array load/store does not allow continuous access, we specify the stride (offset) of the access.

(2) Apply SVE operations to a vectorized loop.
Replace the scalar load, store, add, subtract, and multiply operations using SVE operations.

(3) Modify duplicate loads to load them concurrently.
In the code generated by the SPIRAL, when an array element needs to be accessed multiple times, it is accessed each time instead of storing the array in a variable. However, when the SVE operation is used, the performance is better when the number of read operations is reduced.

(4) Load an array of n-element structures into n vectors.
The SVE has operations for loading two to four element structures of an array at a time (svld2, svld3, svld4). This provides a better performance than loading a single element two to four times, but it can only be used if the memory accesses are contiguous. Combining multiple loads can often form continuous access, and this can be applied to improve the performance.

5. Performance Evaluation

To evaluate the performance of the proposed SVE instruction adaptation, we compared it with the performance of a program generated using standard SPIRAL functions. In the experiments, we used the “Flow” Type I subsystem (FX1000), a supercomputer installed at the Information Technology Center of Nagoya University, to evaluate the performance. The hardware and software configurations are shown in Table 1 and Table 2, respectively. All programs used in the performance comparison were computed using only one core on a single node.

Table 1: Hardware configuration of “Flow” Type I subsystem

Machine Name		FUJITSU Supercomputer PRIMEHPC FX1000
CPU	Processor Name	FUJITSU Processor A64FX
	ISA	Arm v8.2 + SVE
	Frequency	2.2 GHz
	SIMD Width	512 bit
	Number of Cores	48 compute cores and 2/4 assistant cores
	L1I Cache Size	3 MiB (64 KiB/core)
	L1D Cache Size	3 MiB (64 KiB/core)
	L2 Cache Size	32 MiB (8 MiB x 4)
Compute Node	Number of CPUs	1
	Memory	HBM2, 32 GiB
	Peak Flops	3.4T (Double), 6.8T (Single), 13.5T (Half)
Number of Nodes		2,304

Table 2: Software information used in performance measurement

Compiler	fccpx (FCC) 4.2.1 20200820 clang: Fujitsu C/C++ Compiler 4.2.1 (Aug 25 2020 11:42:20) (based on LLVM 7.1.0)
Option	-Nclang -mcpu=a64fx+sve -Ofast
SPIRAL	8.2.0
Python	3.6.8

The inputs to SPIRAL for generating a program referred to as SVE-applied DFT (the proposed method) and scalar DFT programs used in the performance comparison are shown in Figs. 7 and 8, respectively, where the variable ‘SIZE’ denotes the size of data and ‘N’ an integer between 4 and 20.

```
1 SIZE := 2^N;
2 opts := SpiralDefaults;
3 opts.target := rec();
4 opts.target.name := "flow-fx-sve";
5 transform := DFT(SIZE, -1);
6 best := DP(transform, rec(), opts);
7 ruletree := best[1].ruletree;
8 icode := CodeRuleTree(ruletree, opts);
9 PrintCode("dft", icode, opts);
```
Figure 7: Inputs to generate the SVE-applied DFT program

```
1 SIZE := 2^N;
2 opts := SpiralDefaults;
3 opts.target := rec();
4 opts.target.name := "flow-fx";
5 transform := DFT(SIZE, -1);
6 best := DP(transform, rec(), opts);
7 ruletree := best[1].ruletree;
8 icode := CodeRuleTree(ruletree, opts);
9 PrintCode("dft", icode, opts);
```
Figure 8: Inputs used to generate scalar DFT program

ⓒ2021 Information Processing Society of Japan

A comparison of the computational performance of the DFT codes generated from Figs. 7 and 8 is shown in Fig. 9. A comparison of the SIMD instruction rates is shown in Fig. 10. The SIMD instruction rate is the ratio of the number of SIMD instructions to the total number of instructions executed.

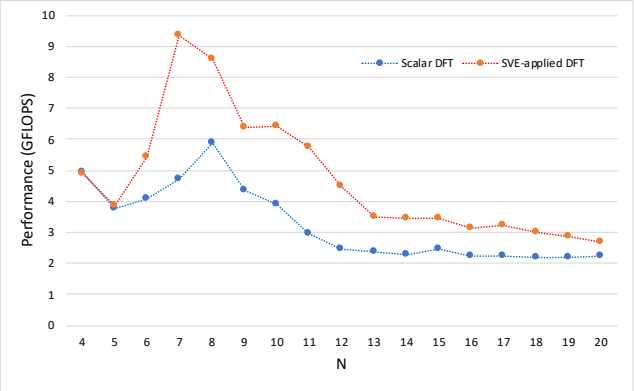

Figure 9: Evaluation of computational performance

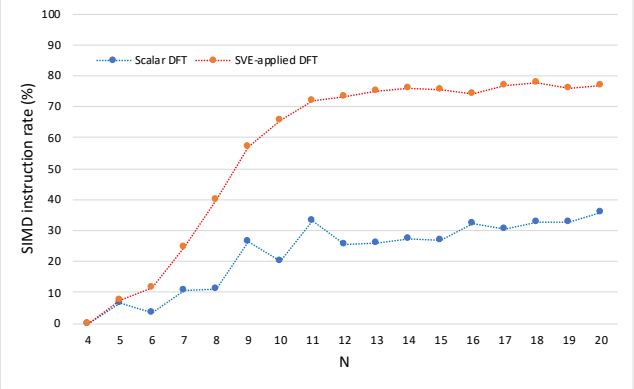

Figure 10: Evaluation of SIMD instruction rate

In the comparison of the computational performance, the SVE-applied DFT shows a performance that is up to 1.98-times better. The largest value was obtained for SIZE = 2^7. Except for the cases of SIZE=2^4 and 2^5, the overall performance of SVE-applied DFT is better than that of scalar DFT. The SIMD instruction rate of the SVE-applied DFT is up to 3.63-times higher than that of scalar DFT. The largest value was obtained for SIZE = 2^8, and is also more than 2-times higher than the SIMD instruction rate of scalar DFT, except for the cases of SIZE = 2^4, 2^5, and 2^{16}.

We also compared the computational performance of the proposed method with that of FFTE on the SVE [9], which uses SPIRAL to generate DFT kernels using a different approach. We used ffte-7.0-spiral.tgz (updated 2020-08-20), which is available at http://www.ffte.jp/ (accessed: 2021-01-01). The input to SPIRAL to generate a program referred to as SVE-applied DFT2 is shown in Fig. 11. The degree of loop unrolling was automatically tuned to improve the performance. The results are shown in Fig. 12.

The computational performance of our approach was up to 11.0 times better than that of FFTE on the SVE. The largest value was obtained for SIZE = 2^4. Except for SIZE = 2^6 and 2^{12}, the performance was higher than that of FFTE on the SVE. As one of the reasons for this result the SIMD instruction rate of our approach is usually higher than that of the FFTE on the SVE, as shown in Fig. 13. Although a higher SIMD instruction rate does not necessarily indicate a better computational performance, it is an important measure of the computational efficiency. In addition, it is important to note here that a simple comparison is not possible because the FFTE on the SVE works on DFTs of any size, whereas our approach generates code that focuses on DFTs of a specific size.

```
1  SIZE := 2^N;
2  opts := SpiralDefaults;
3  opts.target := rec();
4  opts.target.name := "flow-fx-sve";
5  transform := DFT(SIZE, -1);
6  best := DP(transform,
7      rec(globalUnrolling := true,
8        globalUnrollingMax := 1024), opts);
9  ruletree := best[1].ruletree;
10 icode := CodeRuleTree(ruletree, opts);
11 PrintCode("dft", icode, opts);
```

Figure 11: Inputs used to generate SVE-applied DFT2 program

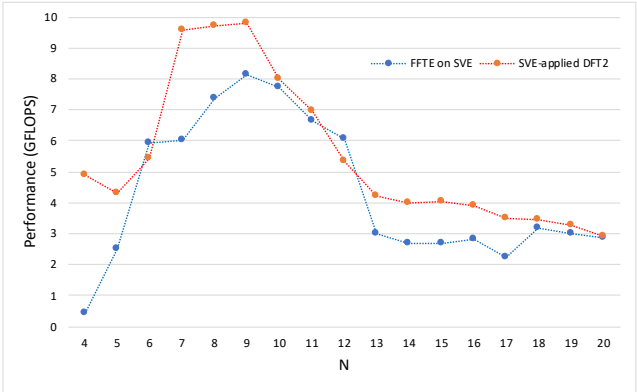

Figure 12: Comparison of computational performance with FFTE on SVE

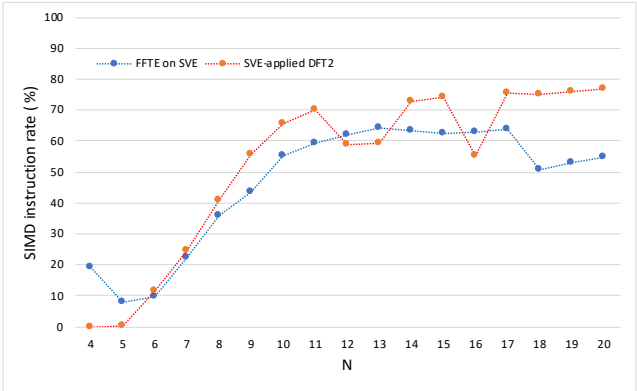

Figure 13: Comparison of SIMD instruction rate with FFTE on SVE

6. Conclusion

We proposed an adaptation of the A64 scalable vector extension for SPIRAL to generate DFT implementations. Using the Flow supercomputer at the Information Technology Center, Nagoya

ⓒ2021 Information Processing Society of Japan

University, we evaluated the performance of programs generated using standard SPIRAL methods (scalar DFT) and our approach (SVE-applied DFT). The SVE-applied DFT codes are up to 1.98-times faster than the scalar DFT code, with up to a 3.63-times higher SIMD instruction rate. In the comparison of the computational performance with the FFTE on the SVE, which generated DFT kernels with the SVE using different approaches available in SPIRAL, the results showed that SVE-applied DFT is up to 11.0-times faster.

In this study, SVE was applied using Python scripts. However, if we can implement the function to generate the SVE codes with the same process as fixed-length vector SIMD instructions such as Intel's AVX, we will be able to generate further optimized SVE codes using SPIRAL. Further studies are needed in order to achieve this function.

Acknowledgments

This work was supported by JSPS KAKENHI (Grant No. JP19H05662).

Reference

[1] Ken Naono, Keita Teranishi, John Cavazos, Reiji Suda (Eds.). Software Automatic Tuning: From concepts to State-of-the-Art Results. 2010, 3-15.

[2] Frigo, M., S. G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. Proceedings of the International Conference on Acoustics, Speech, and Signal Processing vol. 3 (1998), 1381-1384.

[3] R. Clint Whaley, Antoine Petitet, Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS Project. Parallel Computing vol. 27 issue 1-2 (2001), 3-35.

[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Una-May O'Reilly, Saman Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. in 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT) (2014), 303-315.

[5] Hiroyuki Takizawa, Shoichi Hirasawa, Yasuharu Hayashi, Ryusuke Egawa, Hiroaki Kobayashi. Xevolver: An XML-based Code Translation Framework for Supporting HPC Application Migration. IEEE International Conference on High Performance Computing (HiPC) (2014), 1-11.

[6] Takahiro Katagiri, Kenji Kise, Hiroki Honda, and Toshitsugu Yuba. FIBER: A Framework of Installation, Before Execution-invocation, and Run-time Optimization Layers for Auto-tuning Software. UEC-IS-2003-3.

[7] 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, Nicholas Rizzolo. SPIRAL: Code Generation for DSP Transforms. Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" vol. 93 no. 2 (2005), 232-275.

[8] Franz Franchetti, Tze-Meng Low, Thom Popovici, Richard Veras, Daniele G. Spampinato, Jeremy Johnson, Markus Püschel, James C. Hoe and José M. F. Moura. SPIRAL: Extreme Performance Portability. in Proc. of the IEEE, special issue on "From High Level Specification to High Performance Code" vol. 106 no. 11 (2018).

[9] Daisuke Takahashi, Franz Franchetti. FFTE on SVE: SPIRAL-Generated Kernels. International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia) (2020), 114-122.

[10] James W. Cooley, John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. Math. Comp. 19 (1965), 297-301.

[11] Franz Franchetti, Markus Püschel. Encyclopedia of Parallel Computing. Boston, MA: Springer, chapter "Fast Fourier Transform" (2011).

[12] "Arm® Architecture Reference Manual Supplement The Scalable Vector Extension (SVE), for Armv8-A", http://developer.arm.com/-/media/Files/DDI0584A_h_SVE.zip, (accessed 2021-01-01).

[13] "What is the Scalable Vector Extension? | Porting and Optimizing HPC Applications for Arm SVE Version 2.1", https://developer.arm.com/documentation/101726/0210/Learn-about-the-Scalable-Vector-Extension--SVE-/What-is-the-Scalable-Vector-Extension-, (accessed 2021-01-01).

[14] "Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile", https://developer.arm.com/documentation/ddi0487/fc, (accessed 2021-01-01).

[15] "SIMD ISAs | NEON – Arm Developer", https://developer.arm.com/architectures/instruction-sets/simd-isas/neon, (accessed 2021-01-01).

[16] "Introduction to Intel Advanced Vector Extensions", https://software.intel.com/content/dam/develop/external/us/en/documents/intro-to-intel-avx-183287.pdf, (accessed 2021-01-01).

[17] "Vector Length Agnostic (VLA) programming | Porting and Optimizing HPC Applications for Arm SVE Version 2.1", https://developer.arm.com/documentation/101726/0200/SVE-Vector-Length-Agnostic-programming/Vector-Length-Agnostic--VLA--programming, (accessed 2021-01-01).

[18] "Arm C Language Extensions for SVE version 00bet6", https://developer.arm.com/documentation/100987/0000/, (accessed 2021-01-01).

ⓒ2021 Information Processing Society of Japan