An Auto-tuning with Adaptation of A64 Scalable Vector Extension for SPIRAL

Naruya Kitai[1], Daisuke Takahashi[2], Franz Franchetti[3], Takahiro Katagiri[4], Satoshi Ohshima[4], Toru Nagai[4]

1 Graduate School of Informatics, Nagoya University,

2 University of Tsukuba, 3 Carnegie Mellon University,

4 Information Technology Center, Nagoya University

The Sixteenth International Workshop on Automatic Performance Tuning (iWAPT2021), May 21, 2021, Online
(35th IEEE International Parallel & Distributed Processing Symposium)

Outline

- Background
- Aim of This Study
- Spiral Code Generation System
- Scalable Vector Extension
- Adaptation of SVE
- Performance Evaluation
- Summary

Outline

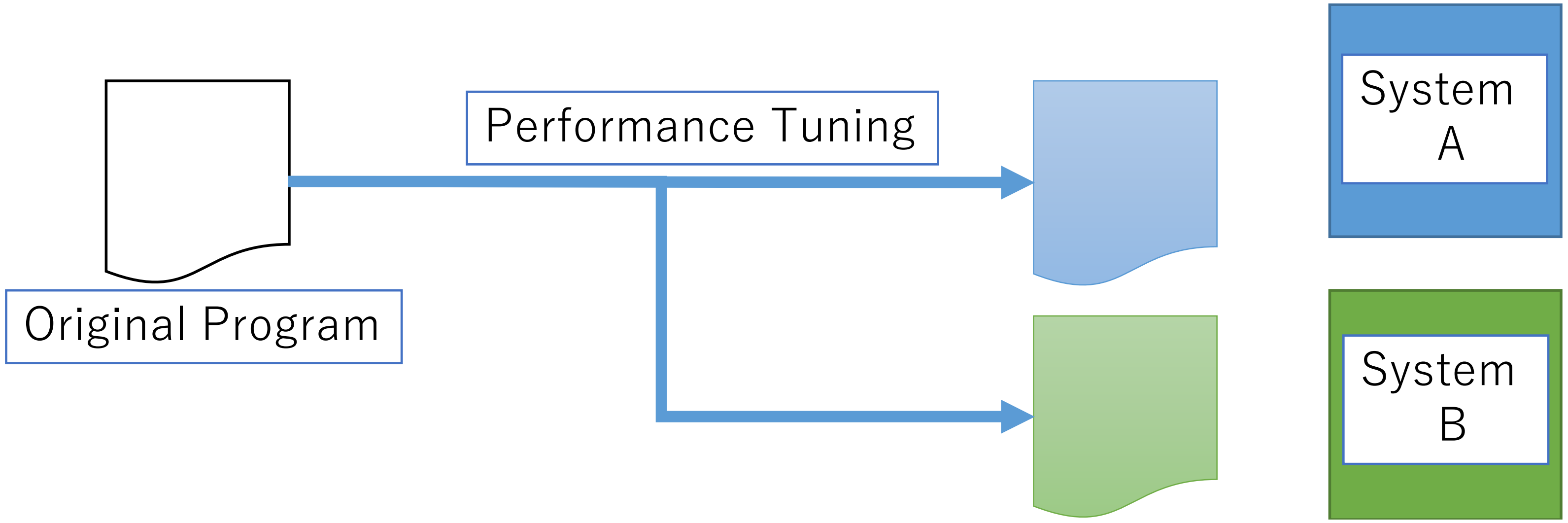
- Background
- Aim of This Study
- Spiral Code Generation System
- Scalable Vector Extension
- Adaptation of SVE
- Performance Evaluation
- Summary

Background（1/2)

- Compute architectures are getting complex…
  - Hierarchical memories for CPUs.
  - Available or unavailable for accelerator GPUs.
- To establish high performance, it requires dedicated tuning to target computers for numerical software.
  - Special knowledge for computer hardware is required to do performance tuning. It is a waste-of-time.
- No performance portability;
  If it is installed on different computer environment.

Background (2/2)

- To establish performance portability, Auto-tuning*1 (AT) is one of promising ways.
- Several code generation systems for numerical kernels are now developing.



*1 Takahiro Katagiri and Daisuke Takahashi: Japanese Autotuning Research: Autotuning Languages and FFT, Proceedings of the IEEE, Vol. 106, No. 11, pp. 2056-2067 (2018).

Outline

- Background
- Aim of This Study
- Spiral Code Generation System
- Scalable Vector Extension
- Adaptation of SVE
- Performance Evaluation
- Summary

Aim of This Study

- Target: Propose a new approach of code generation on Spiral*2 for numerical algorithms.
- Aim: Add the following new function for code generation:

Adaptation of SVE instructions

to the generated code

- Show effectiveness of the function for the generated codes with the proposed approach by performance evaluation.
  - Target numerical computation is DFT (Discrete Fourier Transform) in this study.

*2 Franz Franchetti et al., “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).

Outline

- Background
- Aim of This Study
- Spiral Code Generation System
- Scalable Vector Extension
- Adaptation of SVE
- Performance Evaluation
- Summary

Spiral Code Generation System

- **Goal:** Increase ability of automation and optimization to software/hardware development of signal processing algorithms, or other numerical kernels.
- Establish an automatic code generation for programs/libraires of signal processing.
  - Target: Discrete Fourier Transform (DFT) and similar kernels of DFT.
- Increase performance portability for software.
  - By AT, it can select the best algorithm by measuring target performance.
- Describe generated code with GAP3*3
  - GAP: Groups, Algorithms, Programming
    - a system for computational discrete algebra

*3 Martin Schönert et.al. GAP -- Groups, Algorithms, and Programming -- version 3 release 4 patchlevel 4. Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1997

DFT Description for Spiral (1/3)

- Definition of DFT with sampling with *n* points: $x_0, x_1, \dots, x_{n-1}$

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

- Definition of DFT for Spiral with *n* dimensional vector is as

$$DFT_n \colon \mathbb{C}^n \to \mathbb{C}^n;\ x \longmapsto \left[\omega_n^{ij}\right]_{i,j} x \qquad \dots (2)$$

- The formula (1) can be described with the formula (2) as a matrix-vector multiplication form, such as:

$$y = DFT_n x \qquad \dots (3)$$

DFT Description for Spiral (2/3)

- By decomposition of the formula (2), FFT algorithm can be obtained.
- Cooley-Tukey FFT decomposition algorithm is:

$$DFT_n \rightarrow (DFT_k \otimes I_m) T^n_m (I_k \otimes DFT_m) L^n_k \ , n = km, \quad \cdots (4)$$

where, definition of each matrix and operator is as follows:

Kronecker product

$$A \otimes B = [a_{k,l} B], \text{for } A = [a_{k,l}]$$

Twiddle matrix

$$T^n_m = [t_{k,l}] \ , t_{k,l} = \begin{cases} \omega^k_m & (k = l) \\ 0 & (k \neq l) \end{cases}$$

Stride permutation matrix

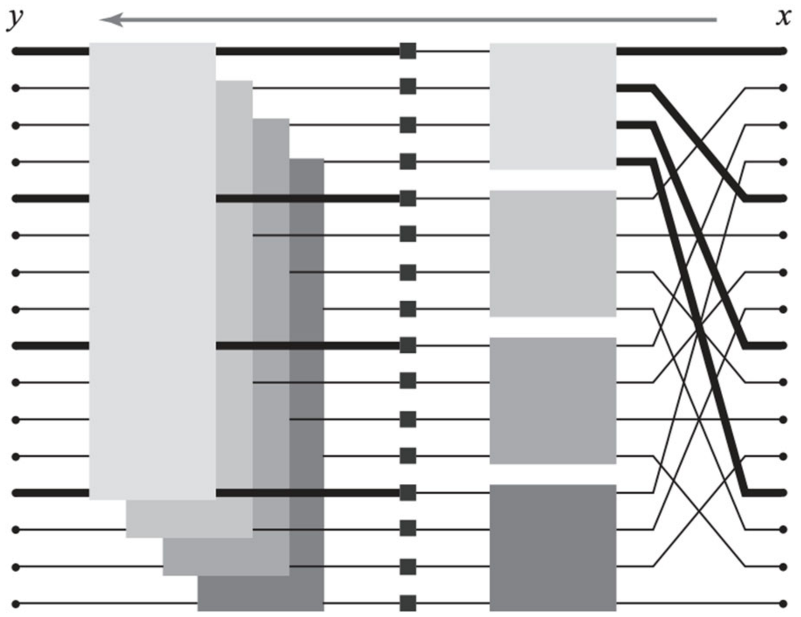
$$L^n_k = [l_{i,j}] \ , l_{i,j} = \begin{cases} 1 \ (k(i \bmod m) + \lfloor j/k \rfloor = ni + j) \\ 0 \ otherwise \end{cases}$$

DFT Description for Spiral（3/3)

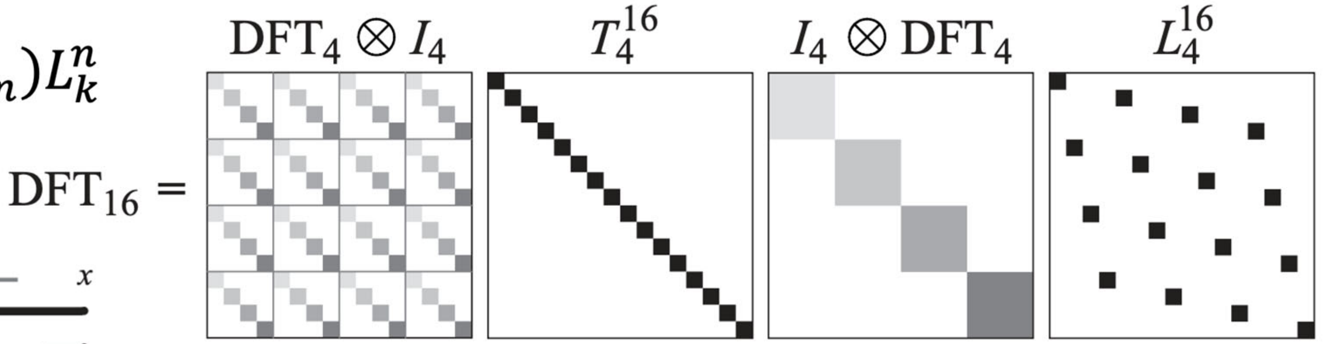
$$DFT_n \rightarrow (DFT_k \otimes I_m) T_m^n (I_k \otimes DFT_m) L_k^n$$

block parallelism (n blocks): $I_n \otimes A$

vector parallelism (n-way): $A \otimes I_n$



(Source: *4)



Source : *4

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

$$L_k^n = [l_{i,j}],$$

$$l_{i,j} = \begin{cases} 1 & (k(i \bmod m) + \lfloor j/k \rfloor = ni + j) \\ 0 & otherwise \end{cases}$$

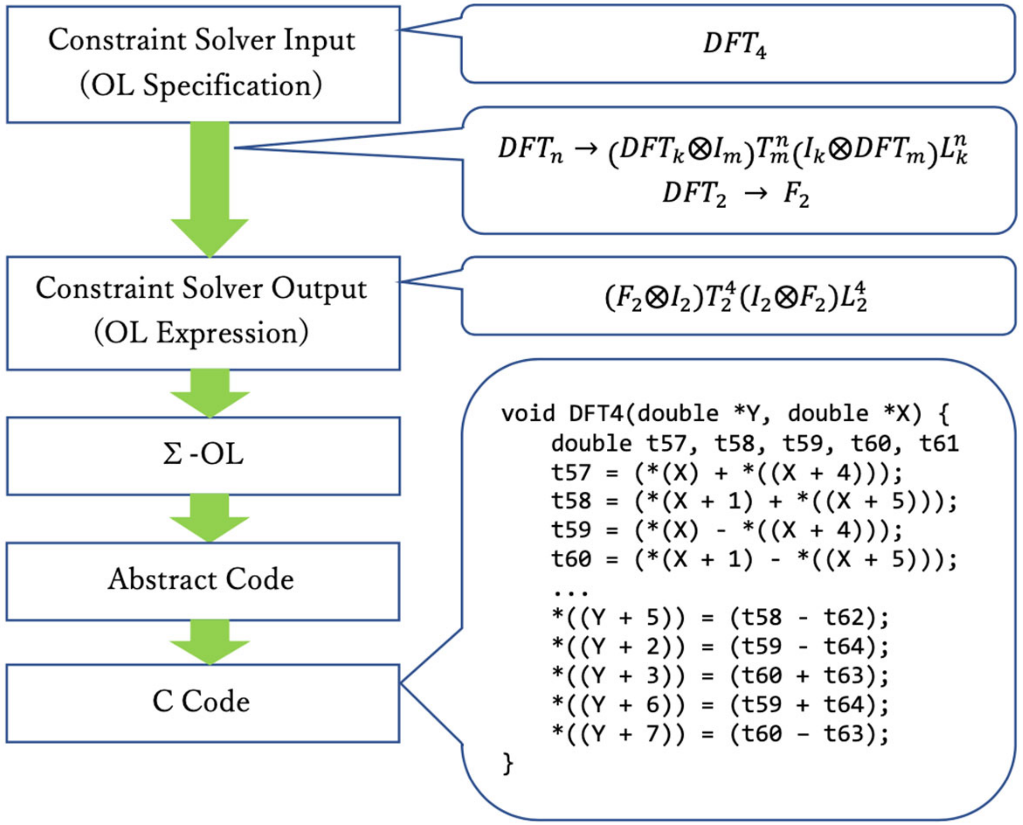
*4 Franz Franchetti, Markus Püschel. Encyclopedia of Parallel Computing. Boston, MA: Springer, chapter “Fast Fourier Transform” (2011).

*5 Franchetti F., de Mesmay F., McFarlin D., Püschel M. (2009) Operator Language: A Program Generation Framework for Fast Kernels. In: Taha W.M. (eds) Domain-Specific Languages. DSL 2009. Lecture Notes in *1 Computer Science, vol 5658. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-03034-5_18

Flow of Code Generation for Inside of Spiral

- An Example:
  C code generation DFT_4

1. **OL Specification**
   - Specify target of code generation.
   - OL; Operator Language *5
2. **OL Expression**
   - Decompose the problem according to rules.
   - Tree structure, named "Rule tree."
3. **C Code**
   - Output a form of function.

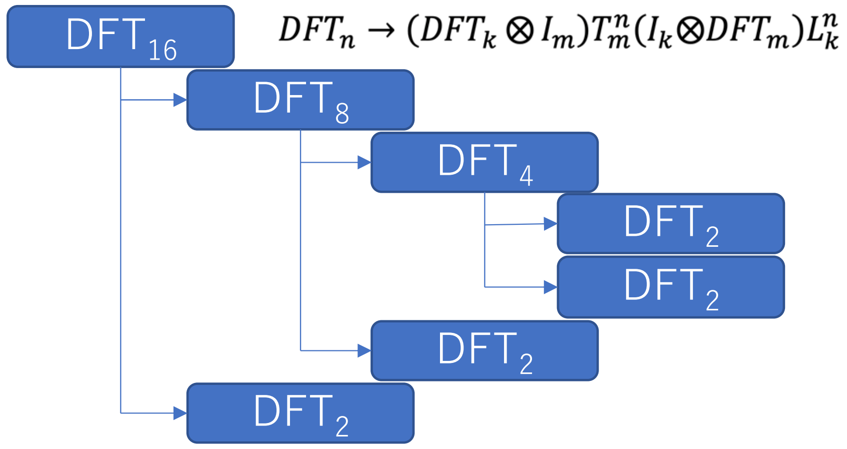


Constraint Solver Input (OL Specification)
DFT_4
$DFT_n \rightarrow (DFT_k \otimes I_m) T^n_m (I_k \otimes DFT_m) L^n_k$
$DFT_2 \rightarrow F_2$
Constraint Solver Output (OL Expression)
$(F_2 \otimes I_2) T^4_2 (I_2 \otimes F_2) L^4_2$
Σ-OL
Abstract Code
C Code

```
void DFT4(double *Y, double *X) {
    double t57, t58, t59, t60, t61
    t57 = (*(X) + *((X + 4)));
    t58 = (*(X + 1) + *((X + 5)));
    t59 = (*(X) - *((X + 4)));
    t60 = (*(X + 1) - *((X + 5)));
    ...
    *((Y + 5)) = (t58 - t62);
    *((Y + 2)) = (t59 - t64);
    *((Y + 3)) = (t60 + t63);
    *((Y + 6)) = (t59 + t64);
    *((Y + 7)) = (t60 - t63);
}
```

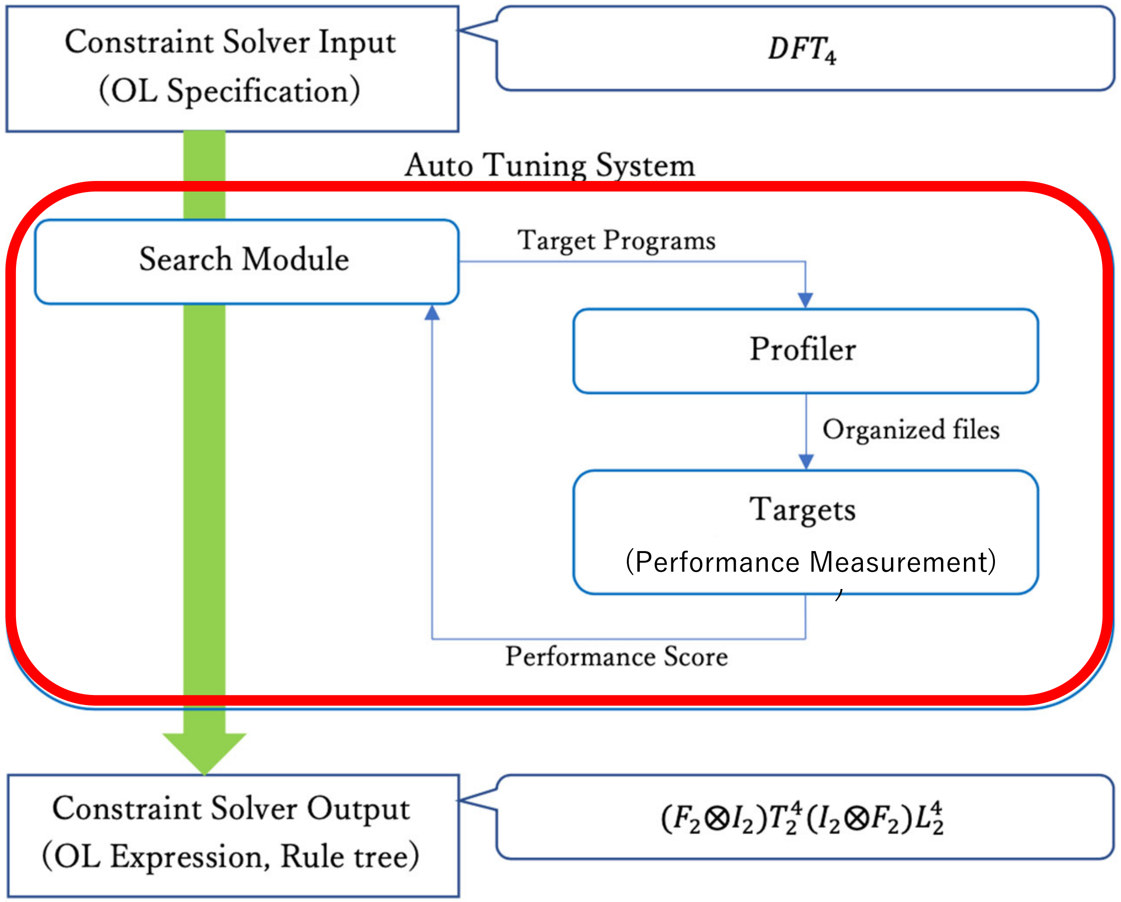
Flow of Auto-Tuning

- The best rule tree is determined by iterative measurement of search module.



DFT$_{16}$
$DFT_n \rightarrow (DFT_k \otimes I_m)T^n_m(I_k \otimes DFT_m)L^n_k$
DFT$_8$
DFT$_4$
DFT$_2$
DFT$_2$
DFT$_2$
DFT$_2$

An example of rule tree for DFT$_{16}$ Rule tree.

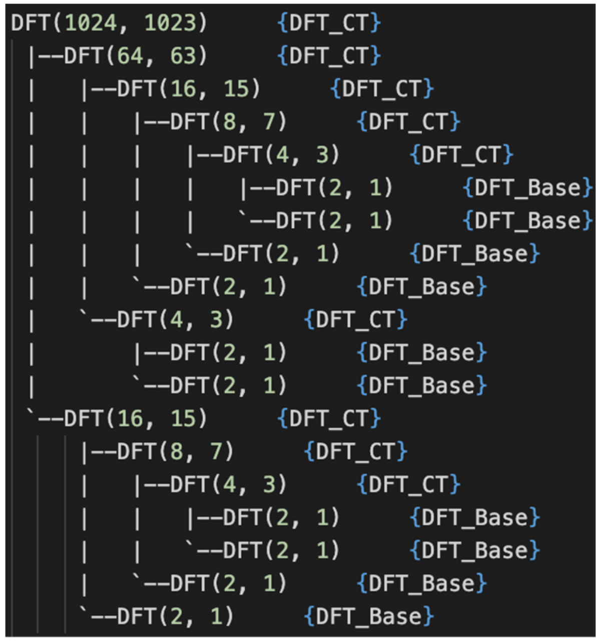


Constraint Solver Input (OL Specification)
DFT_4
Auto Tuning System
Search Module
Target Programs
Profiler
Organized files
Targets (Performance Measurement)
Performance Score
Constraint Solver Output (OL Expression, Rule tree)
$(F_2 \otimes I_2)T^4_2(I_2 \otimes F_2)L^4_2$

Determined Parameters by AT

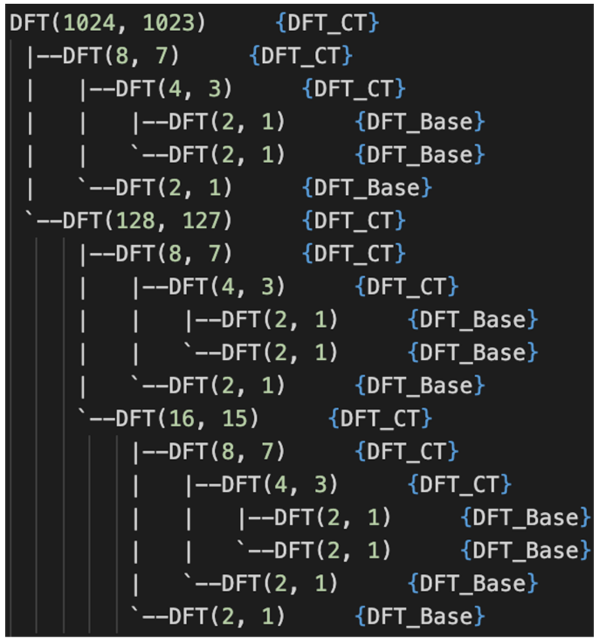
Example of
tunable parameters

Rule tree Number: 2669

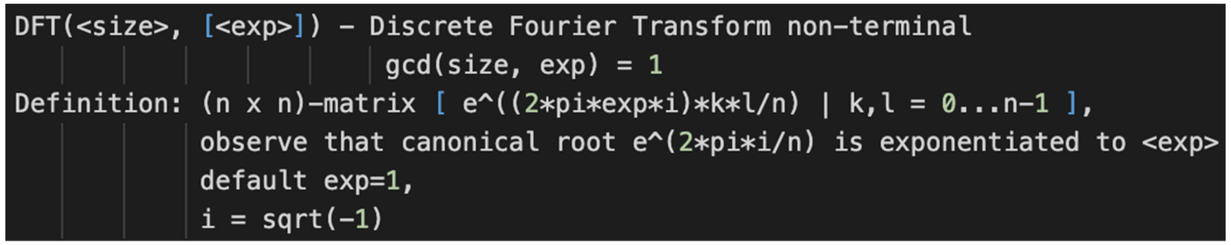


```
DFT(1024, 1023)      {DFT_CT}
 |--DFT(64, 63)      {DFT_CT}
 |    |--DFT(16, 15)      {DFT_CT}
 |    |    |--DFT(8, 7)      {DFT_CT}
 |    |    |    |--DFT(4, 3)      {DFT_CT}
 |    |    |    |    |--DFT(2, 1)      {DFT_Base}
 |    |    |    |    `--DFT(2, 1)      {DFT_Base}
 |    |    |    `--DFT(2, 1)      {DFT_Base}
 |    |    `--DFT(2, 1)      {DFT_Base}
 |    `--DFT(4, 3)      {DFT_CT}
 |         |--DFT(2, 1)      {DFT_Base}
 |         `--DFT(2, 1)      {DFT_Base}
 `--DFT(16, 15)      {DFT_CT}
      |--DFT(8, 7)      {DFT_CT}
      |    |--DFT(4, 3)      {DFT_CT}
      |    |    |--DFT(2, 1)      {DFT_Base}
      |    |    `--DFT(2, 1)      {DFT_Base}
      |    `--DFT(2, 1)      {DFT_Base}
      `--DFT(2, 1)      {DFT_Base}
```

Rule tree Number: 2057



```
DFT(1024, 1023)      {DFT_CT}
 |--DFT(8, 7)      {DFT_CT}
 |    |--DFT(4, 3)      {DFT_CT}
 |    |    |--DFT(2, 1)      {DFT_Base}
 |    |    `--DFT(2, 1)      {DFT_Base}
 |    `--DFT(2, 1)      {DFT_Base}
 `--DFT(128, 127)      {DFT_CT}
      |--DFT(8, 7)      {DFT_CT}
      |    |--DFT(4, 3)      {DFT_CT}
      |    |    |--DFT(2, 1)      {DFT_Base}
      |    |    `--DFT(2, 1)      {DFT_Base}
      |    `--DFT(2, 1)      {DFT_Base}
      `--DFT(16, 15)      {DFT_CT}
           |--DFT(8, 7)      {DFT_CT}
           |    |--DFT(4, 3)      {DFT_CT}
           |    |    |--DFT(2, 1)      {DFT_Base}
           |    |    `--DFT(2, 1)      {DFT_Base}
           |    `--DFT(2, 1)      {DFT_Base}
           `--DFT(2, 1)      {DFT_Base}
```

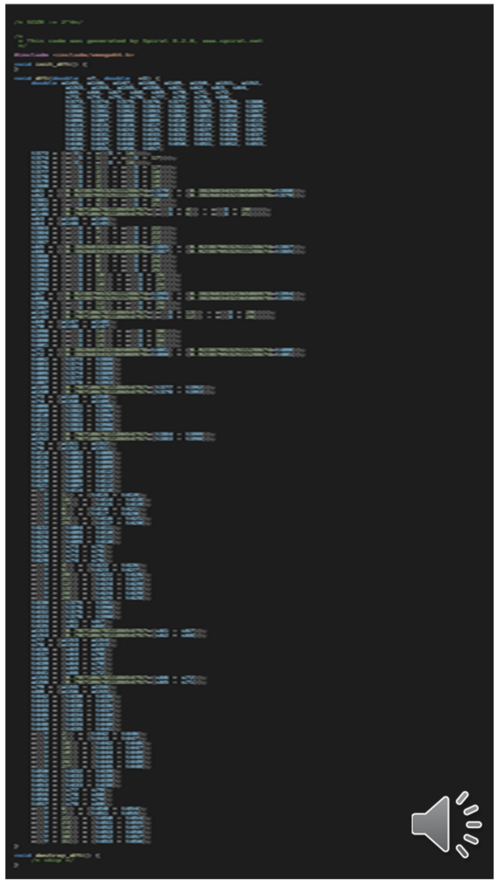


```
DFT(<size>, [<exp>]) - Discrete Fourier Transform non-terminal
                          gcd(size, exp) = 1
Definition: (n x n)-matrix [ e^((2*pi*exp*i)*k*l/n) | k,l = 0...n-1 ],
            observe that canonical root e^(2*pi*i/n) is exponentiated to <exp>
            default exp=1,
            i = sqrt(-1)
```

Unrolling := 8

[illegible]

Unrolling := 1024



[illegible]

Outline

- Background
- Aim of This Study
- Spiral Code Generation System
- Scalable Vector Extension
- Adaptation of SVE
- Performance Evaluation
- Summary

Scalable Vector Extension (SVE)

- Extension of Options for Armv8-A ISA (Only for AArch64)
  - A SIMD extension for HPC.
  - This is unavailable for CPU/SoC, such as Cortex-M3, Apple M1.
- Programming Model of Vector Length Agnostic (VLA)
  - Vector length is not fixed for phase of program description.
  - In phase of compilation or execution, physical vector length on target processor is set.
  - In the A64FX, such as the Supercomputer “Fugaku” (RIKEN), or the Supercomputer “Flow” (Nagoya U.), vector length with 512 bits is implemented .
- Control by Predicate Registers
  - It is not needed for explicit description of number of iterations and end process for loops.
  - Judgement of execution or not is done by elements of vector by assignment of True / False.

Code example by SVE
(z[i] = x[i] + y[i])

```
int64_t i = 0;
svbool_t pg = svwhilelt_b64(i, N);     // predicate: Judge elements of vectors
                                       //   to execute. The elements, which exceed N,
do {                                   //   are set to false.
    svfloat64_t x_sve = svld1(pg, &x[i]); // x_sve = x[i:i + simd_length]
    svfloat64_t y_sve = svld1(pg, &y[i]);
    svfloat64_t z_sve = svadd_x(pg, x_sve, y_sve);
                                   // z_sve[j] = x_sve[j] + y_sve[j] ( 0 <= j < simd_length )
    svst1(pg, &z[i], z_sve);       // z[i:i + simd_length] = z_sve
    i += svcntd();                 // i += simd_length
    pg = svwhilelt_b64(i, N);
} while(svptest_any(svptrue_b64(), pg));   // If pg is true, it continues.
```

Outline

- Background
- Aim of This Study
- Spiral Code Generation System
- Scalable Vector Extension
- Adaptation of SVE
- Performance Evaluation
- Summary

Adaptation of SVE Instructions

- By adaptation of SVE instructions, better code can be generated.

DFT Libraries	Intel SSE	Intel AVX	Arm NEON	VSX (PowerPC)	Arm SVE
Spiral	✓	✓	✓	✓	△[*6]
FFTW	✓	✓	✓	✓	
Spiral (The proposal approach)	✓	✓	✓	✓	✓

*6 Spiral has function of code generation by SVE for computation kernel of FFTE[*7], but it is dedicated function for FFTE.

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

Adaptation of SVE on Spiral

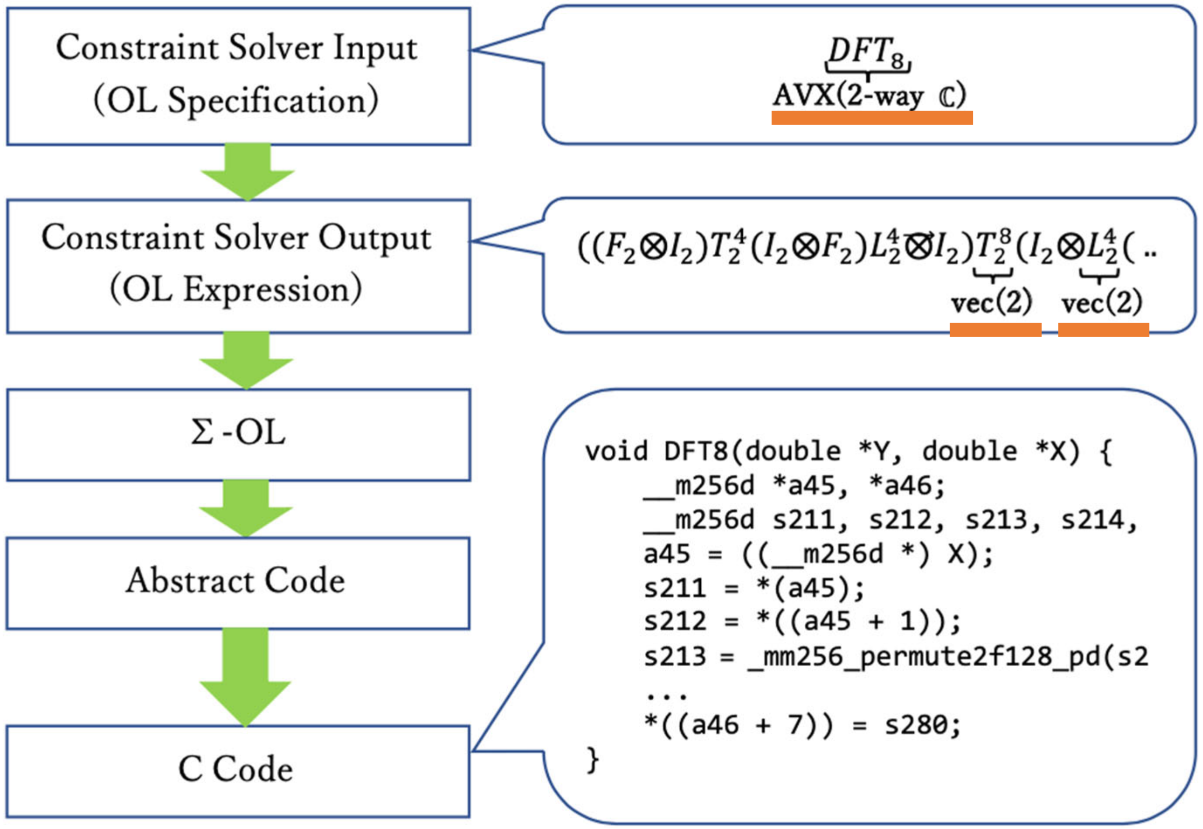
- Spiral supports extended SIMD instructions with fixed vector length, such as Intel AVX, or Arm NEON for code generation.
  - Effective code by the Spiral code generator is based on fixed vector length.
- SVE is non-fixed length of vector with VLA programming model, hence structure of vectorization is very different to conventional SIMD.

- Conventional Spiral approach cannot be utilized.
- In this research, we made a script which can translate generated scaler C codes by Spiral to effective SVE codes.

A Flow of Code Generation: Adaptation of AVX (Convectional approach)

1. Specify DFT$_8$ with AVX as OL.
2. Adaptation of a rule for vectorization.
3. Generate functions with AVX.

Adapt a rule for data structure of vectorization to generate vectored DFT.



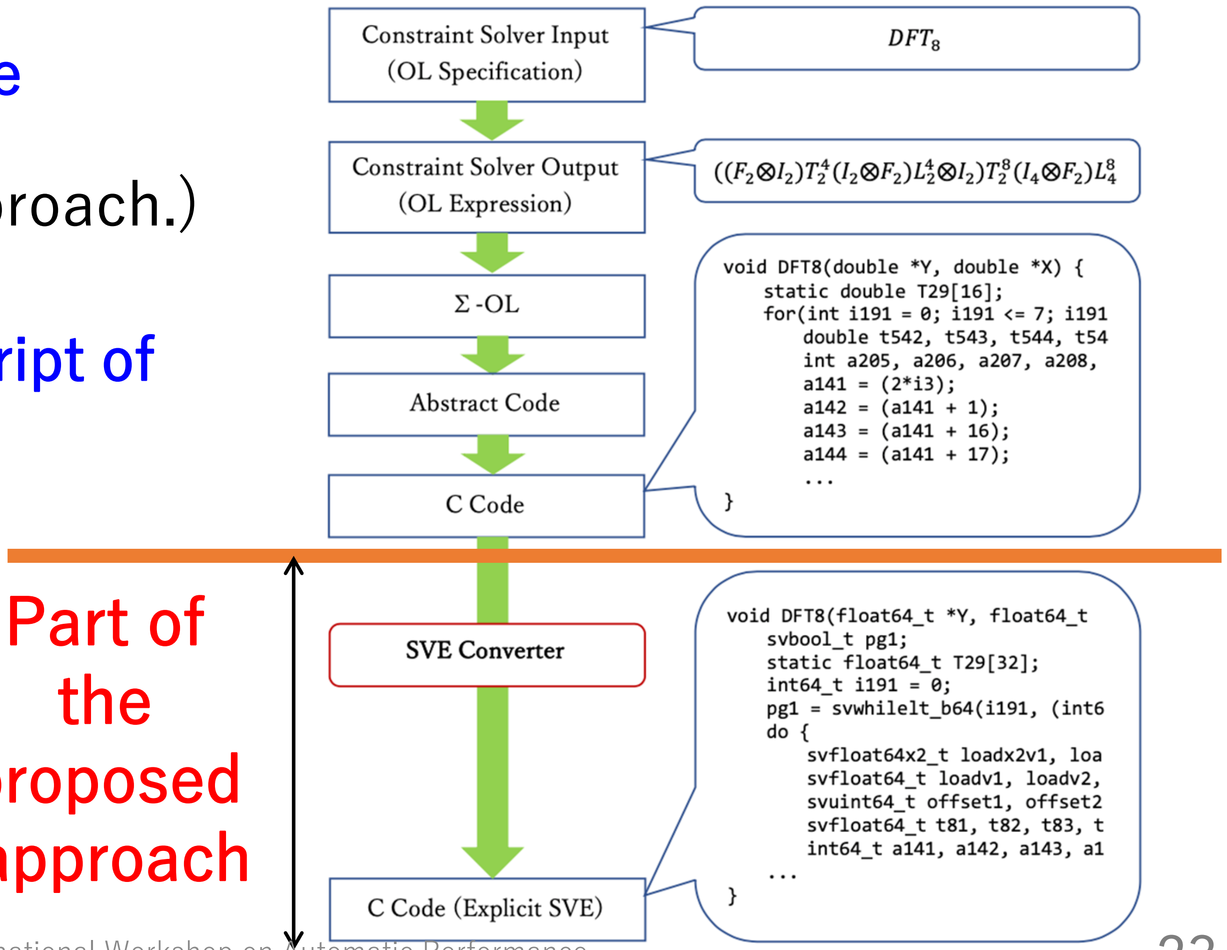
Constraint Solver Input (OL Specification)
$\underbrace{DFT_8}_{\text{AVX(2-way } \mathbb{C})}$
Constraint Solver Output (OL Expression)
$((F_2 \otimes I_2) T_2^4 (I_2 \otimes F_2) L_2^4 \otimes I_2) \underbrace{T_2^8}_{\mathbf{vec}(2)} (I_2 \otimes \underbrace{L_2^4}_{\mathbf{vec}(2)} (..$
Σ-OL
Abstract Code
C Code

```
void DFT8(double *Y, double *X) {
    __m256d *a45, *a46;
    __m256d s211, s212, s213, s214,
    a45 = ((__m256d *) X);
    s211 = *(a45);
    s212 = *((a45 + 1));
    s213 = _mm256_permute2f128_pd(s2
    ...
    *((a46 + 7)) = s280;
}
```

A Flow of Code Generation : Adaptation of SVE (proposed approach)

1. Generate C code by same procedure of scalar DFT. (Conventional Spiral approach.)
2. Generate C code with vectorization by using script of SVE translation.

Due to different data structure to conventional SIMD, we use procedure of scalar DFT in intermediate phase. Then, vectorization is performed in final phase.

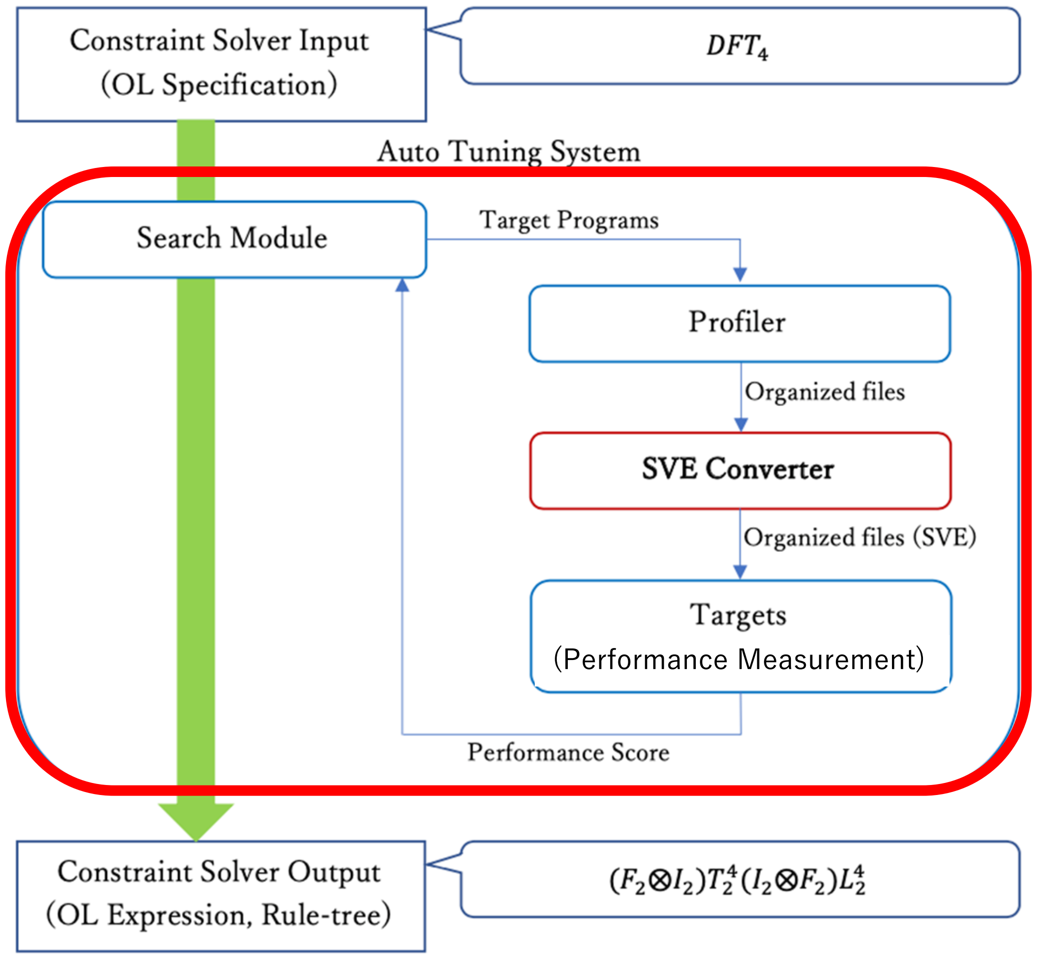

Constraint Solver Input (OL Specification)
DFT_8
Constraint Solver Output (OL Expression)
$((F_2 \otimes I_2) T_2^4 (I_2 \otimes F_2) L_2^4 \otimes I_2) T_2^8 (I_4 \otimes F_2) L_4^8$
Σ-OL
Abstract Code
C Code
Part of the proposed approach
SVE Converter
C Code (Explicit SVE)

```
void DFT8(double *Y, double *X) {
    static double T29[16];
    for(int i191 = 0; i191 <= 7; i191
        double t542, t543, t544, t54
        int a205, a206, a207, a208,
        a141 = (2*i3);
        a142 = (a141 + 1);
        a143 = (a141 + 16);
        a144 = (a141 + 17);
        ...
}
```

```
void DFT8(float64_t *Y, float64_t
    svbool_t pg1;
    static float64_t T29[32];
    int64_t i191 = 0;
    pg1 = svwhilelt_b64(i191, (int6
    do {
        svfloat64x2_t loadx2v1, loa
        svfloat64_t loadv1, loadv2,
        svuint64_t offset1, offset2
        svfloat64_t t81, t82, t83, t
        int64_t a141, a142, a143, a1
    ...
}
```

AT adaptation on the proposed approach

Rule tree is selected from code with SVE.



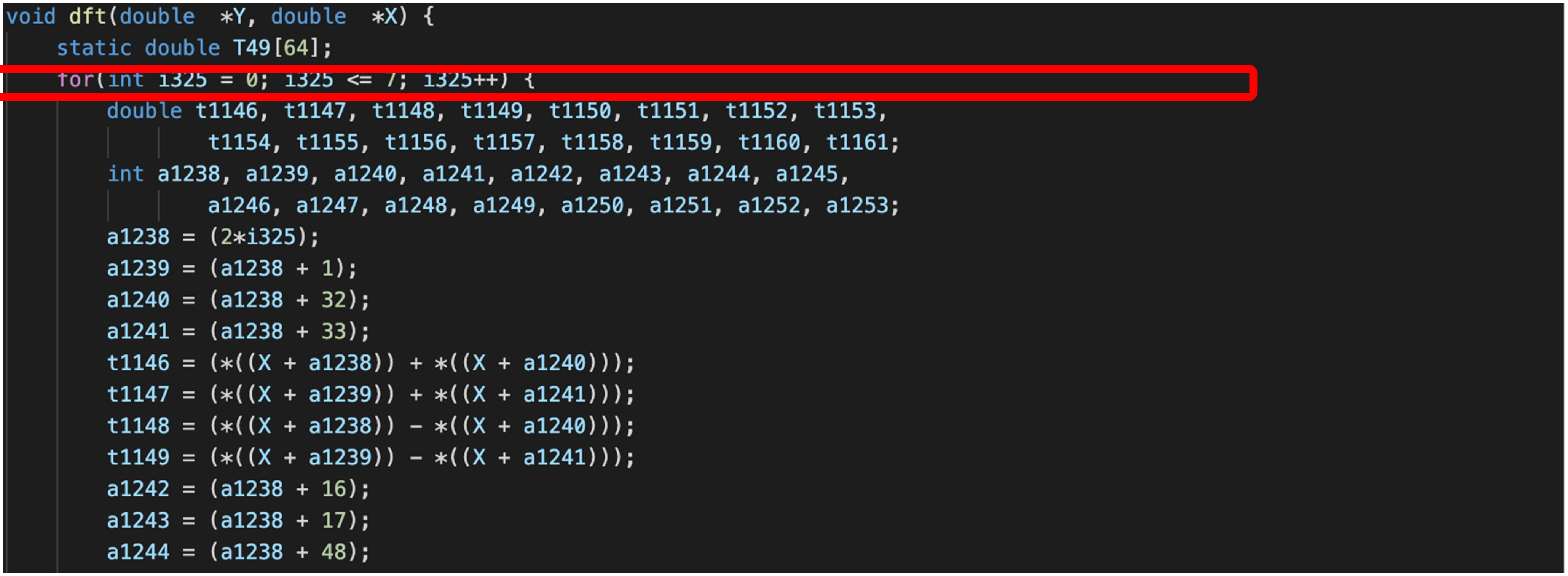
Constraint Solver Input (OL Specification)
DFT_4
Auto Tuning System
Search Module
Target Programs
Profiler
Organized files
SVE Converter
Organized files (SVE)
Targets (Performance Measurement)
Performance Score
Constraint Solver Output (OL Expression, Rule-tree)
$(F_2 \otimes I_2) T_2^4 (I_2 \otimes F_2) L_2^4$

Procedure of the SVE Translation Script

1. **Select the inner loop to do vectorization.**
   - SVE code is generated by using loop structure.
   - Set predicate vectors, loop termination judgements, and additions of loop variables by modifying the target vectorization loop.
2. **Apply SVE instructions for the target vectorization loop.**
   - Adaptation of SVE instructions is established for loads and stores of arrays, additions, substitutions, and multiplications of variables.
3. **Modify multiple loads to a load operation.**
   - Multiple load instructions without stores to variables are generated for scalar code in Spiral .
   - This phase removes these codes when it can vectorize the code.
4. **Apply an instruction of loads to multiple elements.**
   - Several instructions for SVE to load multiple elements, such as every 2-4 elements, with a load instruction is applied, if possible.

An Example of The SVE Translation Script（1/5）

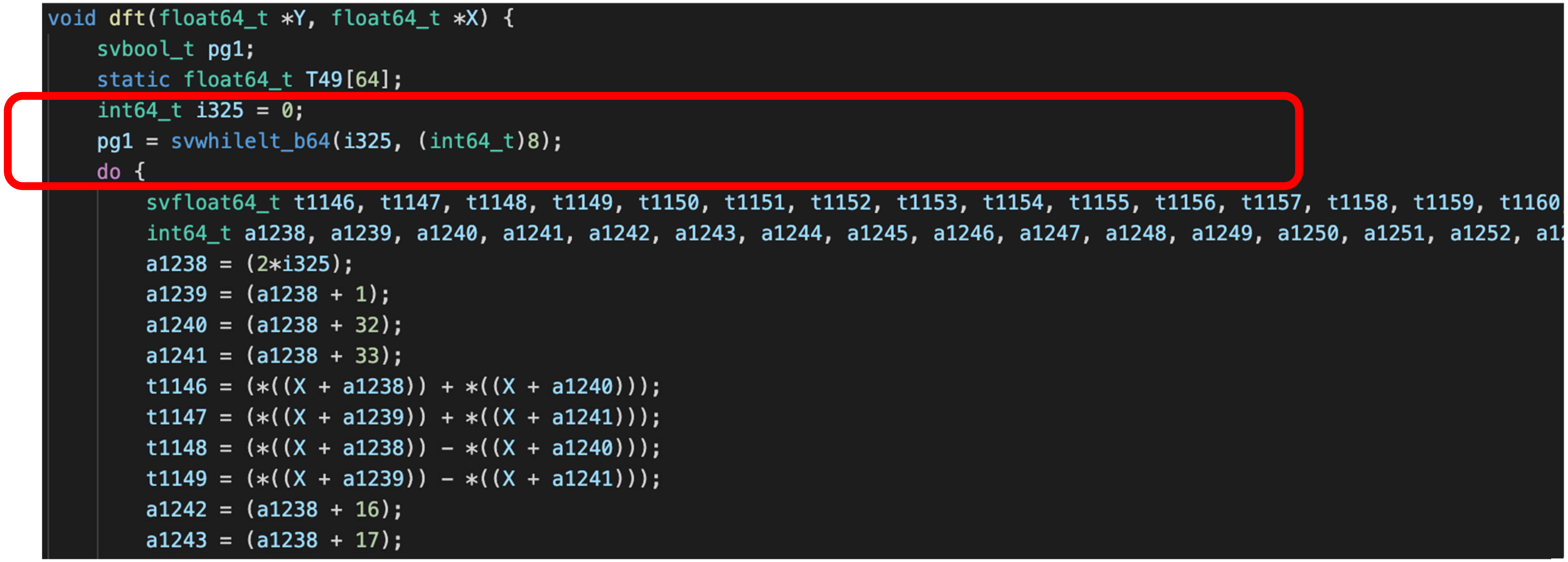
Before Translation



```
void dft(double  *Y, double  *X) {
    static double T49[64];
    for(int i325 = 0; i325 <= 7; i325++) {
        double t1146, t1147, t1148, t1149, t1150, t1151, t1152, t1153,
                t1154, t1155, t1156, t1157, t1158, t1159, t1160, t1161;
        int a1238, a1239, a1240, a1241, a1242, a1243, a1244, a1245,
                a1246, a1247, a1248, a1249, a1250, a1251, a1252, a1253;
        a1238 = (2*i325);
        a1239 = (a1238 + 1);
        a1240 = (a1238 + 32);
        a1241 = (a1238 + 33);
        t1146 = (*((X + a1238)) + *((X + a1240)));
        t1147 = (*((X + a1239)) + *((X + a1241)));
        t1148 = (*((X + a1238)) - *((X + a1240)));
        t1149 = (*((X + a1239)) - *((X + a1241)));
        a1242 = (a1238 + 16);
        a1243 = (a1238 + 17);
        a1244 = (a1238 + 48);
```

An Example of The SVE Translation Script（2/5）

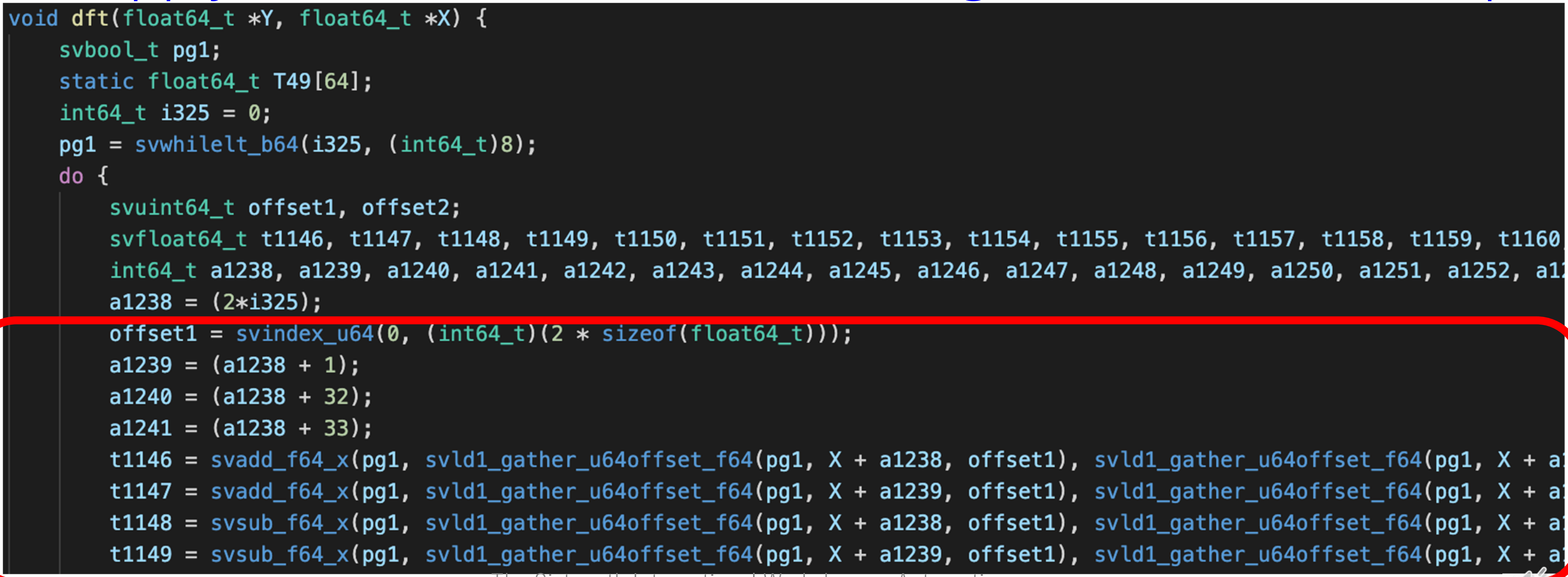
1. Select the inner loop to do vectorization.



```
void dft(float64_t *Y, float64_t *X) {
    svbool_t pg1;
    static float64_t T49[64];
    int64_t i325 = 0;
    pg1 = svwhilelt_b64(i325, (int64_t)8);
    do {
        svfloat64_t t1146, t1147, t1148, t1149, t1150, t1151, t1152, t1153, t1154, t1155, t1156, t1157, t1158, t1159, t1160
        int64_t a1238, a1239, a1240, a1241, a1242, a1243, a1244, a1245, a1246, a1247, a1248, a1249, a1250, a1251, a1252, a1
        a1238 = (2*i325);
        a1239 = (a1238 + 1);
        a1240 = (a1238 + 32);
        a1241 = (a1238 + 33);
        t1146 = (*((X + a1238)) + *((X + a1240)));
        t1147 = (*((X + a1239)) + *((X + a1241)));
        t1148 = (*((X + a1238)) - *((X + a1240)));
        t1149 = (*((X + a1239)) - *((X + a1241)));
        a1242 = (a1238 + 16);
        a1243 = (a1238 + 17);
```

An Example of The SVE Translation Script（3/5）

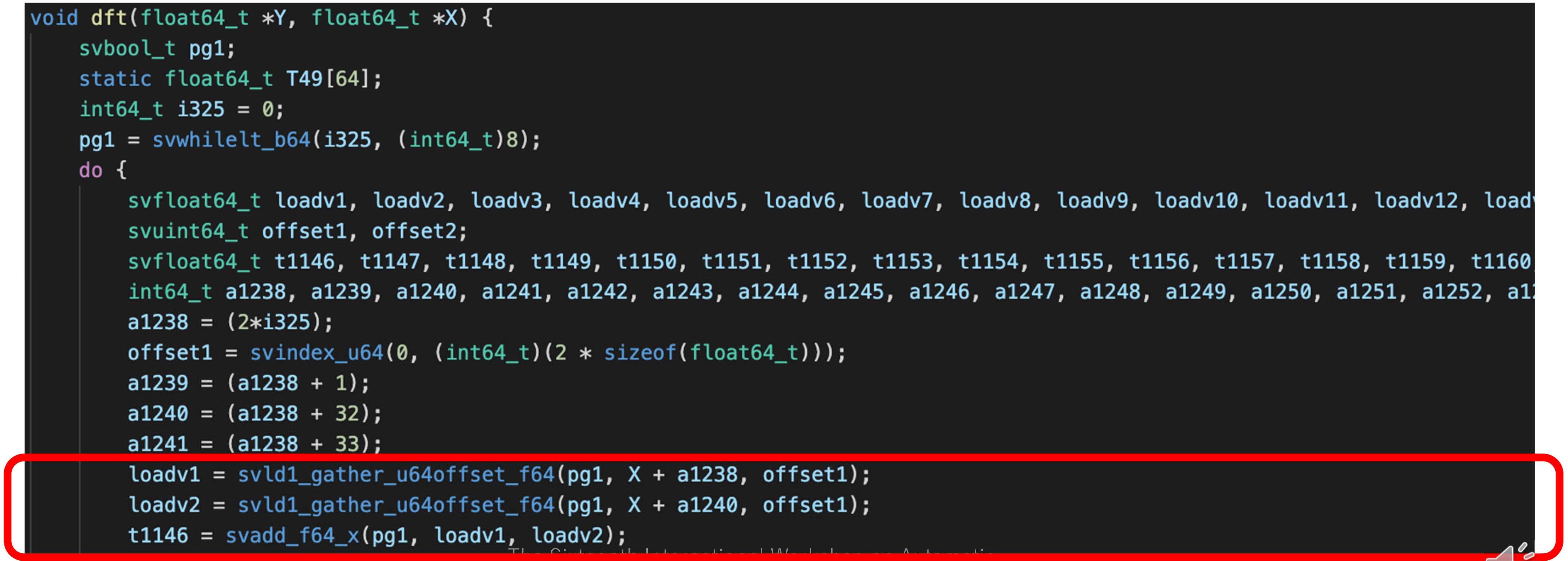
2. Apply SVE instructions for the target vectorization loop.



```
void dft(float64_t *Y, float64_t *X) {
    svbool_t pg1;
    static float64_t T49[64];
    int64_t i325 = 0;
    pg1 = svwhilelt_b64(i325, (int64_t)8);
    do {
        svuint64_t offset1, offset2;
        svfloat64_t t1146, t1147, t1148, t1149, t1150, t1151, t1152, t1153, t1154, t1155, t1156, t1157, t1158, t1159, t1160
        int64_t a1238, a1239, a1240, a1241, a1242, a1243, a1244, a1245, a1246, a1247, a1248, a1249, a1250, a1251, a1252, a1
        a1238 = (2*i325);
        offset1 = svindex_u64(0, (int64_t)(2 * sizeof(float64_t)));
        a1239 = (a1238 + 1);
        a1240 = (a1238 + 32);
        a1241 = (a1238 + 33);
        t1146 = svadd_f64_x(pg1, svld1_gather_u64offset_f64(pg1, X + a1238, offset1), svld1_gather_u64offset_f64(pg1, X + a
        t1147 = svadd_f64_x(pg1, svld1_gather_u64offset_f64(pg1, X + a1239, offset1), svld1_gather_u64offset_f64(pg1, X + a
        t1148 = svsub_f64_x(pg1, svld1_gather_u64offset_f64(pg1, X + a1238, offset1), svld1_gather_u64offset_f64(pg1, X + a
        t1149 = svsub_f64_x(pg1, svld1_gather_u64offset_f64(pg1, X + a1239, offset1), svld1_gather_u64offset_f64(pg1, X + a
```

An Example of The SVE Translation Script（4/5）

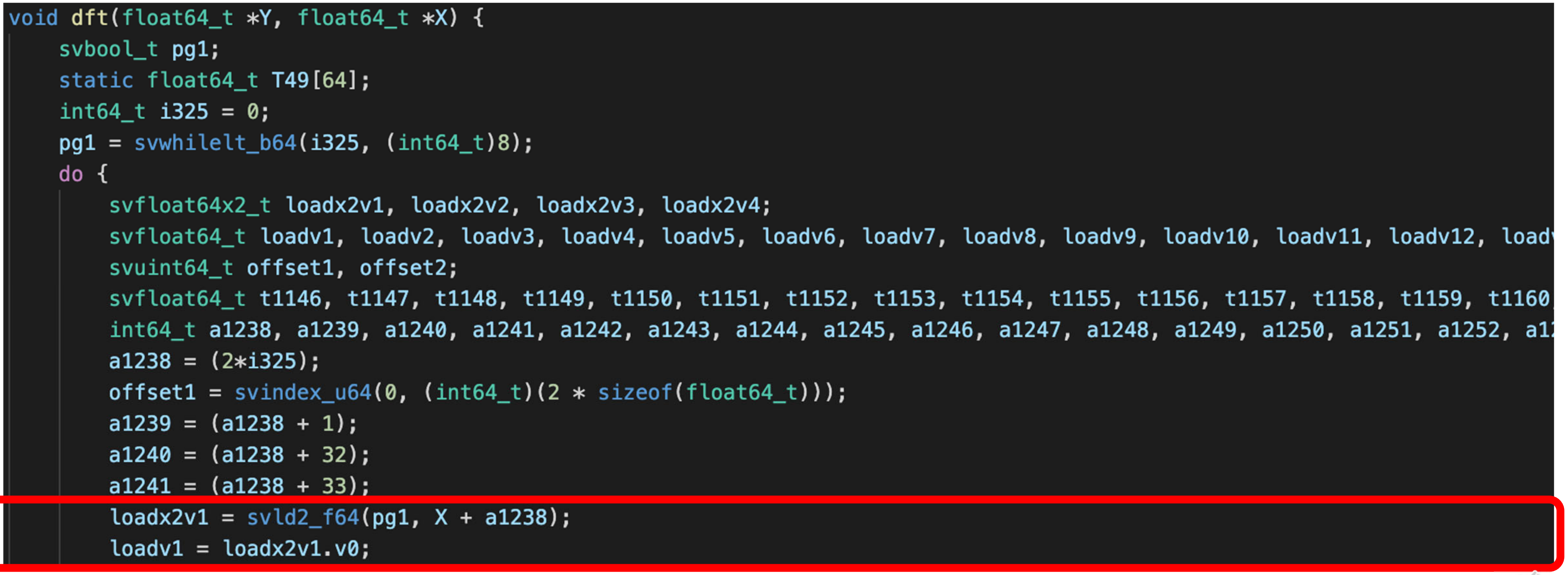
3. Modify multiple loads to a load operation.



```
void dft(float64_t *Y, float64_t *X) {
    svbool_t pg1;
    static float64_t T49[64];
    int64_t i325 = 0;
    pg1 = svwhilelt_b64(i325, (int64_t)8);
    do {
        svfloat64_t loadv1, loadv2, loadv3, loadv4, loadv5, loadv6, loadv7, loadv8, loadv9, loadv10, loadv11, loadv12, load
        svuint64_t offset1, offset2;
        svfloat64_t t1146, t1147, t1148, t1149, t1150, t1151, t1152, t1153, t1154, t1155, t1156, t1157, t1158, t1159, t1160
        int64_t a1238, a1239, a1240, a1241, a1242, a1243, a1244, a1245, a1246, a1247, a1248, a1249, a1250, a1251, a1252, a1
        a1238 = (2*i325);
        offset1 = svindex_u64(0, (int64_t)(2 * sizeof(float64_t)));
        a1239 = (a1238 + 1);
        a1240 = (a1238 + 32);
        a1241 = (a1238 + 33);
        loadv1 = svld1_gather_u64offset_f64(pg1, X + a1238, offset1);
        loadv2 = svld1_gather_u64offset_f64(pg1, X + a1240, offset1);
        t1146 = svadd_f64_x(pg1, loadv1, loadv2);
```

An Example of The SVE Translation Script（5/5)

4. Apply an instruction of load to multiple elements.



```
void dft(float64_t *Y, float64_t *X) {
    svbool_t pg1;
    static float64_t T49[64];
    int64_t i325 = 0;
    pg1 = svwhilelt_b64(i325, (int64_t)8);
    do {
        svfloat64x2_t loadx2v1, loadx2v2, loadx2v3, loadx2v4;
        svfloat64_t loadv1, loadv2, loadv3, loadv4, loadv5, loadv6, loadv7, loadv8, loadv9, loadv10, loadv11, loadv12, load
        svuint64_t offset1, offset2;
        svfloat64_t t1146, t1147, t1148, t1149, t1150, t1151, t1152, t1153, t1154, t1155, t1156, t1157, t1158, t1159, t1160
        int64_t a1238, a1239, a1240, a1241, a1242, a1243, a1244, a1245, a1246, a1247, a1248, a1249, a1250, a1251, a1252, a1
        a1238 = (2*i325);
        offset1 = svindex_u64(0, (int64_t)(2 * sizeof(float64_t)));
        a1239 = (a1238 + 1);
        a1240 = (a1238 + 32);
        a1241 = (a1238 + 33);
        loadx2v1 = svld2_f64(pg1, X + a1238);
        loadv1 = loadx2v1.v0;
```

Outline

- Background
- Aim of This Study
- Spiral Code Generation System
- Scalable Vector Extension
- Adaptation of SVE
- Performance Evaluation
- Summary

Performance Evaluation (Comparison with conventional approach)

- Performance comparison with the followings:
  - **SVE-applied DFT**: Generated Codes with the proposed approach.
  - **Scalar DFT**: **A Scalar DFT** code by conventional approach.
- Condition of the performance evaluation
  - DFT sizes: $2^N (4 \leqq N \leqq 20)$
  - 1 Node (1 core) is used.

Performance Evaluation

Type I Subsystem (FX1000): Supercomputer "Fugaku" Type

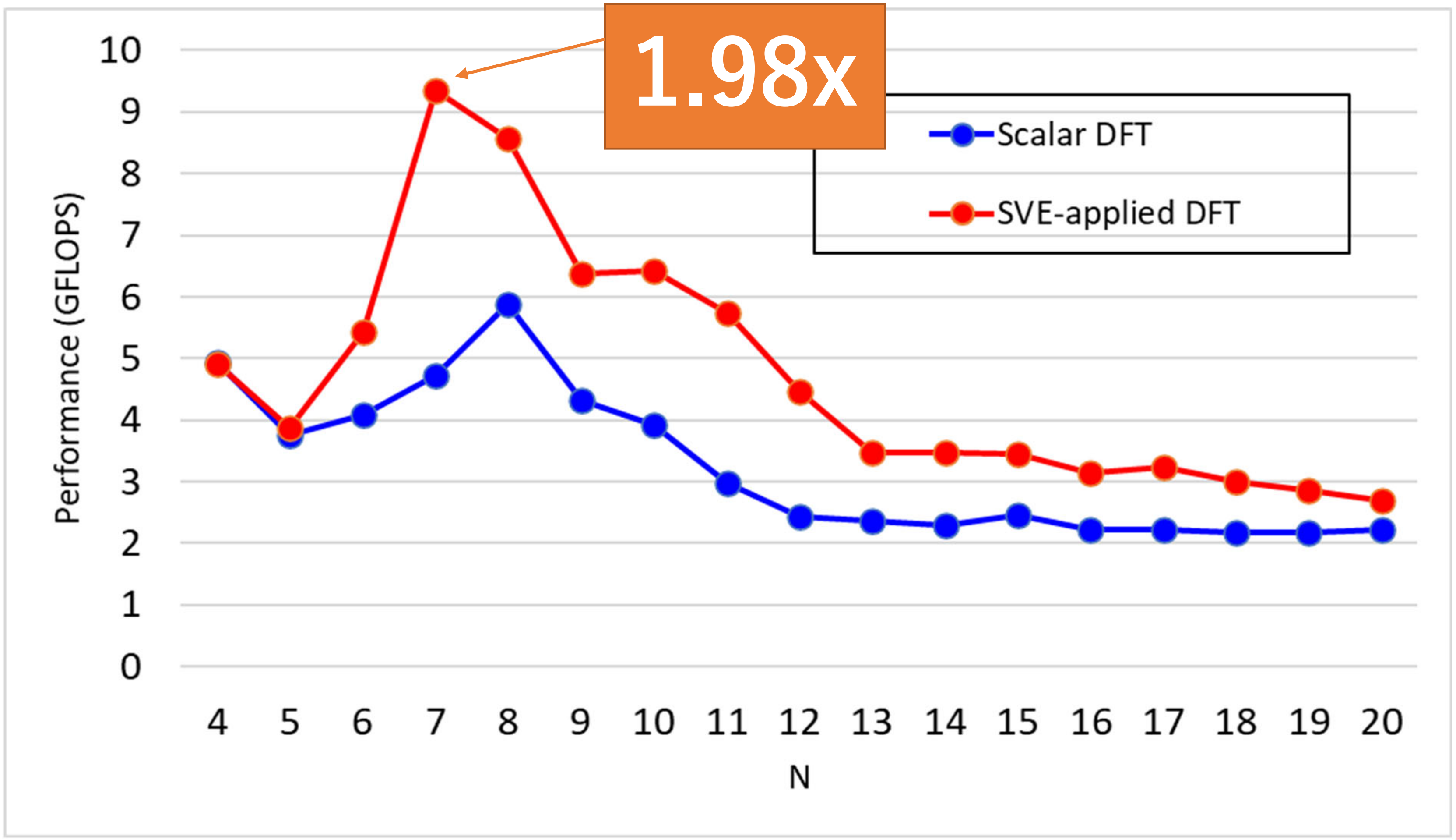
Supercomputer "Flow" at Information Technology Center, Nagoya University.

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

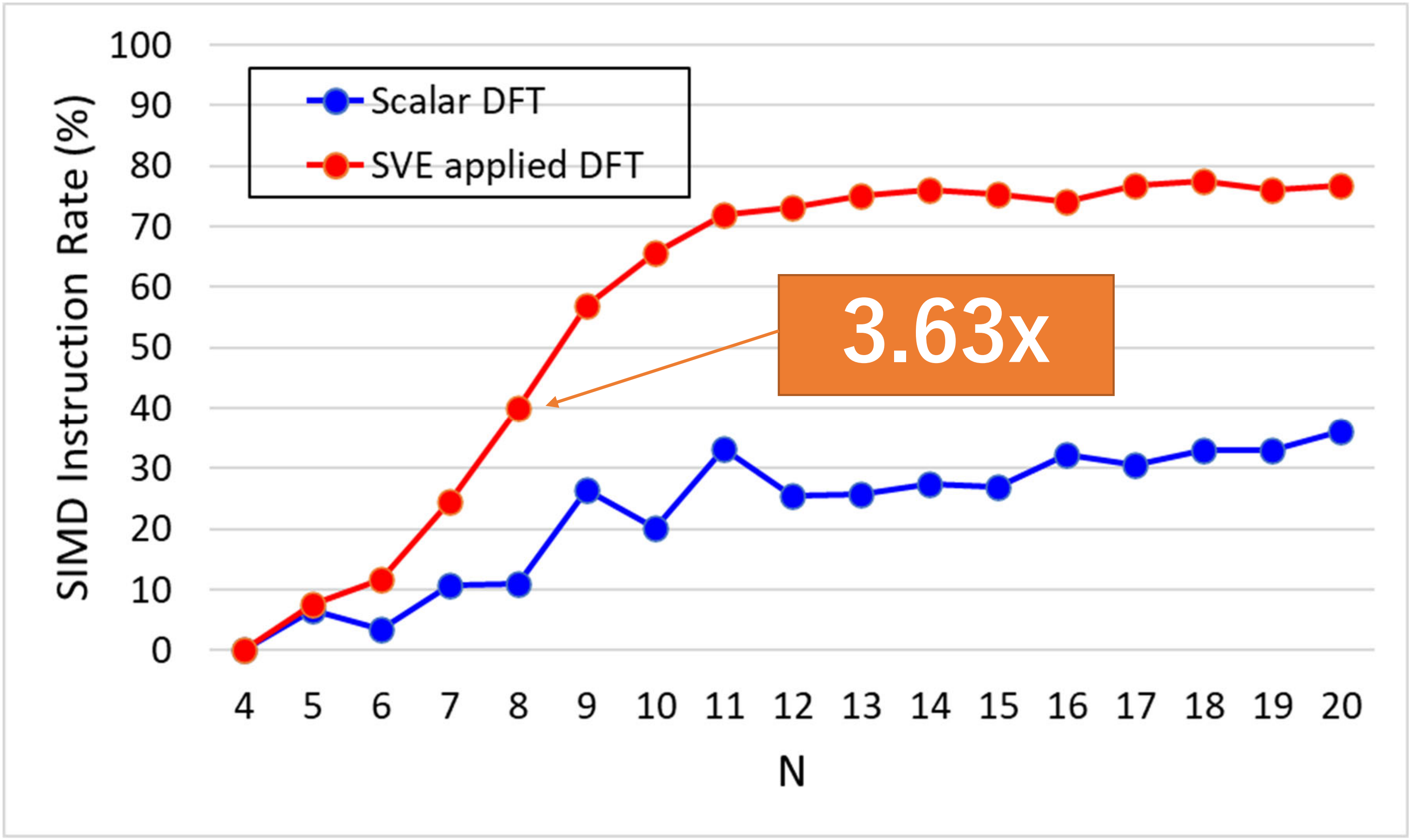
Compiler option and version of software

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

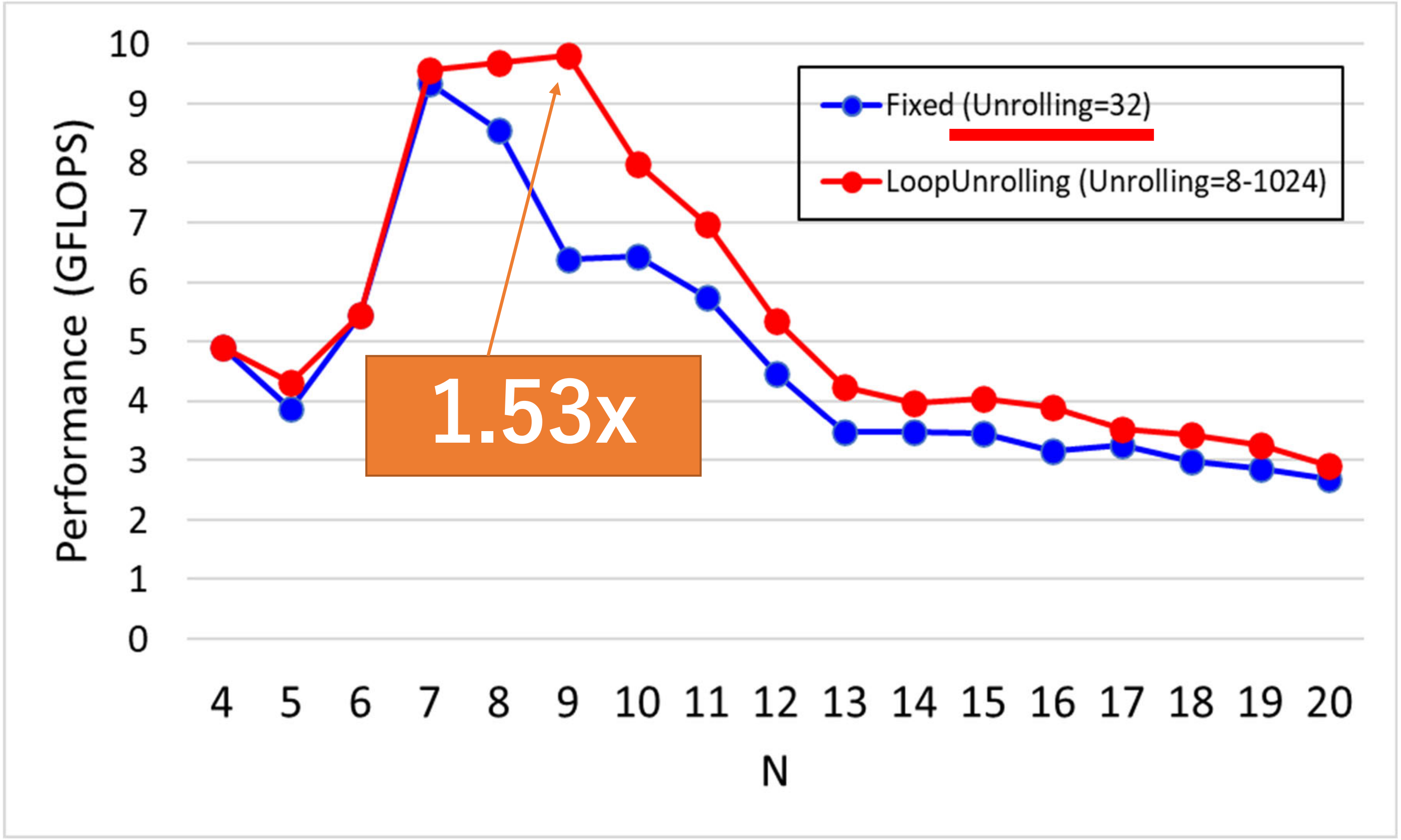
Performance Comparison (Scalar DFT)



SIMD Instruction Rate (Scalar DFT)

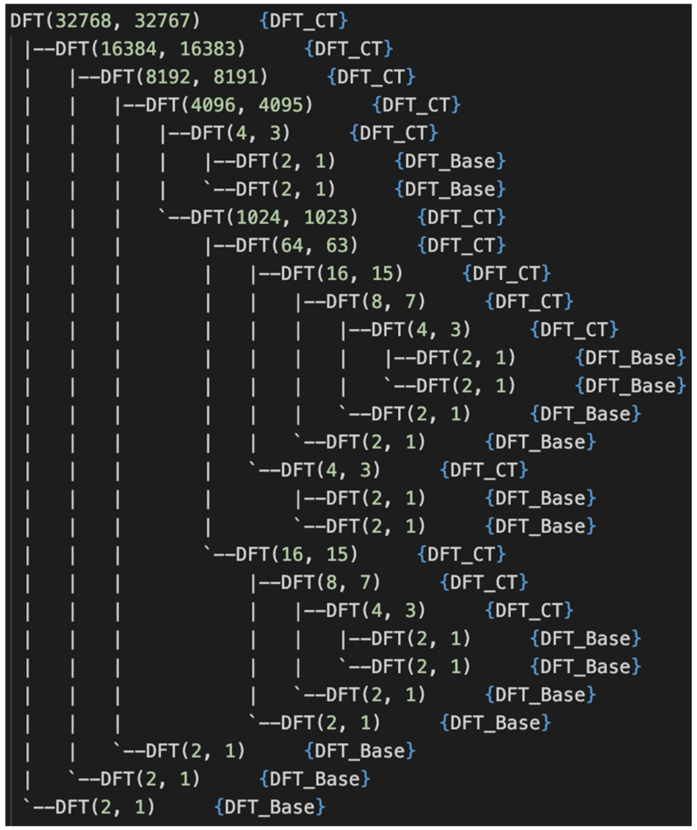


Effect of Auto-tuning



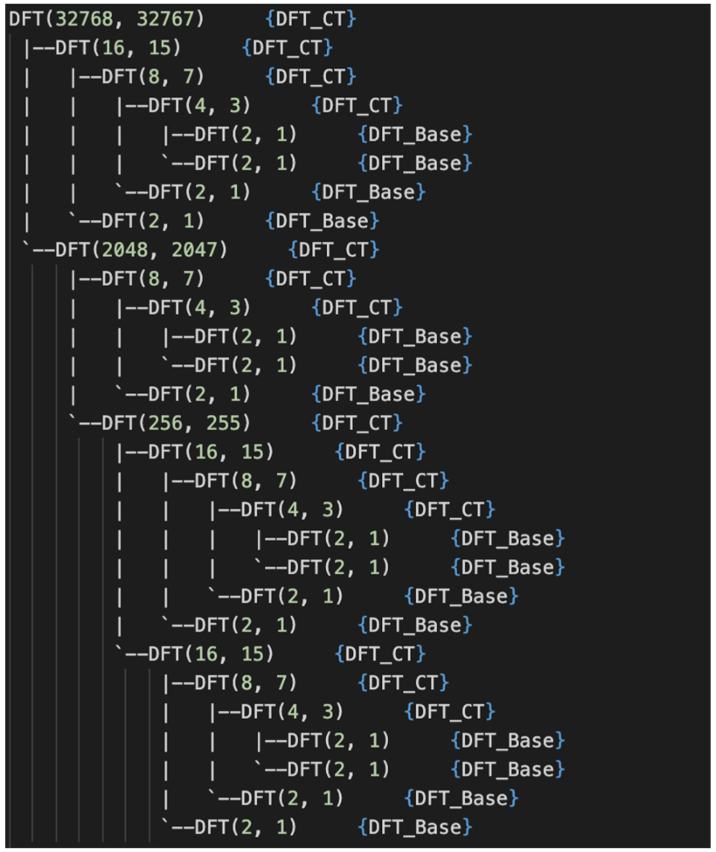
Trends of Selected Rule Tree

Scalar $DFT_{2^{15}}$



```
DFT(32768, 32767)      {DFT_CT}
 |--DFT(16384, 16383)      {DFT_CT}
 |   |--DFT(8192, 8191)      {DFT_CT}
 |   |   |--DFT(4096, 4095)      {DFT_CT}
 |   |   |   |--DFT(4, 3)      {DFT_CT}
 |   |   |   |   |--DFT(2, 1)      {DFT_Base}
 |   |   |   |   `--DFT(2, 1)      {DFT_Base}
 |   |   |   `--DFT(1024, 1023)      {DFT_CT}
 |   |   |       |--DFT(64, 63)      {DFT_CT}
 |   |   |       |   |--DFT(16, 15)      {DFT_CT}
 |   |   |       |   |   |--DFT(8, 7)      {DFT_CT}
 |   |   |       |   |   |   |--DFT(4, 3)      {DFT_CT}
 |   |   |       |   |   |   |   |--DFT(2, 1)      {DFT_Base}
 |   |   |       |   |   |   |   `--DFT(2, 1)      {DFT_Base}
 |   |   |       |   |   |   `--DFT(2, 1)      {DFT_Base}
 |   |   |       |   |   `--DFT(2, 1)      {DFT_Base}
 |   |   |       |   `--DFT(4, 3)      {DFT_CT}
 |   |   |       |       |--DFT(2, 1)      {DFT_Base}
 |   |   |       |       `--DFT(2, 1)      {DFT_Base}
 |   |   |       `--DFT(16, 15)      {DFT_CT}
 |   |   |           |--DFT(8, 7)      {DFT_CT}
 |   |   |           |   |--DFT(4, 3)      {DFT_CT}
 |   |   |           |   |   |--DFT(2, 1)      {DFT_Base}
 |   |   |           |   |   `--DFT(2, 1)      {DFT_Base}
 |   |   |           |   `--DFT(2, 1)      {DFT_Base}
 |   |   |           `--DFT(2, 1)      {DFT_Base}
 |   |   `--DFT(2, 1)      {DFT_Base}
 |   `--DFT(2, 1)      {DFT_Base}
 `--DFT(2, 1)      {DFT_Base}
```

SVE-applied $DFT_{2^{15}}$

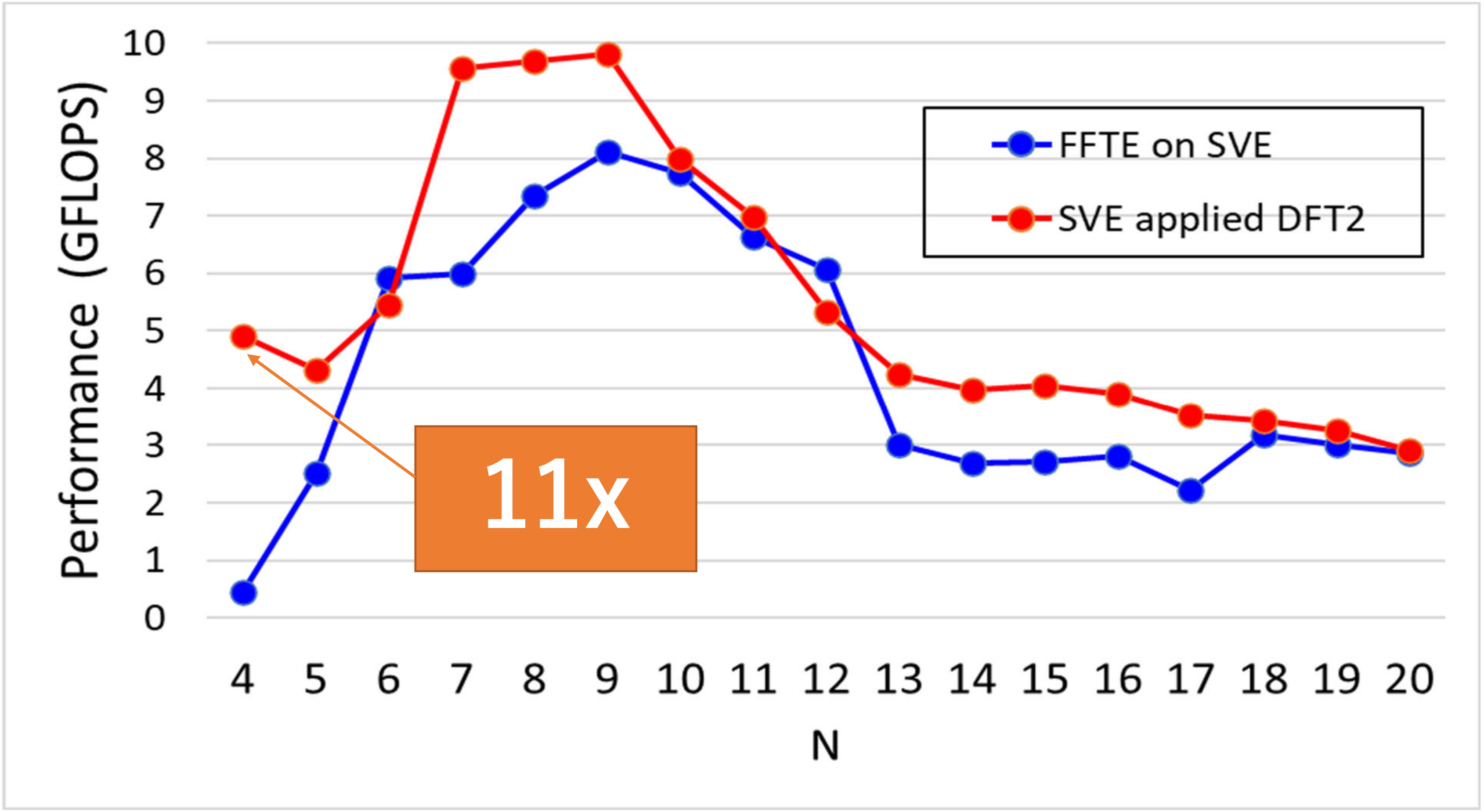


```
DFT(32768, 32767)      {DFT_CT}
 |--DFT(16, 15)      {DFT_CT}
 |   |--DFT(8, 7)      {DFT_CT}
 |   |   |--DFT(4, 3)      {DFT_CT}
 |   |   |   |--DFT(2, 1)      {DFT_Base}
 |   |   |   `--DFT(2, 1)      {DFT_Base}
 |   |   `--DFT(2, 1)      {DFT_Base}
 |   `--DFT(2, 1)      {DFT_Base}
 `--DFT(2048, 2047)      {DFT_CT}
     |--DFT(8, 7)      {DFT_CT}
     |   |--DFT(4, 3)      {DFT_CT}
     |   |   |--DFT(2, 1)      {DFT_Base}
     |   |   `--DFT(2, 1)      {DFT_Base}
     |   `--DFT(2, 1)      {DFT_Base}
     `--DFT(256, 255)      {DFT_CT}
         |--DFT(16, 15)      {DFT_CT}
         |   |--DFT(8, 7)      {DFT_CT}
         |   |   |--DFT(4, 3)      {DFT_CT}
         |   |   |   |--DFT(2, 1)      {DFT_Base}
         |   |   |   `--DFT(2, 1)      {DFT_Base}
         |   |   `--DFT(2, 1)      {DFT_Base}
         |   `--DFT(2, 1)      {DFT_Base}
         `--DFT(16, 15)      {DFT_CT}
             |--DFT(8, 7)      {DFT_CT}
             |   |--DFT(4, 3)      {DFT_CT}
             |   |   |--DFT(2, 1)      {DFT_Base}
             |   |   `--DFT(2, 1)      {DFT_Base}
             |   `--DFT(2, 1)      {DFT_Base}
             `--DFT(2, 1)      {DFT_Base}
```

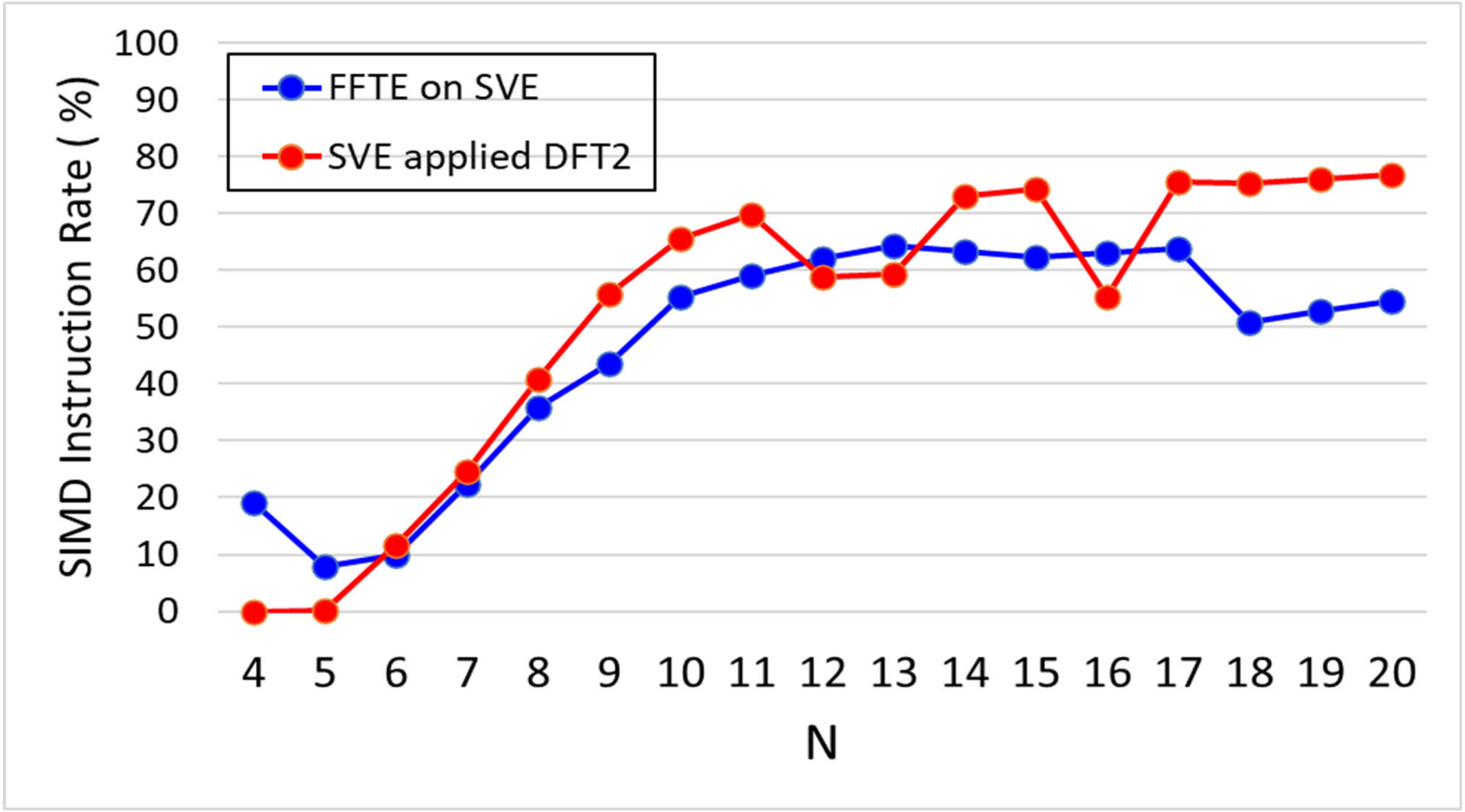
Performance Evaluation (Other Libraries)

- Performance evaluation with the follows:
  - **SVE-applied DFT2**: Generated Codes with the proposed method.
    - AT for depth of loop unrolling is adapted.
  - **FFTE on SVE**: FFTE with generated code by Spiral.
  - **FFTW3.3.8**: The previous version of FFTW. SVE is not supported. (The latest version is 3.3.9.)
- Condition of performance evaluation.
  - DFT size: $2^N (4 \leqq N \leqq 20)$
  - 1 Node (1 core)
- Same environment for the previous evaluation.

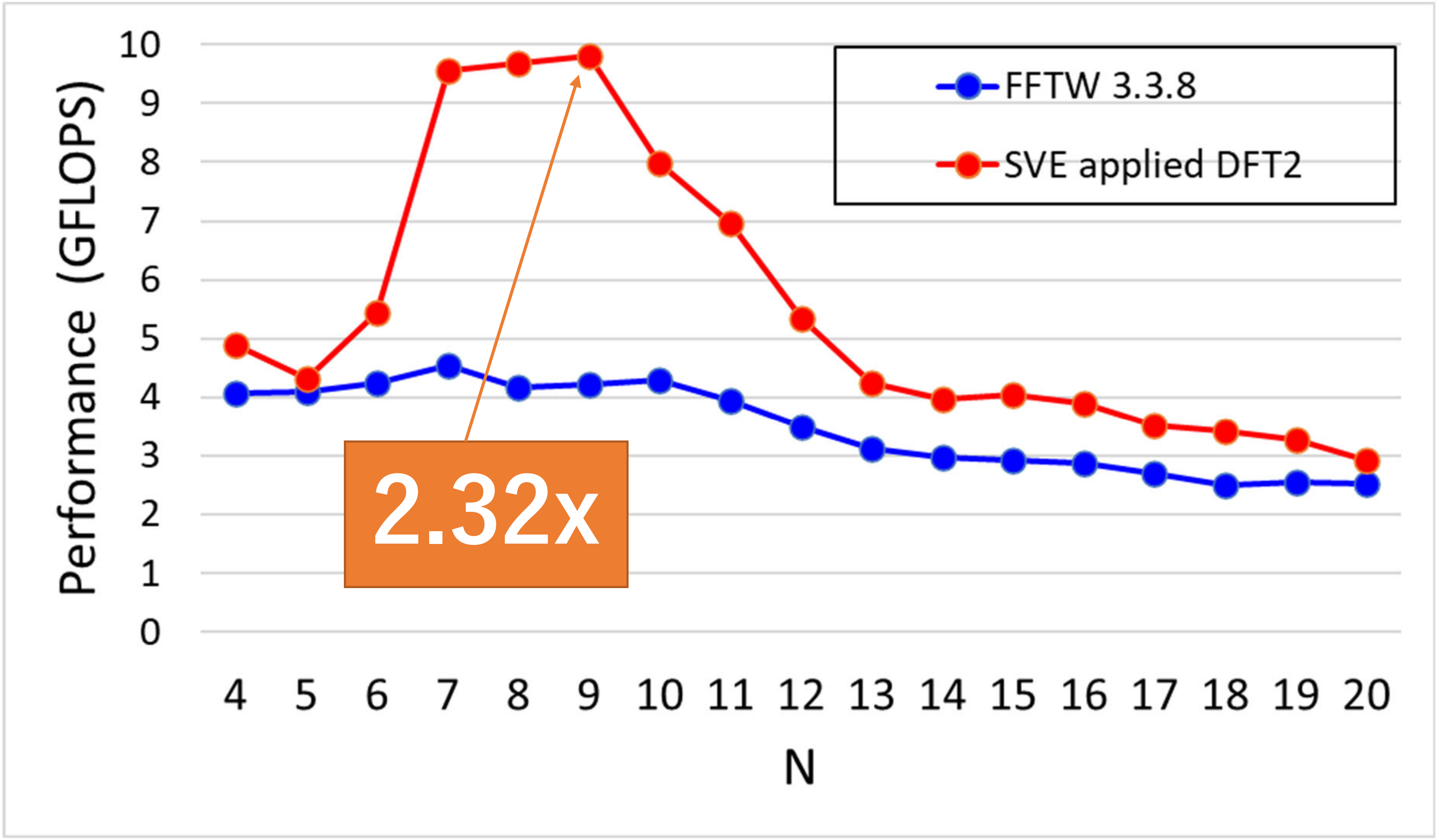
Performance Comparison (FFTE)



SIMD Instruction Rate (FFTE)



Performance Comparison (FFTW)



Outline

- Background
- Aim of This Study
- Spiral Code Generation System
- Scalable Vector Extension
- Adaptation of SVE
- Performance Evaluation
- Summary

Summary

- Propose an approach to generate codes with SVE instructions on Spiral.
- By adaptation of SVE, we found:
  - **Maximum 1.98x speedup** to **conventional approach**,
  - **Maximum 11.0x speedup** to **FFTE on SVE**, and
  - **Maximum 2.32x speedup** to **FFTW3.3.8**.

- Future work
  - Performance evaluation with environment of multicore CPUs or multiple nodes.
    - Combination with OpenMP + SVE.
  - Extend the code generation with adaptation of SVE for general abstraction of Spiral(Operator Language)
    - A rule need to be added for variable vectors on Spiral.