IEICE TRANS. ELECTRON., VOL.E105–C, NO.6 JUNE 2022

PAPER Special Section on Low-Power and High-Speed Chips

A Binary Translator to Accelerate Development of Deep Learning Processing Library for AArch64 CPU

Kentaro KAWAKAMI[†a)], **Kouji KURIHARA**[†], **Masafumi YAMAZAKI**[†], ***Nonmembers***, **Takumi HONDA**[†], ***Member***, ***and*** **Naoto FUKUMOTO**[†], ***Nonmember***

SUMMARY To accelerate deep learning (DL) processes on the supercomputer Fugaku, the authors have ported and optimized oneDNN for Fugaku's CPU, the Fujitsu A64FX. oneDNN is an open-source DL processing library developed by Intel for the x86_64 architecture. The A64FX CPU is based on the Armv8-A architecture. oneDNN dynamically creates the execution code for the computation kernels, which are implemented at the granularity of x86_64 instructions using Xbyak, the Just-In-Time (JIT) assembler for x86_64 architecture. To port oneDNN to A64FX, it must be rewritten into Armv8-A instructions using Xbyak_aarch64, the JIT assembler for the Armv8-A architecture. This is challenging because the number of steps to be rewritten exceeds several tens of thousands of lines. This study presents the Xbyak_translator_aarch64. Xbyak_translator_aarch64 is a binary translator that at runtime converts dynamically produced executable codes for the x86_64 architecture into executable codes for the Armv8-A architecture. Xbyak_translator_aarch64 eliminates the need to rewrite the source code for porting oneDNN to A64FX and allows us to port oneDNN to A64FX quickly.

key words: *deep learning, oneDNN, AArch64, just-in-time assembler, binary translator*

1. Introduction

On March 9, 2021, the supercomputer Fugaku went online [1]–[5]. Fugaku is a CPU-based supercomputer powered by the Fujitsu A64FX [6], [7]. The A64FX complies with the Armv8-A architecture profile [8]. The Armv8-A architecture has additional Scalable Vector Extension (SVE) [9] instructions designed for high-performance computing workloads. The A64FX CPU is the first in the world to support SVE instructions. AArch64 shall be referred to in this paper as "Armv8-A + SVE."

The authors have been working on a software (S/W) stack that can run deep learning (DL) tasks at high speeds on Fugaku. Figure 1 depicts the S/W stack for the DL processes. The S/W stack consists of a front-end framework S/W and a backend library S/W. The framework is responsible for describing neural network definitions and exchanging input and output data between users and computer systems. The library S/W provides the functions needed to perform the large number of computations required for DL operations. TensorFlow [10], developed by Google, and PyTorch [11], developed by Facebook, are the two leading frameworks. TensorFlow and PyTorch both support Linux as an operating system, and their source code is open-source software (OSS). Fugaku's operating system [12] is RedHat Linux. It is simple to develop and use these frameworks from their source code on Fugaku.

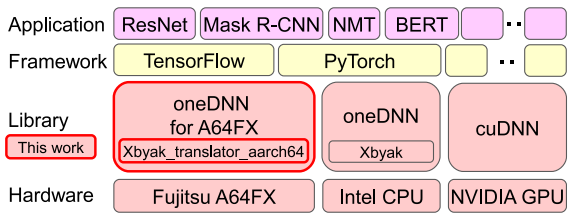


Fig. 1 Software stack of deep learning processes.

Library S/Ws are optimized for each hardware platform to perform the massive amounts of computing required by DL processes at high speeds. CuDNN [13] is provided by NVIDIA for NVIDIA GPUs, and oneDNN [14], [15] is provided by Intel for Intel CPUs. A64FX-optimized DL library S/W is required to realize high-speed DL processes on Fugaku. There is currently no DL library S/W optimized for the AArch64 architecture, and it must be developed from scratch. To address this issue, we have been porting and optimizing oneDNN for A64FX. The following are some of the benefits of porting oneDNN.

- TensorFlow and PyTorch, two of the most popular DL frameworks, both offer oneDNN as a backend.
- oneDNN is optimized for DL processes on CPUs. Multithread processes, for example, are used to accelerate the calculation of computation kernels. OpenMP [16] and Intel Threading Building Blocks [17] are both supported. Because Fugaku includes the OpenMP library, the implementation of oneDNN's multithreaded processes can be used without modification.

We present the development of oneDNN for A64FX in this work. The oneDNN dynamically generates the optimal binary of the computation kernels based on the execution environment and conditions, which is then utilized repeatedly to execute DL processing. The Just-In-Time (JIT) assembler is used to achieve this capability at the x86_64 instruction level. To port it to A64FX, these implementations must be replaced by AArch64 instructions. As oneDNN has tens of thousands of source code lines, rewriting the source code is time-consuming. To address this issue, we created a bi-

Manuscript received June 30, 2021.
Manuscript revised October 19, 2021.
Manuscript publicized December 3, 2021.
[†]The authors are with Fujitsu Limited, Kawasaki-shi, 211–8588 Japan.
a) E-mail: kawakami.k@fujitsu.com
DOI: 10.1587/transele.2021LHP0001

nary translator that dynamically translates x86_64 binaries into AArch64 binaries and integrates it into the oneDNN. The binary translator includes mapping tables connecting x86_64 instructions and AArch64 instruction sequence and allows the current oneDNN to run on AArch64 with very little modification of the source code for x86_64. With the AVX512 instruction, the x86_64 CPU can perform SIMD operations of up to 512 bits. SVE instructions are handled by A64FX as 512-bit SIMD instructions, similar to AVX512 instructions. The binary translator can convert executable code written in AVX512 instructions to SVE instructions without compromising SIMD parallel performance, resulting in executable codes with high processing performance on A64FX.

The remainder of this work is organized as follows. Section 2 describes the current implementation of oneDNN. The porting and developing oneDNN for A64FX is described in Sect. 3. This section also elaborates on the binary translator, Xbyak_translator_aarch64. Section 4 describes the effectiveness of Xbyak_translator_aarch64 in terms of development. The processing performance of the execution code translated for A64FX using the binary translator is shown in Sect. 5. The summary of this paper is provided in Sect. 6.

Table 1 Source code of a sample program written with Xbyak.

```
#include <stdlib.h>
#include "xbyak/xbyak.h"
using namespace Xbyak;
class Generator : public CodeGenerator {
public:
  /* Generate machine code sequence of calculating
     Fibonacci number */                                   Step (a)
  Generator() {
    Label L_begin, L_end;
    mov(r8, 0);  /* F(0) = 0 */
    mov(r9, 1);  /* F(1) = 1 */
    mov(r10, 1);
    sub(rdi, 1); /* 1st argument is passed by RDI register. */

    L(L_begin);
    cmp(rdi, 0);
    jle(L_end);  /* Jump if the value of RDI <= 0 */

    mov(r10, r8);
    add(r10, r9); /* F(n+2) = F(n) + F(n+1) */
    mov(r8, r9);  /* Update F(n) */
    mov(r9, r10); /* Update F(n+1) */
    sub(rdi, 1);  /* Update loop counter */
    jmp(L_begin);

    L(L_end);
    mov(rax, r10); /* Return value is passed by RAX register. */
    ret();
  }
};
int main(int argc, char *argv[]) {
  if(argc != 2 || atoi(argv[1]) <= 0) {
    fprintf(stderr, "%s non-zero_positive_integer¥n", argv[0]);
    exit(1);
  }
  Generator gen;
  gen.ready();
  FILE *fp = fopen("fibonacci.bin", "wb");
  fwrite(gen.getCode(), sizeof(char), gen.getSize(), fp);
  fclose(fp);
  auto f = gen.getCode<int (*)(int)>();
  std::cout << f(atoi(argv[1])) << std::endl;          Step (b)
  return 0;
}
```

2. oneDNN

2.1 Features of oneDNN

oneDNN is a S/W library for high-speed DL operations on Intel architecture (x86_64) CPUs, Intel Processor Graphics, and Xe architecture-based Graphics. Intel created oneDNN, with an open-source source code that is available on GitHub as an OSS. The following is a description of the oneDNN implementation for Intel CPUs.

oneDNN runs on Intel CPUs and supports Microsoft Windows, Linux, and macOS. It is written in C++ and is compatible with GCC, LLVM, Intel Compiler, and Microsoft Visual Studio. In the DL software stack, oneDNN is responsible for the quick execution of computations such as convolution, ReLU, batch normalization, pooling, softmax, and reorder, which are executed repeatedly in DL processes. These functions are known as primitives in oneDNN (e.g., reorder primitive). The oneDNN provides primitive implementations optimized for SSE4.1, AVX, AVX2, and AVX512 instruction sets [18], with the best one chosen according to the environment in which users run DL processes. The implementations of the primitives optimized for each instruction set are defined in the source code at the instruction level using Xbyak [19], the x86_64 architecture's JIT assembler. The implementation with Xbyak is detailed in Sect. 2.1.1

Aside from the Xbyak implementation, each primitive has a reference implementation written in C++. When utilizing oneDNN on non-x86_64 architecture CPUs, none of the SSE4.1, AVX, AVX2, or AVX512 implementations may be utilized, hence the reference implementation is used. The reference implementation performs the required computations for each primitive in a functionally accurate manner, but the processing performance is slow since the implementation prioritizes source code readability. The reference implementation is about 100 times slower than the Xbyak implementation depending on the kind of primitives, the shape of the tensor to be processed (the number of dimensions of the input data array, the number of elements, data precision), and other characteristics. To accelerate DL operations on Fugaku utilizing oneDNN, the primitives optimized for AArch64 architecture must be implemented using the AArch64 JIT assembler.

2.1.1 Runtime Code Generation Using Xbyak

Xbyak is used in oneDNN to implement several primitives for SSE4.1, AVX, AVX2, and AVX512 instruction sets. Xbyak is a S/W library that generates runtime code for the x86_64 architecture. Its source code is available as OSS [19].

The source code for a sample program written with Xbyak is shown in Table 1. C++ programs can take advantage of Xbyak. This sample program returns the N-th Fibonacci number, where N is a nonzero positive integer that is passed as an argument. At runtime, step(a) produces and writes an x86_64 machine code sequence to memory. The sequence is then called as the function in step(b). The results of writing the machine code sequence created in memory to a file and dumping it with the objdump command† [20] are shown in Table 2. Xbyak provides

†objdump -D -b binary -m i386:x86-64 -M intel fibonacci.bin

Table 2 The machine code sequence generated by the sample program.

```
0000000000000000 <.data>:
   0:   41 b8 00 00 00 00       mov     r8d, 0x0
   6:   41 b9 01 00 00 00       mov     r9d, 0x1
   c:   41 ba 01 00 00 00       mov     r10d, 0x1
  12:   48 83 ef 01             sub     rdi, 0x1
  16:   48 83 ff 00             cmp     rdi, 0x0
  1a:   7e 12                   jle     0x2e
  1c:   4d 89 c2                mov     r10, r8
  1f:   4d 01 ca                add     r10, r9
  22:   4d 89 c8                mov     r8, r9
  25:   4d 89 d1                mov     r9, r10
  28:   48 83 ef 01             sub     rdi, 0x1
  2c:   eb e8                   jmp     0x16
  2e:   4c 89 d0                mov     rax, r10
  31:   c3                      ret
```

a set of functions. Each function's name is the same as the x86_64 instruction mnemonics such as "mov," "sub," "cmp," "jle," and "ret" (referred as the mnemonic functions in this paper). The mnemonic functions can be used to produce x86_64 machine code sequences. In memory, calling a single mnemonic function generates a single x86_64 machine code. The functions shown with blue text in Table 1 are the mnemonic functions. The arguments of the mnemonic function specify the instruction operands.

Xbyak has the ability to generate branch instructions. The Label class is given to specify the jump destination labels of branch instructions, and the L function can be used to specify the branch destination address (Table 1 and 2).

If the machine code sequence created by Xbyak conforms to the x86_64 architecture's application binary interface (ABI) [21], [22], it can be utilized as a function with arguments and a return value. The generated function can be invoked at any time using a C++ function pointer, as seen in step(b) of Table 1. In this paper, we refer to the JIT assembler as the library S/W that includes mnemonic functions and runtime-code-generating features, which are similar to Xbyak.

Using Xbyak, oneDNN generates the ideal machine code sequences to speed up the DL processes based on the type of instruction set available in the execution environment, the shape of the tensor to be processed, and other runtime parameters. Table 3 shows the examples of the parameters that are considered during code generation. For example, CPUs that support up to AVX2 have 16 SIMD registers, while CPUs that support AVX512 have 32. If the CPU has more SIMD registers, oneDNN can generate the machine code sequence with a loop body that uses extra SIMD registers. Unrolling with twice the number of registers halves the number of loop iterations, i.e., halves the number of conditional branch instructions executed, resulting in faster processing. Furthermore, for a multidimensional array given as an input to be processed at runtime, if (the number of elements in the innermost dimension) × (data size) is an integer multiple of the SIMD register width, then there is no need to consider the processing elements at the end of the array that do not fit into the SIMD register. On this basis, it is feasible to accelerate processing by creating and executing a machine code sequence that simplifies loop processing. oneDNN attains faster processing by developing optimal machine code sequences for various primitives that are repeatedly executed in the DL processes. The development of runtime code using the JIT assembler is a critical aspect of improving the performance of oneDNN.

Table 3 Example of parameters considered for dynamic code generation.

Parameters	Example
Available instruction set	SSE4.1, AVX, AVX2, AVX512
# of SIMD registers	16, 32
SIMD register width [bits]	128, 256, 512
Input/Output data size [bits]	16, 32
Input/Output data precision	float16, float32, int32, int8
Input/Output array dimension	positive integer
# of array elements	positive integer
Convolution kernel size	positive integer
Scale parameter	1, 1.5, 2

oneDNN creates machine code sequences for each primitive type and parameter combination. If the primitive type and the considered parameters are the same, the created sequences are saved in memory and reused. The time taken to construct a machine code sequence is in milliseconds. The overhead is small compared to the characteristics of the DL processes, which iterate through a significant amount of data.

3. Development of oneDNN for A64FX

3.1 Xbyak_aarch64; JIT Assembler for AArch64

The JIT assembler, as indicated in Sect. 2.1.1, is a key technology that enables fast processing of oneDNN. The authors created the JIT assembler Xbyak_aarch64 [23]–[25] for the AArch64 platform to port oneDNN to A64FX. Runtime machine code generation for the AArch64 architecture, including SVE instructions, is supported by Xbyak aarch64. Similar to Xbyak, Xbyak_aarch64 can be created using a standard C++ compiler and used from C++-based programs.

Table 4 displays the source code rewritten for AArch64 using Xbyak_aarch64, corresponding to Table 1. Other than the "Generator()" implementation, which is the same as in Table 1, they are omitted. The names of the mnemonic functions varies according to the AArch64 instruction set architecture, as do the number and the types of arguments passed to the mnemonic functions to describe the operands and the names of the registers. The Label class and the L function can also be used by Xbyak_aarch64 to specify the jump destination for branch instructions. The Label class and the L function have the same implementations and usage in Xbyak_aarch64 and Xbyak. The syntax and examples are specified in "README.md" file included with Xbyak_aarch64, as well as the cpp files in the "sample" directory.

3.2 Development of Primitives for A64FX

It is now possible to port several primitives to A64FX using JIT assembler technology by developing the Xbyak_aarch64. SSE4.1, AVX, AVX2, and AVX512 optimizations are included in oneDNN. These instruction sets' SIMD widths are 128, 256, and 512 bits, respectively.

Table 4 Source code of the sample program rewritten with Xbyak_aarch64.

```
Generator() {
   Label L_begin, L_end;
   mov(x8, 0);  /* F(0) = 0 */
   mov(x9, 1);  x10, 1);
   sub(x0, x0, 1); /* 1st argument is passed by X0 register. */

   L(L_begin);
   cmp(x0, 0);
   b(LE, L_end);

   add(x10, x8, x9); /* F(n+2) = F(n) + F(n+1) */
   mov(x8, x9);  /* Update F(n) */
   mov(x9, x10); /* Update F(n+1) */
   sub(x0, x0, 1);  /* Update loop counter */
   b(L_begin);

   L(L_end);
   mov(x0, x10); /* Return value is passed by X0 register. */
   ret();
}
```

Table 5 Comparison of # of registers between AVX512 CPU and A64FX.

	AVX512	A64FX (Armv8-A + SVE)
# of general-purpose registers	16	32
SIMD register width	512	512
# of SIMD registers	32	32
# of mask (predicate) registers	8	16

AVX512, the implementation with the largest SIMD width, achieves the fastest processing speed. The SVE instructions are SIMD instructions. The SIMD width of SVE is CPU implementation-dependent and can be chosen from $128 \times N$ ($N = 1, 2, \cdots, 16$) by a CPU development vendor [9]. SVE is implemented as 512-bit SIMD ($N = 4$) on the A64FX. Because AVX512 and A64FX have identical 512-bit SIMD widths, the authors opted to use the AVX512 implementations as the basic implementations of A64FX.

Masked instructions in AVX512 can control whether or not instructions are executed for each SIMD lane. Because masked instructions have been included in SVE[†], AVX512 masked instructions can be replaced with SVE instructions.

The CPU registers of AVX512 and A64FX are compared in Table 5. A64FX features the same or more general-purpose registers, SIMD registers, and mask registers than AVX512 CPUs. As a result, when AVX512 primitive implementations are rewritten for A64FX using Xbyak_aarch64, the data-register relationship might be similar to that of AVX512. For example, assume that an AVX512 implementation generates a machine code sequence that utilizes the 512-bit SIMD registers Zmm0 to 15 for loading input and Zmm16 to 31 for storing the coefficients utilized in the calculation. The machine code sequence for A64FX should be created in the same fashion, with the 512-bit SIMD registers z0 to 15 used for loading input data and z16 to 31 used for retaining the coefficients.

However, care should be taken in primitive AVX512 implementations, where all SIMD registers are assigned to data, and SIMD instructions with memory operands are used. With a few exceptions, all arithmetic and logical instruction operands in AArch64 are register operands. When AVX512 instructions with memory operands are replaced with SVE instructions (Table 6 (a) and (b)), AArch64 must

[†]They are referred to as predicate instructions in AArch64.

Table 6 Example of replacing an instruction with a memory operand.

(a) Statement to generate VPADDD

```
vpaddd(zmm0, zmm1, ptr[r9]);
```

(b) Macnihe code generated by (a).

```
vpaddd zmm0,zmm1,ZMMWORD PTR [r9]
```

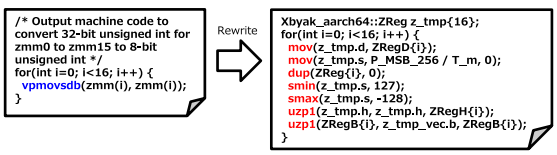
(c) Statements to generate the machine code sequence equivalent to (a) for A64FX.

```
sub(sp, sp, 64);
str(z31, ptr(sp));
ldr(z31, ptr(x9));
add(z0.s, z1.s, z31.s);
ldr(z31, ptr(sp));
add(sp, sp, 64);
```

(d) Macnihe code generated by (c).

```
sub  sp, sp, #0x40
str  z31, [sp]
ldr  z31, [x9]
add  z0.s, z1.s, z31.s
ldr  z31, [sp]
add  sp, sp, #0x40
```

"sp" is the stack pointer register of AArch64. ".s" means that SVE lane width is 32 bits. Since SVE register width of A64FX is 512 bits, the stack address should be shifted by 64 bytes before executing load/store instructions.



Example rewriting a process that generates x86_64 machine code with a process that generates Armv8-A machine code that runs the equivalent process using Xbyak. The function indicated with blue text and functions indicated with red text are provided by Xbyak and Xbyak_aarch64, respectively. Although a single x86_64 instruction can on average be replaced with two Armv8-A instructions, this example shows that complicated x86_64 instructions may result in a larger number of Armv8-A instructions when replaced.

Fig. 2 Example of rewriting the implementation for A64FX.

replace them with multiple instructions specified in Table 6 (c) and (d). First, an instruction must be generated to temporarily load data from memory into a SIMD register, followed by an instruction to perform an operation using that register as the source operand. If the SIMD register that is used as a temporary register contains data that should not be deleted, a store instruction is necessary to temporarily preserve this data to stack memory and a load instruction is required to retrieve it later.

To port oneDNN to A64FX, the source code written in Xbyak for AVX512 will be rewritten in Xbyak_aarch64, as previously mentioned. This has the following drawbacks.

- The number of steps that must be rewritten is substantial. The files that implement the primitive using JIT technology in oneDNN are placed in the "src/cpu/x64" directory, with file names beginning with "jit_." Among these, the files beginning with "jit_avx512_" and "jit_uni_" must be rewritten for A64FX. "jit_avx512_" contains the AVX512 primitive implementation. "jit_uni_" is the implementation of the primitive shared by all instruction sets. The total number of steps in these source codes is in the tens of thousands.
- To rewrite the code, the developer must be knowledgeable in the processing details of each of the x86_64 and AArch64 instruction sets, as well as their correspondence (Fig. 2).
- When there is a rewriting error, it is difficult to debug the source code. Debugging can be accomplished by 1) inspecting the source code written at the instruction level abstraction with Xbyak_aarch64, 2) inspecting the

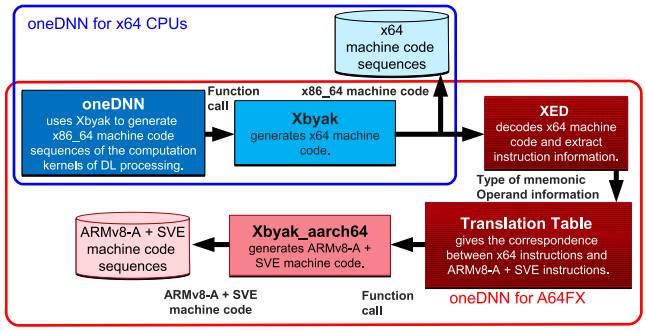


Fig. 3 Processing flow of Xbyak_translator_aarch64.

text file disassembled from the generated executable code like Table 2, and 3) inspecting the generated machine code sequences while stepping through them at the instruction level in GDB. It is difficult to isolate a single instruction error, regardless of the mechanism utilized. In most cases, the number of instructions in the resulting machine code sequences exceeds 1,000. Furthermore, the produced sequences vary depending on the runtime parameters (test parameters).

Furthermore, oneDNN is regularly updated and improved. To boost processing speed, new and current primitives are added and updated, respectively. To stay up with these developments and continue to deliver the most up-to-date oneDNN for A64FX, it is vital to find a technique to lessen the porting effort.

3.3 Binary Translator to Accelerate Development of oneDNN for A64FX

To address the issue indicated in Sect. 3.2, the authors created a binary translator, Xbyak_translator_aarch64 (hereinafter referred to as the translator). When converting oneDNN to A64FX, the translator can considerably reduce the amount of modification required to the source code. The configuration and flow of the translator are depicted in Fig. 3. The translator consists of Xbyak, XED, Translation Table, and Xbyak_aarch64. The interfaces of Xbyak in the translator are the same as those of the original Xbyak. OneDNN for x86_64 can be converted to A64FX by replacing the Xbyak used in oneDNN for x86_64 with the translator. In the original Xbyak, each mnemonic function creates the machine code for one corresponding x86_64 instruction. The method depicted in Fig. 3, will be executed by replacing Xbyak with the translator in oneDNN (except for the mnemonic function for the branch instruction).

1. The Xbyak mnemonic function generates machine code for one x86_64 instruction.
2. The preceding step's information is fed into the translation table. The definitions of the x86_64 instruction and the corresponding AArch64 instruction sequence are provided in this table. The translation table calls the relevant Xbyak_aarch64 mnemonic function based on the information about the mnemonic and operand types output by XED to construct a sequence of AArch64 instructions that are equivalent to the x86_64 instruction. The operand type and value returned by XED are converted to the type and value of the corresponding Xbyak_aarch64 and passed as arguments to the mnemonic function.

Table 7 Instruction information extracted by XED.

Information	Example
Mnemonic	ADD, JMP, VPADDD, RET
# of operands	0, 1, 2, 3, 4
Operand type	Register operand Memory operand Immediate Value
Memory operand or not	True, False
Mask method	No, Zeroing, Merging
Broadcast	True, False

Intel's XED [26], [27] is an encoder/decoder for the x86_64 instructions. XED may be accessible as OSS. The translator outputs the necessary information from the machine code generated by Xbyak using the decoding function of XED.

The Xbyak utilized in the translation is a partially modified version of the original Xbyak. While in the original Xbyak, each mnemonic function generates the corresponding x86_64 machine code and completes the process. Except for the branch instruction, each mnemonic function of Xbyak in the translator is simply added as a single statement "decodeAndTransToAArch64()" function, as shown in Table 9 (a) that processes XED and beyond of Fig. 3.

Instead of creating the machine code for the branch instructions in x86_64, the mnemonic functions for the branch instructions in Xbyak have been altered to directly call the mnemonic functions for the branch instructions in Xbyak_aarch64. Table 9 (b) is an example modification for the x86_64 JG (Jump if greater than) instruction, in which the Xbyak_aarch64 mnemonic function is used to generate the B.GT instruction in AArch64. The Label class of Xbyak and Xbyak aarch64 has the same implementation, as indicated in Sect. 3.1. As a result, the Label class passed to the mnemonic function of the branch instruction of Xbyak from oneDNN can be passed directly to the mnemonic function of the branch instruction of Xbyak_aarch64.

3.3.1 Implementation of Translation Table

The translation table in Fig. 3, is the most crucial component of building the translator. It was necessary to support 229 mnemonics in the x86_64 instructions to support the implementation of oneDNN for AVX512 in the translator. To define the connection between the x86_64 instructions and AArch64 instruction sequences, the authors created Microsoft Excel files for each mnemonic. An example of the x86_64 VPADDD instruction is seen in Table 10. This table elaborates how to convert the VPADDD instruction to

Table 8 Operand information extracted by XED.

Information	Example	Available if Operand is		
		Register	Imm. value	Mem. addr.
Register index	0, 1, ⋯, 31	Yes	No	No
Operand width	32, 64, 128, 256, 512	Yes	No	Yes
Imm. value	Imm. value	No	Yes	No
Base address register index	0, 1, ⋯, 15	No	No	Yes
Index address register index	0, 1, ⋯, 15	No	No	Yes
Index scale	0, 1, 2, 4	No	No	Yes
Index displacement	Integer value	No	No	Yes

Table 9 Example of modification of mnemonic function.

(a) Example of Nonbranch Instruction

```
void Xbyak::CodeGenerator::vpaddd(const Xmm &x1, const Xmm &x2,
                                  const Operand &op) {
  opAVX_X_X_XM(x1, x2, op, T_66 | T_0F | T_EW0 | T_YMM | T_EVEX | T_B32, 0xFE);
  /* Add this statement to kick off the subsequent processing. */
  decodeAndTransToAArch64();
}
```

(b) Example of Branch Instruction

```
void Xbyak::CodeGenerator::jg(const Label &label, LabelType type) {
  /* The mnemonic function of the corresponding
     AArch64 branch instruction are directly called. LabelType is used to
     distinguish between x86_64 near jump instructions and long jump
     instructions; In aarch64, there is only one type of conditional
     branch instruction, regardless of the distance to the jump
     destination, so there is no need to distinguish between them. */
  b(Xbyak_aarch64::GT, label);
}
```

an AArch64 instruction sequence. Cells F4 through F10 are eliminated due to space constraints. The SIMD instruction VPADDD adds two 32-bit integers from two source operands. Furthermore, there are variants with SIMD widths of 128, 256, and 512 bits. Table 10 excerpts only the 512-bit versions.

The instruction format description given in the reference [18] is shown in column A. VPADD instructions have four operands, Zmm1:destination register, {k1}{z}:mask register and masking mechanism (zeroing/merging), Zmm2:first source operand, zmm3/m512/m32bcst:second source operand. "{}" denotes that it can be omitted. A 512-bit SIMD register, 512-bit data in memory, or 32-bit data in memory can be used as the second source operand. If the 32-bit data type "m32bcst" is specified, the same 32-bit data is copied 16 times and used for all SIMD lanes in the second source register. For 512-bit operands, VPADDD instructions would have nine variations: (no mask/zeroing mask/merging mask) × (zmm3/m512/m32bcst).

These nine variations† are addressed in rows 2–10 in Table 10. All nine supported versions are mentioned in this table. However, explanations for variations that do not occur in oneDNN are omitted in the Excel files for other x86_64 instructions to reduce implementation effort.

Rows 2–10 of Table 10 describe the implementation for creating the AArch64 machine code sequence for the nine variations. The information retrieved by XED is compared to the values described in each row's columns B through E, and matching rows are searched. There should be one row that matches all of them. The description in the row's column F is utilized to generate the AArch64 instruction sequence. For example, if the information of the VPADDD instruction collected by XED is "the third operand is a zmm register, the first operand size equals 512 bits, no mask, and no broadcast," the description of the F2 cell is utilized to construct AArch64 machine code sequence. The F2 cell's add function is the mnemonic function of the ADD instruction in Xbyak_aarch64. The ADD function takes as inputs the destination operand and two source operands. The indexes of the three zmm register operands of the VPADDD instruction are acquired via XED and set to member variables of the "a64" structure by the "decodeAndTransToAArch64()" function. As a result, if the instruction to be converted is "VPADDD zmm5, zmm6, zmm7," the instruction "ADD z5.s, z6.s, z7.s" is generated, which is an AArch64 instruction that performs the same action as "VPADDD zmm5, zmm6, zmm7."

The cells framed by the red rectangles in Table 10 should be designed by the translator's developer. The instructions needed to identify the nine variations of VPADDD instructions are detailed in columns B through E. Columns G through U should be filled out to generate the comparable AArch64 machine code sequences for each variation. The description in column F is made up of Excel formulas that are automatically updated when the columns G through U are changed. The AArch64 instruction sequence to be generated for each variant has many similarities. Cells G1 through U1 list the implementation details, and each row explains whether or not to use them (complete the columns G through U with 1 or leave them blank). Table 10 generates the "vpaddd.h" header file automatically. This file includes the implementation that determines which row corresponds to the XED instruction information and executes the description indicated in column F. Except for the branch instructions, the translator [28], [29] offers header files for 221 mnemonic types.

3.4 Advantages of Xbyak_translator_aarch64

As of June 2021, Intel developers were working on oneDNN, in which they were adding new primitives and optimizing current primitives for the Intel architecture. Even if features are introduced or modified for the x86_64 architecture, they can be quickly transferred to A64FX using the translator. When converting the latest oneDNN to A64FX, the new and improved primitives may employ x86_64 mnemonic functions that the translator does not currently support. In this situation, definition tables for new mnemonics and operand variations must be created. When oneDNN is upgraded, there are only a few new operand variations. It only takes a short time to describe and modify the translation definition table to support the latest oneDNN

†The AVX512 primitives in oneDNN do not employ all of the variations mentioned in the reference [18].

Table 10 Example of translation definition table.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	Instruction	Operand 3 Type	Operand width 0	Mask	Broadcast	Implementation	dstIdx = a64.operands[0].regIdx;	srcIdx = a64.operands[2].regIdx;	src2Idx = a64.operands[3].regIdx;	maskIdx = a64.operands[1].regIdx;	zTmpIdx = xt_push_zreg(&a64);	ldr(ZReg(zTmpIdx), ptr(X_TMP_ADDR));	ld1rw(ZReg(zTmpIdx).s, P_ALL_ONE/T_z, ptr(X_TMP_ADDR));	not_(P_TMP_0.b, P_ALL_ONE.b, PRegB(maskIdx));	add(ZReg(dstIdx).s, ZReg(srcIdx).s, ZReg(src2Idx).s);	add(ZReg(dstIdx).s, ZReg(srcIdx).s, ZReg(zTmpIdx).s);	add(ZReg(zTmpIdx).s, ZReg(srcIdx).s, ZReg(src2Idx).s);	add(ZReg(zTmpIdx).s, ZReg(srcIdx).s, ZReg(zTmpIdx).s);	mov(ZRegS(dstIdx), PReg(maskIdx)/T_m, ZRegS(zTmpIdx));	mov(ZReg(dstIdx).s, P_TMP_0/T_m, 0);	xt_pop_zreg();
2		REG3	512	NO	0	dstIdx = a64.operands[0].regIdx; srcIdx = a64.operands[2].regIdx; src2Idx = a64.operands[3].regIdx; add(ZReg(dstIdx).s, ZReg(srcIdx).s, Zreg(src2Idx).s);	1	1	1						1							
3	VPADDD Zmm1 {k1}{z}, Zmm2, zmm3/m512/m32bcst	MEM0	512	NO	0	dstIdx = a64.operands[0].regIdx; srcIdx = a64.operands[2].regIdx; zTmpIdx = xt_push_zreg(&a64); ldr(ZReg(zTmpIdx), ptr(X_TMP_ADDR)); add(ZReg(dstIdx).s, ZReg(srcIdx).s, Zreg(zTmpIdx).s); xt_pop_zreg();	1	1			1	1				1						1
4		MEM0	512	NO	1		1	1	1		1		1			1					1
5		REG3	512	ZERO	0		1	1	1	1				1	1					1	
6		MEM0	512	ZERO	0		1	1	1	1	1	1		1		1				1	1
7		MEM0	512	ZERO	1	(Omit)	1	1	1	1	1		1	1		1				1	1
8		REG3	512	MERG	0		1	1	1	1	1						1		1		1
9		MEM0	512	MERG	0		1	1	1	1	1	1						1	1		1
10		MEM0	512	MERG	1		1	1	1	1	1		1					1	1		1

on A64FX. Xbyak can be updated by adding a "decodeAndTransToAArch64()" statement, as indicated in Sect. 3.3. The translator should always use the most recent version of XED, and no special adjustments are necessary. As a result, keeping up with oneDNN upgrades is simple.

4. Effectiveness of Translator in Porting oneDNN to A64FX

We confirmed how much the translator improves development efficiency. As a comparison, we used oneDNN's reorder primitive. The man-hours required for port development and debugging are depicted in Fig. 4. a) The ordering primitive implemented with Xbyak is manually rewritten for AArch64 using Xbyak_aarch64. b) is the case when the translator is used. Both a) and b) were performed by the same developer, who was familiar with the AArch64 instruction set but not the x86_64 instruction set when the work began. The amount of steps in the source code to be ported is approximately 1,000 lines, which comprises the Xbyak implementation, the C++ implementation, and comment lines. oneDNN includes a test program for continuous integration (CI). We can utilize these to see if the porting is done appropriately. It was not necessary to implement a new test program in either case a) or b), because the CI test program could be utilized to assess whether the porting was done correctly.

In case a), it took 30 days, as described below.

1. Copy the files "cpu/x64/jit_uni_reorder.[h|c]pp" to the "cpu/aarch64" directory. In the copied files, replace "x64" with "aarch64." Include some C++ codes. Add the presence of a reorder primitive for the SVE instruction set, for example, to the primitive list.

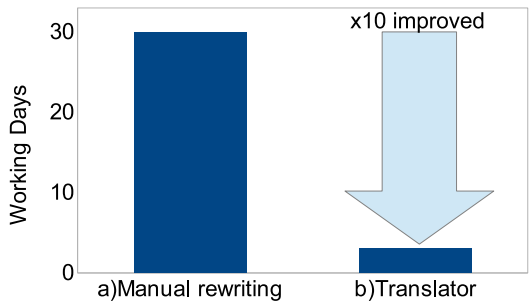


Fig. 4 Efficiency of Xbyak_translator_aarch64 in reducing development time.

2. Read the source code of the "jit_uni_reorder.[h|c]pp" file to understand how the reorder primitive's executable code is generated.
3. Refer to the reference [18] to understand the x86_64 instruction used in the reorder primitive. Rewrite the Xbyak implementation to be compatible with Xbyak_aarch64.
4. Resolve any build errors.
5. Execute the test and review the results. Debug test errors using the method specified in Sect. 3.2.

In a), the number of days required for (2), (3), and (5) was dominant.

(2) and (3) were not required in case b). Instead, the translator has to be substituted for Xbyak of oneDNN by the developer. The developer did not have to manually modify Xbyak mnemonic functions into Xbyak_aarch64 mnemonic functions. (1) and the substitution of Xbyak into the translator took two days and another extra day for the test. The work did not necessitate knowledge of the x86_64 and AArch64 instructions. We also checked that all of the tests passed on the first execution of the test program after the

build error was resolved in (4). According to the data shown above, using translator increases porting development efficiency by tenfold (Fig. 4).

5. Performance Evaluation of the Binaries Translated for A64FX

In this section, we describe the processing performance of the translator's A64FX machine code sequence.

When utilizing the translator, it takes several times longer to generate binaries for A64FX than the original oneDNN. This is due to the suggested method's requirement to build binaries for x86_64, decode them, and then generate binaries for AArch64. However, given the processing characteristics of the DL, as mentioned in Sect. 2.1.1, the time required for this step is insignificant.

Figure 5 depicts the processing performance of oneDNN's 8-bit integer precision convolution primitive. Benchdnn, a microbenchmark included with oneDNN, was utilized for evaluation. The test patterns (Table A· 2) were chosen from a set of 20 convolution patterns that formed when the ResNet-50 [30] model was processed. Figure 5 (a) represents the results of benchmarking the original oneDNN on a Xeon with AVX512 capability, and (b) represents the results of running it on A64FX with the translator. It should be noted that the microarchitecture of the CPU cores, memory system, and operating clock frequency differ between Xeon and A64FX; however, it can be observed that by employing the translator, A64FX can attain almost the same performance as Xeon.

Because the translator converts x86_64 instructions on an instruction-by-instruction basis, there may be redundant instructions or the order of instructions may be inefficient for A64FX when considering the instruction sequence. In this scenario, the part of the converted result that can be optimized can be manually rewritten directly using the Xbyak_aarch64 functions. Even though the source code of oneDNN is partially modified in Xbyak_aarch64, the executable code can be successfully created for A64FX. The machine code of AArch64 is directly written into memory for the component implemented with Xbyak_aarch64, bypassing the Xbyak, XED, and translation table processes. Partially rewritten examples can be found in Appendix B.

Figure 5(c) depicts an example of optimization after approximately two months of evaluating the bottleneck part of the binary generated by the translator and partial rewriting using Xbyak_aarch64. Manual optimization consists of three steps: 1) removing unnecessary instructions, 2) rearranging instructions, and 3) adding software prefetch instructions. 1) and 2) increase the overall processing speed for all patterns. Because the software prefetch instruction has a detrimental effect on some patterns, it was carefully implemented to ensure that the total processing speed of the 20 issues was high. The inclusion of software prefetching instructions to problems 1, 3, and 14 resulted in slow processing speed. Overall, the processing time for the 20 tasks was improved by 5.5%. The performance of the binaries

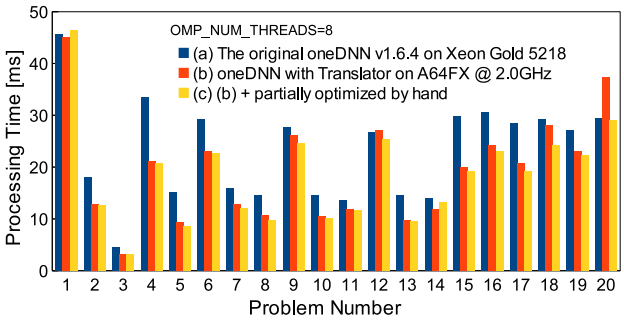


Fig. 5 Processing time of oneDNN 8-bit integer convolution primitive.

created by the translator can be evaluated and profiled using performance counters [7] to detect bottlenecks, and then partially optimized by hand depending on the difficulty of fixing the bottlenecks and the person-hours required.

6. Conclusions

To speed up DL processing on the supercomputer Fugaku, the DL library oneDNN was ported for A64FX CPU. To accelerate DL processing, oneDNN can dynamically generate an executable code that is optimized for runtime conditions. The JIT assembler, Xbyak is used to implement oneDNN's code generation functionality at the x86_64 instruction level. Xbyak_aarch64, the AArch64 JIT assembler, was developed to port oneDNN to A64FX. A binary converter, Xbyak translator_aarch64, has also been developed. It is now feasible to develop oneDNN for A64FX using Xbyak_translator_aarch64 without modifying the source code of the original oneDNN implemented at the x86_64 instruction level. Xbyak_translator_aarch64 has been demonstrated to increase oneDNN's efficiency for A64FX development by a factor of 10. The creation of oneDNN for A64FX has significantly accelerated DL processing on Fugaku. Xbyak_aarch64, Xbyak_translator_aarch64, and oneDNN for A64FX have all shared their source codes on GitHub as OSS [23], [28], [31].

Acknowledgments

The authors thank S. Mitsunari (Cybozu Labs, Inc.), the developer of the original Xbyak. He contributed helpful advice to Xbyak_aarch64 and Xbyak_translator_aarch64, and brushed up the source code. The authors also thank T. Hashimoto (Fujitsu Ltd.). He contributed to evaluating the performance of the binaries translated for A64FX, analyzing bottlenecks, and optimizing them.

References

[1] Japan's Fugaku Retains Title as World's Fastest Supercomputer for three consecutive terms, https://www.fujitsu.com/global/about/resources/news/press-releases/2021/0628-01.html (accessed 2021-06-29)
[2] Fujitsu and RIKEN Complete Joint Development of Japan's Fugaku, the World's Fastest Supercomputer (online), https://www.

fujitsu.com/global/about/resources/news/press-releases/2021/0309-02.html (accessed 2021-06-07).
[3] Shared use of Fugaku begins, https://www.riken.jp/en/news_pubs/news/2021/20210309_2/index.html (accessed 2021-06-21).
[4] Fujitsu and RIKEN Take First Place Worldwide in TOP500, HPCG, and HPL-AI with Supercomputer Fugaku, https://www.fujitsu.com/global/about/resources/news/press-releases/2020/0622-01.html (accessed 2021-06-16).
[5] Japan's Fugaku Retains Title as World's Fastest Supercomputer, https://www.fujitsu.com/global/about/resources/news/press-releases/2020/1117-01.html (accessed 2021-06-16).
[6] T. Yoshida, "Fujitsu High Performance CPU for the Post-K Computer," Proc. Hot Chips 30, Aug. 2018.
[7] A64FX (online), https://github.com/fujitsu/A64FX (accessed 2021-06-07).
[8] Arm Limited or its affiliates, Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile, 2021.
[9] Arm Limited or its affiliates: Arm Architecture Reference Manual Supplement, The Scalable Vector Extension, 2021.
[10] TensorFlow (online), https://www.tensorflow.org/ (accessed 2021-06-14).
[11] PyTorch (online), https://pytorch.org/ (accessed 2021-06-14).
[12] T. Odajima and Y. Kodama, Codesign and System of the Supercomputer, "Fugaku," Proc. CoolCHIPS24, (online), 2021.
[13] NVIDIA cuDNN (online), https://developer.nvidia.com/cudnn (accessed 2021-06-14).
[14] oneAPI Deep Neural Network Library (oneDNN), https://oneapi-src.github.io/oneDNN/ (accessed 2021-06-14).
[15] oneAPI Deep Neural Network Library (oneDNN),https://github.com/oneapi-src/oneDNN (accessed 2021-06-14).
[16] OpenMP, https://www.openmp.org/ (accessed 2021-06-14).
[17] oneAPI Threading Building Blocks (oneTBB), https://github.com/oneapi-src/oneTBB (accessed 2021-06-14).
[18] Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z, 2019.
[19] Xbyak; JIT assembler for x86(IA32), x64(AMD64, x86-64) by C++, https://github.com/herumi/xbyak (accessed 2021-06.16).
[20] GNU Binary Utilities, https://sourceware.org/binutils/docs/binutils/index.html (accessed 2021-06-16).
[21] x86-64 psABI, https://gitlab.com/x86-psABIs/x86-64-ABI (accessed 2021-06-16).
[22] x64 calling convention, https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-160&viewFallbackFrom=vs-2017 (accessed 2021-06-16).
[23] Xbyak_aarch64 (online), https://github.com/fujitsu/xbyak_aarch64 (accessed 2021-06-07).
[24] K. Kawakami, S. Moriyuki, K. Kurihara, and N. Fukumoto, Xbyak_aarch64;JIT Assembler for Next Generation Supercomputer, Proc. CoolCHIPS23, (online), 2020.
[25] K. Kawakami, Xbyak_aarch64; Just-In-Time Assembler for Armv8-A and Scalable Vector Extention, https://connect.linaro.org/resources/lvc21/lvc21-203/ (accessed 2021-06-15), Linaro Virtual Connect 2021, (online), 2021.
[26] Intel X86 Encoder Decoder (Intel XED), https://github.com/intelxed/xed (accessed 2021-06-15).
[27] Intel Corporation, "X86 Encoder Decoder User Guide," https://intelxed.github.io/ref-manual/ (accessed 2021-06-16).
[28] Xbyak_translator_aarch64, https://github.com/fujitsu/xbyak_translator_aarch64 (accessed 2021-06-15).
[29] K. Kawakami, k. Kurihara, M. Yamazaki, T. Honda, and N. Fukumoto, "Just-in-time machine code translator for deep learning processing on supercomputer Fugaku," Proc. CoolCHIPS24, (online), 2021.
[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2016 IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), pp.770–778, Las Vegas, NV, USA, 2016.
[31] oneDNN for A64FX, https://github.com/fujitsu/oneDNN (accessed 2021-06-16).

Appendix A: Test Patterns Evaluated in Fig. 5

Table A·1 displays the benchmarking commands, where "problem" is one of the problems in Table A·2. The data types "u8" and "s8" of "-cfg" are unsigned 8-bit integer and signed 8-bit integer, respectively. For example, "–cfg=u8s8s8" indicates that the input data type is "u8" and the weight and output data types are "s8." The distinct "-cfg" for Xeon and A64FX is because the integer inner product instruction VPDPBUSD (AVX512) and SDOT (SVE) require (input 0, input 1, output) = (unsigned, signed, signed), (signed, signed, signed), respectively. The translator can, of course, convert VPDPBUSD to SVE instructions, but this is unfavorable for A64FX since the translator inserts additional data conversion instructions before and after the SDOT instruction. The original oneDNN features a "s8s8s8" primitive for AVX512, but it also has data conversion instructions before and after the VPDPBUSD instruction, which is inconvenient for Xeon. We contrasted "u8s8s8" and "s8s8s8" to eliminate this inequity. The "s8s8s8" implementation for A64FX may be handled by simply replacing a few steps in the "u8s8s8" code for AVX512 that use the VPDPBUSD instruction with a function of the SDOT instruction in Xbyak_aarch64. This is shown as (b) in Fig. 5.

Table A·1 Benchdnn execution command.

CPU	Command
Xeon	`OMP_NUM_THREADS=8 ./benchdnn --conv --mode=p \ --fix-times-per-prb=5 --reset --dir=FWD_B \ --cfg=u8s8s8 (problem)`
A64FX	`OMP_NUM_THREADS=8 ./benchdnn --conv --mode=p \ --fix-times-per-prb=5 --reset --dir=FWD_B \ --cfg=s8s8s8 (problem)`

Table A·2 Problem list.

Index	Problem
1	mb256ic3ih224iw224oc64oh112ow112kh7kw7sh2sw2ph3pw3n
2	mb256ic64ih56oc256oh56kh1ph0n
3	mb256ic64ih56oc64oh56kh1ph0n
4	mb256ic64ih56oc64oh56kh3ph1n
5	mb256ic256ih56oc64oh56kh1ph0n
6	mb256ic128ih28oc128oh28kh3ph1n
7	mb256ic128ih28oc512oh28kh1ph0n
8	mb256ic512ih28oc128oh28kh1ph0n
9	mb256ic256ih14oc256oh14kh3ph1n
10	mb256ic256ih14oc1024oh14kh1ph0n
11	mb256ic1024ih14oc256oh14kh1ph0n
12	mb256ic512ih7oc512oh7kh3ph1n
13	mb256ic512ih7oc2048oh7kh1ph0n
14	mb256ic2048ih7oc512oh7kh1ph0n
15	mb256ic256ih56oc128oh56kh1ph0n
16	mb256ic128ih56oc128oh28kh3sh2ph1n
17	mb256ic512ih28oc256oh28kh1ph0n
18	mb256ic256ih28oc256oh14kh3sh2ph1n
19	mb256ic1024ih14oc512oh14kh1ph0n
20	mb256ic512ih14oc512oh7kh3sh2ph1n

Table A· 3 Example of optimization by hand.

(a) Original implemenation.

```
for(int i=0; i<8; i++)
  vmulps(Zmm(i), Zmm(i), ptr[r9]);
```

(b) Machine code generated from (a).

```
vmulps zmm0, zmm0, ZMMWORD PTR [r9]
...
vmulps zmm7, zmm7, ZMMWORD PTR [r9]
```

(c) Converted machine code of (b).

```
sub sp, sp, #0x40
str z31, [sp]
ldr z31, [x9]
fmul z0, z0, z31
ldr z31, [sp]
add sp, sp, #0x40
...
sub sp, sp, #0x40
str z31, [sp]
ldr z31, [x9]
fmul z7, z7, z31
ldr z31, [sp]
add sp, sp, #0x40
```

(d) Example of optimizing (a) by hand.

```
xa_->sub(sp, sp, 0x40);
xa_->str(z31, ptr(sp));
xa_->ldr(z31, ptr(x9));
for(int i=0; i<8; i++)
  xa_->fmul(ZRegS(i), ZRegS(i), z31.s);
xa_->ldr(z31, ptr(sp));
xa_->add(sp, sp, 0x40);
```

(e) Machine code generated from (d).

```
sub sp, sp, #0x40
str z31, [sp]
ldr z31, [x9]
fmul z0.s, z0.s, z31.s
...
fmul z7.s, z7.s, z31.s
ldr z31, [sp]
add sp, sp, #0x40
```

Appendix B: Example of Optimization by Hand

An example of manual optimization is shown in Table A· 3. The original implementation (a) produces seven VMULPS instructions with the memory operand "ZMMWORD PTR [r9]]." The translator converts these to (c) for A64FX. Because the data given by the memory operand of VMULPS must be loaded into the register, each VMULPS is translated into six AArch64 instructions in (c). If the original implementation (a) is rewritten by hand with (d), the machine code sequence created for A64FX becomes (e), reducing the total number of instructions generated for A64FX to 13.

Since Xbyak in the translator includes an instance "xa_" of Xbyak_aarch64, we can use the mnemonic functions of Xbyak_aarch64 by writing like "xa_->sub" in (d).

Kentaro Kawakami was born in 1977 in Ishikawa, Japan. From 1995 to 1997, he majored in Physics at Osaka University. He later changed his major and received a B.E. degree in Electrical and Information Engineering in 2002 and M.E. degree in Electronic and Information Systems in 2004 from Kanazawa University, Ishikawa, Japan. He received his Ph.D. degree in engineering from Kobe University, Kobe, Japan, in 2007. He joined Fujitsu Laboratories Ltd. in 2007. He has been involved in R&D of image codec LSIs and wireless sensor nodes and is currently engaged in the R&D of AI software for Arm high-performance computing systems.

Kouji Kurihara received a B.E. from the Department of Electrical Engineering and Computer Science, School of Engineering, Kyushu University in 2007, and M.E. degrees from the Graduate School of Information Science and Electrical Engineering, Kyushu University in 2009. In the same year, he joined the Fujitsu laboratories LTD., Kawasaki, Japan, where he has been engaged in research and development work on embedded multicore processor software, HEVC codec LSIs, wireless sensor nodes system, and visualization of wireless communication interference from 2007 to 2018. His current research interests include AI software for Arm HPC.

Masafumi Yamazaki received his B.E. and M.E. in Mechanical and Materials Engineering from Yokohama National University, Japan in 1994 and 1996, respectively. He joined Fujitsu's semiconductor division in 1996 and worked on the semiconductor circuit design for products such as high-performance custom memory and dynamically reconfigurable processors. Since 2010, he has been involved in the development and operation of large-scale distributed systems. In 2015, he was transferred to Fujitsu Laboratories. His research interests are deep learning applications and acceleration of them using parallel computing.

Takumi Honda received B.E., M.E., and D.E. degrees from the Hiroshima University, Japan, in 2014, 2015, and 2017, respectively. In 2017, he joined Fujitsu Laboratories Ltd., and has engaged in the research and development of high-performance computing. He is currently a senior researcher at the Advanced Computing project in the ICT Systems Laboratory of Fujitsu Ltd. His research interests are in parallel computing and performance optimization on parallel and distributed systems.

Naoto Fukumoto has led the R&D of AI software as a research manager in Fujitsu Ltd. He received his Ph.D. degree in engineering from Kyushu University, Japan, in 2012. His current research interests include high-performance computing systems.