LVC21F-212
Development of Deep Learning Library for AArch64 CPU

Kentaro Kawakami
Scalable Heterogeneous Computing Project,
ICT Systems Lab., Fujitsu Research,
Fujitsu Ltd., Japan
About Me

- Kentaro Kawakami <kawakami.k@fujitsu.com>
- GitHub account: kawakami-k
- Senior Researcher at Fujitsu Research, Fujitsu Ltd., Japan
- Engaged in R&D of AI software for Arm high-performance computing
- Developing the deep learning software stack for the supercomputer Fugaku for the last two years.
  - Fugaku is the world first Arm ISA-based supercomputer.
Table of Contents

● Background & Motivation of porting oneDNN for AArch64
● Calculation kernel generation of oneDNN
● Development of oneDNN for AArch64
  ○ Binary translator to accelerate development
● Summary
Background & Motivation of porting oneDNN for AArch64
Background

● Shared use of the supercomputer Fugaku began in March 2021.
  ○ Three consecutive TOP 500 winner

● Fujitsu A64FX is used as CPU of Fugaku and its derivative supercomputers FX1000/FX700.
  ○ Armv8-A architecture compliant
  ○ Scalable Vector Extension (SVE)
    ■ SIMD instructions for high performance computing
    ■ SVE width of A64FX: 512 bits

Deep learning processing software is required for Arm HPC environment.
Software Stack of Deep Learning Processing

- Software stack of deep learning (DL) consists of two layers.
- Front-end: framework
  - interfaces to deep learning applications and users,
  - handles neural-network model definition.
- Back-end: library
  - provides calculation kernels required for heavy DL processing.

Optimized library for Arm architecture is needed for high-speed DL processing.
Software Stack of Deep Learning Processing

- Software stack of deep learning (DL) consists of two layers.
- Front-end: framework
  - interfaces to deep learning applications and users,
  - handles neural-network model definition.
- Back-end: library
  - provides calculation kernels required for heavy DL processing.

Optimized library for Arm architecture is needed for high-speed DL processing.
Motivation of porting oneDNN for A64FX

- Source code is available as OSS.
- oneDNN can be built with standard C++ compilers.
  - GCC, CLANG, etc.
- Major frameworks support oneDNN.
  - TensorFlow, PyTorch, etc.
- Optimized for CPUs
  - Just-In-Time kernel code generation optimized for runtime parameters
  - Multithreading with OpenMP
  - Input/output tensor data reorder for efficient calculation
Calculation kernels of oneDNN
Calculation Kernels of oneDNN

- Support a variety of calculations required for deep learning processing
  - Convolution, batch normalization, eltwise, pooling, sum, etc.
- Support for various ISAs of Intel CPU
  - AVX512, AVX2, AVX, SSE4.1

<table>
<thead>
<tr>
<th>Convolution</th>
<th>Batch Norm.</th>
<th>Eltwise</th>
<th>Pooling</th>
<th>Sum</th>
<th>Convolution</th>
<th>Batch Norm.</th>
<th>Eltwise</th>
<th>Pooling</th>
<th>Sum</th>
<th>Convolution</th>
<th>Batch Norm.</th>
<th>Eltwise</th>
<th>Pooling</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Calc. kernel generation for AVX512 (512-bit SIMD)</strong></td>
<td>for AVX2 (256-bit SIMD)</td>
<td>for AVX (128/256-bit SIMD)</td>
<td>for SSE4.1 (128-bit SIMD)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xbyak (Just-In-Time assembler for x86_64)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Calculation Kernels Generation

- **Xbyak**: JIT assembler for x86_64
  - Implemented in the standard C++
  - Generate various x86_64 machine code to memory.
    - the scalar instructions, SSE, AVX, AVX2, AVX512 instructions are supported.

- **oneDNN** generates the optimal subroutines (calculation kernels), considering the runtime parameters and the available instruction set (AVX512, AVX2, AVX, SSE4.1).
  - Runtime parameters
    - Input/output tensor shape such as array dimension, array size, data precision, constant coefficients, operation fusion

An entire subroutine can be generated using Xbyak.

Calculation kernels used for DL processing

<table>
<thead>
<tr>
<th>Calculation Kernels</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv. kernel for input (tensor shape and parameters) A</td>
<td></td>
</tr>
<tr>
<td>Con. kernel for input (tensor shape and parameters) B</td>
<td></td>
</tr>
<tr>
<td>Sum kernel for input (tensor shape and parameters) Z</td>
<td></td>
</tr>
</tbody>
</table>
Development of oneDNN for A64FX
Calculation Kernels for A64FX

- Developing oneDNN to A64FX means implementing calculation kernel generation for SVE512.
  - Except for the calculation kernel generation: the interfaces to frameworks, thread control, input/output data order handling, etc., the source code can be used as is, because they are implemented in the standard C++ and independent from H/W architecture.

Calculation kernel generation of oneDNN

- Convolution, Batch Norm., Eltwise, Pooling, Sum
- Calc. kernel generation for AVX512 (512-bit SIMD)
- Calc. kernel generation for SSE4.1 (128-bit SIMD)
- Xbyak (JIT assembler for x86_64)

Memory

- Conv. kernel for input (tensor shape and parameters) A
- Con. kernel for input (tensor shape and parameters) B
- ... Sum kernel for input (tensor shape and parameters) Z

Generate at runtime

Calculation kernels used for DL processing on A64FX (SVE512)
Xbayk_aarch64: JIT Assembler for AArch64

- JIT assembler is the key component of oneDNN.
- JIT assembler for AArch64 Xbayk_aarch64 has been developed.
  - Source code repository https://github.com/fujitsu/xbyak_aarch64
  - Support the scalar, NEON (advanced SIMD), SVE instructions up to Armv8-A v8.2 (v8.3 and higher is in progress.)
- Xbyak_aarch64 works with A64FX, Apple M1, RaspberryPi3, QEMU.
  - Xbyak_aarch64 may work on Android, Jetson Nano and other AArch64 systems as well.
For example, “jit_avx512_*.c|h|pp” files contain the implementation for AVX512.

“jit_uni_*.c|h|pp” files include implementations common to SSE4.1, AVX, AVX2, and AVX512.

Instruction- and register-aware implementation using Xbyak

This example writes 16 “vpmovsdb” x86_64 instructions to memory.

```c
#include "xbyak.h"
...
/* Output machine code to convert 32-bit integers in zmm0~15 registers to 8-bit integers */
for(int i=0; i<16; i++) {
    vpmovsdb(xmm(i), zmm(i));
}
```

Xbyak classes defined by Xbyak

xmm, zmm: 128, 512-bit SIMD register classes defined by Xbyak
The implementation for SVE512 is based on that for AVX512.
- The kernels for AVX512 is fastest.
- AVX512 and SVE512 has similarities.

The source code is relatively easy to reuse.

AVX512 CPU architecture comparison

<table>
<thead>
<tr>
<th></th>
<th>A64FX (SVE512)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD width</td>
<td>512</td>
</tr>
<tr>
<td># of SIMD registers</td>
<td>32</td>
</tr>
<tr>
<td>Available SIMD instructions with mask (predicate) operation?</td>
<td>Yes</td>
</tr>
<tr>
<td># of mask (predicate) registers</td>
<td>8</td>
</tr>
</tbody>
</table>

The source code is relatively easy to reuse.
Example of Rewriting Original Implementation for A64FX

Original source code
"jit_avx512_*(c|h)pp"

/* Output machine code to add 32-bit integers in 512-bit SIMD registers. */
vpadd(zmm0, zmm0, zmm31);

Source code after rewriting for A64FX
"jit_sve_512_*(c|h)pp"

/* Output machine code to add 32-bit integers in SVE registers. */
add(z0, z1, z31);

Simple case

xmm0: 0-th 128-bit SIMD register
zmm31: 31-th 512-bit SIMD register

z0: 0-th SVE register
z30: 31-th SVE register
d, h, b: SIMD lane size = 32, 16, 8 bits
Example of Rewriting Original Implementation for A64FX

Original source code
“jit_avx512_*.cpp”

/* Output machine code to add 32-bit integers in 512-bit SIMD registers. */
vpaddd(zmm0, zmm0, zmm31);

Source code after rewriting for A64FX
“jit_sve_512_*.cpp”

/* Output machine code to add 32-bit integers in SVE registers. */
add(z0, z1, z31);

Simple case

Complex case

xmm0: 0-th 128-bit SIMD register
zmm31: 31-th 512-bit SIMD register

z0: 0-th SVE register
z30: 31-th SVE register
d, h, b: SIMD lane size = 32, 16, 8 bits

Rewrite manually

Rewrite manually

/* Output machine code to convert 32-bit integers in 512-bit SIMD registers to 8-bit integers using signed saturation */
vpmovsdb(xmm0, zmm0);

/* Output machine code to convert 32-bit integers in SVE registers to 8-bit integers using signed saturation */
 mov(z30.d, z0.d);
dup(z0.s, 0);
smin(z30.s, 127); smax(z30.s, -128);
uzp1(z30.h, z30.h, z0.h);
uzp1(z0.b, z30.b, z0.b);
}
Difficulties in Rewriting Source Code

- The number of steps in the source code is large.
- The source code will be expanded as oneDNN is upgraded.

This is a lot of work.

Rewrite manually.

Calculation kernel generation for AVX512
“jit_avx512_*.(c|h)pp”
“jit_uni_*.(c|h)pp”

Xbyak
JIT assembler for x86_64

Calculation kernels for x86_64

oneDNN for AVX512

Calculation kernel generation for SVE512
“jit_sve_512_*.(c|h)pp”
“jit_uni_*.(c|h)pp”

Xbyak_aarch64
JIT assembler for AArch64

Calculation kernels for A64FX

oneDNN for A64FX
Development of Deep Learning Library for AArch64 CPU

Binary Translator to Use AVX512 Implementation Directly for A64FX

oneDNN for AVX512

oneDNN for A64FX

Calculation kernel generation for AVX512
“jit_avx512_*.c|h|pp”
“jit_uni_*.c|h|pp”

Xbyak
JIT assembler for x86_64

Disassembler

Instruction mnemonic, operands

Translation Table

Calculation kernels for A64FX

Xbyak_translator_aarch64

Xbyak_aarch64
JIT assembler for AArch64

Memory

Calculation kernels for x86_64

x86_64 machine code

AArch64 machine code
Binary Translator to Use AVX512 Implementation Directly for A64FX

Development of Deep Learning Library for AArch64 CPU

Xbyak
JIT assembler for x86_64

Disassembler
Instruction mnemonic, operand information

Xbyak_aarch64
JIT assembler for AArch64

Translation Table
Function call

Function call

AArch64 machine code to memory

x86_64 machine code

Branch instructions

LVC21F-212
Binary Translator to Use AVX512 Implementation Directly for A64FX

This interface is identical to the original Xbyak. Therefore, Xbyak in the original oneDNN can be directly replaced with Xbyak_translator_aarch64.

Xbyak has been modified to kick in the disassembler process after generating x86_64 machine code.

These changes are very minor.

For the branch instructions, Xbyak has been changed to call Xbyak_aarch64 functions to directly generate AArch64 branch instructions. It’s easy, because Xbyak and Xbyak_aarch64 are compatible in their implementation of branch destination label handling.
Binary Translator to Use AVX512 Implementation Directly for A64FX

The disassembler is implemented using Intel XED (encoding/decoding library for x86_64).

- **Disassembler**
  - Function call
  - Instruction mnemonic, operand information

- **Translation Table**
  - Function call
  - x86_64 machine code

- **Xbyak**
  - JIT assembler for x86_64
  - Branch instructions

- **Xbyak_aarch64**
  - JIT assembler for AArch64
  - AArch64 machine code to memory

- **Xbyak_translator_aarch64**
Binary Translator to Use AVX512 Implementation Directly for A64FX

Xbyak_aarch64 has not been changed at all from that was originally developed.

- **Xbyak**
  - JIT assembler for x86_64
  - Function call
- **Xbyak_aarch64**
  - JIT assembler for AArch64
  - AArch64 machine code to memory
- **Disassembler**
  - Instruction mnemonic, operand information
  - Function call
- **Translation Table**
  - x86_64 machine code
  - Branch instructions
The translation table is the most difficult part of the translator development. 
- oneDNN generates more than 200 types of x86_64 instructions at runtime. 
- Each instruction has several tens of minor operand variations. 
- The translation table must contain the correspondence between these variations and AArch64 instruction sequence in total.

Xbyak
JIT assembler for x86_64

Xbyak_aarch64
JIT assembler for AArch64

Translation Table

Disassembler

Instruction mnemonic, operand information

Function call

Branch instructions

AArch64 machine code to memory

Function call
Implementation Details of Translation Table

Translation Table

221 Excel files
- add.xlsx
- sub.xlsx
- vpadd.xlsx

221 header files
- add.h
- sub.h
- vpadd.h

Generate
Example: VPADDD(SIMD addition of 32-bit integers)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand variations (zmm:512-bit SIMD registers, k: mask registers, z: mask mode=zeroing, [:] memory operand, [:]b: memory operands with broadcast)</th>
<th>Type of 3rd operand</th>
<th>Masking type?</th>
<th>Memory operand with broadcast?</th>
<th>O</th>
<th>P</th>
<th>Q</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>U</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9 operand type variations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>VPADDD zmm1, zmm2, zmm3</td>
<td>REG3</td>
<td>NO</td>
<td>0 1 1 1 1 1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>VPADDD zmm1, zmm2, m512</td>
<td>MEM0</td>
<td>NO</td>
<td>0 1 1 1 1 1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>VPADDD zmm1, zmm2, m32bcst</td>
<td>MEM0</td>
<td>NO</td>
<td>1 1 1 1 1 1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>VPADDD zmm1, k1z, zmm2, zmm3</td>
<td>REG3</td>
<td>ZERO</td>
<td>0 1 1 1 1 1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>VPADDD zmm1, k1z, zmm2, m512</td>
<td>MEM0</td>
<td>ZERO</td>
<td>0 1 1 1 1 1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>VPADDD zmm1, k1z, zmm2, m32bcst</td>
<td>MEM0</td>
<td>ZERO</td>
<td>1 1 1 1 1 1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>VPADDD zmm1, k1, zmm2, zmm3</td>
<td>REG3</td>
<td>MERG</td>
<td>0 1 1 1 1 1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>VPADDD zmm1, k1, zmm2, m512</td>
<td>MEM0</td>
<td>MERG</td>
<td>0 1 1 1 1 1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>VPADDD zmm1, k1, zmm2, m32bcst</td>
<td>MEM0</td>
<td>MERG</td>
<td>1 1 1 1 1 1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Development of Deep Learning Library for AArch64 CPU

LVC21F-212
### Example: VPADDD (SIMD addition of 32-bit integers)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand variations</th>
<th>Type of 3rd operand</th>
<th>Masking type?</th>
<th>Memory operand with broadcast?</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPADDD zmm1, zmm2, zmm3</td>
<td>(k1){z}, zmm2, zmm3/ m512/ m32bcst</td>
<td>REG3</td>
<td>NO</td>
<td>0</td>
</tr>
<tr>
<td>VPADDD zmm1, zmm2, m512</td>
<td>MEM0</td>
<td>NO</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>VPADDD zmm1, zmm2, m32bcst</td>
<td>MEM0</td>
<td>NO</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>VPADDD zmm1, zmm2, k1z</td>
<td>REG3</td>
<td>ZERO</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>VPADDD zmm1, zmm2, m512</td>
<td>MEM0</td>
<td>ZERO</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>VPADDD zmm1, m32bcst</td>
<td>MEM0</td>
<td>ZERO</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>VPADDD zmm1, k1z, zmm3</td>
<td>REG3</td>
<td>MERG</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>VPADDD zmm1, zmm2, m15</td>
<td>MEM0</td>
<td>MERG</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>VPADDD zmm1, k1z, m32bcst</td>
<td>MEM0</td>
<td>MERG</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Example:** VPADDD (SIMD addition of 32-bit integers)
## Example: VPADDD(SIMD addition of 32-bit integers)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th>N</th>
<th>O</th>
<th>P</th>
<th>Q</th>
<th>R</th>
<th>S</th>
<th>T</th>
<th>U</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Operand variations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(zmm: 512-bit SIMD registers, k: mask registers, z: mask mode = zeroing, [ ]: memory operand, [ ]: memory operands with broadcast)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>VPADDD zmm1, zmm2, zmm3</td>
<td>REG3</td>
<td>NO</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>VPADDD zmm1, zmm2, m512</td>
<td>MEM0</td>
<td>NO</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These columns describe the process of generating an AArch64 instruction using the a64 structure and **Xbyak_aarch64 functions**.

This structure contains the detail information of the operands extracted by the disassembler.

- **Operand type**:
  - Register operand: register type, width, index
  - Memory operand: address register index
  - Immediate value operand: data type, width, value

---

Development of Deep Learning Library for AArch64 CPU

LVC21F-212
### Example: VPADDD(SIMD addition of 32-bit integers)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand variations</th>
<th>Source code output to the header file</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPADDD zmm1, zmm2, zmm3</td>
<td>REG3 NO 0</td>
<td>dstIdx = a64.operands[0].regIdx; srcIdx = a64.operands[2].regIdx; src2Idx = a64.operands[3].regIdx; zTmpIdx = xt_push_zreg(&amp;a64); ldr(ZReg(zTmpIdx), ptr(X_TMP_ADDR)); add(ZReg(dstIdx).s, ZReg(srcIdx).s, ZReg(zTmpIdx).s); xt_pop_zreg();</td>
</tr>
<tr>
<td>VPADDD zmm1, zmm2, m512</td>
<td>MEM0 NO 0</td>
<td>dstIdx = a64.operands[0].regIdx; srcIdx = a64.operands[2].regIdx; src2Idx = a64.operands[3].regIdx; zTmpIdx = xt_push_zreg(&amp;a64); ldr(ZReg(zTmpIdx).s, P_ALL_ONE/T_z, ptr(X_TMP_ADDR)); add(ZReg(dstIdx).s, ZReg(srcIdx).s, ZReg(src2Idx).s); xt_pop_zreg();</td>
</tr>
<tr>
<td>VPADDD zmm1, zmm2, m32bcst</td>
<td>MEM0 1</td>
<td>dstIdx = a64.operands[0].regIdx; srcIdx = a64.operands[2].regIdx; src2Idx = a64.operands[3].regIdx; zTmpIdx = xt_push_zreg(&amp;a64); ld1rw(ZReg(zTmpIdx), 0, P_ALL_ONE/T_z, ptr(X_TMP_ADDR)); add(ZReg(dstIdx).s, ZReg(srcIdx).s, ZReg(zTmpIdx).s); xt_pop_zreg();</td>
</tr>
</tbody>
</table>

This code is generated in the “vpadddd.h” file.
Binary Translator to Use AVX512 Implementation Directly for A64FX

The latest oneDNN

Calculation kernel generation for AVX512
“jit_avx512_*.c|h)pp”
“jit_uni_*.c|h)pp”

Xbyak
JIT assembler for x86_64

Disassembler

Instruction mnemonic, operands

Translation Table

oneDNN for A64FX

Calculation kernels for A64FX

AArch64 machine code

Xbyak_aarch64
JIT assembler for AArch64

Xbyak_translator_aarch64
Translator Constraints

● Only x86_64 instructions used in the current version of oneDNN are supported.
  ○ If future versions of oneDNN will use the new instructions to generate computation kernels, it will be necessary to add translation rule files (Excel files).

● “kmov(q|d|w|b)” instructions must be rewritten manually.
  ○ The correspondence of bit positions in the mask (predicate) registers and SIMD lane is different between AVX512 and SVE. The translator cannot handle this difference.

```
jit_avx512_*.c|h)pp
/* Set flats for 0..15th SIMD lanes */
mov (rax,0xFF);
kmovb (k1,rax);
vpaddd (zmm0,k1,zmm0,zmm1);

jit_sve_512_*.c|h)pp
/* Set flats for 0..15th 32-bit SIMD lanes */
ptrue(p1.s,VL16);
vpaddd (zmm0,k1,zmm0,zmm1);
```

Rewrite manually

Mixed use of Xbyak and Xbyak_aarch64 functions is OK.
Translator Constraints

- If there is a logical instruction before the conditional branch instruction, insert "cmp" instruction.

```
jit_avx512_*.c(h)pp
and(r8, r9);
jnz(jump_destination);
```

Rewrite manually

```
jit_sve_512_*.c(h)pp
and(r8, r9);
cmp(r8, 0);
jnz(jump_destination);
```

AArch64 logical instructions do not change the conditional branch flag so that "cmp" must be inserted to generate the equivalent code for AArch64.

```
/* Machine code for x86_64 */
and r8, r9 /* Set branch flag */
jnz destination_address
```

/* Machine code for AArch64 */
and x8, x8, x9 /* Do not set branch flag */
cmp x8, 0
b.ne destination_address

jnz, b.ne: jump if non-zero flag is set.
Development of Deep Learning Library for AArch64 CPU

Source Code and References

Source codes are available at github.com/fujitsu

- oneDNN for A64FX
  - https://github.com/fujitsu/oneDNN
- Xbyak_translator_aarch64
  - https://github.com/fujitsu/xbyak_translator_aarch64
- Xbyak_aarch64
  - https://github.com/fujitsu/xbyak_aarch64
- PyTorch for A64FX
  - https://github.com/fujitsu/pytorch
- TensorFlow for A64FX
  - https://github.com/fujitsu/tensorflow

References


We are looking forward to your contributions!
Xbyak\_translator\_aarch64 work for SVE256/128?

- Logically speaking, I think it will work.
  - I tried to run the AVX2 implementation “jit\_avx2\_*.c|h\_pp” on SVE256 environment. The convolution test pattern passed. I have not tried other patterns.
  - Similarly, I believe that the implementation for AVX “jit\_avx\_*.c|h\_pp” can be used for SVE128.

- Some instruction conversion rules may be missing.
  - The implementation for AVX2 and AVX may use instructions that are not used for AVX512. In this case, it is necessary to add an Excel file that defines the translation rules.
Xbyak_translator_aarch64 work for SVE256/128?

- The calculation kernel generated by the Translator contains redundant instructions for SVE256/128.

**Calculation kernel generation for AVX2 “jit_avx2_.*(c|h)pp”**

```c
vpadd ymm2, ymm0, ymm1
```

AVX512 CPU

```
0
```

SVE512 CPU (SVE register size=512 bits)

```
ymm2 (256 bits)
zmm2 (512 bits)
```

SVE256 CPU (256 bits)

```
0
```

**“mov z2.s, p8/m, 0” is needed for SVE512 CPU to behave the same as AVX512 CPU.**

```
mov z2.s, p8/m, 0
```

p8 = 0x11..100..0 for SVE512, 0x00..0 for SVE256

**AVX2 instruction clears upper 256 bits of zmm registers.**

**“mov z2.s, p8/m, 0” is redundant for SVE256 CPU.**

```
add z2.s, z0.s, z1.s
mov z2.s, p8/m, 0
```

Unchanged
Summary
Summary

- To achieve high-speed DL processing on HPC systems with A64FX, including the supercomputer Fugaku, we have developed the software stack.
- oneDNN, a de facto DL processing library for CPUs, is ported and optimized for A64FX.
- Since oneDNN is implemented at the instruction level using JIT assembler, it takes a lot of time to port it to A64FX. To accelerate development, we developed the binary translator Xbyak_translator_aarch64, which enables us to divert the implementation of oneDNN to A64FX.
- We hope that our development will be useful for many Arm users.
Acknowledgment

- We would like to express our sincere gratitude to Mr. Shigeo Mitsunari of Cybozu Labs, Inc., the developer of Xbyak.
- Mr. Mitsunari provided us a lot of useful advice on the specification, and he helped us with the development of Xbyak_aarch64 and Xbyak_translator_aarch64.
Thank you

Accelerating deployment in the Arm Ecosystem