Xbyak_aarch64;
Just-In-Time Assembler
for ARMv8-A and
Scalable Vector Extension

Kentaro Kawakami <kawakami.k@fujitsu.com>
Fujitsu Laboratories Ltd., Kawasaki, Japan
About Me

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

- What is Xbyak_aarch64?
  - Sample programs
- Proven working configuration
- Usage of Xbyak_aarch64
- How to debug programs implemented with Xbyak_aarch64
- Summary
What is Xbyak_aarch64?
What is Xbyak_aarch64?

- A JIT assembler for AArch64
- Enables to assemble AArch64 mnemonic at runtime
- We can make generators that produce instruction sequence
- Just-In-Time (JIT) functions are generated by the generators at runtime
- Based on Xbyak that is for x64 CPUs by S. Mitsunari (Cybozu Labs. Inc.)

```cpp
CodeGenerator
Gen_func(*x, N){
    mov(reg0, 0);
    for(i=0; i<N;i++){
        ldr(reg1, x[i]);
        add(reg0, reg0, reg1);
    }
}

Generate JIT func.
Call JIT func.
```

Standard C++ Compiler(g++, clang, …)

C++ application with JIT code gen. and exec.

Various JIT functions can be generated by input parameters:

```
Gen_func(x, 2);
Gen_func(x, 1);
```
/* Write source code in C++11 or later */
#include <xbyak_aarch64/xbyak_aarch64.h>
using namespace Xbyak_aarch64;

class Generator : public CodeGenerator {
public:
    Generator() {
        add(w0, w1, w0);
        ret();
    }
};

int main() {
    Generator gen;
    gen.ready();
    auto f = gen.getCode<int (*)(int, int)>();
    int a = 3, b = 4;
    printf("%d + %d = %d \n", a, b, f(a, b));
    return 0;
}
Generator() {
    add(w0, w1, w0);
    ret();
}

We call these functions as **the mnemonic functions**.
- The function name is one of the mnemonic of ARMv8-A + SVE instruction set.
- Each function outputs a single machine code correspond to the function name, such as “add”, “sub”, “ldr”, “str”, “ret”, etc.
- The instruction operands can be indicated by the function arguments.

Red bold texts are the functions, classes and instances provided by Xbyak_aarch64.

The machine code sequence can be called as a function.
Supported Instructions

- ARMv8-A, ARMv8.1, ARMv8.2, ARMv8.3 instructions
- Scalable Vector Extension (SVE) instructions

~1K mnemonics

Variety of “add” instruction
- 32-bit/64-bit add on general purpose register
- SIMD add on ASIMD register
- SIMD add with/without predicate on SVE register

Variety of “ldr” instruction
- Load general purpose register 32-bit/64-bit
  Post-index/Pre-index/Unsigned offset addressing
- Load ASIMD register
  Post-index/Pre-index/Unsigned offset addressing
- Load SVE register
  With/Without signed immediate offset

Each mnemonic followed by various types of operands can construct a variety of instructions. Xbyak_aarch64 supports all of them.
Advantage of Xbyak_aarch64 compared to Existing Assembler

- Easier to write assembly code
  - Simple assembly description in C++ syntax
  - Loop unrolling is easy to describe

- Optimization using runtime parameters
  - It is possible to change the instructions using parameters

```
for (int j = 0; j < 15; ++j)
fmla(ZRegS(j), PReg(0), ZRegS(j + 15), ZRegS(31));
```

Dynamical unrolling

```
if (isPowOfTwo(param)) {
  int bitPos = 0;
  do { bitPos +=1;
    } while (param = (param >> 1));
  lsr(r0, r1, bitPos); // Logical shift right
} else {
  udiv(r0, r1, param); // unsigned division
}
```

Dynamical instruction selection

Implementation becomes simpler.
(The above code generates 15 “fmla” instructions.)

```
fmla z0.s, p0.s, z15.s, z31.s
fmla z1.s, p0.s, z16.s, z31.s
....
fmla z14.s, p0.s, z29.s, z31.s
```

Considering execution latency, “lsl” is more preferable than “udiv”.
Performance Comparison

- Environment: FX700 / GCC 8.3.1
  - CPU: A64FX (ARMv8-A + 512-bit SVE)
  - Compile options: -march=armv8.2-a+sve -fopenmp -O3

- Measurement conditions
  - Reduction operation for \( N \)-size array \((N = 512)\)
  - iterated 10 million times

Reference code

```c
void generate(int N) {
    size_t offset = sizeof(float) * 16;
    int numZregs = N/16;
    ptrue(PRegS(0));
    for(int i = 0; i < numZregs; i++){
        add(x1, x0, i * offset);
        ldr(ZReg(i), ptr(x1));
    }
    for(int i = numZregs; i > 0; i = i>>1){
        for(int j =0; j < (i/2); j++){
            if((j+(i/2)) < numZregs)
                fadd(ZRegS(j), ZRegS(j), ZRegS(j+(i/2)));
        }
        faddv(SReg(0), PReg(0), ZRegS(0));
    }
    ret();
}
```

Reference code + pragma

```c
float reduction(float* A){
    s = 0;
    for(i = 0; i < N; i++)
        s += A[i];
    return s;
}
```

JIT implementation with `Xbyak_aarch64`

```c
float reduction(float* A){
    s = 0;
    #pragma omp simd reduction (+:s)
    for(i = 0; i < N; i++)
        s += A[i];
    return s;
}
```

Execution time [sec]

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>JIT</td>
<td>0.45</td>
</tr>
<tr>
<td>Reference code</td>
<td>6.30</td>
</tr>
<tr>
<td>Reference code + pragma</td>
<td>23.24</td>
</tr>
</tbody>
</table>

JIT implementation with `Xbyak_aarch64` is x51.6 times faster than the reference code and x14 times faster than the reference code with pragma.
Proven working configuration
Main Development Target
(Since Xbyak_aarch64 is an OSS, it is basically provided as is.)

- H/W: Fugaku, Fujitsu PRIMEHPC FX1000/FX700
  - CPU: Fujitsu A64FX, designed for high-performance computing and complies with the ARMv8-A architecture profile and the Scalable Vector Extension (SVE)
- Compiler: FCC(Fujitsu C/C++ compiler)/GCC/LLVM
- Language: C++11 or later
- OS: Linux (RedHat Enterprise Linux 8.x)
## Proven working configurations

(Since Xbyak_aarch64 is an OSS, it is basically provided as is.)

<table>
<thead>
<tr>
<th>H/W</th>
<th>CPU</th>
<th>OS (64-bit)</th>
<th>Compiler</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX1000*1</td>
<td>Fujitsu A64FX</td>
<td>RedHat Enterprise Linux 8.x</td>
<td>FCC*2</td>
<td>Well-tested oneDNN*3 works</td>
</tr>
<tr>
<td>FX700*1</td>
<td>Fujitsu A64FX</td>
<td>CentOS 8</td>
<td>FCC/GCC/LLVM</td>
<td></td>
</tr>
<tr>
<td>IA server</td>
<td>QEMU 5.0.0 (</td>
<td>(Host OS running on IA server)</td>
<td>GCC</td>
<td>oneDNN works</td>
</tr>
<tr>
<td></td>
<td>Linux user mode)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAC mini, 2020</td>
<td>Apple M1</td>
<td>macOS Big Sur</td>
<td>Apple clang</td>
<td>oneDNN works</td>
</tr>
<tr>
<td>Raspberry Pi3 Model B Rev 1.2</td>
<td>Broadcom BCM2835</td>
<td>Ubuntu 18.04.4 LTS</td>
<td>GCC</td>
<td>Some samples of Xbyak_aarch64 works*4</td>
</tr>
</tbody>
</table>

*2 C/C++ compiler of FUJITSU Software Compiler Package
*3 oneDNN is one of the applications that uses up all the functionality of Xbyak/Xbyak_aarch64.
*4 A limited number of sample programs has been tested.
Usage of Xbyak_aarch64
Usage of Mnemonic Functions

- Prototype declaration of the mnemonic functions
  - https://github.com/fujitsu/xbyak_aarch64/blob/main/xbyak_aarch64/xbyak_aarch64_mnemonic_def.h

```c
void add(const WReg &rd, const WReg &rn, const uint32_t imm, const uint32_t sh = 0);
void add(const WReg &rd, const WReg &rn, const WReg &rm, const ShMod shmod = NONE, const uint32_t sh = 0);
void sub(const XReg &rd, const XReg &rn, const uint32_t imm, const uint32_t sh = 0);
void sub(const XReg &rd, const XReg &rn, const XReg &rm, const ShMod shmod = NONE, const uint32_t sh = 0);
void ldr(const WReg &rt, const AdrReg &adr);
void ldr(const XReg &rt, const AdrReg &adr);
void fmul(const ZRegH &zdn, const _PReg &pg, const ZRegH &zm);
```

- Usage samples

```c
add(w0, w0, 0x2aa); dump();
add(w0, w0, w2); dump();
sub(x0, x0, 0x2aa); dump();
sub(x0, x0, x2); dump();
ldr(w0, ptr(x3)); dump();
ldr(x0, ptr(x3)); dump();
fmul(z0.h, p7/T_m, z7.h); dump();
```
## General Purpose Register Class

<table>
<thead>
<tr>
<th>Class name defined in Xbyak_aarch64</th>
<th>Pre-instantiated variable</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>WReg</td>
<td>w0, w1, ..., w30</td>
<td>32-bit general purpose registers</td>
</tr>
<tr>
<td></td>
<td>wsp, wzr</td>
<td>32-bit stack pointer, zero register</td>
</tr>
<tr>
<td>XReg</td>
<td>x0, x1, ..., x30</td>
<td>64-bit general purpose registers</td>
</tr>
<tr>
<td></td>
<td>sp, xzr</td>
<td>64-bit stack pointer, zero register</td>
</tr>
</tbody>
</table>

WReg dstReg(0);
WReg srcReg0(1);
WReg srcReg1(2);
add(dstReg, srcReg0, srcReg1);
add(w0, w1, w2);
for(size_t i=0; i<16; i++)
    add(WReg(i), WReg(i), WReg(i+1));

(A) Register instances can be freely defined.
(B) These two line generate the same machine code of “add w0, w1, w2”.
(C) Register can be instantiated on the fly.

Xbyak_aarch64 also defines the classes and has the pre-instantiated variables for V (128-bit SIMD), Z (SVE), P (scalable predicate) registers.
Please refer [README.md](#) of Xbyak_aarch64.
Passing parameters to JIT-ed code/
Receiving return value from JIT-ed code

- As JIT-ed code complies the procedure call standard of AArch64, JIT-ed code can freely exchange parameters with the code generated by compiler.

```c
Generator() {
  add(w0, w1, w0);
  ret();
}
```

- The first and second parameters are passed by r0(w0), r1(w1).
- The return value is passed by r0(w0).

---

Table 2. General purpose registers and AAPCS64 usage

<table>
<thead>
<tr>
<th>Register</th>
<th>Special</th>
<th>Role in the procedure call standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td></td>
<td>The Stack Pointer.</td>
</tr>
<tr>
<td>r30</td>
<td>LR</td>
<td>The Link Register.</td>
</tr>
<tr>
<td>r29</td>
<td>FP</td>
<td>The Frame Pointer</td>
</tr>
<tr>
<td>r19...r28</td>
<td></td>
<td>Callee-saved registers</td>
</tr>
<tr>
<td>r18</td>
<td></td>
<td>The Platform Register, if needed; otherwise a temporary register. See notes.</td>
</tr>
<tr>
<td>r17</td>
<td>IP1</td>
<td>The second intra-procedure-call temporary register (can be used by call vneesers and PLT code); at other times may be used as a temporary register.</td>
</tr>
<tr>
<td>r16</td>
<td>IP0</td>
<td>The first intra-procedure-call scratch register (can be used by call vneesers and PLT code); at other times may be used as a temporary register.</td>
</tr>
<tr>
<td>r9...r15</td>
<td></td>
<td>Temporary registers</td>
</tr>
<tr>
<td>r8</td>
<td></td>
<td>Indirect result location register</td>
</tr>
<tr>
<td>r0...r7</td>
<td></td>
<td>Parameter/result registers</td>
</tr>
</tbody>
</table>

From “Procedure Call Standard for the Arm 64-bit Architecture (AArch64)”
Register usage in JIT-ed code

<table>
<thead>
<tr>
<th>Register</th>
<th>Special</th>
<th>Role in the procedure call standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td></td>
<td>The Stack Pointer.</td>
</tr>
<tr>
<td>r30</td>
<td>LR</td>
<td>The Link Register.</td>
</tr>
<tr>
<td>r29</td>
<td>FP</td>
<td>The Frame Pointer</td>
</tr>
<tr>
<td>r19…r26</td>
<td></td>
<td>Callee-saved registers</td>
</tr>
<tr>
<td>r18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r17</td>
<td>IP1</td>
<td>The second intra-procedure-call temporary register (can be used by call veeners and PLT code); at other times may be used as a temporary register.</td>
</tr>
<tr>
<td>r16</td>
<td>IP0</td>
<td>The first intra-procedure-call scratch register (can be used by call veeners and PLT code); at other times may be used as a temporary register.</td>
</tr>
<tr>
<td>r9…r15</td>
<td></td>
<td>Temporary registers</td>
</tr>
<tr>
<td>r8</td>
<td></td>
<td>Indirect result location register</td>
</tr>
<tr>
<td>r0…r7</td>
<td></td>
<td>Parameter/result registers</td>
</tr>
</tbody>
</table>

JIT-ed code
- must save the values on these registers to the stack before use them,
- must restore them before “ret” instruction.

JIT-ed code can be freely use these registers.

“Procedure Call Standard” also defines the usage for V (128-bit SIMD) registers, Z (SVE) resisters and P (scalable predicate) registers of SVE, please refer them.
Register usage in JIT-ed code

Generator() {
    /* stp: store register pair */
    /* pre_ptr: Pre-index addressing */
    /* sp: stack pointer register */
    for(size_t i=19; i<=28; i+=2)
        stp(XReg(i), XReg(i+1), pre_ptr(sp, -16));

    /* Implement what you want to do */
    /* with x0 - x7, x9 - x15, x19 - x28. */

    /* ldp: load register pair */
    /* post_ptr: Post-index addressing */
    for(size_t i=28; i>=19; i-=2)
        ldp(XReg(i-1), XReg(i), post_ptr(sp, 16));

    ret();
}

Red bold texts are the functions, classes and instances provided by Xbyak_aarch64.

- Save the registers before use them
- Restore the registers after use them
Label and Branch Instructions

Generator() {
    Label L1, L2;  // Instancing Label class of Xbyak_aarch64.
    L(L1);         // L function of Xbyak_aarch64 registers JIT-ed code address of this position to Label L1.
    add(w0, w1, w0);
    cmp(w0, 13);   // Compare the register w0 value to the immediate value 13.
    b(EQ, L2);     // Branch to L2, if the register w0 value == 13.
    sub(w1, w1, 1); // Decrement loop counter value.
    b(L1);         // Unconditional branch.
    L(L2);
    ret();
}
Referencing Static Table

```cpp
#include <xbyak_aarch64/xbyak_aarch64.h>
using namespace Xbyak_aarch64;

class Generator : public CodeGenerator {
public:
  Generator() {
    mov(x1, reinterpret_cast<uint64_t>(table));
    add(x1, x1, x0, LSL, 2);
    ldr(w0, ptr(x1));
    ret();
  }

private:
  const uint32_t table[4] =
    {0x00, 0x11111111, 0x22222222, 0x33333333};
};

int main() {
  Generator gen;
  gen.ready();
  auto f = gen.getCode<uint32_t (*)(uint32_t)>();
  uint32_t a = 2;
  printf("table[\%d] = 0x%08\n", a, f(a));
}
```

JIT-ed code

```
B+ 0xffffbe7a0000  mov  x1, #0xf110
0xffffbe7a0004  movk  x1, #0xffff, lsl #16
0xffffbe7a0008  movk  x1, #0xffff, lsl #32
  0xffffbe7a000c  add  x1, x1, x0, lsl #2
0xffffbe7a0010  ldr  w0, [x1]
0xffffbe7a0014  ret
```

native process 41276 In:
(gdb) x/4x $x1
0xfffffffff110: 0x00000000 0x11111111 0x22222222 0x33333333
(gdb)

fx700-01-05.local /home/kawakami/mkldnn/xbyak_aarch64_fujitsu/sample% ./table.exe

```
#include <xbyak_aarch64/xbyak_aarch64.h>
using namespace Xbyak_aarch64;
class Generator : public CodeGenerator {
public:
    Generator() {
        Label table_label;
        adr(x1, table_label);
        add(x1, x1, x0, LSL, 2);
        ldr(w0, ptr(x1));
        ret();
        L(table_label);
        dd(0x0);
        dd(0x01111111);
        dd(0x22222222);
        dd(0x33333333);
    }
};

int main() {
    Generator gen;
    gen.ready();
    auto f = gen.getCode<
        uint32_t (*)(uint32_t)>();
    uint32_t a = 2;
    printf("table[%d] = 0x%08x\n", a, f(a));
}

x700-01-05.local /home/kawakami/mkldnn/xbyak_aarch64_fujitsu/sample% ./table_gen.exe

Generating and Referencing Table

JIT-ed code
Precautions

- Xbyak_aarch64 can output instructions that cannot be executed on the CPU running Xbyak_aarch64.
  - Your CPU may not have support for cryptographic, atomic, SVE instructions etc., but Xbyak_aarch64 running on your CPU can output machine code of these instructions, which raises the illegal instruction exception.
    -> Please check your CPU capability and chose the mnemonic functions.
  - In an extreme case, if Xbyak_aarch64 is run on an x64 machine, any ARMv8-A machine code generated by Xbyak_aarch64 causes the exception.
Precautions

- Xbyak_aarch64 does not validate every argument passed to the mnemonic functions.
  - Example: immediate value of FMOV
    FMOV copies an immediate floating-point constant into every element of SIMD&FP register
    FMOV <Vd>.<T>, #<imm>

Only these constant values are allowed for #<imm> of “FMOV”

From “Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile”
Precautions

- Xbyak_aarch64 does not validate every argument passed to the mnemonic functions.
  - Example: immediate value of FMOV

If you use values that are not listed in the table, the operation of the mnemonic function is undefined. The operand validation is the future work.
How to debug programs implemented with Xbyak_aarch64
Debug JIT-ed Code

- So far, there is no efficient way to debug JIT-ed code 😞
- Basically, it’s the same as debugging assembler.
  - I often use GDB with “asm” layout.
- JIT-ed code can be dump as a file and disassembled by “objdump”.
Debug by GDB

1) Set a break point to the address of the function pointer \( f \), before it is called.
Debug by GDB

2) Set layout to “asm” or “regs”

3) Continue execution

4) The program breaks at the start of JIT-ed code

5) Then, you can step through the instruction level with GDB command “si”.

```
0xffffffff a0000
add w0, w0, w1
0xffffffff a0004
ret
0xffffffff a0008
.inst 0x00000000 ; undefined
0xffffffff a000c
.inst 0x00000000 ; undefined
0xffffffff a0010
native process 38688
Starting program: /b
```

Breakpoint 3, main () at add.cpp:26
(gdb) n
(gdb) p/x f
$f1 = 0xffffffff a0000
(gdb) b *f
Breakpoint 4 at 0xffffffff a0000
(gdb) layout asm
(gdb) c
Continuing.
Breakpoint 4, 0x00000ffffff7a0000 in ?? ()
(gdb) layout regs
```
Dumping JIT-ed Code

```cpp
#include <xbyak_aarch64/xbyak_aarch64.h>
using namespace Xbyak_aarch64;

class Generator : public CodeGenerator {
public:
    Generator() {
        add(w0, w0, w1);
        ret();
    }

    int main() {
        Generator gen;
        gen.ready();
        auto f = gen.getCode<int (*)(int, int)>();
        int a = 3;
        int b = 4;

        FILE *fp = fopen("dump.bin", "wb");
        fwrite(gen.getCode(), gen.getSize(), 1, fp);
        fclose(fp);

        printf("%d + %d = %d\n", a, b, f(a, b));
    }
};
```

```bash
[kawakami@fx700-01-05 sample]$ objdump -D -b binary -m AArch64 dump.bin

dump.bin: file format binary

Disassembly of section .data:

0000000000000000 < data>:
  0: 0b010000 add w0, w0, w1
  4: d65f03c0 ret
```

[kawakami@fx700-01-05 sample]$
Summary
Summary

- Xbyak_aarch64; just-in-time assembler for ARMv8-A + SVE, is introduced, which can dynamically generate optimized code considering runtime parameters and make it easier than the existing assembler to implement optimized code at the instruction level.

- Xbyak_aarch64 is mainly developed to implement the deep learning processing software on the supercomputer Fugaku, but it can be expected to work with a variety of software for ARMv8-a architecture systems.

- Xbyak_aarch64 is being developed as an OSS. I hope that many people will use Xbyak_aarch64 on various platforms and participate in its development.
  - Questions, bug reports, pull requests, etc. on Github are welcome. [https://github.com/fujitsu/xbyak_aarch64](https://github.com/fujitsu/xbyak_aarch64)
Acknowledgment

- The authors thank S. Mitsunari (Cybozu Labs, Inc.), the developer of the original Xbyak. He contributed helpful advice to Xbyak_aarch64 and brushed up the source code.
References

- Xbyak_aarch64: Just-In-Time assembler for ARMv8-A + SVE
  - https://github.com/fujitsu/xbyak_aarch64
- Xbyak: Just-In-Time assembler for x86_64
  - https://github.com/herumi/xbyak
- oneDNN: Deep Learning Processing Library
  - https://github.com/oneapi-src/oneDNN
- oneDNN for A64FX: Deep Learning Processing Library for A64FX
  - https://github.com/fujitsu/oneDNN
- A64FX: CPU designed for high-performance computing and complies with the ARMv8-A architecture profile and the Scalable Vector Extension (SVE)
- “Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile”
- “Procedure Call Standard for the Arm 64-bit Architecture (AArch64)”
- TechBlog
  - https://blog.fltech.dev/entry/2020/11/19/fugaku-onednn-deep-dive-en
Thank you

Accelerating deployment in the Arm Ecosystem
What is Xbyak_aarch64?

- Xbyak_aarch64 ([https://github.com/fujitsu/xbyak_aarch64](https://github.com/fujitsu/xbyak_aarch64)) is the Just-In-Time (JIT) assembler for ARMv8-A + Scalable Vector Extension (SVE), inheriting the concept of Xbyak, written in C++11.
- Xbyak ([https://github.com/herumi/xbyak](https://github.com/herumi/xbyak)) is the Just-In-Time assembler for x86_64 instruction set architecture (ISA),
  - developed by S. Mitsunari (Cybozu Labs, Inc.),
  - pronounced “kai-bja-k” (I'm not sure the correct spelling by IPA), [https://translate.google.com/?hl=ja&sl=ja&tl=en&text=kaibyaku](https://translate.google.com/?hl=ja&sl=ja&tl=en&text=kaibyaku)
  - The word “Xbyak” is derived from Japanese word “開闢”.
    - Its meanings is "the beginning of the world", "exploring the unexplored", etc.
- The main purpose of developing Xbyak_aarch64 is to port oneDNN, a deep learning processing library for x86_64, to A64FX (ARMv8-A + SVE).