Vol.2021-HPC-180 No.12
2021/7/20

AArch64 CPU向けディープラーニング ライブラリ開発を加速するバイナリトランスレータ

川上 健太郎[1,a)]　栗原 康志[1]　山崎 雅文[1]　本田 巧[1]　福本 尚人[1]

概要：筆者らは、スーパーコンピュータ富岳上でのディープラーニング (DL) 処理を高速化するため、CPU向け DL 処理ライブラリの oneDNN を Fujitsu A64FX 向けに移植，最適化している。富岳は Armv8-A アーキテクチャに準拠した Fujitsu A64FX を CPU として搭載している CPU ベースのスーパーコンピュータである．一方，oneDNN は Intel が x86_64 アーキテクチャ向けに開発し OSS 化している DL ライブラリである．oneDNN は，計算カーネル部分の実行コードを動的に生成する機能を持つ．実行コードの動的生成機能は，x86_64 アーキテクチャ向け Just-In-Time(JIT) アセンブラの Xbyak を用いて x86_64 命令の粒度で実装されている．oneDNN を A64FX 向けに移植するためには，これを Armv8-A アーキテクチャ向け JIT アセンブラの Xbyak_aarch64 を用いて Armv8-A 命令に書き換える必要がある．しかし，これは書き換え対象のステップ数が数万行以上あり困難である．本稿では Xbyak を用いて x86_64 アーキテクチャ向けに生成した実行コードを Armv8-A アーキテクチャ向けの実行コードに変換するバイナリトランスレータの Xbyak_translator_aarch64 を紹介する．Xbyak_translator_aarch64 により，oneDNN を A64FX に移植する際にソースコードを書き換える作業がほぼ不要となり，oneDNN の A64FX 向けの移植開発効率を改善できる．

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

Kawakami Kentaro[1,a)]　Kurihara Kouji[1]　Yamazaki Masafumi[1]　Honda Takumi[1]
Fukumoto Naoto[1]

Abstract: To accelerate deep learning (DL) processing on the supercomputer Fugaku, the authors have ported and optimized the oneDNN for Fujitsu A64FX. The oneDNN is the DL processing library developed by Intel for the x86_64 architecture as an open-source software. Fujitsu A64FX is the CPU that complies with the Armv8-A architecture. The oneDNN dynamically generates the execution code of 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 the oneDNN to A64FX, it needs to be rewritten into Armv8-A instructions using Xbyak_aarch64, the JIT assembler for the Armv8-A architecture. This is difficult, because the number of steps to be rewritten is more than several tens of thousands of lines. In this paper, Xbyak_translator_aarch64 is introduced. Xbyak_translator_aarch64 is the binary translator that convert the executable code dynamically generated for the x86_64 architecture into the executable code for the Armv8-A architecture at runtime. Xbyak_translator_aarch64 eliminates the need to rewrite the source code for porting oneDNN to A64FX and enables us to port the oneDNN for A64FX in a short time.

1. はじめに

スーパーコンピューター富岳の供用が 2021 年 3 月 9 日に開始された [1], [2]．富岳は，富士通 A64FX CPU[3], [4] を搭載した CPU ベースのスーパーコンピュータである．

[1] 富士通株式会社
4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki, Kanagawa 211–8588, Japan
[a)] kawakami.k@fujitsu.com

A64FX は Armv8-A architecture profile[5] に準拠している．Armv8-A アーキテクチャには High performance computing(HPC) 用途のために追加定義された Scalable Vector Extension(SVE) 命令 [6] がある．A64FX は, SVE 命令をサポートする世界初の CPU である．本稿では以下, Armv8-A + SVE を AArch64 と表記する．

筆者らは 2019 年 4 月から，富岳の上でディープラーニング (DL) 処理を高速に実行できるソフトウェア (S/W) スタックの開発を行っている．**図 1** に DL 処理の S/W スタックを示す．S/W スタックは，ユーザーと計算機システムの間でニューラルネットワーク定義の記述や入出力データのやりとりを行うフロントエンド側のフレームワーク S/W と，DL 処理で必要とされる大量の計算を実行する機能を提供するバックエンド側のライブラリ S/W からなる．主要なフレームワーク S/W としては Google が開発する TensorFlow[7] と Facebook が開発する PyTorch[8] がある．TensorFlow，PyTorch ともに OS として Linux をサポートしており，そのソースコードはオープンソースソフトウェア (OSS) として公開されている．富岳は OS として RedHat Linux を採用しているため [9]，各種フレームワーク S/W のソースコードをビルドするのは容易である．富岳で用いるフレームワーク S/W としてはこれらを用いることができる．

ライブラリ S/W は，DL 処理で要求される大量の計算を高速に実行するため，ハードウェアプラットフォーム毎に最適化した S/W が提供されている．NVIDIA GPU 向けには NVIDIA から cuDNN[10] が提供されており，Intel CPU 向けには Intel から oneDNN[11], [12] が提供されている．富岳での DL 処理高速化を実現するためには A64FX に最適化した DL ライブラリ S/W が不可欠である．これまで AArch64 アーキテクチャに最適化された DL ライブラリ S/W は存在しておらず，新規に開発が必要である．筆者らはこの状況を解決するため，Intel CPU 向けに開発されている oneDNN を A64FX 向けに移植，最適化することを選択し，開発を進めている．oneDNN を移植することのメリットは以下の点が挙げられる．

- 主要な DL フレームワーク S/W である TensorFlow, PyTorch が oneDNN をバックエンドとしてサポートしている．
- CPU 上での DL 処理に最適化されている．例えば，計算カーネルの処理はマルチスレッド処理 (Intel Threading Building Blocks[13] または OpenMP[14] が利用可) で実装されており, 処理が高速化されている．富岳では OpenMP ライブラリが提供されているので，oneDNN のマルチスレッド処理の実装をそのまま流用できる．
- オープンソースソフトウェアとしてソースコードが公開されている．

本稿では，富岳での高速な DL 処理を実現するために開発している oneDNN for A64FX について述べる．本稿の構成は以下の通りである．2 章で既存の oneDNN の実装について述べる．3 章で oneDNN for A64FX の移植開発について述べる．oneDNN では処理高速化のため計算カーネルは x86_64 命令のレベルで実装されている．oneDNN for A64FX でも同様の高速化を実現するためには x86_64 CPU 向けの実装を AArch64 CPU 向けの実装に書き換える必要があるが，書き換え対象のステップ数やデバッグの点から容易ではない．3 章ではこれを解決するバイナリトランスレータ Xbyak_translator_aarch64 についても述べる．4 章で Xbyak_translator_aarch64 による開発効率向上の効果についてについて述べる．5 章で本稿のをまとめを述べる．

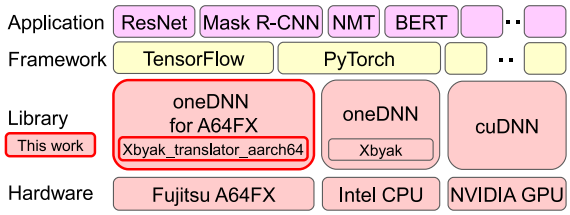


図 1 ディープラーニング処理のソフトウェアスタックの構成
Fig. 1 Software stack of deep learning processing.

2. oneDNN

2.1 oneDNN の特徴

oneDNN は Intel アーキテクチャ (x86_64) の CPU, Intel Processor Graphics, Xe architecture-based Graphics の上で DL 処理を高速に実行するために開発されているライブラリ S/W である．oneDNN は Intel が開発しており，OSS として Github 上でソースコードが公開されている．以下，oneDNN の Intel CPU 向けに実装されている部分について述べる．

oneDNN は Windows, Linux, macOS(Intel CPU) 上での動作をサポートしている．C++で実装されており，GCC, LLVM, Intel Compiler, Microsoft Visual Studio でビルドすることができる．oneDNN は DL ソフトウェアスタックにおいて, DL 処理で繰り返し実行される convolution, ReLU, batch normalization，pooling，softmax，reorder などの計算を高速に実行する機能を担っている．oneDNN ではこれらの機能は primitive と呼ばれている (例:reorder primitive)．oneDNN では，primitive は SSE4.1，AVX，AVX2，AVX512 の各命令セット向けに最適化された実装が用意されており，ユーザーが DL 処理を実行する環境に応じて最適な実装が選択される．各命令セット向けに最適化した primitive の実装は x86_64 アーキテクチャ用 Just-In-Time(JIT) アセンブラ Xbyak[15], [16] を用いて命令レベルでソースコードに記述されている．Xbyak を用いた実装については § 2.1.1 で説明する．

各 primitive は Xbyak を用いた実装とは別に，純粋な

(Xbyak を用いていない)C++で記述されたリファレンス実装がある．x86_64 アーキテクチャでない CPU 上で oneDNN を用いた場合，SSE4.1，AVX，AVX2，AVX512 向けのいずれの実装も使うことができないので，リファレンス実装が選択され使われる．リファレンス実装は各 primitive に求められる計算を機能的には正しく実行するが，ソースコードの読みやすさを優先した実装となっているため処理は低速である．そのため，primitive の種類や処理対象のテンソル形状 (多次元配列の次元，要素数，データ精度) 等にもよるが，Xbyak を用いて最適化された実装と比較して処理速度は数百分の 1 以下である．oneDNN を用いて富岳上での DL 処理を高速化するためには，JIT アセンブラを用いて AArch64 命令セット向けに最適化して実装した primitive が不可欠である．

2.1.1 Xbyak を用いた実行時コード生成

oneDNN では種々の primitive が SSE4.1，AVX，AVX2，AVX512 の各命令セット向けに Xbyak を用いて実装されている．Xbyak は実行時にコード生成する機能を提供するライブラリ S/W で，x86_64 アーキテクチャに対応している．Xbyak のソースコードは OSS として公開されている [16]．

表 1 に Xbyak を用いてプログラムのサンプルコードを示す．表 1 は 引数として 1 以上の整数 N を与えると，N 番目のフィボナッチ数を標準出力に出力する．このサンプルプログラムでは，Xbyak を用いて表の Step (a) の部分でフィボナッチ数を計算する x86_64 機械語列をメモリ上に生成し，Step (b) の部分をそれを関数として呼び出して使用している．Xbyak では x86_64 命令の mnemonic を名前とする関数群 (本稿ではこれを mnemonic 関数と呼ぶ) が実装されており，これらを用いて所望の機械語列を生成するように実装する．表 1 では mnemonic 関数は青色文字で表記されている．命令のオペランドは mnemonic 関数の引数で指定する．x86_64 アーキテクチャでは命令のオペランドとしてレジスタオペランド，メモリオペランド，即値などをとることができる．Xbyak ではいずれにも対応している [15], [16]．また，Xbyak は分岐命令の生成にも対応している．分岐命令のジャンプ先を指定するために Label クラスが用意されており，L 関数を用いて分岐先を指定することができる．Xbyak を用いて生成する機械語列を x86_64 アーキテクチャの application binary interface(ABI) に準拠させれば，引数や返値を持つ関数を生成することができる．x86_64 アーキテクチャ向け Linux では，整数の引数を 1 つを受け取り，整数の返値を返す関数の場合，引数は RDI レジスタで，返値は RAX レジスタを介して受け渡すことが定められている [17]．表 1 が生成する機械語列は，これに準拠している．

表 1 の例では Xbyak が生成する機械語列は常に同じだが，実行環境や処理対象データのパラメータに合わせて生成する機械語列を変えることもできる．**表 2** に oneDNN がコード生成時に考慮するパラメータの例を示す．例えば，SIMD レジスタは AVX2 までをサポートする CPU では 16 個，AVX512 をサポートする CPU では 32 個である．レジスタが多ければ，ループ処理部分のループボディでより多くのレジスタを使ってレジスタアンローリングした実行コードを生成できる．2 倍のレジスタ数でアンローリングすると，ループの繰返し実行回数が半分になり，すなわち条件分岐命令の実行回数が半分になり，処理が高速化される．また，実行時に処理対象として与えられた多次元配列について，(最内次元の要素数) × (データサイズ) が SIMD レジスタ幅の整数倍であれば，SIMD レジスタに収まらない配列末端の要素の処理を考慮する必要がなくなる．これを前提としてループ処理部分の機械語列を簡単化して生成し，これを用いることで，処理を高速化できる．このように oneDNN では，DL 処理で繰り返し実行される primitive の実行コードを，実行時のパラメータを考慮して最適化して生成することで高速化を実現している．JIT アセンブラを用いた primitive の実行コードの動的生成は，oneDNN の高速化を支えるキー技術である

JIT アセンブラを用いて生成される関数は primitive の種類およびパラメータの組合せ毎に生成される．生成された関数はメモリ上に保持され，関数生成時に考慮したパラメータが同一であれば再利用される．大量のデータに対して繰り返し処理を実行する DL の特性を考えると，関数生成に必要な時間は無視できる．

表 1 Xbyak を用いたサンプルプログラムのソースコード
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[]) {
  Generator gen;
  gen.ready();
  auto f = gen.getCode<int (*)(int)>();                  Step (b)
  std::cout << f(atoi(argv[1])) << std::endl;
  return 0;
}
```

表 2　実行コード生成で考慮されるパラメータの例
Table 2　Example of Parameters Considered for Dynamic Code Generation.

Parameters	Example
Availabel instruction set	SSE4.1, AVX, AVX2, AVX512
# of SIMD registers	16, 32
SIMD register width	128, 256, 512
Input/Output data size	16, 32
Input/Output data precision	float16, float32, int32, int8

表 3　Xbyak_aarch64 を用いて書き換えたソースコードの例
Table 3　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);  /* F(1) = 1 */
   mov(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();
}
```

3. oneDNN for A64FX の開発

3.1 Xbyak_aarch64

§ 2.1.1 で述べたように，JIT アセンブラは oneDNN の高速処理を支える技術である．oneDNN を A64FX に移植するために，筆者らは AArch64 アーキテクチャに対応した JIT アセンブラ Xbyak_aarch64 を開発した [18], [19], [20]. Xbyak_aarch64 は SVE 拡張を含む AArch64 アーキテクチャで実行可能な機械語列の生成に対応している．Xbyak と同様，Xbyak_aarch64 も標準的な C++コンパイラでビルドでき，C++で実装するプログラムから使用できる．

表 3 に表 1 を Xbyak_aarch64 を用いて書き換えたソースコードを示す．赤文字の関数は Xbyak_aarch64 の mnemonic 関数を表す．Generator 関数以外については，inlcude ファイルの名前や namespace を Xbyak_aarch64 に変更する必要があるが，それ以外は表 1 と同じである．AArch64 の命令セットやアーキテクチャに合わせて mnemonic 関数の名前，オペランドを指定するために mnemonic 関数に与える引数の数や型，レジスタの名前が異なる．Xbyak_aarch64 でも分岐命令のジャンプ先の指定は Label クラスと L 関数を用いて行うことができる．Label クラスの実装や使い方は Xbyak_aarch64 と Xbyak で同じである．

表 4　AVX512 命令対応の CPU と A64FX のレジスタ数の比較
Table 4　Comparison of # of registers between AVX512 CPU and A64FX.

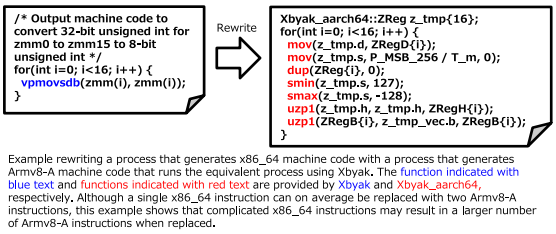
	AVX512	A64FX (Armv8-A + SVE)
# of general purpose registers	16	32
SIMD register width	512	512
# of SIMD registers	32	32
# of mask (predicate) registers	8	16

3.2 primitive の A64FX 向け移植

Xbyak_aarch64 の開発により，JIT アセンブラ技術を用いて A64FX 向けに各種 primitive を移植できる環境が整った．oneDNN には SSE4.1, AVX, AVX2, AVX512 のそれぞれに最適化した実装がある．SIMD 幅はそれぞれ 128，128，256，512 ビットであるが，SIMD 幅が最も大きい AVX512 の実装が最も処理速度が高速である．AArch64 の SVE 命令は SIMD 命令であるが，SIMD 幅は CPU の実装依存で，CPU の開発ベンダが $128 \times N \quad (N = 1, 2, \cdots, 16)$ の中から選択できる．A64FX では SVE を $N = 4$ の 512 ビット SIMD として実装している．A64FX 向けの primitive の実装として，同じ 512 ビット幅の SIMD 命令向けに実装している AVX512 向けの実装をベースとして用いることにした．AVX512 では SIMD のレーン毎に命令実行をする/しないを制御するマスク付きの命令が導入されている．SVE でも同様にマスク付きの命令が導入されているので，マスク付きの AVX512 命令の実行コードを生成する箇所では，マスク付きの SVE 命令[*1]の実行コードを生成するように置き換えることで対応できる．

表 4 に AVX512 をサポートする CPU と A64FX のレジスタ数の比較を示す．A64FX は汎用レジスタ，SIMD レジスタ，マスクレジスタのいずれも AVX512 をサポートする CPU と同じかそれ以上の数を持つ．従って，AVX512 向けに Xbyak を用いて実装している primitive を Xbyak_aarch64 を用いて A64FX 向けに書き換えていく際，各レジスタをどの用途で使用するかの割り当ては AVX512 向けの割り当てをそのまま適用することができる．例えば，AVX512 向けの実装で 512 ビット SIMD レジスタである ZMM レジスタの 0 から 15 番を入力データの load 先と計算途中の値の保持に，16 から 31 番を計算で使用する係数を保持に使うような実行コードを生成していたとする．A64FX 向けの実装では同じように 512 ビット SIMD レジスタの Z レジスタの 0 から 15 番を入力データの load 先と計算途中の値の保持に，16 から 31 番を計算で使用する係数の保持に使う様な実行コードを生成するようにすればよい．ただし，AVX512 向けの実装において，すべての SIMD レジスタに用途が割り当てられており，かつメモリオペランドを伴う SIMD 命令が使われている primitive では注意する必

[*1] AArch64 では predicate 付きの SVE 命令と呼ぶ．



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.

図 2 AVX512 向けの実装を A64FX 向けの実装に置き換える例
Fig. 2 Example of rewriting the implementation.

要がある．AArch64 では一部の例外を除いて，オペランドはすべてレジスタオペランドである．メモリオペランドを伴う AVX512 命令を SVE 命令に置き換える際，AArch64 ではメモリオペランドの代わりにメモリから一時的にデータを SIMD レジスタに load する命令と，そのレジスタをソースオペランドとする SVE 命令とに置き換える必要がある．一時的なデータの load 先として使用する SIMD レジスタが破棄してはいけないデータを保持している場合，このデータを一時的にスタックメモリに退避する store 命令と，後にそれを復元する load 命令が追加で必要になる．

oneDNN の A64FX 向けの移植開発のためには，§ 3.1 で説明したように，AVX512 向けに Xbyak を用いて実装されているソースコードを Xbyak_aarch64 を用いて書き換えればよい．ただし，これは以下にあげる課題がある．

- 書き換え対象のステップ数が大きい．oneDNN では JIT 技術を用いた primitive が実装されたファイルは src/cpu/x64 ディレクトリ配下の jit_で始まるファイル名として置かれている．このうち，A64FX 向けに移植が必要なソースコードは AVX512 用の実装を含む jit_avx512_で名前が始まるファイルと，各命令セット共通の実装を含む jit_uni_で始まるファイルである．これらのソースコードの合計ステップ数は 8 万行 (コメント，空行を含む) 以上ある．
- 書き換えには，x86_64 命令セットと AArch64 命令セットのそれぞれの処理内容と，それらの対応関係の知識を習得しなければならず，困難である．例えば x86_64 の VPMOVSDB 命令は，AArch64 命令列に置き換えると 7 命令になる (**図 2**)．
- 書き換えミスがあった際のデバッグが難しい．デバッグは，Xbyak_aarch64 を用いて命令レベルの抽象度で記述されたソースコードを確認する，生成された実行コードを逆アセンブルしたテキストファイルを確認する，生成した実行コードを GDB で命令レベルでステップ実行しながら確認するなどで行うことができる．いずれの方法を用いても，1 命令単位の記述間違いを特定するのは簡単ではない．

また，oneDNN は，日々機能拡張と改善が行われいる．新しい種類の primitive が追加されたり，既存の primitive

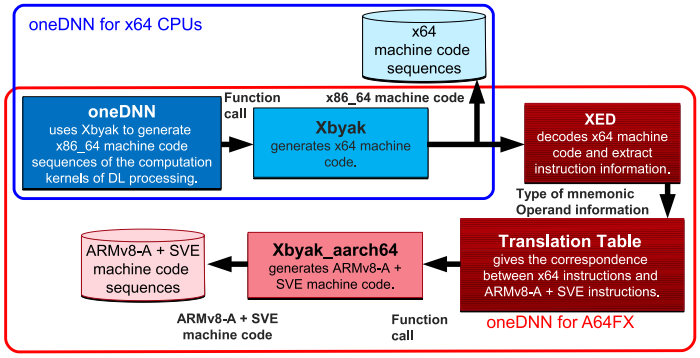


図 3 Xbyak_translator_aarch64 の処理フロー
Fig. 3 Flow of Xbyak_translator_aarch64.

も速度改善のためにソースコードが修正されている．これらに追従して A64FX 向けに最新 oneDNN を提供し続けるためにも，移植開発の負荷を削減できる方策が必要である．

3.3 移植開発を加速するバイナリトランスレータ

§ 3.2 で述べた課題を解決するため，筆者らはバイナリトランスレータ Xbyak_translator_aarch64(以下，Translator と表記する)[21], [22] を開発した．Translator は，oneDNN を A64FX 向けに移植する際に必要な oneDNN のソースコードの修正量を大きく削減する．**図 3** に Translator の構成とフローを示す．Translator は Xbyak，XED，Translation Table，Xbyak_aarch64 を含む．Translator 内の Xbyak はオリジナルの Xbyak と同一のインターフェースを持つ．x86_64 向けの oneDNN で使用されている Xbyak を Translator と置き換えることで，x86_64 向けの oneDNN を A64FX 向けの oneDNN に改変することができる．オリジナルの Xbyak は mnemonic 関数が 1 回 call されるごとに，関数名に対応する 1 つの x86_64 命令の機械語を生成する．この処理が，oneDNN の中の Xbyak を Translator に差し替えることにより，図 3 に示す処理に置き換わる (分岐命令を名前に持つ mnemonic 関数は除く)．

(1) Xbyak の mnemonic 関数が x86_64 の 1 命令分の機械語を生成する．
(2) 生成した機械語から XED を用い，命令の mnemonic，オペランドの数とタイプなど (**表 5**) と，各オペランドの詳細情報 (**表 6**) を取得する．
(3) 前ステップで得た情報を Translation Table に入力する．Translation Table は x86_64 命令と，それと同等の処理を実現する AArch64 命令列の定義を持っており，入力された情報に基づき，必要な Xbyak_aarch64 の関数を call し，AArch64 命令列を生成する．詳細は § 3.3.1 で説明する．

XED[23] は Intel が OSS としてソースコードを公開している x86_64 命令のエンコード/デコード機能を提供するライブラリである．Translator では，XED のデコード機能を用いて Xbyak が生成する機械語から必要な情報を出力

表 5　XED を用いて取得する命令の情報
Table 5　Instruction Information Extracted by XED.

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

表 6　XED を用いて取得するオペランドの情報
Table 6　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
Scale	0, 1, 2, 4	No	No	Yes
Displacement	Integer value	No	No	Yes

させている．

Translator に使用している Xbyak はオリジナルの Xbyak から一部修正している．オリジナルの Xbyak の各 mnemonic 関数では対応する x86_64 の機械語を生成して処理を終えるのに対し，Translator 内の Xbyak では分岐命令以外の各 mnemonic 関数の最後で，図 3 の XED 以降の処理を行う decodeAndTransToAArch64 関数を call する 1 文を追加している．

Xbyak の分岐命令に関する mnemonic 関数については x86_64 の命令の機械語を生成せずに，直接 Xbyak_aarch64 の分岐命令の mnemonic 関数を call するように変更している．**表 7** は x86_64 の JG(Jump if greater than) 命令の mnemonic 関数で，AArch64 で対応する B.GT 命令の機械語を生成するように Xbyak_aarch64 の mnemonic 関数を call する様に修正した例である．Translator の中の Xbyak の Label クラスは Translator の Label クラスの子クラスとして実装しているため，oneDNN から Xbyak の mnemonic 関数に与えられた Label クラスの引数をそのまま Xbyak_aarch64 の mnemonic 関数に渡すことができる．

3.3.1　Translation Table の実装

Translator の開発において最も重要なのは図 3 の Translation Table である．oneDNN の AVX512 向けの実装を

表 7　分岐命令の mnemonic 関数の修正例
Table 7　Example of Modifying Mnemonic Function in a Branch Instruction.

```
void Xbyak::CodeGenerator::jg(const Label &label, LabelType type) {
  b(Xbyak_aarch64::GT, label);
}
```

Translator で対応するためには，x86_64 の 229 種類の mnemonic に対応する必要があった．筆者らは mnemonic ごとに Microsoft Excel ファイルを作成し，そこに変換対象の x86_64 命令と対応する AArch64 命令列を生成するために必要な Xbyak_aarch64 の mnemonic 関数の対応を記述する方法を用いた．**表 8** に，x86_64 の VPADDD 命令とそれに対応する AArch64 命令列の定義を記述した例を示す．紙面のスペースの都合により，F4 から F10 セルは省略表記している．VPADDD 命令は 32 ビット整数同士を加算する SIMD 命令であり，SIMD 幅が 128/256/512 ビットのバリエーションがある．表 8 ではそのうち，512 ビットのバリエーションの部分のみを抜粋している．

A 列の「VPADDD Zmm1, k1z, Zmm2, zmm3/ m512/ m32bcst」は文献 [24] に定義されている命令記述の書式である．Zmm1 はデスティネーションオペランドを，k1 はマスク付き SIMD 命令の場合のマスクレジスタ，z は zeroing マスクか merging マスクかを，Zmm2 は 1 つめのソースオペランドを表す．{} は省略できることを表す．zmm3/m512/m32bcst は 2 つめのソースオペランドを表す．それぞれ 2 つめのソースオペランドが zmm レジスタ，メモリ上の 512 ビットデータ，メモリ上の 32 ビットデータを SIMD レーン分コピーして使うことを表す．「VPADDD Zmm1, {k1}{z}, Zmm2, zmm3/ m512/ m32bcst」は，マスクなし/zeroing マスクあり/merging マスクあり) × (2 つめのソースオペランドが zmm3/m512/m32bcst) の合計 9 パターンのバリエーションがあることになる．Translator では 2 行目から 10 行目にて，この 9 種類のバリエーションに対応している[*2]

表 8 では 9 種類のバリエーションに対して，2 から 10 行目に AArch64 機械語列を生成するための実装が記述されている．XED で取得した命令の詳細情報と B から E 列の記述を比較し，一致する行の I 列の記述を用いて AArch64 機械語列が生成される．例えば変換対象の VPADDD 命令の詳細情報で，第 3 オペランド zmm3，第 1 オペランドのサイズが 512 ビット，マスクなし，ブロードキャストなしの場合，I2 セルの記述を用いて AArch64 機械語列が生成される．I2 セルに記述されている add 関数は Xbyak_aarch64 の ADD 命令の mnemonic 関数であり，デスティネーションオペラ

[*2] oneDNN の AVX512 向けの primitive の実装では，文献 [24] で定義されているすべてのバリエーションが出現するわけではない．Translator では実装工数を削減するために，出現していないバリエーションに対してはエラーを返し，AArch64 機械語列への変換は行わない．

表 8　命令変換テーブルの例
Table 8　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(src1Idx).s, ZReg(src2Idx).s);	add(ZReg(dstIdx).s, ZReg(srcIdx).s, ZReg(zTmpIdx).s);	add(ZReg(zTmpIdx).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	VPADDD Zmm1 {k1}{z}, Zmm2, zmm3/m512/m32bcst	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		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

ンド，2 つのソースオペランドにすべて Z レジスタを指定する引数が与えられている．VPADDD 命令の 3 つの zmm レジスタオペランドの index は decodeAndTransToAArch64 関数が XED を用いて取得し，a64 構造体のメンバ変数にセットしている．以上により，第 3 オペランド zmm3，第 1 オペランドのサイズが 512 ビット，マスクなし，ブロードキャストなしの命令である「VPADDD zmm5, zmm6, zmm7」に対して，同じ処理を AArch64 で実現する「ADD z5.s, z6.s, z7.s」命令の機械語が生成される．

表 8 において，Translation Table の開発者が記述する必要があるのは赤罫線で囲ったセルである．VPADDD 命令の 9 つのバリエーションを特定するために用いる命令の詳細情報を B から E 列に記述する．各バリエーションと等価な AArch64 機械語列を生成するように G から U 列を記述する．F 列の記述は，Excel の数式を用い，G から U 列に記述すると自動で更新されるようになっている．VPADDD 命令の 9 つのバリエーションはそれぞれ，同等の処理を実現する AArch64 機械語列生成に必要な実装が共通する部分が多いので，G1 から U1 セルに実装内容を記述し，各行でそれらを用いる (G から U 列に 1 を記入する) か用いないか (G から L 列を空白にする) を記述する書式にしている．表 8 からヘッダーファイル vpaddd.h が自動で生成される．vpaddd.h ファイルは，VPADDD 命令の変換処理において，XED から得た命令やオペランドの情報が B から E 列記載の条件に該当するときに，対応する F 列の処理を行い，変換対象の x86_64 命令と同じ処理を行う AArch64 機械語列を生成する．B から E 列，G から U 列は必要に応じて列数を増やして用いる．Translator[22] では，分岐命令を除く 221 種類の mnemonic に対応するヘッダーファイルが提供されている．

3.4　Translator のメリット

2021 年 6 月現在，oneDNN は Intel の開発者を中心に x86_64 向けに新しい primitive の追加，既存の primitive の最適化が続けられている．Translator により，x86_64 向けに機能追加，修正がなされても，短時間で A64FX 向けに移植することができる．最新の oneDNN を A64FX 向けに移植する際，以前の oneDNN と比較して Translator が未対応の x86_64 命令の mnemonic 関数やオペランドのバリエーションが使われることがある．この場合，Translator が未対応の x86_64 命令に対応して，新たに対応する AArch64 命令列の定義を加えることになる．これまで，oneDNN のバージョンアップに際し，Translator が新規に対応する必要が生じた x86_64 命令のバリエーションは数個程度であった．これに必要な Excel ファイルの追加修正は数日である．

4.　Translator による oneDNN の移植開発の効率の向上

Translator による移植開発の改善効果を確認した．oneDNN の reorder primitive の実装を対象とし，a)Xbyak を用いて実装されている部分を人手で Xbyak_aarch64 を用いて書き換えた場合と，b)Translator を用いて移植する場合の工数を比較した．a)，b) のいずれも作業開始時点では AArch64 命令セットの知識を有するが，x86_64 命令セットの知識は有していない同一の開発者が行った．移植対象となるソースコードのステップ数は約 1,200 行 (Xbyak

の mnemonic 関数 call 部分，その他の C++で実装した部分，コメント行を含む) である．oneDNN では Continuous Integration のためのテストプログラムが含まれている．a)，b) いずれも，正しく移植できたかはこのテストプログラムを用いればよいので，新規にテストプログラムを実装する必要なかった．

図 4 に移植開発およびデバッグに要した工数を示す．a) では 30 日間を要した．作業内容は次の通りである．

(1) cpu/x64/jit_uni_reorder.[h|c]pp ファイルを cpu/aarch64 ディレクトリにコピーし，コピーしたファイル内のいくつかの x64 という単語を aarch64 という単語に置換したり，SVE 命令セット向けの reorder primitive の実装を oneDNN に登録するなど，C++レベルで必要な修正を行う．
(2) jit_uni_reorder.[h|c]pp ファイルのソースコードを読み，どのように reorder primitive の実行コードを生成しているかの理解する，
(3) reorder primitive の実行コード生成で使用されている x86_64 命令を文献 [24] を参照して理解する．それに対応する AArch64 命令列を検討しながら Xbyak_aarch64 の mnemonic 関数に書き換える．
(4) ビルドエラーとなる実装エラーを修正し，テストプログラムを実行する．
(5) エラーが生じたをテストパターンを確認し，3.2 で述べた方法でデバッグを行い，バグを修正する．

a) では (3)，(4)，(5) にかかる日数が支配的であった．

b) では (2)，(3)，(4) の作業は不要である．代わりに oneDNN の Xbyak を Translator に差し替えが必要になる．reorder primitive ではマスクレジスタを使用する x86_64 命令は使われていなかったため，Xbyak を使って実装されている部分は修正不要である．(1) および Translator への差し替えに 2 日，テストプログラムの実行に 1 日の合計 3 日間で完了した．作業において，x86_64 命令や AArch64 命令の知識は不要であった．また，(4) でビルドエラーを解消した後，初回のテストプログラム実行ですべてのテストがパスすることを確認できた．以上より，Translator を用いると移植開発の効率が 10 倍高速になることを確認した．

5. まとめ

x86_64 CPU を対象として開発されている DL 処理ライブラリ S/W の oneDNN を，AArch64 アーキテクチャに準拠した CPU である富士通 A64FX 向けに移植開発した．oneDNN は DL 処理を高速化するため，実行コードを動的に生成する機能を持つ．コード生成は JIT アセンブラ Xbyak を用いて，x86_64 命令レベルで実装されている．筆者らは，oneDNN を A64FX 向けに移植開発に必要な AArch64 向け JIT アセンブラ Xbyak_aarch64 を開発した．Xbyak_aarch64 により，

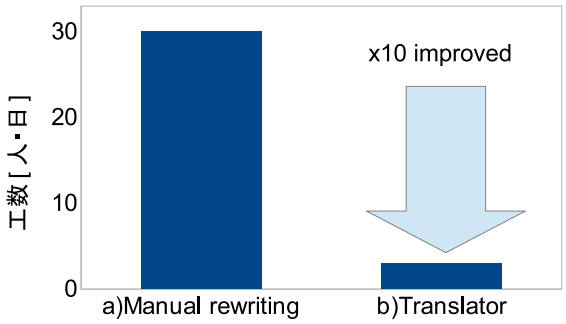


図 4 Translator による移植開発効率化の効果
Fig. 4 Efficiency of Translatorin reducing developemnt time.

x86_64 向けに実装された oneDNN を A64FX 向けに移植開発が可能になった．さらに，Xbyak_aarch64 を直接用いて命令レベルでのソースコード書き換えることなく，A64FX 上で oneDNN を高速に動作可能とするバイナリトランスレータ Xbyak_translator_aarch64 を開発した．Xbyak_translator_aarch64 により，x86_64 向けに実装された oneDNN のソースコードをほとんど書き換える必要がなくなり，移植開発の効率が 10 倍改善されることを確認した．筆者らは oneDNN for A64FX の開発により，スーパーコンピュータ富岳上での DL 処理高速化に大きく貢献できたと考える．Xbyak_aarch64，Xbyak_translator_aarch64，oneDNN for A64FX はいずれも，OSS として github 上でソースコードを公開しており，富岳ユーザーをはじめ，多くのユーザーが利用できるようになっている．

謝辞 Xbyak_aarch64, Xbyak_translator_aarch64 の開発においては、サイボウズ・ラボ株式会社の光成 滋生氏に仕様面，技術面，実装面で多くのアドバイスを頂きました．筆者一同，心より感謝申し上げます．

参考文献

[1] 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).
[2] Shared use of Fugaku begins, https://www.riken.jp/en/news_pubs/news/2021/20210309_2/index.html (accessed 2021-06-21).
[3] Toshio Yoshida: “Fujitsu High Performance CPU for the Post-K Computer,” in Proc. Hot Chips 30, Aug. 2018.
[4] A64FX (online), https://github.com/fujitsu/A64FX (accessed 2021-06-07).
[5] Arm Limited or its affiliates: Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile, 2021.
[6] Arm Limited or its affiliates: Arm Architecture Reference Manual Supplement, The Scalable Vector Extension, 2021.
[7] TensorFlow (online), https://www.tensorflow.org/ (accessed 2021-06-14).
[8] PyTorch (online), https://pytorch.org/ (accessed 2021-06-14).

[9] T. Odajima and Y. Kodama: Codesign and System of the Supercomputer “Fugaku”, *Proc. CoolCHIPS24*, (online), (2021).
[10] NVIDIA cuDNN (online), https://developer.nvidia.com/cudnn (accessed 2021-06-14).
[11] oneAPI Deep Neural Network Library (oneDNN), https://oneapi-src.github.io/oneDNN/ (accessed 2021-06-14).
[12] oneAPI Deep Neural Network Library (oneDNN), https://github.com/oneapi-src/oneDNN (accessed 2021-06-14).
[13] oneAPI Threading Building Blocks (oneTBB), https://github.com/oneapi-src/oneTBB (accessed 2021-06-14).
[14] OpenMP, https://www.openmp.org/ (accessed 2021-06-14).
[15] 光成 滋生: 後期のアセンブラによる x86/x64 CPU 向け高速化テクニック, 夏のプログラミング・シンポジウム 2012「ビューティフルコード」 (2012).
[16] Xbyak; JIT assembler for x86(IA32), x64(AMD64, x86-64) by C++, https://github.com/herumi/xbyak (accessed 2021-06.16).
[17] x86-64 psABI, https://gitlab.com/x86-psABIs/x86-64-ABI (accessed 2021-06-16).
[18] Xbyak_aarch64 (online), https://github.com/fujitsu/xbyak_aarch64 (accessed 2021-06-07).
[19] K. Kawakami, S. Moriyuki, K. Kurihara and N. Fukumoto: Xbyak_aarch64;JIT Assembler for Next Generation Supercomputer, *Proc. CoolCHIPS23*, (online), (2020).
[20] 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).
[21] 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).
[22] Xbyak_translator_aarch64, https://github.com/fujitsu/xbyak_translator_aarch64 (accessed 2021-06-15).
[23] Intel X86 Encoder Decoder (Intel XED), https://github.com/intelxed/xed (accessed 2021-06-15).
[24] Intel Corporation: Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z, 2019.