Vol.2021-HPC-178 No.7
2021/3/15

ARM SVE向けOpenCLの実装

李 珍泌[1,a)]　佐藤 三久[1]

概要：計算機アーキテクチャのトレンドとして、計算資源の並列化と階層化によってスループット性能を上げるスループット志向アーキテクチャが広く採用されている。近年の汎用プロセッサの性能向上も命令セットのデータ幅の拡大やマルチコア化、高速なメモリ階層の導入が進み、Graphics Processing Unit (GPU) のような従来のスループット志向アーキテクチャに近づいてきている。本研究では GPU 向け並列プログラミングモデルとされてきた OpenCL を富岳スーパーコンピューターの A64FX プロセッサに適用することで、スループット性能を最大限に引き出せるプログラミングモデルの実現を目指す。処理系は OpenCL カーネル関数の ARM Scalable Vector Extension (SVE) 命令によるベクトル化やランタイムによるスレッド並列実行を行う。その結果、命令およびスレッドレベル並列化、NUMA メモリ向け性能最適化などの階層的な並列プログラミングを提案手法のみで記述することが可能になる。STREAM triad を用いた性能評価では 560 GB/sec の性能を達成することを確認した。

Development of OpenCL for ARM SVE

Jinpil Lee[1,a)]　Mitsuhisa Sato[1]

1. はじめに

近年、高い性能を達成するハードウェアを実現するために多様なハードウェア階層で並列化を行うことでスループット性能を向上させる**スループット志向アーキテクチャ**が普及している。Graphics Processing Unit (GPU) のような高度に並列化されたスループット志向アーキテクチャは互いに独立して実行可能なハードウェアスレッドを持つ。スレッドの実行を Single Instruction Multiple Thread (SIMT) 機構で切り替えながら動作させることでメモリ参照のレイテンシを隠ぺいし、高い計算スループットを達成する。

CUDA や OpenCL のような SIMT 型プログラミングモデルではユーザが各スレッドで行われる並列計算を独自のプログラミング言語で記述する。これを SIMT カーネル関数といい、ハードウェアのスケジュール機構によって並列に実行される。SIMT 型プログラミングモデルは逐次コードからの書き換えが必要であるが、計算の並列性やデータ参照の局所性を明示的に記述することでスループット志向アーキテクチャの性能を十分に引き出すことができる。

スループット志向アーキテクチャは GPU など高性能計算向けアクセラレータなどに用いられていたが、近年は汎用プロセッサのアーキテクチャにも採用されている。たとえば富岳スーパーコンピューターに用いられている A64FX はチップ内に 4 つの NUMA ノード (12 コア) で構成され、各コアは 512-bit の Single Instruction Multiple Data (SIMD) 命令 Scalable Vector Extension (SVE) が利用可能であるが、これは 384 個の倍精度演算 ($4 \times 12 \times 8$) を同時に処理できることを表す。このような高い並列性を持つ CPU は従来の CPU と比べてスループット性能を強調しており、GPU 寄りのアーキテクチャとして捉えることもできる。

このようなトレンドが進むと同時に CPU のハードウェアが複雑化し、命令実行やメモリ参照のレイテンシが増加する結果になっている。また、多様な資源を活用する並列プログラミングが必須になり、汎用プロセッサの並列プログラミングモデルとして OpenMP が広く使われている。最新仕様には SIMD 並列化や NUMA メモリを意識し

[1] 理化学研究所 計算科学研究センター
RIKEN Center for Computational Science
[a)] jinpil.lee@riken.jp

たスレッド制御が可能であるが、逐次コードに指示文を挿入するアプローチには性能最適化に限界がある。例えば、キャッシュや NUMA メモリを意識した最適化を行うときはループ文の修正を行うなどのコードの書き換えが必要であるため、OpenMP の逐次コードを保持しながら並列化を行うモデルは成り立たない。また、実行レイテンシやスループットを意識した命令スケジュールは OpenMP 言語仕様の対象外である。

本研究では汎用プロセッサアーキテクチャにおけるスループット志向のトレンドに対応すべく、アーキテクチャの多様なレベルの並列性を活用するプログラミングモデルの実現を目指す。SIMT 型プログラミング言語で記述されたプログラムから汎用プログラム向けの並列化や命令スケジュールを行うコード生成技術の研究を行う。そのために A64FX プロセッサ向け OpenCL 処理系を実装し、その性能評価を行った。

本稿の構成は以下のようである。第 2 章では先行研究や関連研究について述べる。第 3 章では本研究の概要として A64FX プロセッサ向け OpenCL 処理系について説明する。第 4 章ではスレッドの並列実行やベクトルコード生成など、OpenCL 処理系の実現手法について述べる。第 5 章では STREAM ベンチマークを用いた性能評価の結果を示す。第 6 章で本稿をまとめ、今後の課題について述べる。

2. 関連研究

OpenCL[1] は Khronos Group によって仕様の作成が行われているクロスプラットフォームの並列プログラミングモデルである。GPU のようなメニーコアアクセラレータ向けに開発が始められ、現在は CPU や Field-Programmable Gate Array (FPGA) をサポートする処理系も存在する。開発の歴史から GPU で効率よく動作するとされているが、Intel[2] や AMD[3] による x64 アーキテクチャ CPU 向けの処理系も開発されている。Jaejin Lee[4] らは CPU、GPU クラスタ向け OpenCL 処理系の開発を進めており、ヘテロジニアスクラスタ環境で仮想共有メモリによる並列プログラミングを実現している。また同じ著者によって ARM Neon 命令をサポートする OpenCL 処理系の開発も行われている [5]。本研究は A64FX の OpenCL 処理系を開発するものであり、SVE 命令をサポートする初めての処理系となる。

3. ARM アーキテクチャ向けスループット志向プログラミングモデルの導入

OpenCL は多数の計算コアを持つデバイスと、それを操作するホストの存在を想定したプログラミングモデルである。図 1 に OpenCL のコード例を示す。OpenCL ではターゲットデバイス上で並列に実行されるスレッドの動作をカーネル関数に記述する。カーネル関数はハードウェア

```
1  __kernel void
2  func(const int n,
3       const __global float* A,
4       __global float* B) {
5    int lid = get_local_id(0);
6    int gid = T * get_group_id(0) + lid;
7  // local memoryによるメモリ参照の局所化
8    __local double buffer[T];
9    buffer[lid] = A[gid];
10 // work group内同期
11   barrier(CLK_LOCAL_MEM_FENCE);
12   double result = buffer[ ...;
13   B[gid] = result;
14 }
```

図 1 OpenCL カーネル関数のコード例

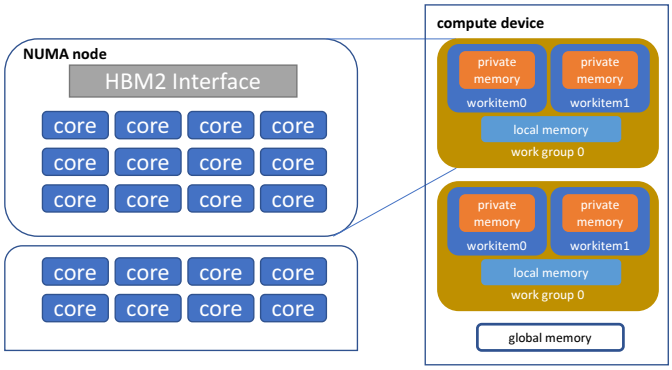


図 2 OpenCL の実行モデルと A64FX への対応

のスケジュール機構によって計算コアに割り当てられて実行されるが、複数の計算コアが同じカーネル関数を実行することで SIMT 実行モデルを実現する。各スレッドが異なるメモリ領域を参照するようにするため、スレッド ID をランタイム API (get_group_id(), get_local_id()) で取得する。高速なローカルメモリを持つデバイスでメモリ局所化を行うために__local キーワードによるローカルメモリの宣言とスレッド間の同期 (barrier()) を記述する。

カーネル関数の実行やデバイスのメモリ管理はホスト側の API によって行われる。ホスト側 API は C 言語のライブラリとして提供される。

本研究では図 1 のようなカーネル関数で記述されたコードを A64FX 上で並列実行するための OpenCL の処理系を実装する。図 2 に OpenCL の実行モデルと A64FX プロセッサへの対応を示す。OpenCL の work item は並列実行の基本単位であり、カーネル関数を実行する。work group は複数の work item から構成され、local memory と呼ばれる高速なメモリを共有する。GPU アーキテクチャでは計算コアのハードウェアスレッドを work item に対応させて並列実行を行う。work group に対応する高速なメモリの実装とハードウェアスレッドのグループ化が行われており、グループ化されたスレッドブロックの数を増やすことによってスケーラブルな性能向上を実現する。

OpenCL の実行モデルを A64FX の CPU アーキテクチャ

表 1　A64FX プロセッサのスペック

Item	Name/Value
ISA	Arm v8.2-A + SVE
SIMD Width	512-bit
NUMA Node (CMG)	12 個の計算コア + 1 個のアシスタントコア 4 つの CMG がリングバスで接続
Memory	HBM2 32GiB 1,024 GB/s

上で実現するために NUMA アーキテクチャと SVE ベクトル命令を利用する。表 1 に A64FX プロセッサのスペックを示す。A64FX は Arm v8.2-A 命令セットを採用した Arm プロセッサである。高性能計算向けベクトル拡張命令である SVE を実装しており、SIMD 幅は 512-bit である。

A64FX の NUMA アーキテクチャは Core Memory Group (CMG) を基本単位とし、チップ内に 4 つの CMG が実装されている。1 つの CMG は 12 個の計算コアと高速にアクセスできる HBM2 メモリを持つ。CMG 間はリングバスで接続されているため、他の CMG メモリにアクセスすることも可能であるが、リモートメモリアクセスによる遅延が発生する。

図 2 のように A64FX の CMG を OpenCL の work group に対応させる。work group の中の work item を CMG の中の計算コアに割り当てることで CMG 内並列化を実現する。work item の並列実行は OS スレッドによって実装するが、カーネル関数がベクトル化できる場合は SVE 命令を用いて一部の work item をまとめて処理する。

global memory と local memory の実装は HBM2 メモリを利用する。global memory のメモリ割り当ては CMG ノードを意識せずに行うのに対して、local memory の割り当ては NUMA API を用いて CMG に接続されたローカルメモリを利用する。これによって work group の中の work item が local memory を参照することでリモートメモリアクセスを発生させないメモリ最適化が可能である。

work item の高速な専用バッファーである private memory には A64FX のセクターキャッシュを用いることが考えられるが、本研究では実装していない。

従来、CPU の並列プログラミングには OpenMP が広く使われている。OpenMP はコア間のスレッド並列化を記述するための構文は提供しており、最新の仕様では SIMD 命令生成のための指示文が導入されている。しかし、逐次コードを保持しながらハードウェアの多様な並列性を引き出すことには限界があり、最適化のためにはコードの修正が必要な場合がある。SIMD 化に関しては処理系のベクトル並列化を意識したコード変換が必要で、NUMA 最適化のためにはデータの初期化を NUMA ライブラリを用いて行わなければならない。OpenCL による並列プログラミングは逐次コードから並列カーネルコードへの書き換えが必要であるが、コンパイラやハードウェアが必要とする並列性をプログラムの中で明示的に表現できるというメリットがある。GPU の普及が進み、CUDA などによる SIMT プログラミングモデルで得られた性能最適化のノウハウを本研究で実現する OpenCL の処理系を用いることで CPU でも適用できると期待する。

図 3　コード変換の流れ

4. SVE 命令に対応した OpenCL 処理系の実装

本章では第 3 章で示した実行モデルを実現する OpenCL 処理系の実装方法を示す。処理系の実装にはオープンソースコンパイラである LLVM と軽量スレッドライブラリ Argobots[6] を用いる。

4.1 コード変換の流れ

ソースコードは LLVM をベースに実装された処理系によって A64FX のバイナリーに変換される。図 3 に処理系によるコード変換の流れを示す。OpenCL ソースコードとは C 言語で記述されたホスト側コードと OpenCL カーネル関数コードで構成される。カーネル関数コードは別途コンパイラされたバイナリー形式とホスト側コードに文字列として組み込まれたテキスト形式で与えることができる。本研究の実装ではテキスト形式のみをサポートする。OpenCL ヘッダーファイルは Khronos グループが提供しているものを利用した。処理系のランタイムライブラリは OpenCL 1.2 のホスト API を部分的に実装している。

OpenCL のホスト側コードは標準仕様の C 言語で書かれたものであるため、C 言語をサポートするコンパイラであればコンパイル可能である。しかし、ソース内で文字列として与えられたカーネル関数をバイナリーに変換して実行するために LLVM の Clang と JIT コンパイラを用いた。OpenCL の clCreateProgramWithSource() は文字列のカーネル関数からバイナリーを生成するホスト側 API である。処理系のランタイムは clCreateProgramWithSource() の中で文字列のカーネル関数の内容をテキストのソースコードに変換し、LLVM の C/C++言語フロントエンドである Clang に与える。Clang は OpenCL のキーワードをサポートするため、カーネル関数コードを中間言語である LLVM IR に変換することができる。そのため特別な実装を行わず、Clang の機能をそのまま用いることでカーネル関数から LLVM IR への変換を行う。

変換された LLVM IR は独自に実装した LLVM のモジュールを利用して並列コードに変換される。具体的には

```
1  void
2  OpenCL_run(size_t groups[3],
3             size_t items[3],
4             void *kernel, void *args) {
5  // work group loop
6    for (size_t g2=0; g2<groups[2]; g2++)
7      for (size_t g1=0; g1<groups[1]; g1++)
8        for (size_t g0=0; g0<groups[0]; g0++)
9  // work item loop
10   for (size_t i2=0; i2<items[2]; i2++)
11     for (size_t i1=0; i1<items[1]; i1++)
12       for (size_t i0=0; i0<items[0]; i0++)
13         kernel(args);
14 }
```

図 4 OclExec

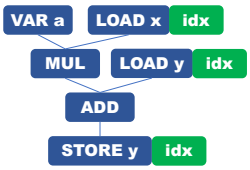
カーネル関数コードをターゲットデバイスで実行可能な関数に変換し、それを呼び出すループ文を生成する。OpenCL で並列性を記述するためにはホスト側 API によって work group と work item の形状を宣言するが、work group と work item の大きさがループ文のイテレーションステップ数を決定する[*1]。図 4 に 3 次元の work group と work item が与えられたカーネルコードを逐次実行する C 言語コードを示す。このように与えられた形状通りにループ文を生成し、イテレーション毎に work group ID と work item ID を変化させながらカーネル関数を実行することで OpenCL の実行モデルを実現する。

実際の処理系も図 4 のコードのように多重ループ文を生成して実行を行うが、一部のループ文を並列化する。現在の実装は work group loop の各イテレーションを CMG ノードに round-robin 方式で割り当てる。work group loop の内部の work item loop はすべてのイテレーションが 1 つの CMG で実行される。現在の実装では最外ループ文はスレッド並列化を行い、最内ループ文は可能であれば SVE ベクトル化を行う[*2]。

4.2 SVE コード生成

カーネル関数コードは LLVM によって Arm アーキテクチャ向けのバイナリーに変換されるが、ベクトル化可能である場合は SVE 命令を用いたベクトル化関数が生成される [7]。図 5 に daxpy 関数の OpenCL コードと演算命令の依存グラフを示す。

処理系は戻り値が void の OpenCL カーネル関数で副作用が発生するのはメモリの store 命令と仮定し、store 命令のリストを作成する。その後、リストの中の命令を取り出し、その命令のオペランドを再帰的に探索することでメモリ演算に利用される命令を探す。見つかったすべての命令に対応する SVE 命令があればベクトル化関数の生成に




```
__kernel
void daxpy(__global double *y,
           __global double *x, double a) {
  int idx = get_global_id(0);
  y[idx] = a * x[idx] + y[idx];
}
```

図 5 Code

```
1  void
2  daxpy(double *y,
3        double *x, double a) {
4    int idx = get_global_id(0);
5    int N = get_global_size(0);
6    svbool_t p = svwhilelt_b64(idx, N);
7    svfloat64_t vx = svld1(p, &x[idx]);
8    svfloat64_t vy = svld1(p, &y[idx]);
9    svfloat64_t va = svdup_f64_x(p, a);
10   svfloat64_t mul = svmul_x(p, va, vx);
11   svfloat64_t add = svadd_x(p, mul, vy);
12   svst1(p, &y[idx], add);
13 }
```

図 6 SVEDaxpy

進む。

現在の LLVM IR は SVE の可変ベクトル長を表現するためのデータ型を持っているため、スカラ命令を LLVM IR の組み込み命令に変換することは可能である。しかし、AArch64 のバックエンドで可変データ型の組み込み命令を用いた SVE 命令生成が実装されておらず、predicate マスクを組み込み命令で表現することもできないため、ベクトル化関数を LLVM IR の組み込み命令で表現することはできない。そのため、処理系はスカラ命令を SVE の intrinsic 関数に変換することでベクトル化関数を生成する[*3]。図 6 に SVE intrinsic 関数を用いて記述した daxpy コードを示す[*4]。

4.3 ループ文の生成

実行可能なカーネル関数が得られたら、それを並列実行するループ文を生成する。work group loop はランタイムライブラリによって処理される。図 7 に 1 次元の work group を処理するランタイムコードを示す。処理系は work group の各イテレーション毎にスレッドブロックを生成し、CMG ノードに割り当てる。現在の実装では複雑なスケジュールは行わず、round-robin で CMG のノード番号を決定する。

daxp y_wrapper_1() は work item loop を並列実行するカーネルの wrapper 関数である。処理系はカーネル関数毎にこのような wrapper 関数を生成する。図 8 にスカラ版 daxpy の wrapper 関数を示す。処理系は wrapper 関数

*1 両方とも最大 3 次元の形状で与えられるが、次元数は同じでないといけない。
*2 work group と work item が 1 次元の場合は 1 つのループ文に両方の並列化を適用する。
*3 バックエンドが intrinsic 関数からのバイナリー生成には対応しているため
*4 説明を簡単にするために簡略化された C 言語のコードを用いているが、処理系のコード変換は LLVM IR で行っている。

```
void
OpenCL_run(size_t *groups,
           size_t *items,
           void *kernel, void *args) {
  for (size_t g = 0; g < groups[0]; g++)
    thread_block_create(daxpy_wrapper_1,
                        args, items, g);
}
```

図 7 ScalarWrapper

```
void
daxpy_wrapper_1(double *y, double *x,
                double a,
                size_t st, size_t ed) {
  for (int i = st; i < ed; i++)
    daxpy(y, x, a);
}
```

図 8 スカラ版 daxpy の wrapper 関数

```
void
daxpy_wrapper_1_SVE(double *y, double *x,
                    double a,
                    size_t st, ed) {
  for (int i = st; i < ed; i += svcntd()) {
    svbool_t p = svwhilelt_b64(i, D[0]);
    svfloat64_t vx = svld1(p, &x[i]);
    ...
  }
}
```

図 9 SVE 版 daxpy の wrapper 関数

の中で work item ループ文を生成し、各イテレーション毎にカーネル関数呼び出しを行う。ループ変数の st と ed はスレッドの並列実行のために用いられるもので、create_thread_block() 関数によって実行範囲が与えられる。現在の実装では work item のサイズをブロック分割して CMG 内のコアに割り当てる。

ベクトル化された daxpy にも wrapper 関数を生成する。図 9 に SVE 命令でベクトル化された daxpy の wrapper 関数コードを示す。work item ループ文のイテレーション stride は SVE ベクトル命令の 64-bit 要素数分進むため、svcntd() が用いられる。処理系は関数呼び出しのオーバーヘッドを減らすため、カーネル関数を work item ループ文の中にインライン展開する。そのとき、0 次元の global work item ID はループ変数 i に書き換えられる。スカラ版の wrapper 関数でも同様の処理が行われる。

5. 性能評価

本章では STREM ベンチマークを用いて実装した処理系の性能評価を行う。STREAM はシステムのメモリバンド幅を測定するために用いられるベンチマークソフトウェアであり、複数の測定関数で構成される。本研究では triad カーネル関数による評価を行った。評価環境は表 1 で示し

```
__kernel void
triad_opencl(__global int *a, int scalar,
             __global int *b,
             __global int *c) {
  size_t idx = get_global_id(0);
  a[idx] = b[idx] + scalar * c[idx];
}
```

図 10 OclStream

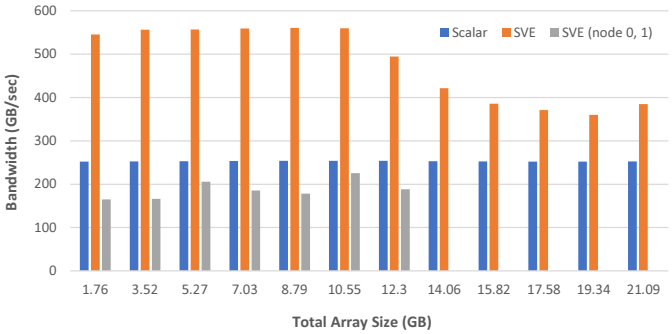


図 11 STREAM ベンチマークの評価結果

たものと同じである。

図 10 に OpenCL で記述された STREAM triad カーネル関数のコードを示す。計算は 2 つの配列のロードと 1 つの配列のストア、足し算とかけ算で構成される。足し算とかけ算のコストは無視し、3 つのメモリアクセスを配列の要素数分行うと考えることができる[*5]。triad_opencl() は 1 次元の OpenCL カーネル関数であるため、work group と work item も 1 次元を指定する。本研究では work group size が NUMA ノードの数と同じになるようにパラメータの設定を行った。

評価結果を図 11 に示す。配列を 157,286,400 個 (総データサイズ 1.76GB) 単位で増加させながらメモリバンド幅を計測したものである。スカラはベクトル化されていないカーネル関数を用いた場合、SVE はベクトル化カーネル関数を用いた場合の性能である。SVE (node 0, 1) は numactl 命令を使って利用するメモリを NUMA ノードを 0, 1 に限定したものである。

スカラカーネル関数を用いた評価ではどのデータサイズでも 250 GB/sec 程度の性能を達成した。表 1 で示したようにシステムのトータルメモリバンド幅は 1,024 GB/sec であるため、ピーク性能の 1/4 の結果になっているが、原因としてはスカラ命令では十分なデータをシステムに供給することができないことが考えられる。

カーネル関数をベクトル化した SVE の評価では最大で 560 GB/sec の性能を達成した。スカラカーネル関数と比べて 2 倍以上の性能を達成しているが、ピーク性能の半分程度である。富士通 [8] による評価結果ではピーク性能

[*5] double データ型は OpenCL のオプション機能であり、本研究で実装した処理系ではサポートしていないため、32-bit 整数型の STREAM triad 関数を作成して評価を行った。

の 80%以上を達成しているため、現在原因の把握と改善を行っている。

NUMA ノードの利用状況を調べるために、NUMA ライブラリの API を用いてアクセスする配列のデータが NUMA ノードに均等に割り当てられていることや計算コアとメモリの NUMA ノードが同じであることを確認した。比較のために NUMA ノードの 0 と 1 のメモリだけを使った評価結果 (SVE (node 0, 1)) を示す[*6]。計算コアは NUMA ノード 0 から 3 までをすべて利用するため、リモートメモリアクセスが発生する。計算コアとメモリの NUMA ノードを一致させた SVE と比べると性能が低下していることが分かる。

先行研究では演算主体のアプリケーションでソフトウェアパイプライニングによる命令の入れ替えが有効であるとされる [9]。シミュレーターを用いた STREAM triad の SVE 性能評価ではカーネルがメモリ性能律速であること [10] が示されているため、マイクロアーキテクチャやコンパイラの命令スケジュールに影響される可能性は低い。今後の課題として SVE 向け命令スケジュールの実装とプロファイラによる有効性の確認を行う。

6. おわりに

本研究では A64FX プロセッサ向け OpenCL 処理系の開発を行い、STREAM ベンチマークによる性能評価を行った。これによって OpenCL の実行モデルを汎用 CPU で実現し、NUMA ノードや SVE ベクトル命令による階層的な並列アーキテクチャを活用できることを確認した。今後の課題としては以下のようなことが考えられる。

- マイクロアーキテクチャを考慮した命令スケジュールの実装によって実行効率を改善する。
- ローカルメモリを宣言する__local キーワードを実装し、メモリ局所化による性能改善を確認する。
- セクターキャッシュによる work item の private memory の実現可能性について検討を行う。

参考文献

[1] OpenCL - Khronos Group: https://www.khronos.org/opencl/.
[2] Intel OpenCL SDK: https://software.intel.com/content/www/us/en/develop/tools/opencl-sdk.html.
[3] AMD ROCm: https://www.amd.com/ja/graphics/servers-solutions-rocm-hpc.
[4] Kim, J., Seo, S., Lee, J., Nah, J., Jo, G. and Lee, J.: SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters, *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, New York, NY, USA, Association for Computing Machinery, p. 341–352 (online), DOI: 10.1145/2304576.2304623 (2012).
[5] Jo, G., Jeon, W., Jung, W., Taft, G. and Lee, J.: OpenCL framework for ARM processors with NEON support, (online), DOI: 10.1145/2568058.2568064 (2014).
[6] Argobots - A Lightweight Low-level Threading Framework: https://www.argobots.org/.
[7] Karrenberg, R. and Hack, S.: Whole-function vectorization, *International Symposium on Code Generation and Optimization (CGO 2011)*, pp. 141–150 (online), DOI: 10.1109/CGO.2011.5764682 (2011).
[8] Yoshida, T.: Fujitsu high performance CPU for the Post-K Computer, Hot Chips 30: 30th Symposium on High Performance Chips (2018).
[9] 修一千葉，正樹青木，俊　鎌塚，雅人松井，尚　八代：Evaluation of Software Pipelining for Architecture with Small Number of Registers，情報処理学会論文誌，Vol. 61, No. 2, pp. 429–439 (online), available from ⟨https://ci.nii.ac.jp/naid/170000181703/en/⟩ (2020).
[10] 哲哉小田嶋，祐悦児玉，元彦松田，珍泌　李，美和子辻，三久佐藤：ベクトル長を可変とする SVE アーキテクチャの評価，技術報告 11，理化学研究所計算科学研究機構, 理化学研究所計算科学研究機構, 理化学研究所計算科学研究機構, 理化学研究所計算科学研究機構, 理化学研究所計算科学研究機構, 理化学研究所計算科学研究機構 (2017).

[*6] メモリ容量の関係で 10 GB までの評価になっている。