A64FXプログラミング講習会
入門編

2023.10.31

富士通株式会社

目次

13:00 - 13:30	スパコンの使い方など
13:30 - 14:20	A64FXアーキテクチャと並列プログラミング（座学）
14:30 - 15:40	OpenMPを活用した並列プログラミング演習 3重ループを含むプログラムで、OpenMPの指示文挿入によるスレッド並列や SIMD演算を活用するコードを取り扱います。 また、Intrinsic（ACLE）を使ってSIMD化する例も取り上げます。
15:50 - 17:00	プロファイラを活用したチューニング演習

FUJITSU

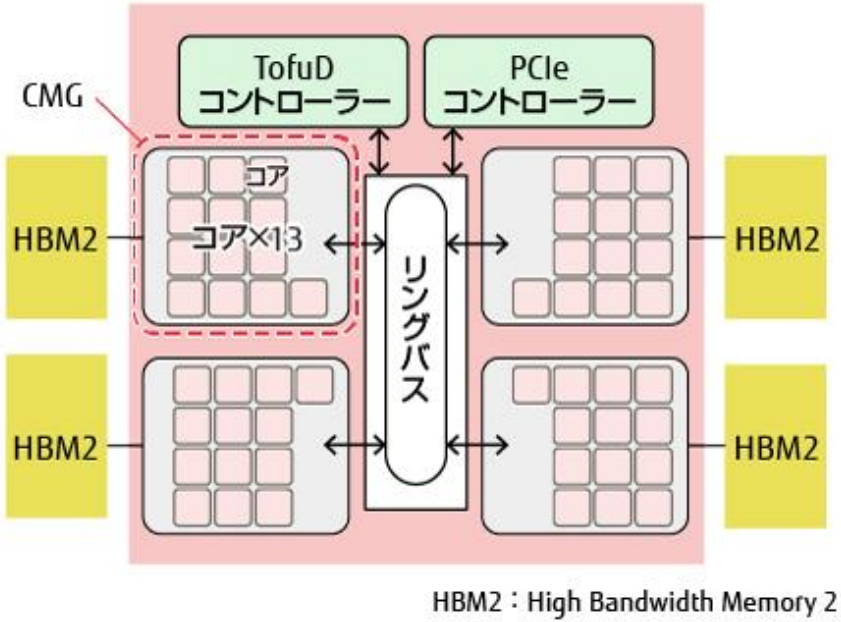
A64FXアーキテクチャと並列プログラミング

A64FXアーキテクチャ

A64FXの概要

- 富士通が2019年にリリースしたメニーコアCPU
- 48 (+アシスタント)コア、コアあたり512bits SIMD演算器 x2
- 高バンド幅のメモリを搭載
- 2023年6月第2位の「富岳」に使われるCPU
- 2019年にはGPUのシステムを抑えてGreen500で1位を取るなど、非常に電力効率がよい

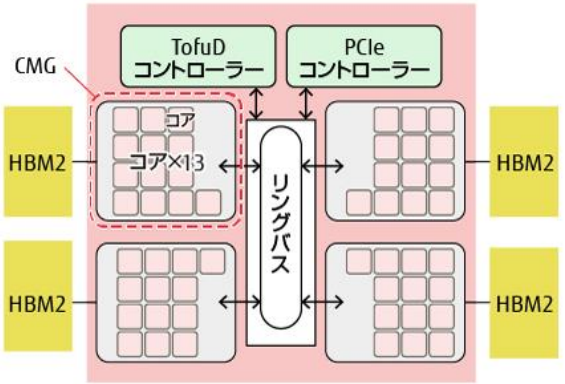
A64FX CPU



https://www.fujitsu.com/jp/about/resources/publications/technicalreview/2020-03/article03.html

A64FXのハードウェアの特徴

- 汎用CPUとしては非常に高い電力性能
  - 高い理論演算性能
  - 高いメモリバンド幅
- 一方で使いづらいところがある
  - メモリ容量が小さい（32GiB）
  - 命令レイテンシが長い（FMAは9cycles）
  - ラストレベルキャッシュメモリサイズが小さい
  - すべてのコアとSIMD演算器を使わないと高性能は達成できない



HBM2：High Bandwidth Memory 2

コア数	48 + アシスタントコア2~4
周波数	2.0GHz (or 2.2GHz)
理論演算性能	3.4 TFLOPS
メモリ容量	32 GiB
メモリバンド幅	1024GB/s
L1 Cache	64 KiB/core (Inst/Data)
L2 Cache	8 MiB / CMG

A64FXのソフトウェアから見た特徴

FUJITSU

- 得意なプログラムをチューニングすれば非常に高性能
  - 例1：最適化されたライブラリが主体のプログラム
  - 例2：メモリバンド幅ネックかつ並列性が十分にあるプログラム
  - 例3：演算ネックで、並列性があり、レイテンシを隠蔽できるプログラム
- チューニングなしでは性能が出にくい
  - 既存CPU向けに記述されたコードでは性能が活かせないことが多い（コアとSIMDの並列性の抽出が必須。演算ネックのプログラムだとレイテンシの隠蔽も必要）
  - アルゴリズムとデータ構造の見直しも必要かもしれない
- 苦手なプログラムは諦めたほうがいい（こともある）
  - 演算間の依存関係が強いループボディの大きいプログラムは性能が出しづらい

FUJITSU

A64FXアーキテクチャと並列プログラミング

A64FXの性能の引き出し方

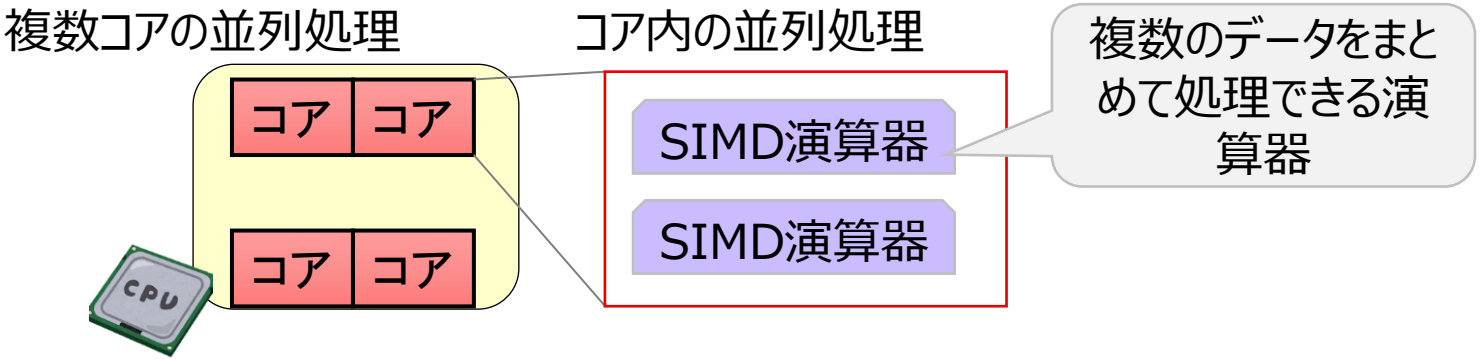
A64FXに最適化された計算ライブラリの利用

FUJITSU

- A64FXのHW性能を十分引き出した計算ライブラリが公開されている
- まずは計算ライブラリの活用を検討する
- 有名な計算ライブラリ
  - SSL2：富士通が最適化したHPC向けライブラリ（行列演算系などがある）。マルチコアで使用するときは、富士通のOpenMPライブラリを使う必要がある
  - OneDNN: Intel発のDeep learning向けライブラリ。主に、富士通とサイボウズ・ラボの光成さんでA64FX向け最適化を行い、本家OSSにアップストリームした
  - その他
    - BLAS, LAPACK, FFTなど：A64FX向けに最適化されていないものでも実行効率30-50%程度はあると思うので活用したほうが良いと思います

並列処理は必須

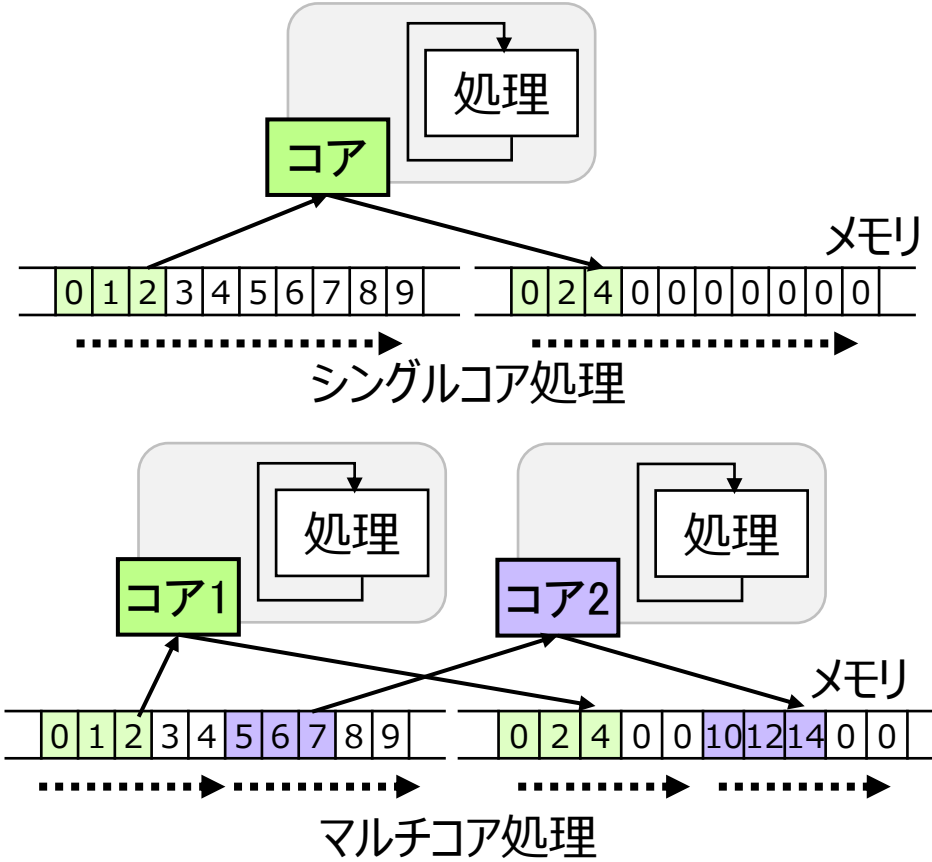
- 現代のプロセッサは、並列性を活かさなければ性能がでない



- 複数コアの並列処理とSIMD演算器の並列処理は、コンパイラによる最適化があまり期待できないので、プログラマが記述しなければならない
  - プログラムから、依存関係のない（＝並列処理可能）処理を抽出する必要がある

データ並列とスレッド並列

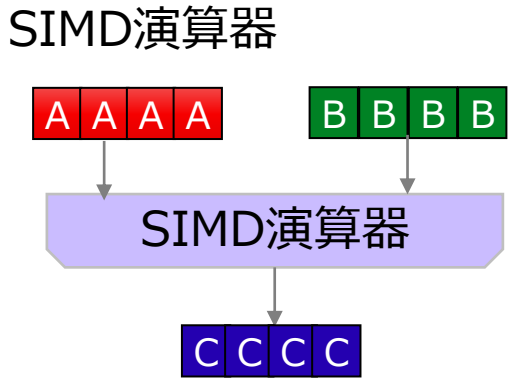
- データ並列
  - 最もよく使われる並列性
  - 異なるデータに対して同じ処理を行う
  - コア間の並列処理でもSIMD演算器の並列処理でも利用可能
- スレッド並列
  - コア間の並列処理に使われる並列性（SIMD演算器では利用できない）
  - コア間で同じコードを処理させる
- ~~タスク並列~~
  - ~~ややこしいのでなるべく考えない~~



SIMD演算器での並列処理の記述

FUJITSU

- メモリ上で連続したデータをまとめて演算する
- 最も内側のループで、連続しているデータに対して処理するように記述されていればコンパイラがSIMD演算器を活用するコードを生成してくれる



SIMD化されるコード

```
for (i = 0; i<N; i++)
 c[i] = a[i] + b[i];
```

多分SIMD化されないコード

```
for (i = 0; i<N; i++)
 c[i] = a[2*i] + b[2*i];
```

並列性の抽出

FUJITSU

- すべてのコアを使う（A64FXはコア数が多いので重要）
  - ループにOpenMPディレクティブを挿入
  - ループの回転数を確認して足りない場合は工夫する
    - OpenMPのcollapse節を使って、多重ループに対して並列化を行う
    - 処理方法を見直す
    - プロセス並列やタスク並列の導入を検討する（今回の演習の対象外）
- SIMD演算器を活用する（A64FXではメモリバンド幅ネックのプログラムでもSIMD演算器の活用は必須）
  - 最も内側のループで以下を満たせるようにする
    - 連続データに対して処理できるようにする
    - 十分な回転数があり、なるべく512bitsの倍数でデータが処理できるようにする
    - if文はなるべくループの外に追い出す
    - 関数呼び出しは避ける（コンパイラによる最適化が難しくなる）
  - もしくは、アセンブリコードレベルで記述する

FUJITSU

A64FXアーキテクチャと並列プログラミング

すべてのコアを使う

複数コアを使った並列処理の記述方法

FUJITSU

- OpenMPで記述
  - サーバ内（共有メモリ）での並列処理を記述するときに使う
  - 逐次プログラムにディレクティブと呼ばれる指示文を入れることで並列化する
  - 共有メモリが前提のため、明示的なデータのやり取りは記述しなくて良い
  - 記述が簡単で、コードの可読性があまり落ちないのでよく使われる

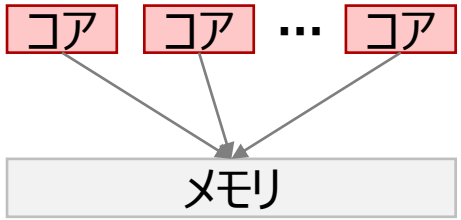
本講習の対象

- MPIで記述
  - 主に複数サーバ（分散メモリ）での並列処理を記述するときに使う
  - 共有メモリがない前提で、明示的にデータのやりとりを記述する
  - 複数サーバでの並列処理を使うときは、ほぼ必須なのでOpenMPを使わずにMPIだけ使うコードもある
- 他の言語で記述
  - Pthread, OpenCL, SYCLなどがある
  - 興味のある方は調べてみると良いと思います

OpenMPとは

- 共有メモリ型の計算機において並列処理を可能とするAPI
- 逐次プログラムにディレクティブという指示文を挿入することで並列化可能
- OpenMPディレクティブは、コンパイルオプションを指定しない限り、無視されるため同じコードで逐次プログラムと並列プログラムを生成できる

共有メモリ型計算機
＝同じデータにアクセス可能

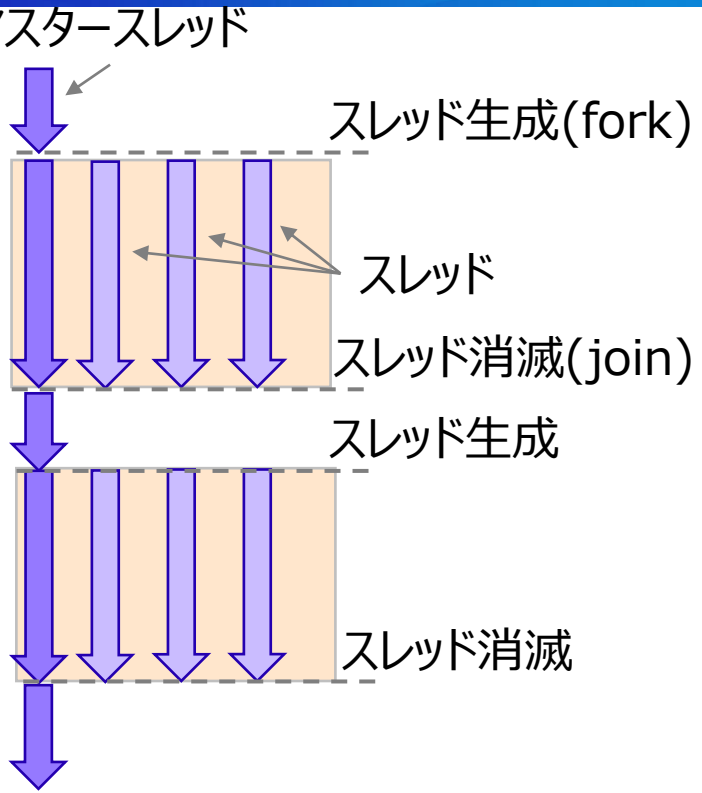


OpenMPプログラム例

```
#pragma omp parallel for
for(i=0;i<M;i++){
  for(j=0;j<N;j++){
   for(k=0;k<K;k++){
     C[i][j]+=a[i][k]*b[k][j];
   } } }
```

OpenMPでよく記述される並列モデル

```
// 逐次処理
…
#pragma omp parallel
{
   // 並列処理A
   …
}
// 逐次処理
…
#pragma omp parallel
{
  // 並列処理B
  …
}
// 逐次処理
…
```



OpenMPでよく使われる並列化

FUJITSU

- ループを対象とした並列処理

逐次コード

```
for(i=0;i<N;i++){
C[i]=a[i]+b[i];
}
```

並列化コード

```
#pragma omp parallel for
for(i=0;i<N;i++){
C[i]=a[i]+b[i];
}
```

- タスク並列なども記述できるが、あまり使わない

コアの並列処理効率の確認

FUJITSU

1. ループの回転数が十分あれば、複数コアの並列処理の効率は高いことが多いため、まずはループの回転数が十分かを確認する
   - printf文をソースコード中に挿入するなどして確認
2. キャッシュメモリなどの共有資源での競合で性能が向上しないこともあるため、並列化させるコア数を変更して性能を確認
   - デフォルトの設定のOpenMPのスレッド生成数＝コア数
   - スレッド生成数は、環境変数 OMP_NUM_THREADSや omp_set_num_threads(int num)関数で変更可能

コアの並列処理効率の向上

FUJITSU

- 並列化対象のループの回転数を増やす
  - 多重ループであれば複数のループを対象にする（collapse節を挿入）
  - 対象となるループを変更する
  - データ構造とアルゴリズムレベルで見直す
- 細かい最適化をする
  - 立ち上げるスレッド数を減らす
  - omp parallel節の範囲をなるべく広げて、スレッド生成コストを下げる
  - 自動で挿入されるバリア同期を削る（nowait節を挿入）

```
#pragma omp parallel for collapse (2)
for(i=0;i<M;i++){
  for(j=0;j<N;j++){
   for(k=0;k<K;k++){
     C[i][j]+=a[i][k]*b[k][j];
   } } }
```

FUJITSU

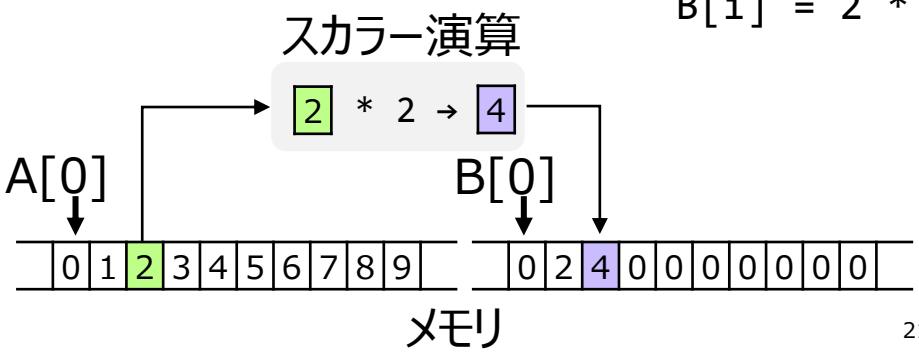
A64FXアーキテクチャと並列プログラミング

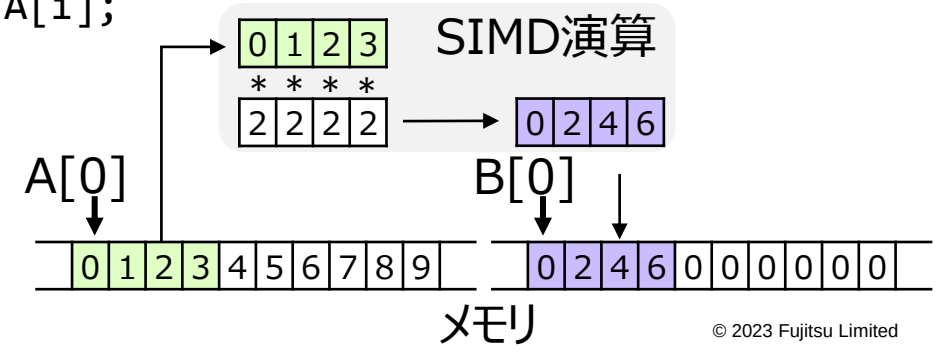
SIMD

Single Instruction Multiple Data (SIMD)

- 1命令を複数のデータに同時に適用する機構
  - 複数データを一度に処理することで演算性能を向上
- 近年のCPUの多くはSIMD機構を搭載
  - x86: SSE, AVX2, AVX512
  - ARM: Neon, **Scalable Vector Extensions(SVE)**


```
for(i=0;i<N;i++)
  B[i] = 2 * A[i];
```

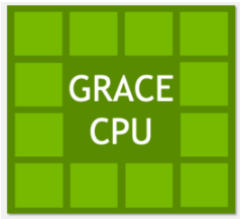




Scalable Vector Extensions (SVE)

- ARMアーキテクチャの拡張SIMD命令セット
- Vector Length Agnostic (VLA)
  - SVE命令はベクトル長として128-bit ~ 2048-bitまで128-bitの倍数をサポート
    - ハードウェアは任意のベクトル長を実装することが可能
  - ベクトル長に依存しないプログラムを記述可能

    → 汎用性がある高速なプログラムの作成を可能に



Grace CPU
128-bit SVE

Graviton3
256-bit SVE

A64FX
512-bit SVE

SVEの特徴

- SVEレジスタの長さ(要素数)は実行時に取得可能
  - cnt[b,h,w,d]命令を使用
    - b: 8-bit, h: 16-bit, w:32-bit, d:64-bit要素換算

```
cntd x1;
```

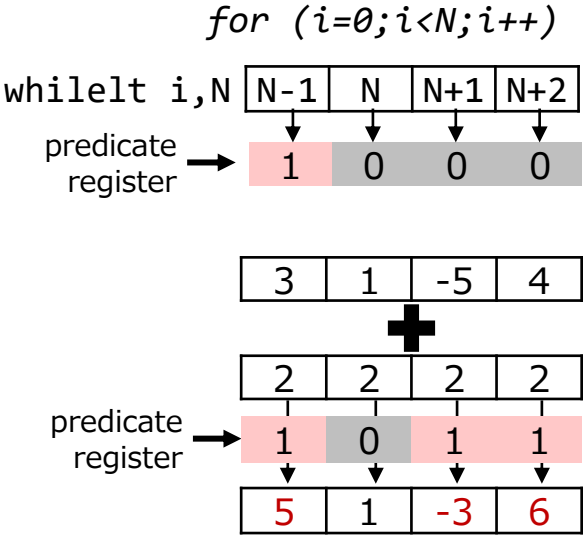
→ 8 (=512/64)

- プレディケーション
  - SVEレジスタの要素ごとに命令の適用の有無を変えられる
  - 多くのSVE命令がプレディケートレジスタ付きの命令
  - 利点

    → 任意長のループのSIMD化が容易 (端数処理)

    → if文を含む処理のSIMD化が可能

C/C++コードでの使用方法は後述



A64FXのSVE演算部

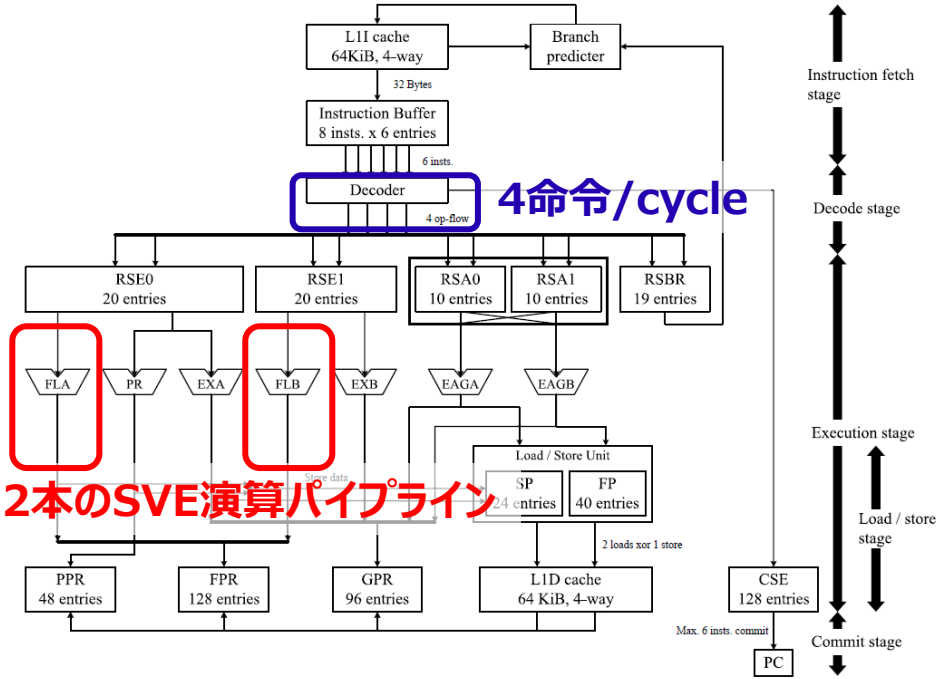
- A64FXの各コアに、2本のSVE演算パイプライン
- SVE幅: 512-bit
  - FP64: 8要素分
  - FP32: 16要素分
  - FP16: 32要素分

1度にまとめて
処理可能

- 命令発行部
  - 最大4命令発行/cycle

アプリ性能を高めるためには

SVE命令x2/cycleを目指して
SVE命令の割合が高いプログラムを作る



A64FXのコア内のプロセッシングステージ

コンパイラ最適化: SVE命令が使われるようにする

FUJITSU

- コンパイラ最適化の確認: 最適化メッセージ出力オプションを追加
  - fcc/FCCのメッセージ出力オプション: -Koptmsg=2
- 最適化されない → コンパイラ最適化が効きやすいようにコードを修正
  - 最適化できない原因は多種多様 (例: ループ間での依存関係がある)

```
43 void ReLU(dtype *inout, int num_elements) {
44
45   for (int i = 0; i < num_elements; i++)
46     inout[i] = (inout[i] >= 0) ? inout[i] : 0.0;
47 }
```

```
FCC main.cpp ¥
    -Kfast -Koptmsg=2
```

最適化メッセージ

```
jwd6001s-i  "main.cpp", line 45: SIMD conversion is applied to this
loop with the loop variable 'i'.
jwd8204o-i  "main.cpp", line 45: This loop is software pipelined.
jwd8205o-i  "main.cpp", line 45: The software-pipelined loop is chosen
at run time when the iteration count is greater than or equal to 64.
```

指示文: SVE命令が使われるようにする

FUJITSU

- 最適化でSIMD化されないコードでも、指示文でSIMD化できる可能性あり
- SIMD化を適用するforループにOpenMP指示文を挿入
  - 指示文: #pragma omp simd

```
36  for (int m = 0; m < ic; m++)
37    for (int l = 0; l < oc; l++) {
38      Out(i,j,k,l) += W(l,m) * In(i,j,k,m);
39    }
```

```
FCC main.cpp -Kfast -Koptmsg=2
```

```
jwd6208s-i  "main.cpp", line 38: SIMD conversion is not applied to this loop because the uncertain order of the definition and reference to variable 'out' may cause different results from serial execution.
```

```
36  for (int m = 0; m < ic; m++)
37 #pragma omp simd
38    for (int l = 0; l < oc; l++) {
39      Out(i,j,k,l) += W(l,m) * In(i,j,k,m);
40    }
```

```
FCC main.cpp -Kfast,openmp -Koptmsg=2
```

```
jwd6501s-i  "main.cpp", line 38: This loop with variable 'l' is vectorized by OpenMP SIMD.
```

SVE命令の使用状況確認 1/3

- コンパイラの最適化メッセージ
  - Fujitsuコンパイラのオプション”-Nlst=t”と”-Koptmsg=2”を追加
  - “-Nlst=t”: ソースコードに最適化情報を付与した*.lstファイルを生成
  - “-Koptmsg”: SIMD化等の最適化機能が動作したことをメッセージ出力

```
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    SIMD(VL: 16)
                    <<<    PREFETCH(HARD) Expected by compiler :
                    <<<      (unknown)
                    <<< Loop-information  End >>>
38          2v          for (int l = 0; l < oc; l++) {
39          2v            ConvOut(i, j, k, l) += ConvWei(l, m) * ConvIn(i, j, k, m);
40    p     2v          }
```

直後のfor文に適用した最適化情報
- 16要素のSIMD化
- ハードウェアにプリフェッチ利用推奨

- アンローリング:2
- SIMD化

*.lstファイルの例

SVE命令の使用状況確認 2/3

FUJITSU

- **fippプロファイラによる確認**
  1. プロファイルデータの生成: `fipp -C -d <dir name> -Icall ./a.out`
  2. プロファイルデータの解析: `fipppx -A -Icpupa -d <dir name>`
  - 最適なプロファイルオプションはドキュメント参照

```
 Execution                    Floating-point  Mem throughput  Mem throughput
         time(s)       GFLOPS   peak ratio(%)         (GB/s)   peak ratio(%)
-------------------------------------------------------------------------------
         26.5010       1.9377          0.2294         0.1940          0.0189   Application
-------------------------------------------------------------------------------
         26.5010       1.9377          0.2294         0.1940          0.0189   Process      0

  Effective       Floating-point  SIMD inst.      SVE operation
     instruction       operation        rate(%)         rate(%)
-------------------------------------------------------------------------------
    315855961257     51350899939          0.2046  <-------->  7.1878           Application
-------------------------------------------------------------------------------
    315855961257     51350899939          0.2046         7.1878                  Process      0
```

SIMD命令に占めるSVE命令の割合

解析出力例

SVE命令の使用状況確認 3/3

FUJITSU

- **逆アセンブルによるアセンブリ命令列の確認**
  - コマンド: objdump –d <ファイル>
  - ベクトルレジスタの確認: “z”+数字
    - zで始まるレジスタが使われていない場合、SVE命令ではない

```
864:   ldr     d0, [x20, x1]
868:   fmul    d0, d0, d0
86c:   ldr     d1, [x19, x1]
870:   fadd    d0, d1, d0
874:   str     d0, [x19, x1]
878:   add     x1, x1, #0x8
87c:   cmp     x1, #0x320
880:   b.ne    864 <main+0x50>
```

スカラー命令のみの命令列

```
7b0:   ld1d    {z1.d}, p0/z, [x19, x1, lsl #3]
7b4:   ld1d    {z0.d}, p0/z, [x20, x1, lsl #3]
7b8:   fmad    z0.d, p1/m, z0.d, z1.d
7bc:   st1d    {z0.d}, p0, [x19, x1, lsl #3]
7c0:   add     x1, x1, x2
7c4:   whilelo p0.d, w1, w0
7c8:   b.ne    7b0 <main+0xb0>  // b.any
```

SVE命令を含む命令列

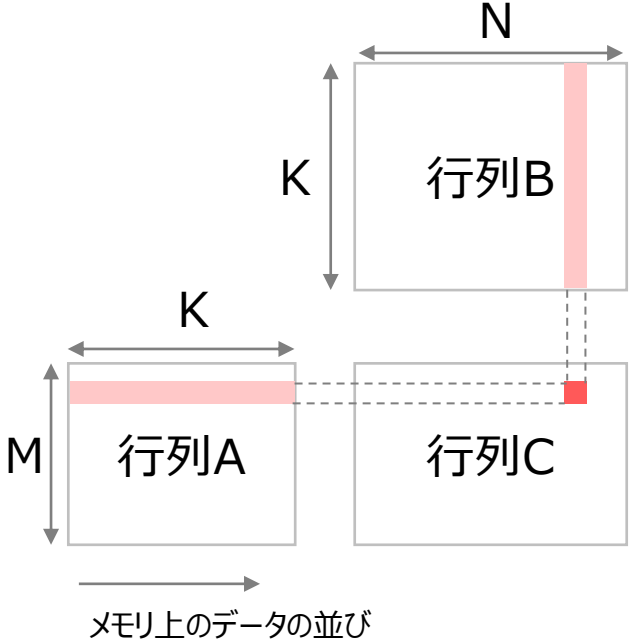
FUJITSU

A64FXアーキテクチャと並列プログラミング

行列積を例とした並列プログラムの記述方法の説明

例題：密行列積

- 密行列と密行列の積: C=A x B



単純に記述した行列積

```
for(i=0;i<M;i++){
  for(j=0;j<N;j++){
   for(k=0;k<K;k++){
     C[i][j]+=a[i][k]*b[k][j];
   }
} }
```

複数コアの並列処理の記述

FUJITSU

- OpenMPで記述することが多い
- 並列化したいループの直前にOpenMPの指示文を挿入
  - parallel節：スレッドを生成する
  - for節：直後のfor文を対象に並列処理
    privateやsharedで変数の特徴を指定したほうがよい（ループ変数ぐらいであれば不要）

```
for(i=0;i<M;i++){
  for(j=0;j<N;j++){
   for(k=0;k<K;k++){
     C[i][j]+=a[i][k]*b[k][j];
   } } }
```

```
#pragma omp parallel for
for(i=0;i<M;i++){
  for(j=0;j<N;j++){
   for(k=0;k<K;k++){
     C[i][j]+=a[i][k]*b[k][j];
   } } }
```

- M>>Nのときは、コア間の並列処理の効率が高いからこちらがよい

```
for(i=0;i<M;i++){
#pragma omp parallel for
  for(j=0;j<N;j++){
   for(k=0;k<K;k++){
     C[i][j]+=a[i][k]*b[k][j];
   } } }
```

- N>>MのときはNのループを対象にしたほうがよい
- omp parallel節はループ内部に無い方がよいので外に追い出したほうがよい（次ページ）

複数コアの並列処理の記述

FUJITSU

修正例1:parallel節を外に出す　修正例2: ループの順番の入れ替え

```
for(i=0;i<M;i++){
#pragma omp parallel for
  for(j=0;j<N;j++){
   for(k=0;k<K;k++){
     C[i][j]+=a[i][k]*b[k][j];
   } } }
```

```
#pragma omp parallel
for(i=0;i<M;i++){
#pragma omp for
  for(j=0;j<N;j++){
   for(k=0;k<K;k++){
     C[i][j]+=a[i][k]*b[k][j];
   } } }
```

```
#pragma omp parallel for
for(j=0;j<N;j++){
  for(i=0;i<M;i++){
    for(k=0;k<K;k++){
     C[i][j]+=a[i][k]*b[k][j];
  } } }
```

- A64FXではスレッド生成やバリア同期のコストが大きいため、ループ内部にparallel節やfor節はないほうがよい
  - 今回の例では、修正例1のようにparallel節を外に出すだけのではなく、ループの順序を入れ替えるほうがよい

SIMD演算器の活用

FUJITSU

- 最も内側のループでメモリ上の連続したデータに対して処理したい

```
#pragma omp parallel for
for(i=0;i<M;i++){
  for(j=0;j<N;j++){
   for(k=0;k<K;k++){
     c[i][j]+=a[i][k]*b[k][j];
   } } }
```

- 配列aはk方向で連続アクセスとなっているのでOK
- 配列b,cはk方向で連続アクセスとなっていないのNG

- j方向のループを一番内側にする

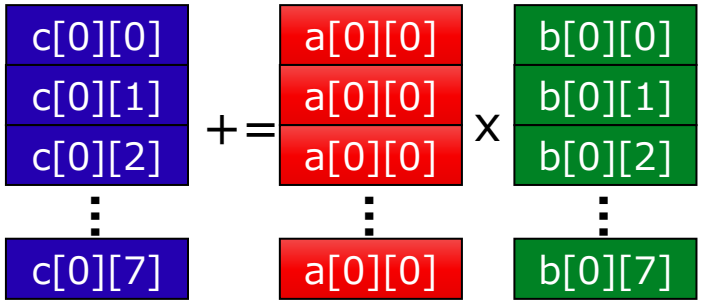
```
#pragma omp parallel for
for(i=0;i<M;i++){
  for(k=0;k<K;k++){
    for(j=0;j<N;j++){
      c[i][j]+=a[i][k]*b[k][j];
   } } }
```

- 配列b,cはj方向で連続となったためOK
- 配列aは最も内側のループでは同じデータを処理し続ける（おそらくSIMD化はされない）

（参考）前ページのコードの問題と改善

```
#pragma omp parallel for
for(i=0;i<M;i++){
  for(k=0;k<K;k++){
    for(j=0;j<N;j++){
      c[i][j]+=a[i][k]*b[k][j];
  } } }
```

- aを8要素複製する形でレジスタ上にロードしたい
- ソースコード上で記述すると、メモリ上でのデータの複製になるので遅くなる
  → ACLEやインラインアセンブラで書いたほうがよいはず



さらに高速な行列積を記述したいときは、配列a, b, cをキャッシュヒットするように適切なサイズでブロック化して、最内ループをある程度ループ展開して記述する

FUJITSU

A64FXアーキテクチャと並列プログラミング

プロファイラを使ったチューニング手順

チューニングの流れ

FUJITSU

1. プログラムのホットスポット（実行時間の割合が大きい箇所）を見つける
2. 対象ホットスポットにおいて、期待している性能がでているかを確認する
3. 性能が出ていなければ、性能が出ていない理由を分析する
4. 性能向上方法を考えて実装して、性能を測定する
5. 十分な性能が出ていなければ3に戻り、性能に満足できていれば次に進む
6. 全体性能が満足できる値となっていなければ、次のホットスポットを対象とし、2に戻る
7. 終わり

1. ホットスポットを見つける

FUJITSU

- プロセッサ内にあるHardware counterを使うプロファイリングツールを使う
  - 富士通コンパイラを使っているなら、fipp, fapp
  - そうでないなら、Linux Perfがおすすめ
- ホットスポット解析のコマンドを使って関数ごとの実行時間を集計する

2. 期待している性能がでているか確認する

FUJITSU

- 方法1：他のCPUで実行時間を計測して比較する
  - 同一コードを他のCPUで実行する
  - コンパイラ最適化の影響が出にくいように同じコンパイラと最適化オプションを使うほうがよい
- 方法2：理論性能値と比較する
  - 演算回数とデータ転送量を人手で計算する
  - 実行時間を取得し、時間あたりの演算数と時間あたりのデータ転送量を算出
    → どちらかの実行効率が50%出ていれば十分

理論演算性能（倍精度）	3.4 TFLOPS
理論演算性能（単精度）	6.8 TFLOPS
メモリバンド幅	1024 GiB/s

3. 性能が出ていない理由を分析する

FUJITSU

- プロファイラで詳細を調べる前に以下を確認する
  - コア間の並列化ができているか（=コア数を増やしたときの性能向上効果は高いか）
  - SIMD化ができているか（=SIMD命令が発行されているかを確認する）
- プロファイラで詳細を確認
  - （FAPPによる詳細プロファイルの取り方を調べて書く）

チューニングの流れ

FUJITSU

1. プログラムのホットスポット（実行時間の割合が大きい箇所）を見つける
2. 対象ホットスポットにおいて、期待している性能がでているかを確認する
3. 性能が出ていなければ、性能が出ていない理由を分析する
4. 性能向上方法を考えて実装して、性能を測定する
5. 十分な性能が出ていなければ3に戻り、性能に満足できていれば次に進む
6. 全体性能が満足できる値となっていなければ、次のホットスポットを対象とし、2に戻る
7. 終わり

目次

13:00 - 13:30	スパコンの使い方など
13:30 - 14:20	A64FXアーキテクチャと並列プログラミング（座学）
14:30 - 15:40	OpenMPを活用した並列プログラミング演習 3重ループを含むプログラムで、OpenMPの指示文挿入によるスレッド並列や SIMD演算を活用するコードを取り扱います。 また、Intrinsic（ACLE）を使ってSIMD化する例も取り上げます。
15:50 - 17:00	プロファイラを活用したチューニング演習

FUJITSU

プログラミング演習

複数コアの活用

SIMD演算器の活用

演習前の準備

- 下記ファイルをコピー
  cp /work/gt00/share/lecture_a64fx.tgz /work/gt00/{アカウント名}/
- 解凍
  cd /work/gt00/{アカウント名}/
  tar zxvf lecture_a64fx.tgz

演習概要

- 演習1: OpenMP 基本 （15分程度）
- 演習2: OpenMPディレクティブ挿入（25分程度）
- 演習3: ACLEを活用したSIMD化 (30分程度）

演習1の概要

- OpenMPディレクティブを挿入して、プログラムを高速化する
- 対象プログラム：単純なサンプルプログラム

```
 sum = 0;
#define LOOP_SIZE 1280
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      a[i] = b[i] + c[i];
    }

    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      d[i] = b[i] + c[i];
    }

    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      sum += b[i] + c[i];
    }
```

演習1 実行方法

- 作業ディレクトリ
  - /work/gt00/{アカウント名}/omp1/asis
  - 対象ファイル: main.c (55行 ~ 68行目)
- コンパイル
  - pjsub -g gt00 make.sh
- 実行
  - pjsub -g gt00 submit.sh
- 実行結果

```
Validation check is OK

total time:  63.029 [msec]
```

演習1の内容

FUJITSU

- 演習1-1: 対象コードにOpenMPディレクティブを挿入して結果検証をクリアする
  - 3つ目のループはリダクション演算なので注意（並列化しないか、reductionの指示文を使う。使い方は検索してみてください）

- 演習1-2: OpenMPディレクティブを工夫して高速化する
  - 目標実行時間：30 msec

14:30 - 13:45 演習1 (13:42に解説を始めます）
14:45 - 15:10 演習2 (15:03に解説を始める予定です）

演習1-1 回答例

```
#pragma omp parallel for
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      a[i] = b[i] + c[i];
    }
#pragma omp parallel for
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      d[i] = b[i] + c[i];
    }

#pragma omp parallel for reduction
(+:sum)
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      sum += b[i] + c[i];
    }
```

63 msec

```
#pragma omp parallel
    {

#pragma omp for
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      a[i] = b[i] + c[i];
    }
#pragma omp for
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      d[i] = b[i] + c[i];
    }
#pragma omp for reduction (+:sum)
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      sum += b[i] + c[i];
    }
    }
```

38 msec

スレッド生成回数が少ない右のコードのほうが高速

演習1-2回答例

```
#pragma omp parallel
    {

#pragma omp for nowait
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      a[i] = b[i] + c[i];
    }
#pragma omp for nowait
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      d[i] = b[i] + c[i];
    }
#pragma omp for reduction (+:sum) nowait
    for ( i = 0; i < LOOP_SIZE ; i++ ) {
      sum += b[i] + c[i];
    }

    }
```

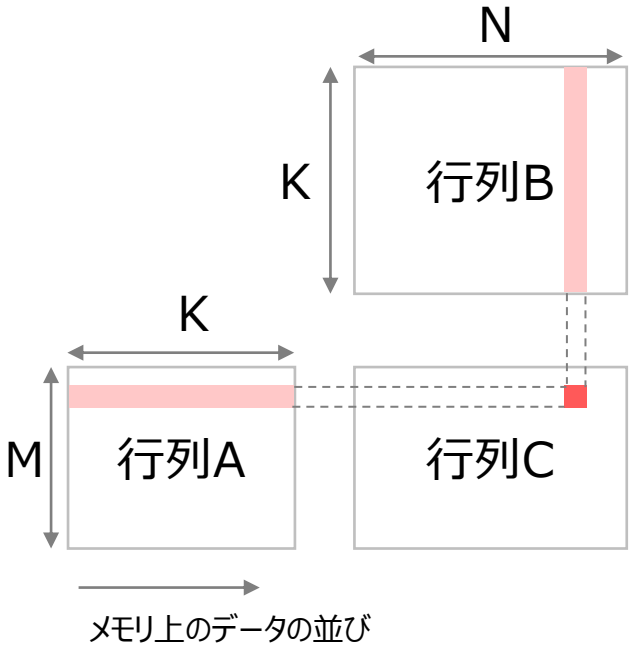
- nowait節をつけることで暗黙のバリア同期を削れる
- 38msec → 25msecへと改善

演習概要

- 演習1: OpenMP TIPS （10分程度）
- 演習2: OpenMPディレクティブ挿入（20分程度）
- 演習3: ACLEを活用したSIMD化 (30分程度)

演習2の概要

- OpenMPディレクティブを挿入して、行列積を高速化する
- 対象プログラム：密行列と密行列の積: C=A x B




```
#define a(J,I)  A[J*lda + I]
#define b(J,I)  B[J*ldb + I]
#define c(J,I)  C[J*ldc + I]
// 90行目あたり
for ( i = 0; i < M; i++ ) {
    for ( j = 0; j < N; j++ ) {
      for ( k = 0; k < K; k++) {
        c(i,j) += a(i,k) * b(k,j);
      } } }
```

演習 実行方法

- 作業ディレクトリ
  - /work/gt00/{アカウント名}/omp2/asis
  - 対象ファイル: main.c (90行 ~ 97行目)
- コンパイル
  - pjsub -g gt00 make.sh
- 実行
  - pjsub -g gt00 submit.sh
- 実行結果

```
Validation check is OK
matrix size, elapsed time, GFLOPS
M=10 N=4096 K=4096, 14.322, 0.023
```

演習内容

- 演習2-1: 対象コードにOpenMPディレクティブを挿入して高速化する
  - 目標性能: 0.7GFLOPS
  - M, N, Kの値に注意

- 演習2-2: 自由にコードを変形して高速化する (Mの値を大きくしてもよい)
  - 目標性能：3GFLOPS

14:45 - 15:10 演習2 (15:03に解説を始める予定です)

```
#define a(J,I)  A[J*lda + I]
#define b(J,I)  B[J*ldb + I]
#define c(J,I)  C[J*ldc + I]
// 90行目あたり
for ( i = 0; i < M; i++ ) {
    for ( j = 0; j < N; j++ ) {
      for ( k = 0; k < K; k++) {
        c(i,j) += a(i,k) * b(k,j);
      } } }
```

演習2-1: 解答例

```
#pragma omp parallel for  collapse(2)
  for ( i = 0; i < M; i++ ) {
    for ( j = 0; j < N; j++ ) {
      for ( k = 0; k < K; k++) {
        c(i,j) += a(i,k) * b(k,j);
      }
    }
  }
```

- iループの回転数が少ないため、collapse節を挿入して並列化対象ループを増やす

matrix size, elapsed time, GFLOPS
M=10 N=4096 K=4096, 0.461, 0.728

演習2-2: 解答例

N方向をブロック化して最内ループにすることで
同時に実行できる命令数を増やす

```
#pragma omp parallel for  collapse(2)
  for ( i = 0; i < M; i++ ) {
    for ( jj = 0; jj < N; jj+=N_BLK) {

      for ( k = 0; k < K; k++) {
        for ( j = jj; j < jj+N_BLK; j++ ) {
          c(i,j) += a(i,k) * b(k,j);
        }
      }
    }
  }
```

matrix size, elapsed time, GFLOPS
M=10 N=4096 K=4096, 0.121, 2.779

ライブラリをコールする

```
cblas_dgemm (
              CblasRowMajor,
              CblasNoTrans,
              CblasNoTrans,
              M, N, K,
              alpha,
              A, lda,
              B, ldb,
              beta,
              C, ldc );
```

matrix size, elapsed time, GFLOPS
M=10 N=4096 K=4096, 0.006, 56.290

演習2-2: 解答例2

Mを4096にしてライブラリをコールする(その2)

```
cblas_dgemm (
            CblasRowMajor,
            CblasNoTrans,
            CblasNoTrans,
            M, N, K,
            alpha,
            A, lda,
            B, ldb,
            beta,
            C, ldc );
```

Validation check is OK
matrix size, elapsed time, GFLOPS
M=4096 N=4096 K=4096, 0.073, 1894.299

演習概要

- 演習1: OpenMP TIPS （10分程度）
- 演習2: OpenMPディレクティブ挿入（20分程度）
- 演習3: ACLEを活用したSIMD化 (30分程度)

演習用プログラムの格納先

FUJITSU

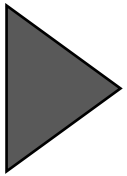
- /work/gt00/share/ACLE
  - memcpy
  - H-gate
  - PConvReLUBNorm

ACLE

FUJITSU

- C/C++で組み込み関数を用いてSVE命令を使用可能にする拡張機能
- VLA(Vector Length Agnostic)プログラミング
  - SVEは128-bit ~ 2048-bitのベクトル長をサポートする命令体系
  - 可変のベクトル長を意識したプログラミングが可能
  - 同一コードで、異なるベクトル長のSVEをサポートするプロセッサで動作可能

```
for(int i=0; i < N; i++){
  float val = array[i];
  val += val*val;
  array[i] =val;
}
```



```
#include <arm_sve.h>

for(int i=0; i < N; i+=svcntw()){
  svbool_t pg = svwhilelt_b32(i, N);
  svfloat32_t sv_val = svld1(pg, &array[i]);
  svmla_x(pg, sv_val, sv_val, sv_val);
  svst1(pg, &array[i], sv_val);
}
```

任意のベクトル長の
SVE搭載プロセッサで実行可能

ACLEを使うための事前準備

FUJITSU

- ヘッダファイルの追加
  - arm_sve.h ← SVE data typeとintrinsicの追加
- コンパイラオプションの追加
  - ターゲットアーキをSVEサポートのアーキに
    - -march=armv8.2-a+sveなど

コンパイル例
gcc –O3 **–march=armv8.2-a+sve** main.cpp

```
#include <arm_sve.h> //ヘッダファイル追加

for(int i=0; i < N; i+=svcntw()){
  svbool_t pg = svwhilelt_b32(i, N);
  svfloat32_t sv_val = svld1(pg, &array[i]);
  svmla_x(pg, sv_val, sv_val, sv_val);
  svst1(pg, &array[i], sv_val);
}
```

ACLE: SVEレジスタの要素数の取得

FUJITSU

- svcnt[b, h, w, d]()関数
  - 8-, 16-, 32-, 64-bitデータ型それぞれに対応する関数がある
  - svcntw(): 32-bitデータ換算の要素数を取得

```
#include <arm_sve.h>

for(int i=0; i < N; i+=svcntw()){
  //処理
}
```

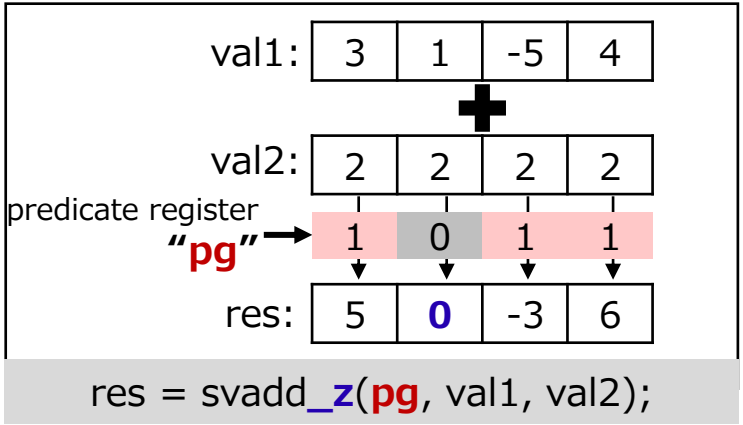
ループ変数の増加量に使用する例

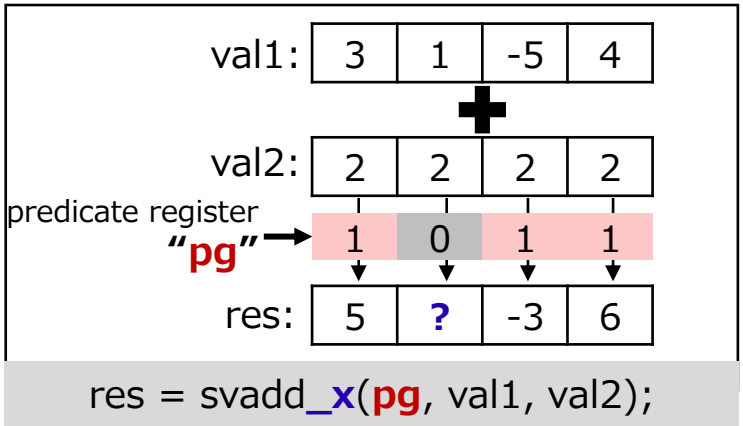
関数	Grace CPU	Graviton3	A64FX
svcntb()	16	32	64
svcnth()	8	16	32
svcntw()	4	8	16
svcntd()	2	4	8

ACLEでのプレディケート変数

- データ型: svbool_t
- 命令を適用するSVEレジスタの要素を指定するためのフラグ
  - SVE命令セットと同じでACLEの多くの組み込み関数の引数となっている
  - フラグがfalseのときの要素の扱いによってACLE組み込み関数のプレフィックスが異なる
    - “_x”: falseの要素の値を保証しない
    - “_z”: falseの要素は0が代入される
    - “_m”: 入力要素が代入されるなど…

→今後使用しない要素の場合は”_x”がおすすめ





ACLE: プレディケート変数の設定

- 全要素分のフラグをtrueに設定
  - 関数名: svptrue_bX()
  - “_bX”: X-bitの全要素分のフラグがtrue

predicate register “pg” 1 1 1 1 1 1 1 1

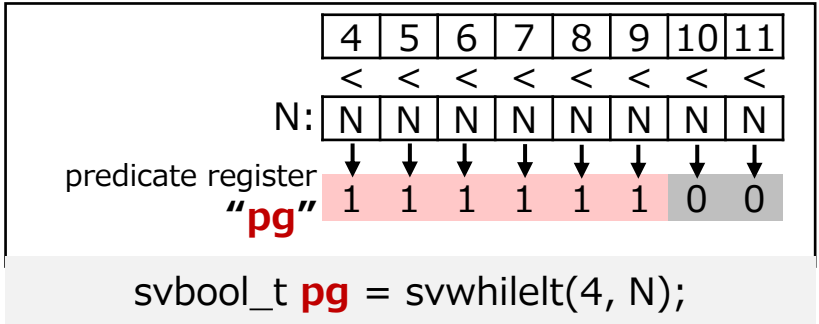
```
svbool_t  pg = svptrue_b64();
```

- 特定パターンのプレディケートの設定
  - 関数名: svptrue_pat_b64(pattern);
  - パターン: SV_VL[1,2,4,8,16,32]など
    - SV_VL2 → 下位2要素分がtrueとなるのプレディケートレジスタ設定

predicate register “pg” 1 1 1 1 0 0 0 0

```
svbool_t pg =svptrue_pat_b64(SV_VL4);
```

- 動的なプレディケートレジスタの設定
  - オペランド比較でプレディケートレジスタを設定
  - 関数例
    - svwhile[le, lt, ge, gt]
    - svcmp[eq, ne, le, lt, ge, gt]




```
svbool_t pg = svwhilelt(4, N);
```

ACLE: SVEレジスタへのデータ読み込み

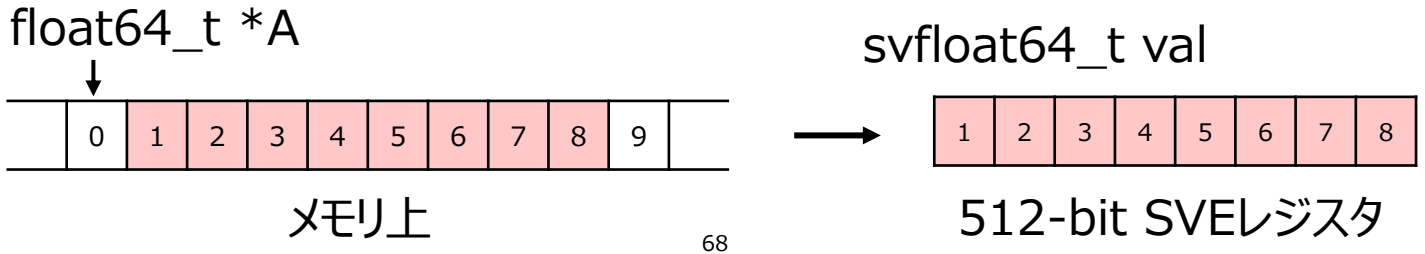
FUJITSU

- float64_tの配列からSVEレジスタにデータを読み込む命令

```
svfloat64_t svld1(svbool_t pg, float64_t *base);
```

  - 戻り値 svfloat64_t: SVEレジスタ用のデータ型
  - 引数1 svbool_t：使用するプレディケートレジスタ
  - 引数2 float64_t*: 読み込みデータの先頭アドレス

- 使用例: svfloat64_t val = svld1(svptrue_b64(), &A[1]);
  - svptrue_b64()：すべての要素がactiveのプレディケートレジスタを返す関数



ACLE: SVEレジスタからのデータ書き込み

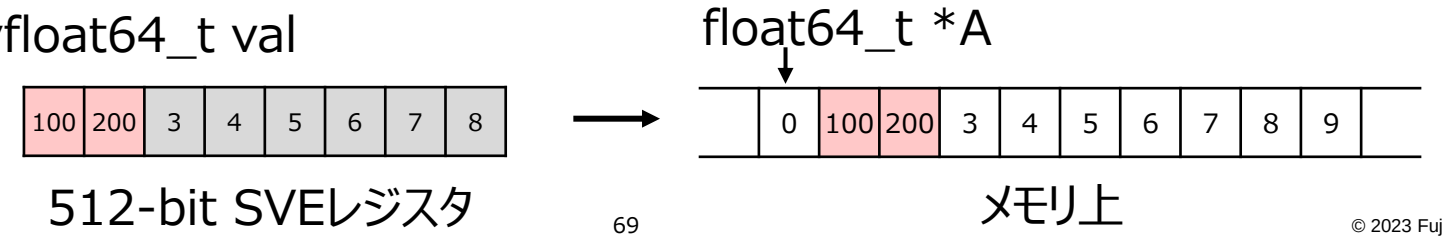
FUJITSU

- SVEレジスタからfloat64_tにデータを書き込む命令

```
svst1(svbool_t pg, float64_t *base, svfloat64_t val);
```

  - 引数1 svbool_t：使用するプレディケートレジスタ
  - 引数2 float64_t*: 読み込みデータの先頭アドレス
  - 引数3 svfloat64_t: 書き込むデータを保持しているSVEレジスタ

- 使用例: svst1(svptrue_pat_b64(SV_VL2), &A[1], val);
  - svptrue_pat_b64()：特定パターンの要素がactiveのプレディケートレジスタを返す関数
    - SV_VL2: 2要素目までがactiveになるプレディケートパターン



ACLE: スカラーレジスタからSVEレジスタへブロードキャスト

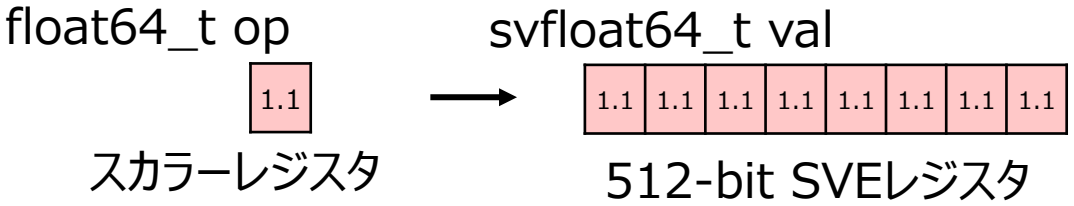
FUJITSU

- スカラー値をsvfloat64_tにブロードキャスト

```
svfloat64_t svdup_f64(float64_t op);
```

  - 戻り値 svfloat64_t：SVEレジスタ用のデータ型
  - 引数 float64_t: ブロードキャストするオペランド

- 使用例: svfloat64_t val = svdup_f64(op);



ACLE: 加・減・乗算/乗加算

FUJITSU

- svfloat64_tに対するACLE関数を示す
  - 異なる型でも同様に記載可能

処理	ACLE関数	プレフィックス	戻り値	引数	計算内容
加算	svadd	_[x, z, m]	計算結果 svfloat64_t	svbool_t pg, svfloat64_t op1, svfloat64_t op2	op1 + op2
減算	svsub				op1 – op2
乗算	svmul				op1 * op2
除算	svdiv				op1 / op2
乗加算	svmla			svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3	op1 + op2 * op3

ACLE演習：配列コピー

FUJITSU

- 内容：配列Aから配列Bへデータをコピー
- 配列の各要素はcomplex<double>型：64-bit x 2
- 注意事項
  - ACLE化では、complexのload/storeがないためdoubleポインタへ変換が必要

  → double *ptr = (double *)in.data();

```
void memcpy(vector<complex<double>> &in,
vector<complex<double>> &out) {

  for (int i = 0; i < out.size(); i++)
    out[i] = in[i];

}
```

AsIsコード

Step1
ACLEによるSIMD化

Step2
OpenMPによる
スレッド並列

ACLE演習: 配列コピー 実行方法

- 作業ディレクトリ
  - /work/gt00/{アカウント名}/ACLE/memcpy/asis
  - 対象ファイル: memcpy.cpp
- コンパイル
  - pjsub -g gt00 make.sh
- 実行
  - pjsub -g gt00 submit.sh
- 実行結果

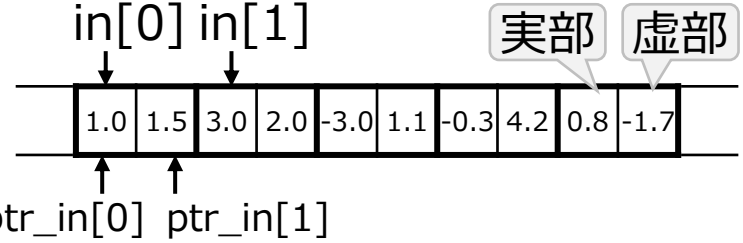
```
== Memcpy benchmark ==
  Array size[GB]: 2
  memcpy exection time[msec]: XXXX
  Bandwidth [GB/sec]: XXXX
    Validation: PASSED
```

- 演習
  - Step1: ACLEによるSIMD化
    → 目標: 約 40 GB/sec
  - Step2: +OpenMPによるスレッド並列
    → 目標(12コア, 1CMG): 約125 GB/sec
- 注意事項
  - ACLE化では、complexのload/storeがないため、doubleポインタへ変換が必要
    - double *ptr = (double *)in.data();
  - double型でアクセスする場合は要素数が2倍
  - スレッド数設定
    - export OMP_NUM_THREADS=12

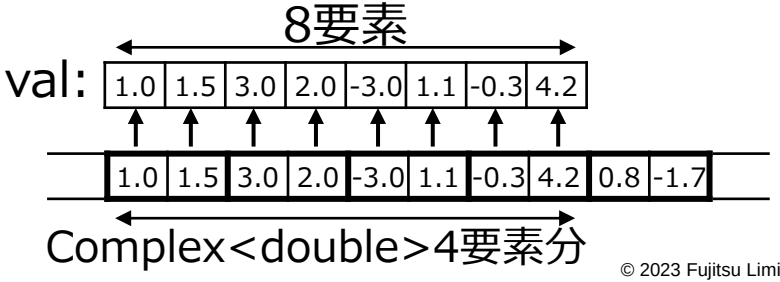
ACLE演習: 配列コピー ACLE+OpenMP例

```
void memcpy(vector<complex<double>>& in, vector<complex<double>>& out){

  // double型のポインタ作成
  double *ptr_in = (double *)in.data();
  double *ptr_out = (double *)out.data();

  // double型換算の要素数算出
  size_t num = out.size() * 2;

  // double型換算の要素数でforループ実行
#pragma omp parallel for
  for (int i = 0; i < num; i += svcntd()) {
    // [i, i+1, …, i+7]がnum未満でtrueとなる
    // プレディケートレジスタを作成
    svbool_t pg = svwhilelt_b64(i, num);

    svfloat64_t val;
    // double型で連続(最大)8要素の読み込み
    val = svld1(pg, &ptr_in[i]);
    // double型で連続(最大)8要素の書き込み
    svst1(pg, &ptr_out[i], val);
}}
```

- double型ポインタの配列の見え方
  - ptr_inからは2倍の要素数の配列にみえる

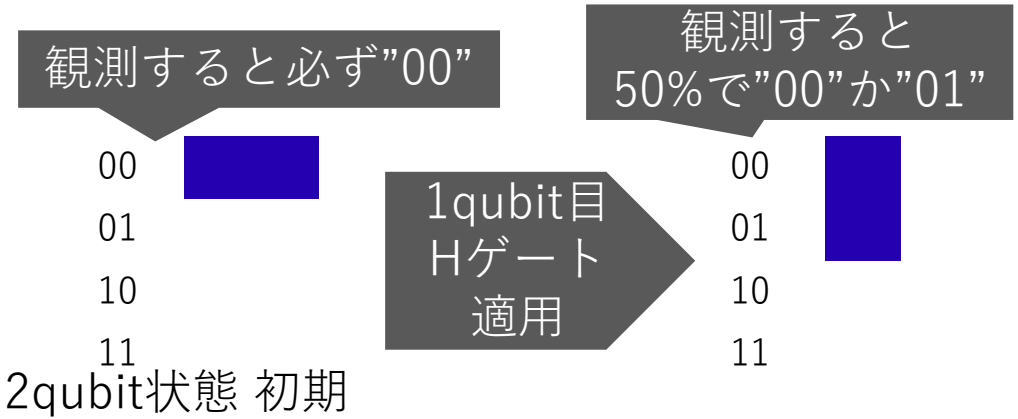


- val = svld1(pg, &ptr_in[0])



ACLE演習2:量子ゲート操作 (H-gate)

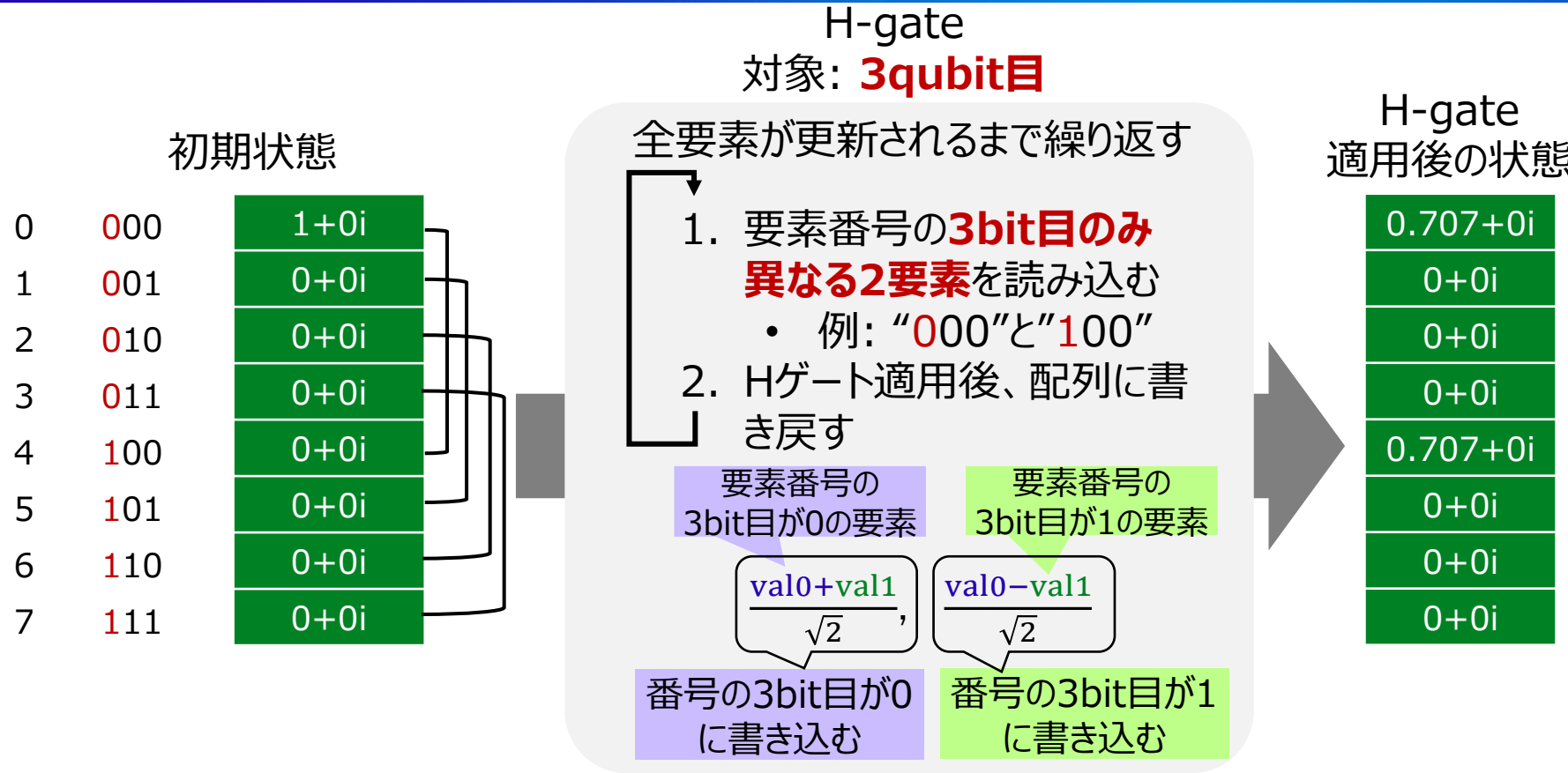
- State Vector方式の量子シミュレーション
  - 全ての量子状態を保持し量子ゲート操作による更新を行う
  - n量子ビットのシミュレーション→ 2^n要素の複素配列に対する演算
- Hゲート(アダマールゲート)操作
  - 重ね合わせ状態を作るゲート
  - 配列の2要素間の演算を全ペアに対して実行



疑似コード
target: 対象qubit

```
mask = (1 << target)
mask_l = mask - 1
mask_h = ~mask_l

for i=0 to 2^(n-1)
  idx0 ← (i & mask_l) + ((i & mask_h)<<1)
  idx1 ← idx0 + mask
  val0 ← state[idx0];  val1 ← state[idx1]

  state[idx0] ← (val0+val1)/√2
  state[idx1] ← (val0−val1)/√2
end for
```

ACLE演習2: Hゲート処理



ACLE演習2: 実行条件

FUJITSU

- 配列サイズ: 2^{27} = 128M要素
- 配列データ型: Complex128 (実部, 虚部ともにfloat64_t)
- Hゲート対象qubit: 20qubit目
  - 要素番号で20bit目が異なる要素がペアとなり、Hゲート操作を適用
  - SVEベクトル長内での複雑なデータ移動は必要ない
- 注意事項
  - ACLE化では、complexのload/storeがないため、doubleポインタへ変換が必要
    - double *ptr = (double *)in.data();
  - スレッド数設定
    - export OMP_NUM_THREADS=12

ACLE演習2: H_gate 実行方法

- 作業ディレクトリ: asis
  - /work/gt00/{アカウント名}/ACLE/H-gate/asis
  - 対象ファイル: hgate.cpp
- コンパイル
  - pjsub -g gt00 make.sh
- 実行
  - pjsub -g gt00 submit.sh
- 実行結果

```
== H-gate benchmark ==
 Array size[GB]: 2
 Target qubit: 20
H-gate exection time[msec]: XXXX
Validation: PASSED
```

- 演習
  - AsIs
    - 性能: 約 25 sec
  - Step1: ACLEによるSIMD化
    → 目標: 約 1 sec以下
  - Step2: +OpenMPによるスレッド並列
    → 目標(12コア, 1CMG): 約 0.3 sec以下

使用するACLE命令一覧

FUJITSU

- SVEレジスタの要素数取得
  - svcntd(): 64-bit換算での要素数を取得
- プレディケートレジスタの設定
  - svbool_t pg = svptrue_b64()
- データ読み込み
  - svfloat64_t sv_val = svld1(svbool_t, float64_t *base)
- データ書き込み
  - svst1(svbool_t, float64_t *base, svfloat64_t val)
- SVEレジスタの各要素にスカラー値をブロードキャスト
  - svfloat64_t sv_val = svdup_f64(float64_t val)
- 浮動小数点演算
  - 加算: svfloat64_t sv_val3 = svadd_[x, z, m](svbool_t pg, svfloat64_t sv_val1, svfloat64_t sv_val2)
  - 減算: svfloat64_t sv_val3 = svsub_[x, z, m](svbool_t pg, svfloat64_t sv_val1, svfloat64_t sv_val2)
  - 乗算: svfloat64_t sv_val3 = svmul_[x, z, m](svbool_t pg, svfloat64_t sv_val1, svfloat64_t sv_val2)
  - 除算: svfloat64_t sv_val3 = svdiv_[x, z, m](svbool_t pg, svfloat64_t sv_val1, svfloat64_t sv_val2)

ACLE演習2: H-gate ACLE+OpenMP例

FUJITSU

```
void H_gate(vector<complex<double>>& array, int target_qubit){
  …
  const int vec_len = svcntd();
  svbool_t pg = svptrue_b64();
  svfloat64_t factor = svdup_f64(sqrt2inv);

#pragma omp parallel for
  for (int i = 0; i < loop_dim; i += (vec_len >> 1)) {
    svfloat64_t input0, input1, output0, output1;
    int basis_index_0 =(i & mask_low) + ((i & mask_high) << 1);
    int basis_index_1 = basis_index_0 + mask;

    input0 = svld1(pg, &ptr[basis_index_0*2]);
    input1 = svld1(pg, &ptr[basis_index_1*2]);

    output0 = svadd_x(pg, input0, input1);
    output0 = svmul_x(pg, output0, factor);
    output1 = svsub_x(pg, input0, input1);
    output1 = svmul_x(pg, output1, factor);

    svst1(pg, &ptr[basis_index_0*2], output0);
    svst1(pg, &ptr[basis_index_1*2], output1);
}}
```

①SVEレジスタの要素数取得(64-bitで8要素)
②全要素分trueのプレディケート作成
③全要素が$\frac{1}{\sqrt{2}}$のSVEレジスタ作成

SVE load命令
(A64FX:8要素まとめて読み込み)

$$\frac{\mathrm{input0} + \mathrm{input1}}{\sqrt{2}}$$

$$\frac{\mathrm{input0} - \mathrm{input1}}{\sqrt{2}}$$

SVE store命令
(A64FX:8要素まとめて書き込み)

目次

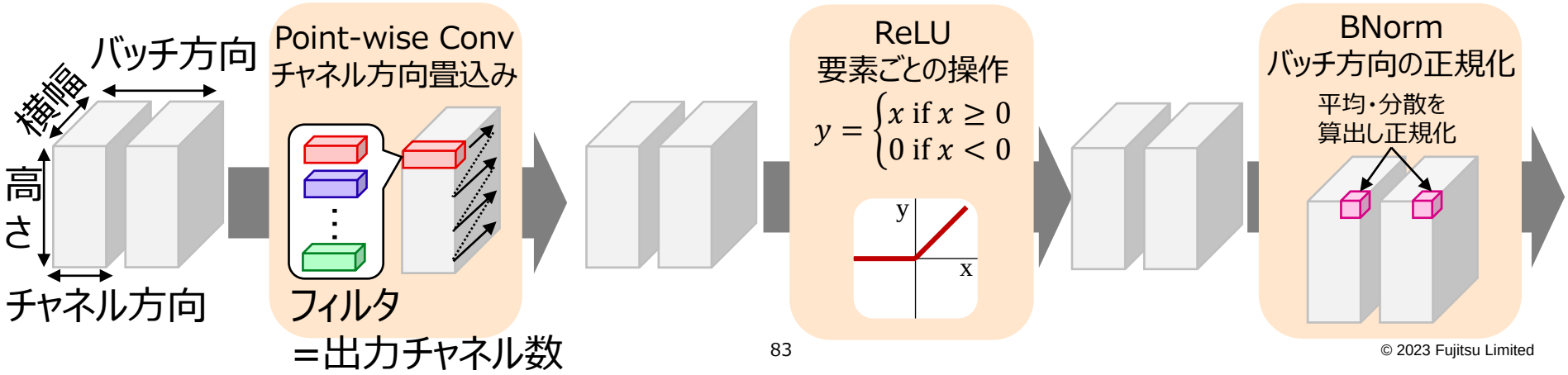
13:00 - 13:30	スパコンの使い方など
13:30 - 14:20	A64FXアーキテクチャと並列プログラミング（座学）
14:30 - 15:40	OpenMPを活用した並列プログラミング演習 3重ループを含むプログラムで、OpenMPの指示文挿入によるスレッド並列や SIMD演算を活用するコードを取り扱います。 また、Intrinsic（ACLE）を使ってSIMD化する例も取り上げます。
15:50 - 17:00	プロファイラを活用したチューニング演習

FUJITSU

プロファイラを活用したチューニング演習

演習内容

- 目的: プロファイラによる高コスト関数の特定＆チューニング
- 演習コード: 深層学習の複数種類のレイヤ処理
  - Point-wise Convolution → ReLU → Batch Normalization
    - Point-wise Convolution: チャネル方向のみの畳み込み
    - ReLU: 活性化関数の１つ(要素ごとの操作)
    - Batch Normalization: バッチ方向の正規化



演習 実行方法

- 作業ディレクトリ:
  - /work/gt00/{アカウント名}/ACLE/PConvReLUBNorm/asis
  - 対象ファイル: layers.cpp
    - PointConv, ReLU, BNorm
- コンパイル
  - pjsub -g gt00 make.sh
- 実行
  - pjsub -g gt00 submit.sh
- 実行結果

- 演習
  - Step1: 高コスト関数の特定
  - Step2: SVE+OpenMPチューニング

```
== PointConv+ReLU+BNorm layers benchmark ==
  Exection time[msec]: 8115.26
 -- Validation --
  ref time[msec]: 876.264
  validation time[msec]: 8.352
  ***Validation: PASSED
```

プロファイラの使用方法

FUJITSU

1. プロファイルデータの取得
   - fipp –C –d <dir name> -Icall ./ConvReLUBNorm
2. プロファイルデータの解析
   - fipppx –A –Icpupa –Sregion –d <dir name> > **result.txt**
   - -Icpupa:スレッド単位のメモリースループット、演算数などCPU動作状況を出力
   - -Sregion: **fipp_start/stop区間**のみの解析
3. “**result.txt**”の確認

main.cpp

```
fipp_start();
for (int i = 0; i < Loop; i++) {
  PointConv(conv_in, conv_out, weight, batch, conv_in_h, conv_in_w, conv_in_c,
            conv_out_h, conv_out_w, conv_out_c);
  ReLU(conv_out, batch * conv_out_h * conv_out_w * conv_out_c);
  BNorm(conv_out, batch, conv_out_h * conv_out_w * conv_out_c);
}
fipp_stop();
```

AsIsコードのプロファイル結果

FUJITSU

- 最もコストが高い関数：PointConv
  - 処理時間の60%以上を占める → 2倍高速化すると、処理時間が30%短縮
- 次にコストが高い関数：BNorm
  - 処理時間の20%を占める → 2倍高速化すると、処理時間は10%短縮

```
********************************************************************************
       Application - procedures
         Application and Process outputs the total value of the cost of each thread.
         Procedure outputs the total value of the procedure cost of each thread.
********************************************************************************

            Cost          %  Operation (s)        Barrier          %    Start      End
--------------------------------------------------------------------------------
              97   100.0000         9.8403             14    14.4330       --       --   Application
--------------------------------------------------------------------------------
              61    62.8866         6.1883              0     0.0000        3       15   PointConv(…
              20    20.6186         2.0289              0     0.0000       23       47   BNorm(…
```

PointConv関数のスレッド並列化例

- 5重のforループの外側3重分は依存関係がない
  - for (int m = 0; m < ic; m++)は書き込み先が重複するため並列化できない
- “parallel for” + collapseでスレッド並列化

```
== PointConv+ReLU+BNorm layers benchmark ==
  Exection time[msec]: 8115.26
 -- Validation --
  ref
  vali
  ***V
```

12スレッド

```
== PointConv+ReLU+BNorm layers benchmark ==
  Exection time[msec]: 2846.09
 -- Validation --
  ref time[msec]: 878.331
  validation time[msec]: 8.352
  ***Validation: PASSED
```

```
#pragma omp parallel for collapse(3)
  for (int i = 0; i < b; i++)
    for (int j = 0; j < oh; j++)
      for (int k = 0; k < ow; k++)
        for (int m = 0; m < ic; m++)
          for (int l = 0; l < oc; l++)
            out[((i * oh + j) * ow + k) * oc + l] =
                out[((i * oh + j) * ow + k) * oc + l] +
                wei[l + m * oc] * in[((i * ih + j) * iw + k) * ic + m];
```

PointConv並列化版のプロファイル結果

スレッド並列では、実行時間はスレッド並列数の合計値
→実際の処理時間とは異なる
→スレッド毎の処理コストを確認

スレッド並列プログラムの場合、スレッド間同期待ちコストが該当
→Bnorm待機コスト

```
 Cost              %  Operation (                          art
------------------------------                          -------------------------
          337   100.0000                                 7656          --         --   Appl      ion
------------------------------
          225    66.7656          .6056          225    66.7656          --         --   __?unknown
           85    25.2226         8.5399            0     0.0000           6         16   PointConv
           21     6.2315         2.1099            0     0.0000          24         48   BNorm

*******************************************************************************
    Thread       0 - procedures
*******************************************************************************

         Cost          %  Operation (s)       Barrier          %
----------------------------------------------------------------------
           29   100.0000         2.9136            0     0.0000          --         --             0
----------------------------------------------------------------------------
           21    72.4138         2.1099            0     0.0000          24         48   BNorm
            8    27.5862         0.8038            0     0.0000           6         16   PointConv
```

BNorm関数が最もコストが高い

BNorm関数のスレッド並列化の例

- 外側ループにループ間の依存関係はない
  - 外側ループにparallel for指示文を挿入しスレッド並列化可能

```
#pragma omp parallel for
for (int i = 0; i <
num_elements_inbatch; i++)
```

```
== PointConv+ReLU+BNorm layers benchmark ==
  Exection time[msec]: 2846.09
 -- Validation --
  re
  va
  **
```

12スレッド

```
== PointConv+ReLU+BNorm layers benchmark ==
  Exection time[msec]: 865.35
 -- Validation --
  ref time[msec]: 877.017
  validation time[msec]: 8.352
  ***Validation: PASSED
```

BNorm関数並列化版のプロファイル結果

FUJITSU

- BNorm関数が高コスト関数ではなくなる
- 高コスト関数は再びPointConv関数に
- これまで高コスト関数ではなかったReLU関数が全体の10%を占める

```
********************************************************************************
    Thread       0 - procedures
********************************************************************************

           Cost          %  Operation (s)        Barrier          %    Start      End

-------------------------------------------------------------------------------
              9   100.0000         0.9153              0     0.0000       --       --   Thread    0
-------------------------------------------------------------------------------
              8    88.8889         0.8136              0     0.0000        6       16   PointConv
              1    11.1111         0.1017              0     0.0000       18       22   ReLU
```