A64FX CPU向けコンパイラとチューニング事例

富士通株式会社
ミッションクリティカルシステム事業本部
ＨＰＣシステム事業部　言語ソフトウェア部
原口正寿

自己紹介

- スーパーコンピュータ京、富岳などのコンパイラ開発を担当
  - 京や富岳において、現在のtradモードのコンパイラ開発をリード
  - 現在は、OSSコンパイラをベースとするclangモードも含めてコンパイラ全体を担当

はじめに

FUJITSU

- 富岳・FX1000のA64FXは、省電力を目指した設計で、x86と特性が異なるため、最高性能を引き出すにはコツがいる
- 本日は、コンパイラ機能の活用やアプリチューニングでA64FXの性能を引き出すコツのいくつかを紹介する

アウトライン

- A64FXの特徴（電力性能）とアーキテクチャ特性
- コンパイラ機能活用による性能改善
- clangモードについて
- プログラムチューニングによる性能改善
- チューニング事例

A64FXの特徴（電力性能）

FUJITSU

- 性能／電力を重視で設計
  - 下表は、SPEC CPU2017における性能と電力の測定結果
  - ratio値単体では富岳はCascade Lakeの約6割の性能
  - 最大電力あたりのratio値（ratio/最大電力）において、富岳はCascade Lakeの約2倍

SPEC CPU2017 fp	富岳			PRIMERGY RX2530M5 (Intel Xeon Platinum 8268 Cascade Lake)			ratio/最大電力の富岳とXeonの比(1.0以上で富岳が良い)
	48コア			24コア×2ソケット			
	2.0GHz(ノーマルページ)			2.9GHz(Turbo off)			
	SPEC ratio	最大電力(W)	ratio/最大電力	SPEC ratio	最大電力(W)	ratio/最大電力	
幾何平均	69.2	118.5	0.584	118.5	434.2	0.273	2.14

※注：本電力性能測定データは、弊社で実測した参考値で、SPEC未登録です。

アプリ性能観点から見たアーキテクチャ特性

FUJITSU

- アーキテクチャ特性から言える性能向上のポイント
  - レイテンシの隠蔽（ソフトウェアパイプライニング、プリフェッチ）
  - メモリバンド幅の有効活用
  - SIMD化

項目	A64FX	Cascade Lake	性能向上のポイント
周波数	2.0GHz	2.7/2.2/1.8GHz(int/avx2/avx512)	
Turbo周波数	2.2GHz	上限4.0/3.8/3.7GHz	
L1Dキャッシュ	64KiB/core, 4way	32KiB/core, 8way	
命令レイテンシ(simd)	load 11	load 4～6	**レイテンシの隠蔽が重要スカラ命令のレイテンシも長く、SIMD化も重要**
	fma 9	fma 4	
	add/logic 4	add/logic 1	
メモリバンド幅(ノード)	256GB/s×4	140.78GB/s× 2	**A64FX有利だが実アプリではより必要なケースも**
レイテンシ隠蔽手段	OoO※1+コンパイラによるSWP※2等	短レイテンシ+大OoO※1+SMT	**SWP有効であれば有利**

※1 OoO:Out of Order資源、※2 SWP:ソフトウェアパイプライニング

コンパイラ機能活用による性能改善

FUJITSU

- レイテンシの隠蔽
  - ソフトウェアパイプライニング
    - 基本原理、実装、効果
    - 評価方法
    - チューニング機能
    - 促進機能と効果
    - ジレンマ（性能向上しないケース）
    - レジスタ不足の影響対策、ループ分割
  - キャッシュ活用
    - ソフトウェアプリフェッチ
- メモリバンド幅ネックの対策
  - zfillの活用

ソフトウェアパイプライニング：基本原理

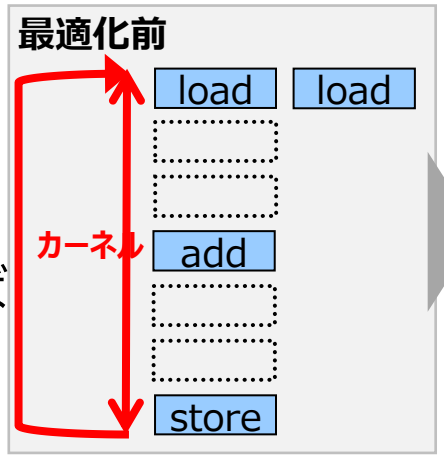
FUJITSU

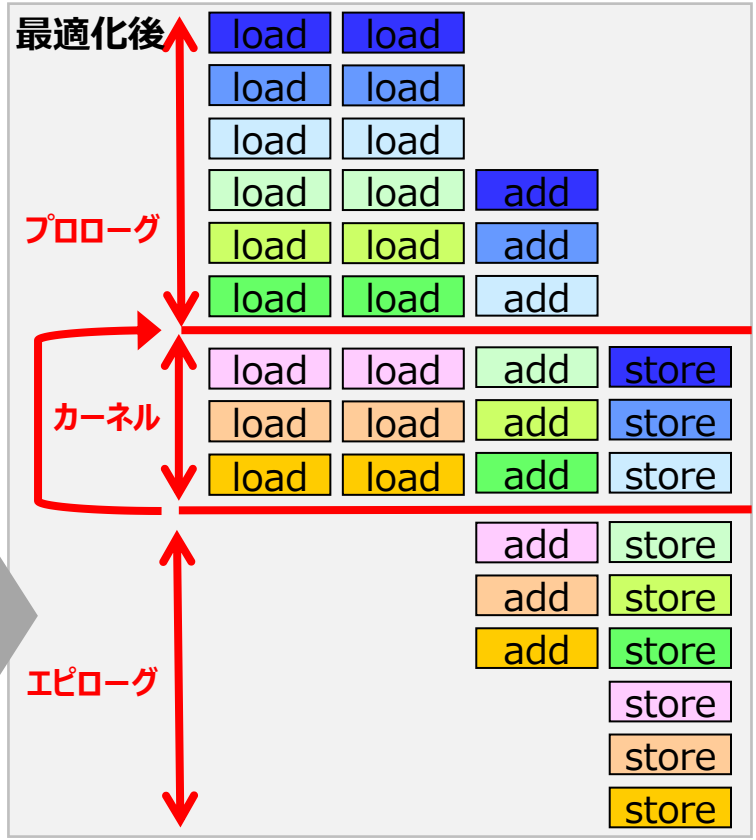
- ループの次回転以降を重ね合わせて命令レベルの並列性を向上

概念説明のためのマシンモデル
load の latency = 3サイクル
add の latency = 3サイクル
store の latency = 1サイクル
演算器の数(load,store用)= 3
commit 数 = 4 (load,storeは3つまで同時発行可能)

プログラム例
```
do i=1,n
   a(i) = b(i)+ c(i)
enddo
```

- このマシンモデルであれば右図のようにカーネルはきっちり埋まる





ソフトウェアパイプライニング：実装（１）

FUJITSU

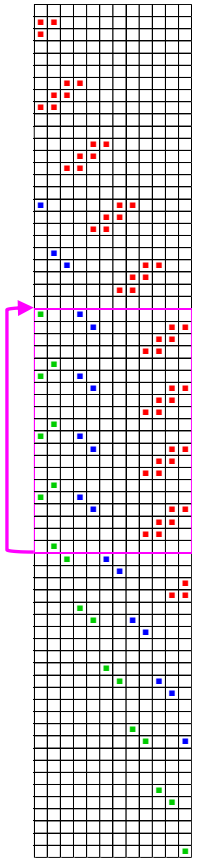
- スケジューラ実装
  - データアクセスはオンキャッシュを想定
  - カーネルでハードリソースを使い切るように命令スケジュール
- 翻訳時適用判断など
  - ソフトウェアパイプライニング時点での最内ループが対象
    （事前にフルアンローリング等が動作してループでなくなっている場合は親ループが対象）
  - ソフトウェアパイプライニング時点で、関数・サブルーチンCALL、分岐があったら抑止
  - レジスタ不足と判定されたループは抑止
  - ループが巨大（命令数が多い、メモリアクセスが多い）な場合は抑止
  - 有限の時間内に良いスケジューリング結果が求まらない、もしくはデータ依存が原因でスケジューリング効果がないと判断した場合は抑止
- 実行時の動き
  - ある程度のループ回転数がないと高速ルートを通らない
    そのルート以外を通る場合にはかえって遅くなるケースもある

ソフトウェアパイプライニング：実装（２）

FUJITSU

```
do i=1,n
  a(i) = b(i) + c(i)*d(i)
enddo
```

ソフトウェアパイプライニングのカーネルループ



- A64FXでの実際の動き
  - 演算器数、レジスタ数、命令レイテンシ／パイプラインを配慮し、カーネルでムダなくリソースを使うスジューリング結果を探し出す

ソフトウェアパイプライニングのマシンモデル

  - 倍精度浮動小数点レジスタ：３２個
  - SIMD loadは２つまで同時実行可、storeとの同時実行は不可
  - レイテンシ（下表）

ld1d(SIMDﾛｰﾄﾞ)	■	11T
fmla(FMA)	■	9T
st1d(SIMDｽﾄｱ)	■	1T(突き放しのため)

  - このカーネルルートを通るには、ある程度の回転数が必要
    - 左例では144回転必要

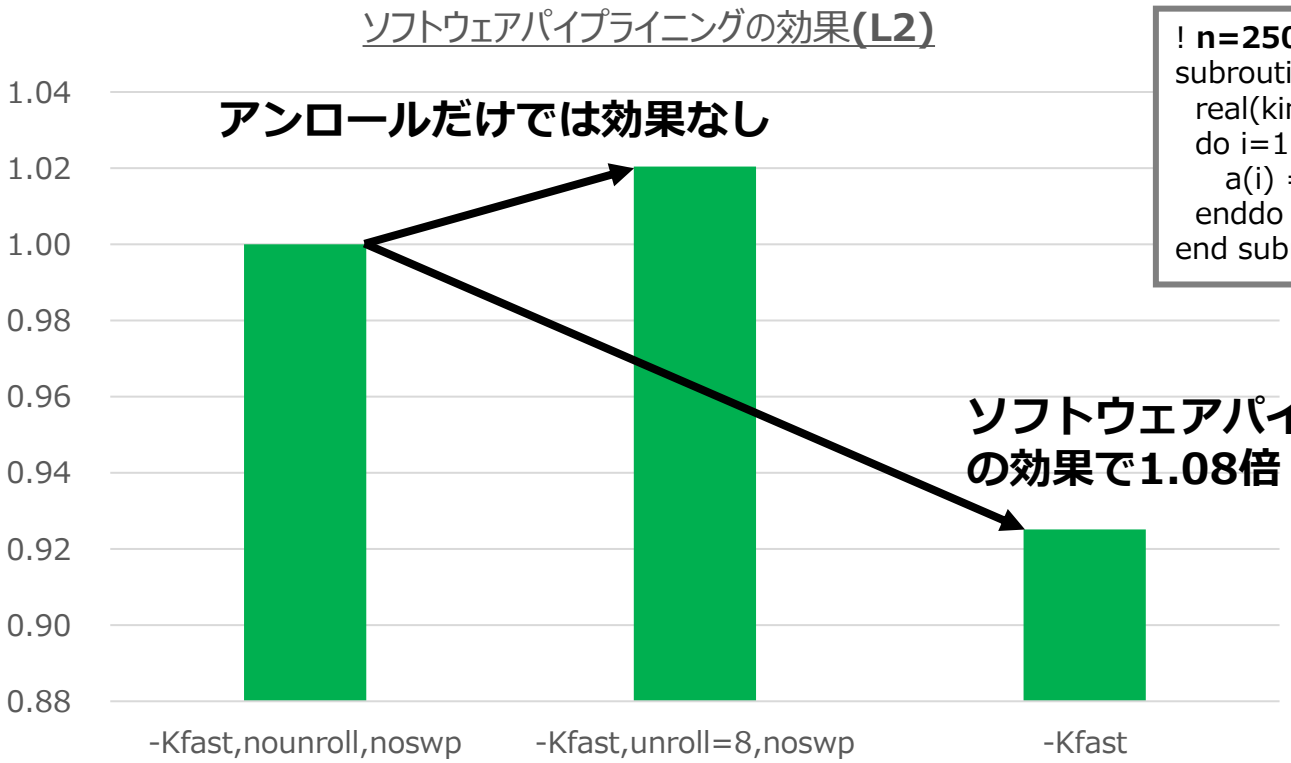
-Koptmsg=2で出力される最適化メッセージ

jwd8205o-i "fma.c", line 3: ループの繰返し数が144回以上の時、ソフトウェアパイプライニングを適用したループが実行時に選択されます。

-Nlst=tで生成される *.lst ファイルの最適化出力情報

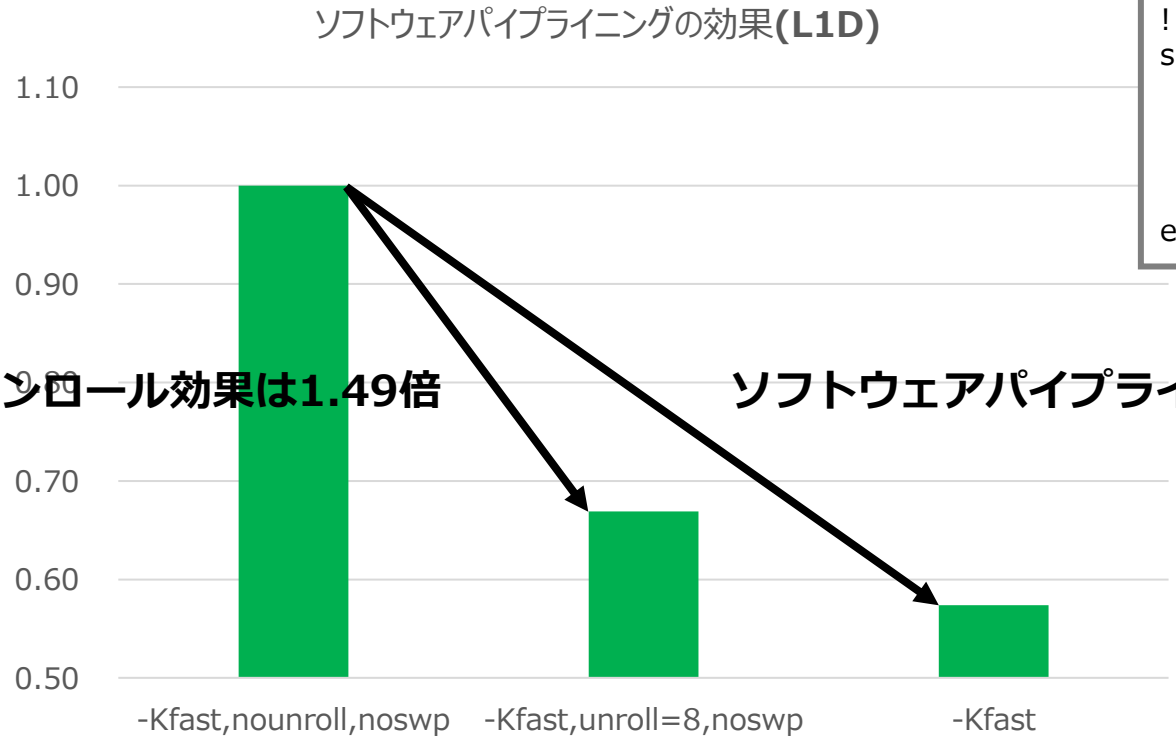
SOFTWARE PIPELINING(IPC: 3.20, ITR: 144, MVE: 4, POL: S)

ソフトウェアパイプライニング：効果(1)




```
! n=250000 で sub を 10000回呼出し
subroutine sub(a,b,c,d,n)
  real(kind=8),dimension(1:n) :: a,b,c,d
  do i=1,n
     a(i) = b(i) + c(i)*d(i)
  enddo
end subroutine
```

ソフトウェアパイプライニング：効果(2)




```
! n=500 で sub を 5000000回呼出し
subroutine sub(a,b,c,d,n)
  real(kind=8),dimension(1:n) :: a,b,c,d
  do i=1,n
     a(i) = b(i) + c(i)*d(i)
  enddo
end subroutine
```

アンロール効果は1.49倍

ソフトウェアパイプライニング効果は1.74倍

ソフトウェアパイプライニング：評価方法

FUJITSU

-Nlst=t指定による翻訳時情報出力

※詳細情報はtrad固有

- **IPC**（Instruction Per Cycle）
  最重要。良いスケジューリング結果かどうかの目安。4.0に近い値で大きい値のほうが良い
- **ITR**（ITeRation）
  ソフトウェアパイプライニングのカーネルを通るために必要な1スレッドあたりの回転数
- **MVE**（Modulo Variable Extension）
  ソフトウェアパイプライニングのカーネルに何回転分が入っているかを示す
- **POL**（Policy）
  ソフトウェアパイプライニングの採用アルゴリズムを示す。詳細後述。
  S:small : IMS（Iterated Modulo Scheduling）
  L:large : SMS（Swing Modulo Scheduling）

```
1               subroutine sub(a,b,c,d,n)
                real(kind=8),dimension(1:n) :: a,b,c,d
                < Loop-information Start >>>
                <  [OPTIMIZATION]
                <    SIMD(VL: 8)
                <    SOFTWARE PIPELINING(IPC: 3.20, ITR: 144, MVE: 4, POL: S)
                <    PREFETCH(HARD) Expected by compiler :
                <      a, d, c, b
                   <<< Loop-information  End >>>
                  do i=1,n
                    a(i) = b(i) + c(i)*d(i)
                  enddo
                nd subroutine
```

ソフトウェアパイプライニング：チューニング機能

FUJITSU

- trad固有のチューニング機能を紹介
  - -Kswp_policy={auto|small|large}
  - -Kswp_{weak|strong}
  - -Kswp_{freg|ireg|preg}_rate=N

ソフトウェアパイプライニング：アルゴリズム選択

FUJITSU

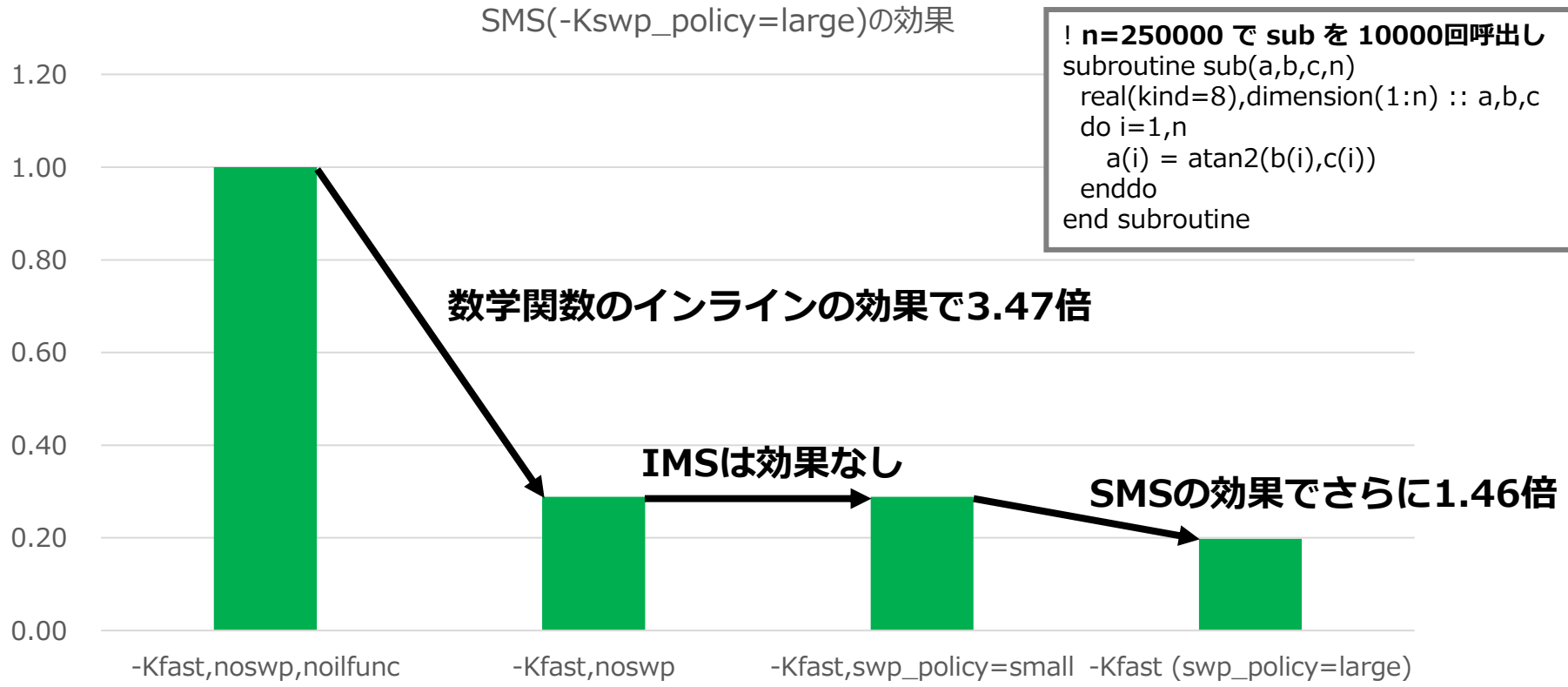
○-Kswp_policy={auto|small|large}　※OCL/pragmaもあり

翻訳オプション	マニュアル記載	採用アルゴリズム	詳細
-Kswp_policy=small	小さなループ(例えば、必要レジスタ数が少ないループ)に適した命令スケジューリングアルゴリズムを使用します。	IMS (Iterative Modulo Scheduling)	京でも採用している従来アルゴリズムをA64FX向けに対応。レジスタ不足とならない(SPILL/FILLが出ない）範囲で解を求めます ※clangのソフトウェアパイプライニングは-ffj-swp指定で動作し、実装はIMSのみ。
-Kswp_policy=large	大きなループ(例えば、必要レジスタ数が多いループ)に適した命令スケジューリングアルゴリズムを使用します。	SMS (Swing Modulo Scheduling)	A64FX向けに改良した新アルゴリズム レジスタプレッシャ削減とクリティカルパス優先のヒューリスティックを持つモジュロスケジューラ。特徴として、FILLロードはリソースが空いているなら許容します。 **レジスタプレッシャが高いループでより良い命令スケジューリングが期待できます。**
-Kswp_policy=auto	ループ毎に命令スケジューリングアルゴリズムを自動で選択します。	IMSとSMSの自動選択	デフォルト。基本、IMSとSMSの自動選択だが、ヒューリスティックにIMS効果が望めそうと判断したケースはIMSを採用

SMSの効果

FUJITSU

○atan2を含むループで評価（自動でSMSが選択されるケース）




```
! n=250000 で sub を 10000回呼出し
subroutine sub(a,b,c,n)
  real(kind=8),dimension(1:n) :: a,b,c
  do i=1,n
     a(i) = atan2(b(i),c(i))
  enddo
end subroutine
```

ソフトウェアパイプライニング：チューニング機能（続）

FUJITSU

○ ソフトウェアパイプライニングルートを通すためのチューニングオプション

翻訳オプション（以下OCL/pragma有）	最適化指示子	マニュアル記載	詳細（ワンポイント）
-Kswp_weak	○	ソフトウェアパイプライニングを調整し、ループ内の実行文の重なりを小さくすることを指示します。	**ITRを小さくしたい** IPCを下げて、ソフトウェアパイプライニングのカーネルに入る回転数(ITR)を小さくする。
-Kswp_strong	○	ソフトウェアパイプライニングの条件を緩和し、ソフトウェアパイプライニングを促進すうことを指示します。	**ソフトウェアパイプライニングの答えが見つからない(jwd8662o-i)** スケジューリングの解が見つからなかった時、もう少し時間をかけて解を探させる。
-Kswp_freg_rate=N 他に -Kswp_ireg_rate=N, -Kswp_preg_rate=N もある	○	ソフトウェアパイプライニングを適用する際、浮動小数点レジスタならびにSVEのSIMDレジスタのN%が使用可能であると指示します。	**レジスタ不足でソフトウェアパイプライニングできない(jwd8665o-i,jwd8666o-i,jwd8673o-i)** 特にデータ依存がないケースで、メモリへの退避復元であるSPILL/FILLが出ても良いからソフトウェアパイプライニングを実行してみる。 （例）浮動小数点レジスタ不足時、レジスタ数が1.5倍あると思ってソフトウェアパイプライニング実行 -Kswp_freg_rate=150

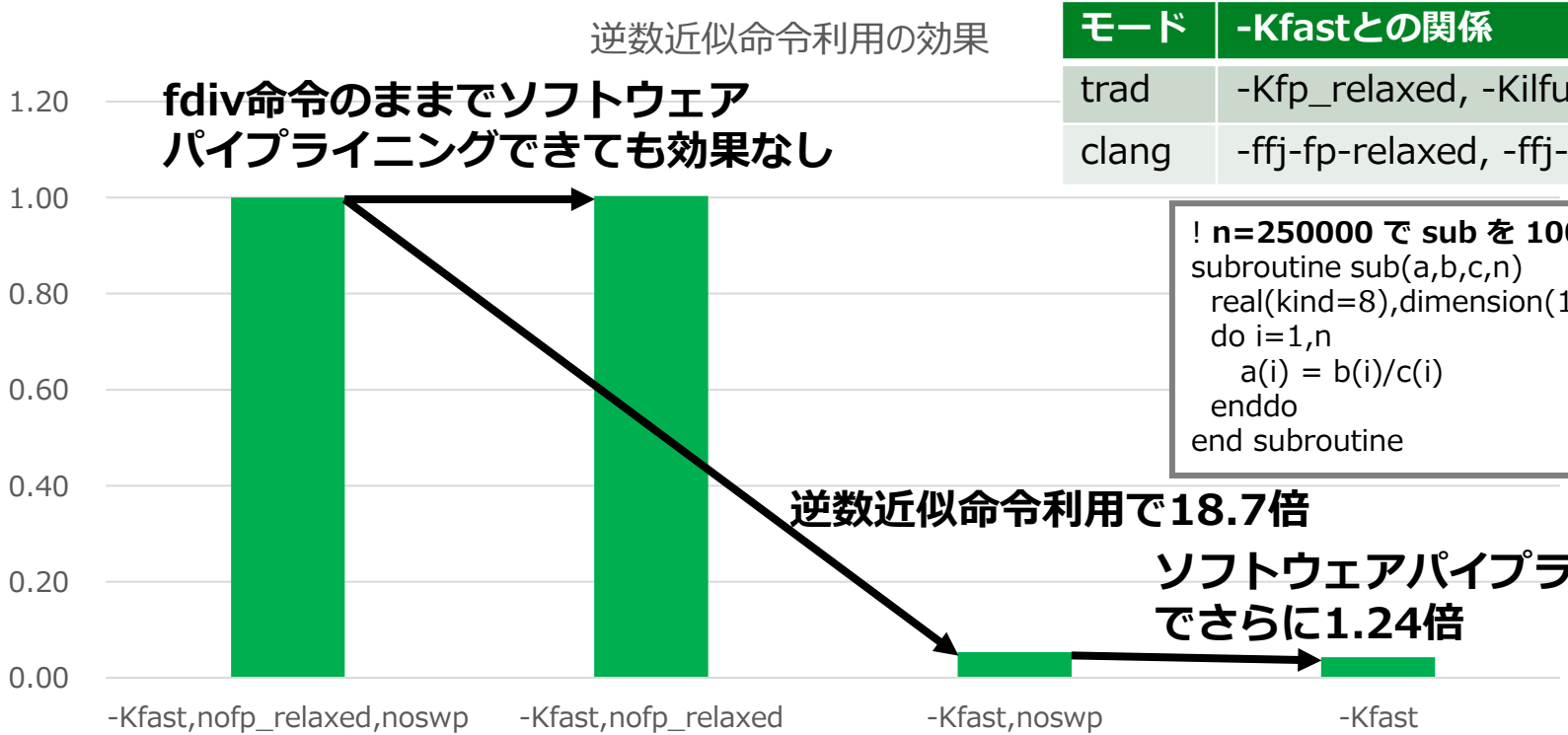
ソフトウェアパイプライニング：促進機能と効果
～逆数近似命令の利用～

FUJITSU

○逆数近似命令の利用(-Kfp_relaxed指定)による性能改善

○三角関数・指数関数補助命令利用などによる数学関数のインライン(-Kilfunc指定)でも似た傾向

モード	-Kfastとの関係
trad	-Kfp_relaxed, -Kilfuncを誘導
clang	-ffj-fp-relaxed, -ffj-ilfuncを誘導




```
! n=250000 で sub を 10000回呼出し
subroutine sub(a,b,c,n)
  real(kind=8),dimension(1:n) :: a,b,c
  do i=1,n
     a(i) = b(i)/c(i)
  enddo
end subroutine
```

ソフトウェアパイプライニングのジレンマ

FUJITSU

- ソフトウェアパイプライニングは万能ではない
  ➡適用されれば必ず性能が上がるというわけではない
  - **ループ回転数がITR以下**の場合は逆効果のケースあり
  - **データ依存のあるループ**はIPCが低く性能向上しないケースあり

```
 <<<    SOFTWARE PIPELINING(IPC: 3.20, ITR: 144, MVE: 4, POL: S)
        :
3     1       2v        do i=1,n    ! 回転数n＜144
4     1       2v          a(i) = b(i) + c(i)*d(i)
5     1       2v        enddo
```

```
 <<<    SOFTWARE PIPELINING(IPC: 0.55, ITR: 6, MVE: 2, POL: S)
       :
3     1       2s        do i=1,n
4     1       2s          a(i) = a(i-1) + b(i)
5     1       2s        enddo
```

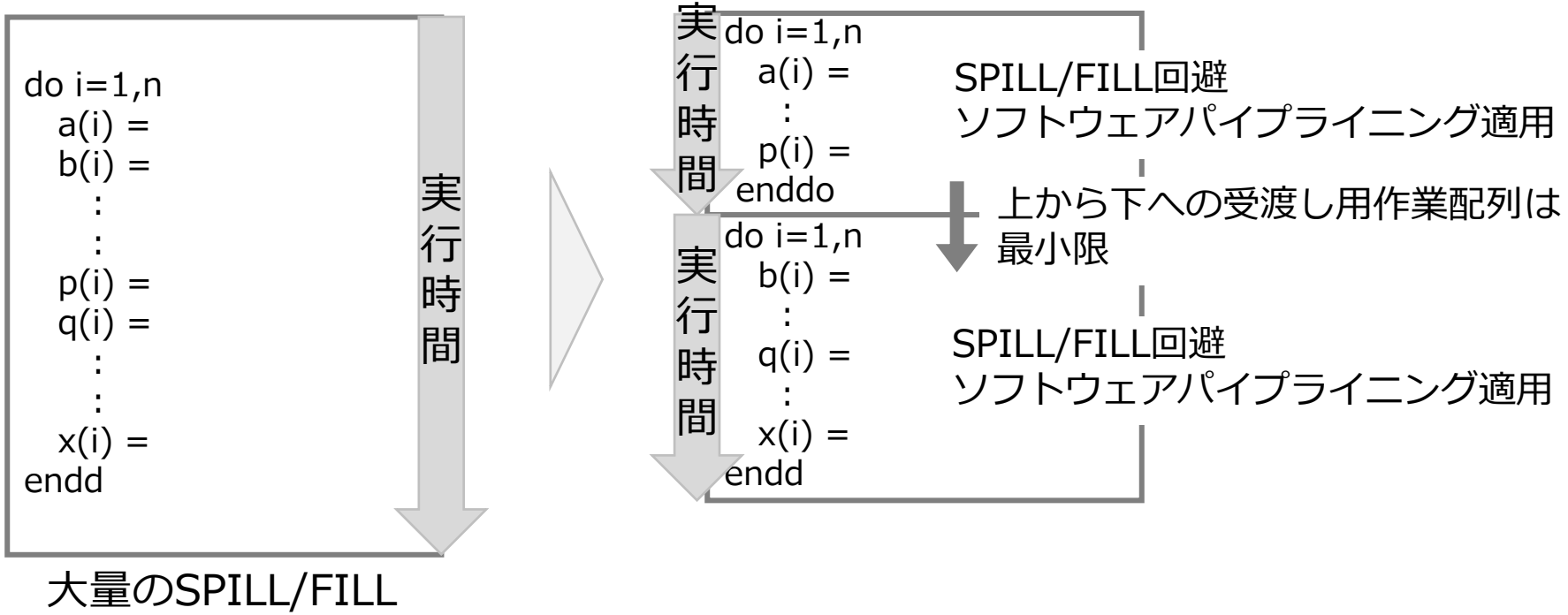
加算のレイテンシ

レジスタ不足の影響対策

FUJITSU

○ループ分割による最適化促進

回転数nで回るループを２つ以上に分け、理想的には分割したループ間の受渡し用作業配列を最小化し、個々のループに対する最適化促進により、性能向上を狙う



ループ分割：２つのアルゴリズム

FUJITSU

○ループ分割による最適化促進（-Kloop_fission,ocl有効時）

```
!OCL LOOP_FISSION_TARGET(CL)
!OCL FISSION_THRESHOLD(n)
do i=1,m
    ～
enddo
```

CL：クラスタリングアルゴリズムでループ分割
分割ループ間で利用する作業配列削減を優先
（default）

```
!OCL LOOP_FISSION_TARGET(LS)
!OCL FISSION_THRESHOLD(n)
do i=1,m
    ～
enddo
```

LS：局所探索アルゴリズムでループ分割
ソフトウェアパイプライニング促進を優先

THRESHOLD(n)
閾値nは1～100でデフォルトは50。小さい値を指定したほうが分割数が増える

キャッシュ活用：ソフトウェアプリフェッチ

FUJITSU

- ソフトウェアプリフェッチ機能の基本動作（出力確認はリスタ機能で）
  - 16ストリームまではハードウェアプリフェッチにお任せ
  - 17ストリーム以上でソフトウェアプリフェッチ動作
  - インダイレクトプリフェッチ、ストライドプリフェッチは個別オプション指定時に動作

```
                <<< Loop-information Start >>>
                    :
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      idx, a
                <<<    PREFETCH(SOFT) : 8
                <<<     INDIRECT : 8
                <<<      b: 8
                <<< Loop-information  End >>>
4     1      2v        do i=1,n
5     1      2v          a(i) = b(idx(i))
6     1      2v        enddo
```

idx,aをハードウェアプリフェッチに任せるのは
プリフェッチのデフォルト動作

bに対するソフトウェアプリフェッチ出力は、
-Kprefetch_indirect オプション指定時の
オプショナル動作

- 次頁ではインダイレクトプリフェッチの活用事例を示す

インダイレクトプリフェッチの活用事例

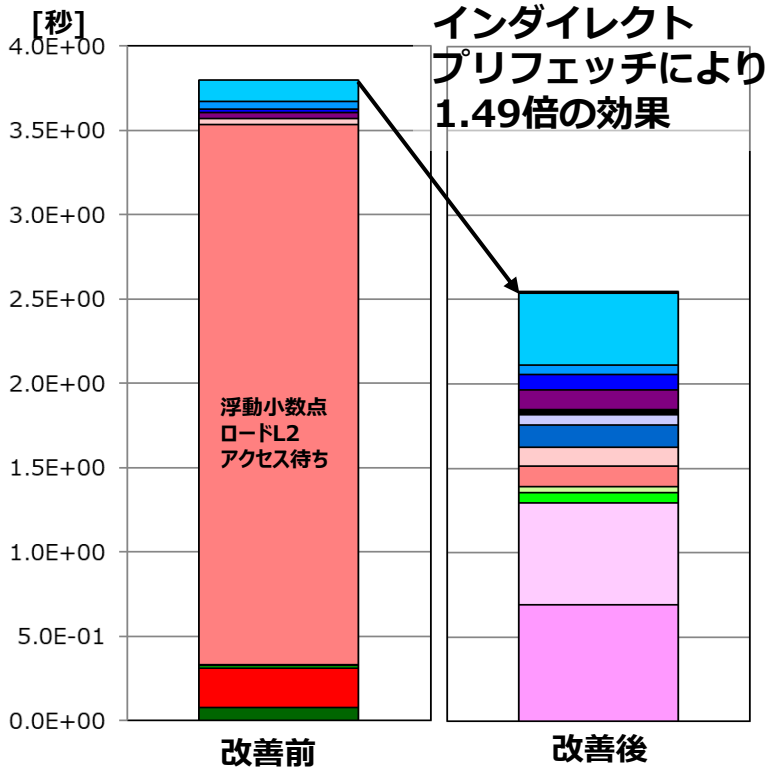
FUJITSU

○ PA情報のL1D,L2のdemand miss率を確認し、Memory throughputの余裕を見て、インダイレクトプリフェッチを活用

```
!ocl prefetch
  do i=1,n
    a(i) = b(d(i)) + scalar * c(e(i))
  enddo
```

Cache	‥	L1D miss	‥	L1D miss demand rate (%) (/L1D miss)	‥	L2 miss	‥	L2 miss demand rate (%) (/L2 miss)	‥
改善前	‥	3.77E+09	‥	**98.23%**	‥	4.29E+08	‥	**54.19%**	‥
改善後	‥	3.82E+09	‥	2.94%	‥	1.77E+09	‥	2.02%	‥

	Memory throughput (GB/s)
改善前	**32.67**
改善後	183.71

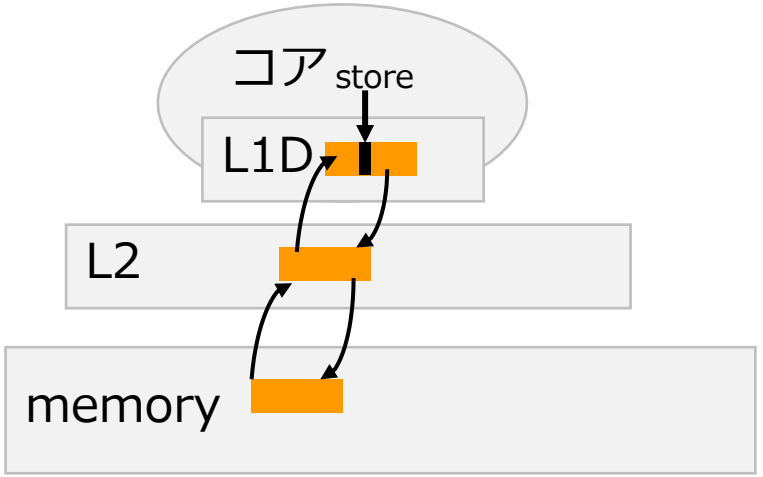


メモリバンド幅ネックの対策

FUJITSU

- メモリバンド幅ネックのプログラムの傾向
  - 命令スケジューリングは効果無し
  - 最適化で多少命令を削っても効果無し、最適化がいまいちで多少命令が増えても変化無し

- 性能向上のためには、メモリアクセス量を減らすしかない
- キャッシュマシンは、読み書きをキャッシュライン単位で行うため、ストアでもREADする。しかし、キャッシュライン全体に書き込むことが分かっているなら、そのREADは不要なはず。



この考えから生まれたのがZFILL
データの中身をREADせず、キャッシュ上の位置だけ確定するプリフェチもどき

zfillの活用（１）

FUJITSU

○最適化指示子 zfill (オプションは-Kzfill)

```
subroutine foo(a,b,c,d,n)
  real(kind=8),dimension(1:n) :: a,b,c
  real(kind=8) ::d
!ocl zfill
  do i=1,n
     a(i) = b(i) + c(i)*d
  enddo
end subroutine foo
```

- zfill実装はSIMD化を利用
- SIMD化できないものはzfillできない
- 固定長SIMDのみ
- 参照・定義の配列に重なりがないことが条件

※clangモードは固定長SIMD指定(-msve-vector-bits=512)が必要
最適化指示行は、「#pragma [fj] loop zfill」

○無闇に指定しても性能向上せず、逆効果

zfill活用は、キャッシュライン全体を書き替えることを保証しなければならないため、
キャッシュライン全体を書き替えることを保証するコードを出力する必要がある。
また、ハードウェアプリフェッチに任せず、zfillとL1Dへのソフトウェアプリフェッチを
出力するため、バンド幅ネックでない場合、これらは全く余分な命令。したがって、ＰＡ
情報を見て活用することが重要。

zfillの活用（２）

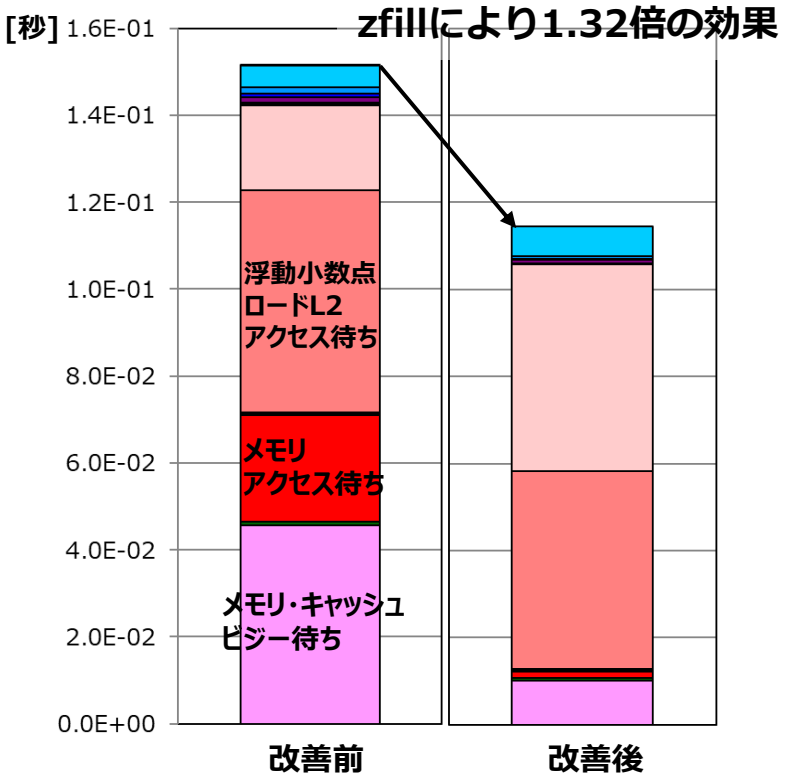
FUJITSU

- 逆効果を避けるため、zfillはPA結果の「Memory Throughput(GB/s)」が大きいアプリかどうかを調べて活用

	Memory Throughput (GB/s)
改善前	**211.59**
改善後	209.96

- 改善効果は「L2 miss」の削減を確認

	‥	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	‥
改善前	‥	9.39E+07	0.24	27.78%	72.21%	0.01%	9.38E+07	‥
改善後	‥	9.39E+07	0.21	16.80%	49.98%	33.22%	6.25E+07	‥



clangモードについて

FUJITSU

- tradモードとclangモードの機能比較
- clangモード優位な機能例
- strict aliasing

tradモードとclangモードの機能比較

FUJITSU

○下表にtradモードとclangモードの主要機能比較を示す

項目		tradモード	clangモード	備考
スレッド並列化	OpenMP	○	○	
	自動	○	×	
SIMD化	可変長	○	○(default)	ポインタ等のデータ依存解析はclangモード優位 SIMD化バリエーションはややtradモード優位 **固定長は-msve-vector-bits=512で動作**
	固定長	○(default)	○	
インライン展開		△	○	C++のインライン展開はclang優位
ソフトウェアプリフェッチ		○	△	clangモードは一部未サポートあり
数学関数のインライン展開		○	○	
ソフトウェアパイプライニング		○	△	clangモードは-ffj-swp指定時に動作、tradモードはチューニング機能が充実
LTO(LinkTimeOptimization)		×	○	
ACLE, fp16		×	○	
zfill		○	○	
セクタキャッシュ		○	×	

- **C++翻訳は、インライン展開優位のclangモードがお勧め**
- **データ依存が明確であれば、SIMD化やソフトウェアパイプライニング性能が安定してるtradモードがお勧め**

clangモード優位な機能例

FUJITSU

- インライン展開
  - 仮想関数テーブル経由コールのインライン
  - ラムダ式のインライン
  - 不要な関数削除（翻訳時間に影響）など
- LTO(Link Time Optimization)
  - -fltoオプションで動作
    - リンク時であれば、実行モジュールを構成するファイルはすべて揃う。そこで、オブジェクトファイル(*.o)の中にソースプログラム相当を埋め込み、リンク時に再翻訳する。
      これにより、**ファイルを跨ぐインライン展開**などの最適化が可能となる。
- データ依存解析(以下は一例)
  - 型種別を利用したデータ依存解析（詳細次頁）

（例）仮想関数テーブル経由コールのインライン

```
// override sample
#include <iostream>
class Calc
{
public: virtual double calc(double a, double b) { return a; }
};
class Add : public Calc
{
public: double calc(double a, double b) { return a+b; }
};
      :
int main(void) {
  Calc *x = new Add;
  // calcをインラインし定数計算
  std::cout << x->calc(2.0,1.0) << std::endl;
}
```

strict aliasing

FUJITSU

- データ型種別を利用したデータ依存の解消
  下記プログラムにおいて、
  - 文法通りだと、a[i]への代入は、b[m]の領域を破壊するかもしれない
    ➡b[m]の不変式不可、SIMD化も不可
  - しかし、そんなコードは通常書かない

```
void foo(double *a, long *b, int n, int m) {
  for (int i=0; i<n; ++i) {
    a[i] = a[i] + (double)b[m];
  }
}
```

- trad　**-Kstrict_aliasing**：個別オプション指定時動作
- clang **-fstrict_aliasing** ：-O2以上で動作

（注）上例(C言語)は**文法のrestrict指定を推奨**。
　　また、この例では、-Krestp=argでも最適化可能

b[m]のloadを最内ループの外に追い出し、SIMD化

プログラムチューニングによる性能改善

FUJITSU

- データ依存の解消
- コーディングの工夫

データ依存の解消（C/C++）

FUJITSU

- 自動SIMD化(or 自動並列化)できない原因の多くが、データ依存解析の問題
- コンパイラがデータ依存を見抜けない場合、restrict指定、最適化指示行、OpenMPによるデータ依存の解消が必要

```
void foo(double *a, double *b, int n) {
    for (i=0; i<n; ++i) {
        a[i] = a[i] + b[i];
    }
}
```

```
void foo(double * restrict a, double * restrict b, int n) {
    for (i=0; i<n; ++i) {
        a[i] = b[i] + c[i]*d[i];
    }
}
```

```
void foo(double *a, double *b, int n) {
#pragma loop novrec
// #pragma clang loop vectorize(assume_safety)
    for (i=0; i<n; ++i) {
        a[i] = a[i] + b[i];
    }
}
```

```
void foo(double *a, double *b, int n) {
#pragma omp simd
    for (i=0; i<n; ++i) {
        a[i] = a[i] + b[i];
    }
}
```

データ依存の解消（Fortran）

FUJITSU

○ Fortranのデータ依存の解消は、C/C++同様の最適化指示子「!ocl novrec」やOpenMP記述の他に、以下のような文法的対応が可能

```
subroutine foo(a,b,c,n)
  real(kind=8),dimension(:),pointer :: a,b,c
  do i=1,n
     a(i) = b(i) + c(i)
  enddo
end subroutine foo
```

```
subroutine foo(a,b,c,n)
  real(kind=8),dimension(:),pointer,contiguous :: a,b,c
  do concurrent(i=1:n)
     a(i) = b(i) + c(i)
  enddo
end subroutine fo
```

do concurrentがお勧めなのですが…

※コンパイラに障害があり、最適化が促進されていない。
今年度末～来年度頭で提供するパッチで対応予定

コーディングの工夫

FUJITSU

- C/C++の２次元以上の配列をFortranと同じように連続領域として確保し、restrict指定で最適化促進

```
void boo(int n, double **a, double **b) {
 for (int j=0; j<n; ++j) {
    for (int i=0; i<n; ++i) {
      a[j][i] = a[j][i] + b[j][i];
    }
  }
}

void foo(int n) {
  double **a = (double **)malloc(sizeof(double *)*n);
  double **b = (double **)malloc(sizeof(double *)*n);
  for (int i=0; i<n; ++i) {
    a[i] = (double *)malloc(sizeof(double)*n);
    b[i] = (double *)malloc(sizeof(double)*n);
  }
  calc_a(n,a);
  calc_b(n,b);
  boo(n, a, b);
}
```

a[j]のload + a[j][i]へのstore

nxnの配列生成

高速化

```
void boo(int n, double (* restrict a)[n], double (* restrict b)[n]) {
 for (int j=0; j<n; ++j) {
    for (int i=0; i<n; ++i) {
      a[j][i] = a[j][i] + b[j][i];
    }
  }
}

void foo(int n) {
  double (*a)[n] = (double (*)[n])malloc(n*sizeof(double[n]));
  double (*b)[n] = (double (*)[n])malloc(n*sizeof(double[n]));

  calc_a(n,a);
  calc_b(n,b);
  boo(n, a, b);
}
```

a[j][i]へのstore

nxnの配列生成

チューニング事例

FUJITSU

- ABINIT_MPアプリのチューニング事例

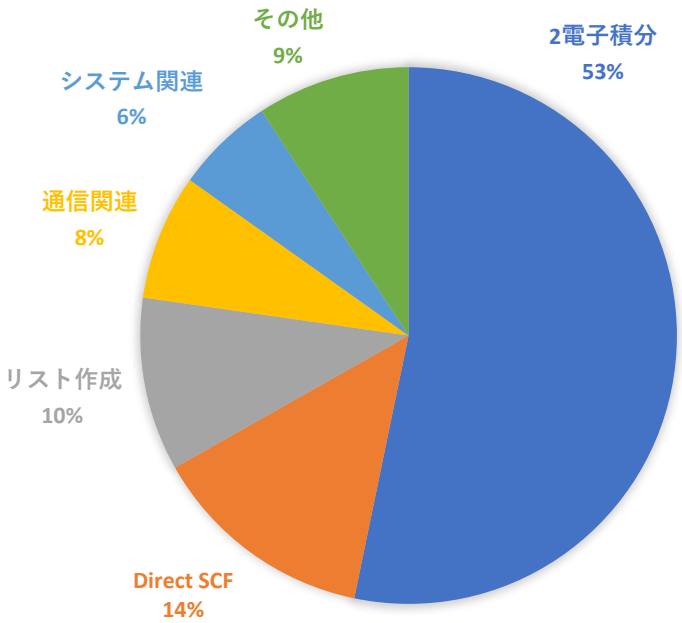
チューニング事例

FUJITSU

- 対象アプリケーション
  - ABINIT-MP
    - フラグメント分子軌道（FMO）計算を高速に行えるソフトウェア
    - 4体フラグメント展開（FMO4）による2次摂動計算も可能
    - MPIおよびOpenMPにより超大規模計算にも対応
    - http://www.cenav.org/abinit-mp-open_ver-2-rev-4/
- 測定環境
  - 計算機: FX1000 CPU:A64FX(2.0GHz x 48cores)
  - コンパイラ: Fujitsu Fortran compiler 4.5.0 tcsds-1.2.31
  - MPI: Fujitsu MPI

コスト分布

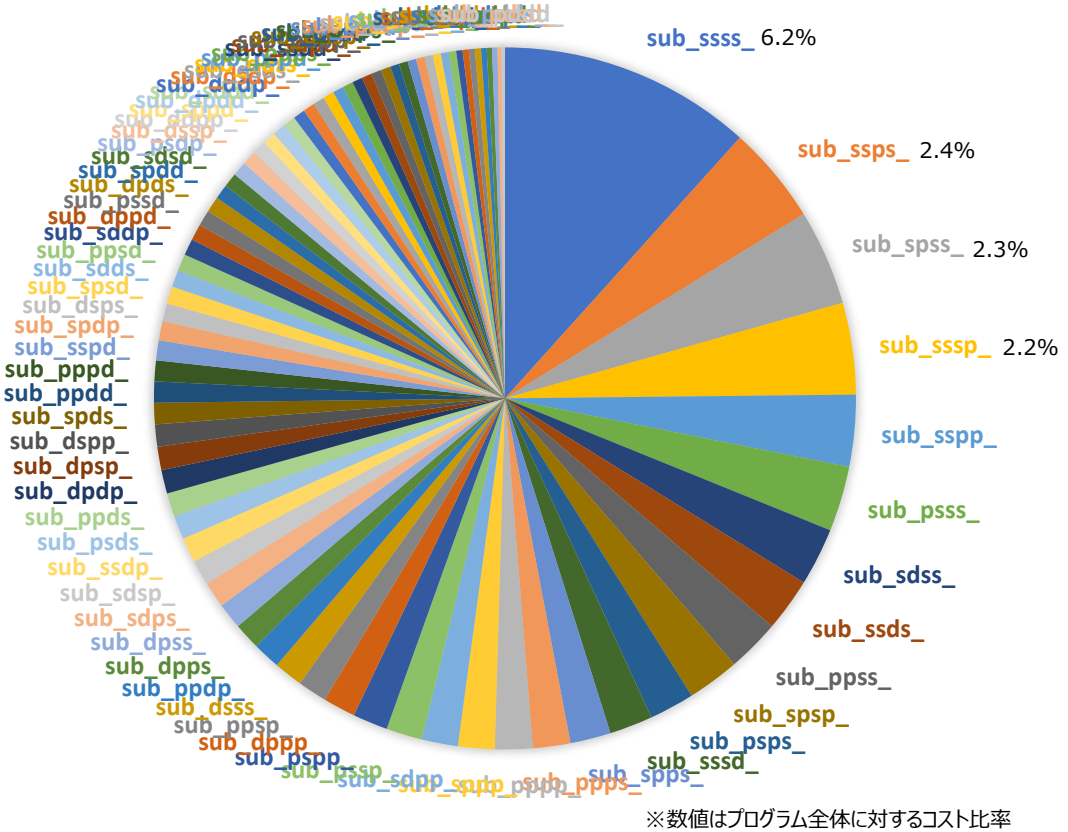
- プログラム全体のコスト分布
  - 基本プロファイラによるプロセス0番、スレッド0番のコスト分布
  - 2電子積分処理が全体の約半分を占める
    ただし、81種の処理の総和であるため、
    1種あたりのコストは1%前後と非常に小さい
  - 通信に関連したコストは8%程度と小さい



2電子積分：81種のサブルーチン(sub_*)のコスト総和
Direct SCF：サブルーチンdirect_scf_gmatのコスト
リスト作成：3種のサブルーチン(get_tei_rs_fix, get_tei_pq_fix, get_ixijcs_to_proc_pqfix)のコスト総和
通信関連：通信に関連した処理(putofu_*, opal_*, mca_*)のコスト総和
システム関連：ライブラリやOSなどに関連した処理のコスト総和
その他：上記以外の処理の総和

【ご参考】コスト分布

- 81種の2電子積分処理コスト
  - プログラム全体に対するコスト比率が2%以上の処理は上位4個
  - 上位4個のコストを合わせるとプログラム全体に対して13.2%



※数値はプログラム全体に対するコスト比率

2電子積分処理のソースコード分析

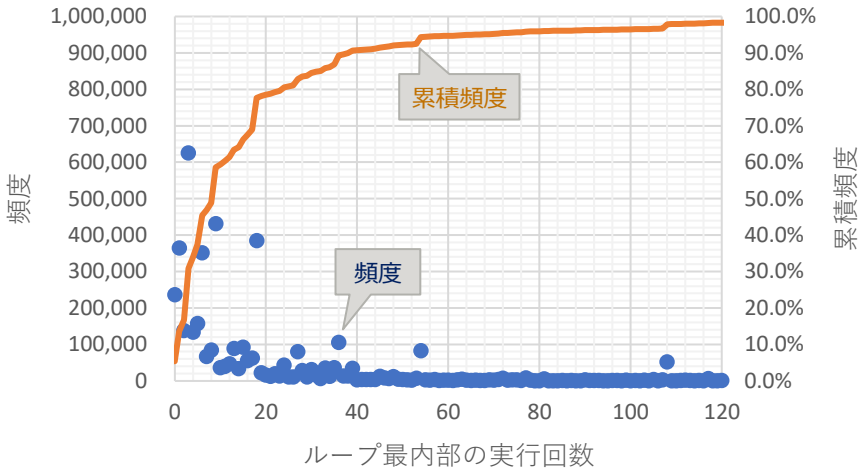
FUJITSU

- ループ構造分析
  - 81種の処理全てで下記の2重ループ構造

```
do npq=1,ngij
  if (abs(dkabm(npq)) > tv) then
    do nrs=1,ngkl
      if (abs(dkabm(npq)*dkcdm(nrs)) > tv) then
                [ループ最内部：2電子積分計算]
      end if
    end do
  end if
end do
```

  - sub_ssssでは（プロセス0番、スレッド0番）
    - sub_ssssは4,425,459回実行される
    - ngij、ngklは36
      ループ最内部は最大でも1296回(36*36)の実行
      ただし、条件判定により1296回よりも減少する

■ sub_ssssのループ最内部の実行回数



- 実行回数40回以下が全実行の90%を占める
- 仮に、二重ループを一重化してソフトウェアパイプライニングなどの最適化を促進させようとしても、その効果は限定的と推測される

sub_ssssの最適化状況分析（1/2）

FUJITSU

- 最適化情報を確認
  - 最適化情報

```
39                    sint(1)      = 0.0_8
40
41    1               do npq=1,ngij
42    2                 if (abs(dkabm(npq)) > tv) then
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      dkcdm
                  <<< Loop-information  End >>>
43    3     s           do nrs=1,ngkl
44    4     m             if (abs(dkabm(npq)*dkcdm(nrs)) > tv) then
45    4     s               ze  = 1.0_8/(zetam(npq)+etam(nrs))
46    4     s               a0  = dkabm(npq)*dkcdm(nrs)*sqrt(ze)
47    4     s               rz  = etam(nrs)*ze
48    4     s               rho = zetam(npq)*rz
49    4     s               tt  = ((qm(1,nrs)-pm(1,npq))*(qm(1,nrs)-pm(1,npq)) &
50    4                         +(qm(2,nrs)-pm(2,npq))*(qm(2,nrs)-pm(2,npq)) &
51    4                         +(qm(3,nrs)-pm(3,npq))*(qm(3,nrs)-pm(3,npq)))*rho
52    4
53    5     s               if (tt <= 36.0_8) then ! Tf = 2*m+36 (for the case of m=0)
54    5     s                 ts    = 0.5_8+tt*fmt_inv_step_size
55    5     s                 delta = ts*fmt_step_size-tt
```

SIMD化されていない

■ 最適化メッセージ（一部抜粋）

- jwd6229s-i "integral/sub_ssss.F90", line 43: IF文が存在するため、このDOループのSIMD化を抑止しました。
- jwd8670o-i "integral/sub_ssss.F90", line 43: ループ内に分岐命令があるため、ソフトウェアパイプライニングを適用できません。
- jwd8209o-i "integral/sub_ssss.F90", line 49: 多項式の演算順序を変更しました。
- jwd8209o-i "integral/sub_ssss.F90", line 56: 多項式の演算順序を変更しました。

コンパイラがSIMD化を抑止している

⇒ OCL指示子から、SIMD化を促進する

sub_ssssの最適化状況分析（2/2）

FUJITSU

- OCL指示子でSIMD化を促進
  - チューニング後の最適化情報

```
39                    sint(1)      = 0.0_8

43                  !ocl noswp
44                  !ocl eval_concurrent
45                  !ocl SIMD
46    1               do npq=1,ngij
47    2                 if (abs(dkabm(npq)) > tv) then
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    SIMD(VL: 8)
                    <<<    PREFETCH(HARD) Expected by compiler :
                    <<<      dkcdm, etam, qm
                    <<< Loop-information  End >>>
48    3     v           do nrs=1,ngkl
49    4     v             if (abs(dkabm(npq)*dkcdm(nrs)) > tv) then
50    4     v               ze  = 1.0_8/(zetam(npq)+etam(nrs))
51    4     v               a0  = dkabm(npq)*dkcdm(nrs)*sqrt(ze)
52    4     v               rz  = etam(nrs)*ze
53    4     v               rho = zetam(npq)*rz
54    4     v               tt  = ((qm(1,nrs)-pm(1,npq))*(qm(1,nrs)-pm(1,npq)) &
55    4                         +(qm(2,nrs)-pm(2,npq))*(qm(2,nrs)-pm(2,npq)) &
56    4                         +(qm(3,nrs)-pm(3,npq))*(qm(3,nrs)-pm(3,npq)))*rho
57    4
58    5     v               if (tt <= 36.0_8) then ! Tf = 2*m+36 (for the case of m=0)
59    5     v                 ts    = 0.5_8+tt*fmt_inv_step_size
60    5     v                 delta = ts*fmt_step_size-tt
```

SIMD化された

OCL指示子	説明
noswp	ソフトウェアパイプライニング機能を無効に
eval_concurrent	tree-height-reduction最適化において 演算の並列性を優先することを指示
SIMD	SIMD 化を有効に

✓ **noswp**を指定している理由

- nglは36で固定
- swpは回転数が40以上のループにしか適用されない

■ チューニング前後の性能比較

実行時間[s]		性能比 対asis
asis	tune	tune
9.17	4.43	**2.07**

sub_sspsの最適化状況分析（1/4）

FUJITSU

○ sub_ssssと同じチューニングを試みる

■ 最適化情報

```
40          f       sint(1:3)      = 0.0_8
41
42    1              do npq=1,ngij
43    2                if (abs(dkabm(npq)) > tv) then
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      dkcdm
                  <<< Loop-information  End >>>
44    3     s            do nrs=1,ngkl
45    4     m              if (abs(dkabm(npq)*dkcdm(nrs)) > tv) then
 …
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    FULL UNROLLING
                  <<< Loop-information  End >>>
51    5     fs             do i=1,3
52    5     fs               qc(i) = qm(i,nrs)-c(i)
53    5     fs               pq(i) = qm(i,nrs)-pm(i,npq)
54    5     fs               wq(i) =-re*pq(i)
55    5     fs             end do
```

SIMD化されていない

oclを追加

■ 最適化情報

```
40          f       sint(1:3)      = 0.0_8

44                  !ocl noswp
45                  !ocl eval_concurrent
46                  !ocl SIMD
47    1               do npq=1,ngij
48    2                 if (abs(dkabm(npq)) > tv) then
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      dkcdm
                  <<< Loop-information  End >>>
49    3     s            do nrs=1,ngkl
50    4     v              if (abs(dkabm(npq)*dkcdm(nrs)) > tv) then
…
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    FULL UNROLLING
                  <<< Loop-information  End >>>
56    5     fs             do i=1,3
57    5     fm               qc(i) = qm(i,nrs)-c(i)
58    5     fv               pq(i) = qm(i,nrs)-pm(i,npq)
59    5     fm               wq(i) =-re*pq(i)
60    5     fv             end do
```

配列qc, wqを求める計算がSIMD化されていない

sub_sspsの最適化状況分析（2/4）

FUJITSU

- 最適化情報を観察
  - 最適化メッセージ（一部抜粋）
    - jwd6208s-i "integral/sub_ssps.F90", line 57: 定義引用の順序が分からないため、定義引用順序が逐次実行と変わる可能性があり、このDOループはSIMD化できません。(名前:qc)
    - jwd6208s-i "integral/sub_ssps.F90", line 59: 定義引用の順序が分からないため、定義引用順序が逐次実行と変わる可能性があり、このDOループはSIMD化できません。(名前:wq)
    - jwd6208s-i "integral/sub_ssps.F90", line 65: 定義引用の順序が分からないため、定義引用順序が逐次実行と変わる可能性があり、このDOループはSIMD化できません。(名前:f)
  - 最適化メッセージより、定義引用の順序が分からないことが原因だと判明
    - qc, wq, fは配列であるため、コンパイラが依存関係を認識できなかったと推察

    ⇒ **配列をスカラ変数に変更し、依存関係を明示化する**

sub_sspsの最適化状況分析（3/4）

FUJITSU

○配列qc,wq,fをスカラ変数に変更

※変更した行：赤色

```
51        do i=1,3
52          qc(i) = qm(i,nrs)-c(i)
53          pq(i) = qm(i,nrs)-pm(i,npq)
54          wq(i) =-re*pq(i)
55        end do
56        tt = (pq(1)*pq(1)+pq(2)*pq(2)+pq(3)*pq(3))*rho
57        if (tt <= 38.0_8) then        ! Tf = 2*m+36 (for the case of m=1)
58          ts    = 0.5_8+tt*fmt_inv_step_size
59          delta = ts*fmt_step_size-tt
60          f(0) = ((fmt_table(3,ts)*inv6 *delta &
61               + fmt_table(2,ts)*inv2)*delta &
62               + fmt_table(1,ts))     *delta &
63               + fmt_table(0,ts)
64          f(1) = ((fmt_table(4,ts)*inv6 *delta &
65               + fmt_table(3,ts)*inv2)*delta &
66               + fmt_table(2,ts))     *delta &
67               + fmt_table(1,ts)
68        else
69          t_inv = inv2/tt
70          f(0)  = sqrt(pi_over2*t_inv)
71          f(1)  = t_inv * f(0)
72        end if
…
81        ssss(0:1)=f(0:1)*a0
82        do k=1, 3
83          sint(k) = sint(k)+qc(k)*ssss(0)+wq(k)*ssss(1)
84        end do
```

オリジナルコード

```
qc1 = qm(1,nrs)-c(1)
qc2 = qm(2,nrs)-c(2)
qc3 = qm(3,nrs)-c(3)
pq1 = qm(1,nrs)-pm(1,npq)
pq2 = qm(2,nrs)-pm(2,npq)
pq3 = qm(3,nrs)-pm(3,npq)
wq1 =-re*pq1
wq2 =-re*pq2
wq3 =-re*pq3
tt = (pq1*pq1+pq2*pq2+pq3*pq3)*rho
```

```
f0 = ((fmt_table(3,ts)*inv6 *delta &
```

```
f1 = ((fmt_table(4,ts)*inv6 *delta &
```

```
f0  = sqrt(pi_over2*t_inv)
f1  = t_inv * f0
```

```
ssss0=f0*a0
ssss1=f1*a0
sint(1) = sint(1)+qc1*ssss0+wq1*ssss1
sint(2) = sint(2)+qc2*ssss0+wq2*ssss1
sint(3) = sint(3)+qc3*ssss0+wq3*ssss1
```

変更後

sub_sspsの最適化状況分析（4/4）

FUJITSU

- チューニング結果の確認
  - 最適化情報

```
 53   3   v      do nrs=1,ngkl
 54   4   v        if (abs(dkabm(npq)*dkcdm(nrs)) > tv) then
 55   4   v          ze  = 1.0_8/(zetam(npq)+etam(nrs))
 56   4   v          a0  = dkabm(npq)*dkcdm(nrs)*sqrt(ze)
 57   4   v          rz  = etam(nrs)*ze
 58   4   v          re  = zetam(npq)*ze
 59   4   v          rho = zetam(npq)*rz
 …
 65   4   v            qc1 = qm(1,nrs)-c(1)
 66   4   v            qc2 = qm(2,nrs)-c(2)
 67   4   v            qc3 = qm(3,nrs)-c(3)
 68   4   v            pq1 = qm(1,nrs)-pm(1,npq)
 69   4   v            pq2 = qm(2,nrs)-pm(2,npq)
 70   4   v            pq3 = qm(3,nrs)-pm(3,npq)
 71   4   v            wq1 =-re*pq1
 72   4   v            wq2 =-re*pq2
 73   4   v            wq3 =-re*pq3
 …
 75   4   v          tt = (pq1*pq1+pq2*pq2+pq3*pq3)*rho
 …
 80   5   v            f0 = ((fmt_table(3,ts)*inv6 *delta &
 81   5                   + fmt_table(2,ts)*inv2)*delta &
 82   5                   + fmt_table(1,ts))     *delta &
 83   5                   + fmt_table(0,ts)
 …
```

依存がないことが認識され、SIMD化された

```
 …
 85   5   v          f1 = ((fmt_table(4,ts)*inv6 *delta &
 86   5                 + fmt_table(3,ts)*inv2)*delta &
 87   5                 + fmt_table(2,ts))     *delta &
 88   5                 + fmt_table(1,ts)
 …
 93   5   v          f0  = sqrt(pi_over2*t_inv)
 94   5   v          f1  = t_inv*f0
 …
105   4   v        ssss0=f0*a0
106   4   v        ssss1=f1*a0
 …
110   4   v          sint(1) = sint(1)+qc1*ssss0+wq1*ssss1
111   4   v          sint(2) = sint(2)+qc2*ssss0+wq2*ssss1
112   4   v          sint(3) = sint(3)+qc3*ssss0+wq3*ssss1
 …
114   4   v        end if
115   3   v      end do
```

■ チューニング前後の性能比較

実行時間[s]		性能比 対asis
asis	tune	tune
3.56	2.38	**1.49**

ABINIT-MP チューニングまとめ

FUJITSU

- ソースコードの分析
  - 基本プロファイラにより最も実行時間を要している個所を特定する
  - ソースコード等から処理の特徴を見極め、チューニングの指針を決定する
  - ループ回転数等の条件によってはソフトウェアパイプライニングを諦める
- SIMD化による高速化
  - ソースリストの最適化出力情報からSIMD化の状況を確認する
  - コンパイラによる自動SIMD化が阻害されている場合には、最適化メッセージから原因を特定し、指示行の挿入などの対策を実施する
- 謝辞

本チューニング事例は、JHPCNのjh210036-NAH課題のご協力のもと、
立教大学 望月祐志教授よりABINIT-MPのソースコードをご提供いただき、
SS研「A64FXシステムアプリ性能検討WG」での活動成果を再構成しました。

その他の事例のご紹介

FUJITSU

- GitHubでのA64FX向けアプリケーションの公開
  - https://github.com/fujitsu
  - A64FX向けOSSパッチ

FrontISTR	LAMMPS	MPAS
OpenFOAM	Quantum Espresso	SPECFEM3D

  - AIフレームワーク

tensorflow	pytorch

- HPCAsia2022 Best Paper Awardの受賞
  - 152K-computer-node parallel scalable implicit solver for dynamic nonlinear earthquake simulation
  - https://sighpc.ipsj.or.jp/HPCAsia2022/index.html#awards
  - 「富岳」のコデザイン活動を通じて、有限要素法ソルバに対する単体性能および並列性能チューニングで貢献

Thank you