第５回計算科学技術特論B(2026)
スレッド並列＋コア単体性能最適化詳細(２)

株式会社 waveZ
熊畑清

2026年5月14日（木）13:00－14:30

主催：高度情報科学技術研究機構(RIST)
次世代HPC・AI研究開発支援センター(HAIRDESC)
共催：東京大学物性研究所
後援：理化学研究所計算科学研究センター、
計算物質科学人材育成コンソーシアム(PCoMS)

計算科学特論B 第5回

スレッド並列＋コア単体性能最適化詳細(2)
アプリケーションの性能最適化の実例

株式会社waveZ
熊畑 清

2026.5.14(木)

概要

waveZ

- 4月9日 スーパーコンピュータとアプリケーションの性能
- 4月16日 アプリケーションの性能最適化1(高並列性能最適化)
- 4月23日 スレッド並列＋コア単体性能最適化概要
- 5月7日 スレッド並列＋コア単体性能最適化詳細(1)
- 5月14日 スレッド並列＋コア単体性能最適化詳細(2)

実アプリ

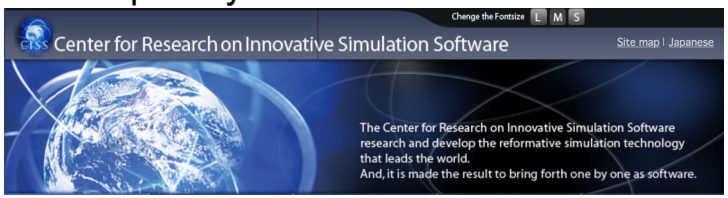
キャッシュ利用効率の改善
スレッド並列化の方法の工夫

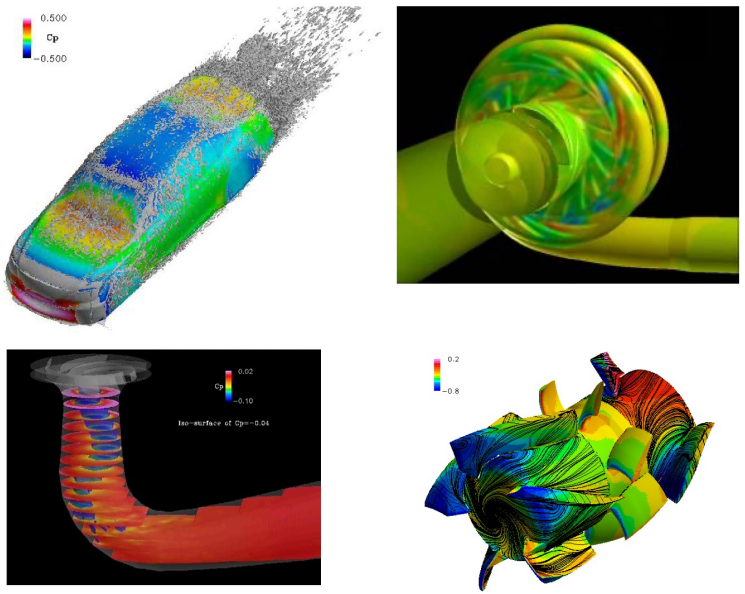
実アプリ) 流体解析ソルバー

waveZ

- FrontFlow/blue (FFB) は非圧縮性流れシミュレーションコード
- 東京大学生産技術研究所によって開発され、パソコンから各種スパコンで利用可能

Developed by



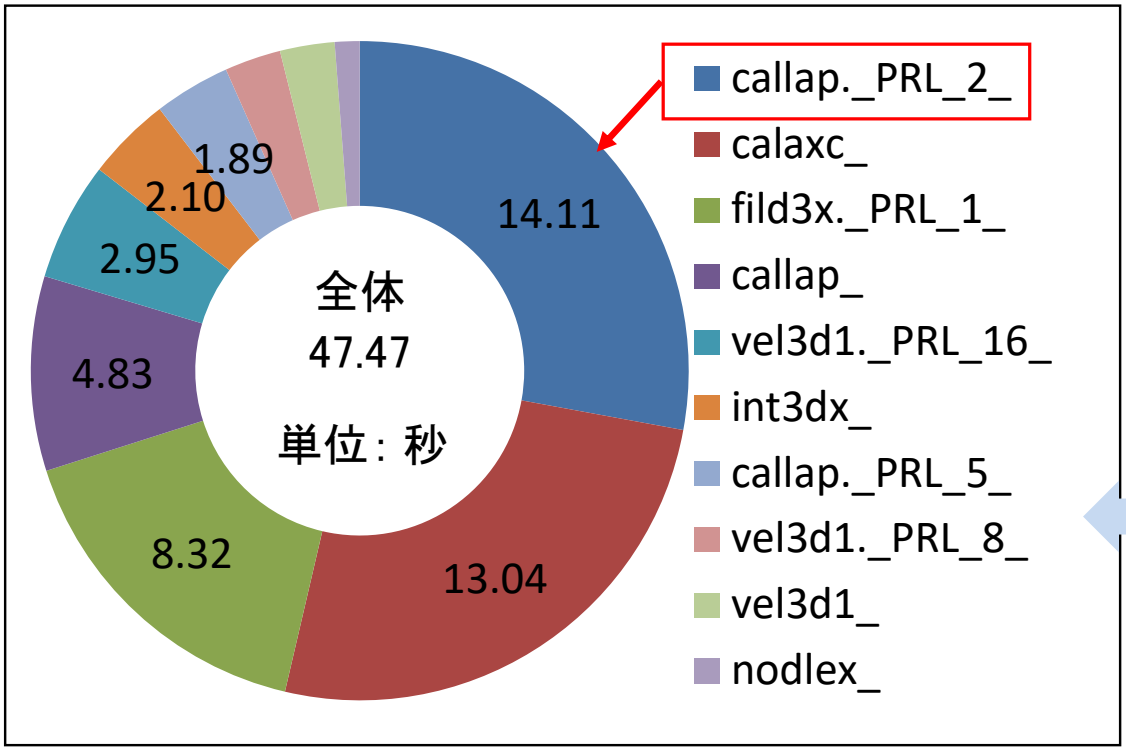


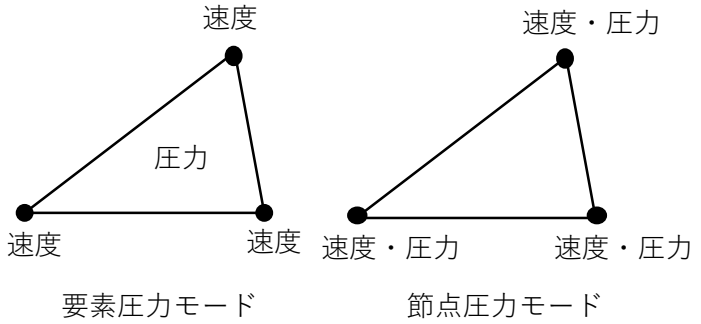
- 有限要素法・分離解法・LES(Large Eddy Simulation)に基づく
- 超並列スーパーコンピュータで高い並列性能を持っている
- 産業界で重要なアプリで、京や富岳開発プロジェクトにおける重要アプリとして選定
- 数万プロセスのオーダーで高い並列性能
- 単体性能チューニングを実施

チューニング対象カーネル

waveZ

要素圧力モードでの処理時間内訳，上位10件





- 10万個の4面体要素＋α
- 14.11s(30%)が4面体要素の勾配計算
- 13.04s(27%)が疎行列ベクトル積
- 8.32s（17%)が4面体要素の発散計算
- Callap._PRL_2_のチューニングを実施

チューニング対象カーネル

waveZ

```
DO ICOLOR=1,NCOLOR(1)
   IES=LLOOP(ICOLOR  ,1)+1
   IEE=LLOOP(ICOLOR+1,1)
   DO IE=IES,IEE・・・・・・・ここで並列
      IP1=NODE(1,IE)
      IP2=NODE(2,IE)
      IP3=NODE(3,IE)
      IP4=NODE(4,IE)
      SWRK=S(IE)
      FX(IP1)=FX(IP1)-SWRK*DNX(1,IE)
      FX(IP2)=FX(IP2)-SWRK*DNX(2,IE)
      FX(IP3)=FX(IP3)-SWRK*DNX(3,IE)
      FX(IP4)=FX(IP4)-SWRK*DNX(4,IE)

      FY(IP1)=FY(IP1)-SWRK*DNY(1,IE)
      FY(IP2)=FY(IP2)-SWRK*DNY(2,IE)
      FY(IP3)=FY(IP3)-SWRK*DNY(3,IE)
      FY(IP4)=FY(IP4)-SWRK*DNY(4,IE)

      FZ(IP1)=FZ(IP1)-SWRK*DNZ(1,IE)
      FZ(IP2)=FZ(IP2)-SWRK*DNZ(2,IE)
      FZ(IP3)=FZ(IP3)-SWRK*DNZ(3,IE)
      FZ(IP4)=FZ(IP4)-SWRK*DNZ(4,IE)
   ENDDO
ENDDO
```

4角形要素について∇pの有限要素近似を計算するループ

$$\nabla p = \frac{\partial p}{\partial x_i} \cong \sum_{j=1} \frac{\partial N_j}{\partial x_i} p_e$$

型	配列名とサイズ	内容
INTEGER*4	NODE(9,NE)	節点リスト
REAL*4	S(NE)	圧力
REAL*4	DNX(9,NE), DNY(9,NE), DNZ(9,NE)	形状関数の導関数
REAL*4	FX(NP), FY(NP), FZ(NP)	圧力勾配ベクトル

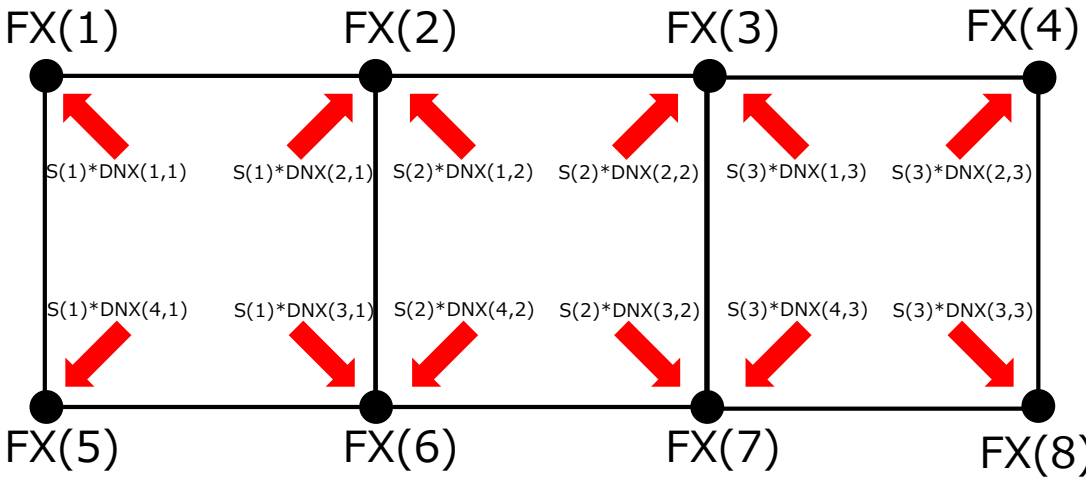
チューニング対象カーネル

waveZ

```
DO ICOLOR=1,NCOLOR(1)
   IES=LLOOP(ICOLOR  ,1)+1
   IEE=LLOOP(ICOLOR+1,1)
   DO IE=IES,IEE・・・・・・・ここで並列
      IP1=NODE(1,IE)
      IP2=NODE(2,IE)
      IP3=NODE(3,IE)
      IP4=NODE(4,IE)
      SWRK=S(IE)
      FX(IP1)=FX(IP1)-SWRK*DNX(1,IE)
      FX(IP2)=FX(IP2)-SWRK*DNX(2,IE)
      FX(IP3)=FX(IP3)-SWRK*DNX(3,IE)
      FX(IP4)=FX(IP4)-SWRK*DNX(4,IE)

      FY(IP1)=FY(IP1)-SWRK*DNY(1,IE)
      FY(IP2)=FY(IP2)-SWRK*DNY(2,IE)
      FY(IP3)=FY(IP3)-SWRK*DNY(3,IE)
      FY(IP4)=FY(IP4)-SWRK*DNY(4,IE)

      FZ(IP1)=FZ(IP1)-SWRK*DNZ(1,IE)
      FZ(IP2)=FZ(IP2)-SWRK*DNZ(2,IE)
      FZ(IP3)=FZ(IP3)-SWRK*DNZ(3,IE)
      FZ(IP4)=FZ(IP4)-SWRK*DNZ(4,IE)
   ENDDO
ENDDO
```

4角形要素について∇pの有限要素近似を計算するループ

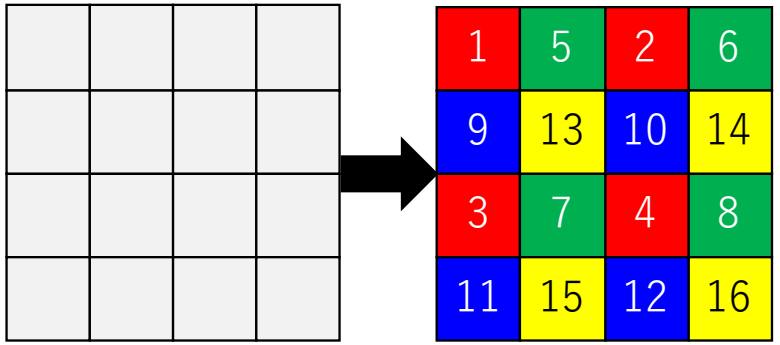
$$\nabla p = \frac{\partial p}{\partial x_i} \cong \sum_{j=1} \frac{\partial N_j}{\partial x_i} p_e$$



チューニング対象カーネル

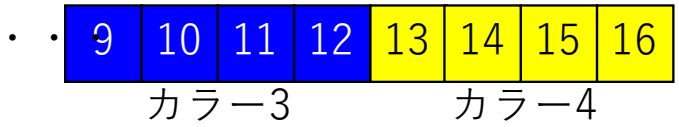
```
DO ICOLOR=1,NCOLOR(1)
   IES=LLOOP(ICOLOR  ,1)+1
   IEE=LLOOP(ICOLOR+1,1)
   DO IE=IES,IEE・・・・・・・ここで並列
      IP1=NODE(1,IE)
      IP2=NODE(2,IE)
      IP3=NODE(3,IE)
      IP4=NODE(4,IE)
      SWRK=S(IE)
      FX(IP1)=FX(IP1)-SWRK*DNX(1,IE)
      FX(IP2)=FX(IP2)-SWRK*DNX(2,IE)
      FX(IP3)=FX(IP3)-SWRK*DNX(3,IE)
      FX(IP4)=FX(IP4)-SWRK*DNX(4,IE)

      FY(IP1)=FY(IP1)-SWRK*DNY(1,IE)
      FY(IP2)=FY(IP2)-SWRK*DNY(2,IE)
      FY(IP3)=FY(IP3)-SWRK*DNY(3,IE)
      FY(IP4)=FY(IP4)-SWRK*DNY(4,IE)

      FZ(IP1)=FZ(IP1)-SWRK*DNZ(1,IE)
      FZ(IP2)=FZ(IP2)-SWRK*DNZ(2,IE)
      FZ(IP3)=FZ(IP3)-SWRK*DNZ(3,IE)
      FZ(IP4)=FZ(IP4)-SWRK*DNZ(4,IE)
   ENDDO
ENDDO
```

カラーリングによりデータ依存が生ずる
要素を別のグループ(カラー)に分ける



要素(メッシュ)

配列 LLOOP/NODE/DNX～Zメモリ上配置



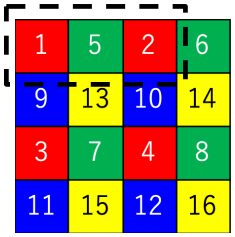
性能評価　要求バイト数の算出

waveZ

配列FX,FY,FZ

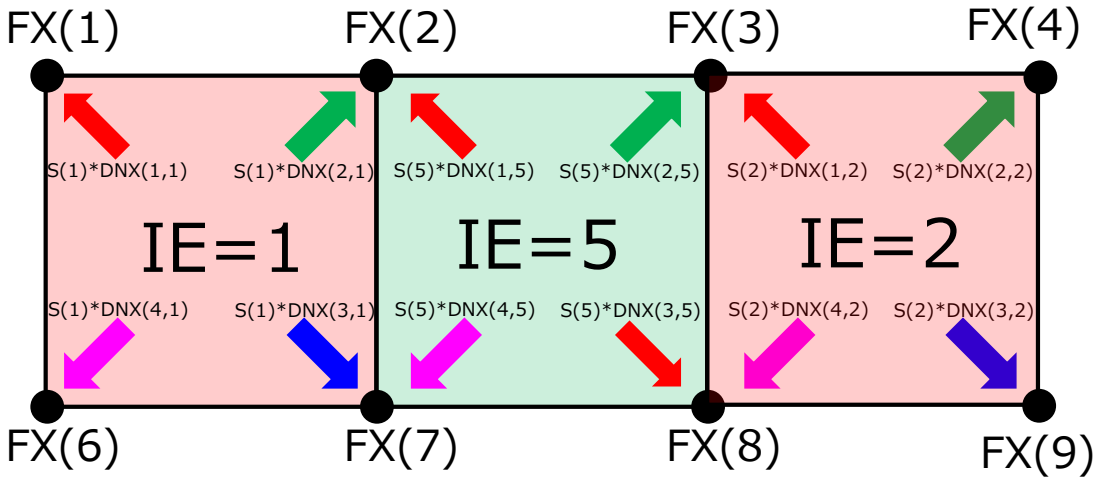
- 配列NODEの値に依存するリストアクセス故にシーケンシャルではない
- 再利用性がある
- リストによって変化してしまい一概には算出できない

例としてこの部分




```
DO  IE=IES,  IEE
    IP1=NODE(1,IE)
    IP2=NODE(2,IE)
    IP3=NODE(3,IE)
    IP4=NODE(4,IE)

    FX(IP1)=FX(IP1)-SWRK*DNX(1,IE)
    FX(IP2)=FX(IP2)-SWRK*DNX(2,IE)
    FX(IP3)=FX(IP3)-SWRK*DNX(3,IE)
    FX(IP4)=FX(IP4)-SWRK*DNX(4,IE)
ENDDO
```

リストアクセス

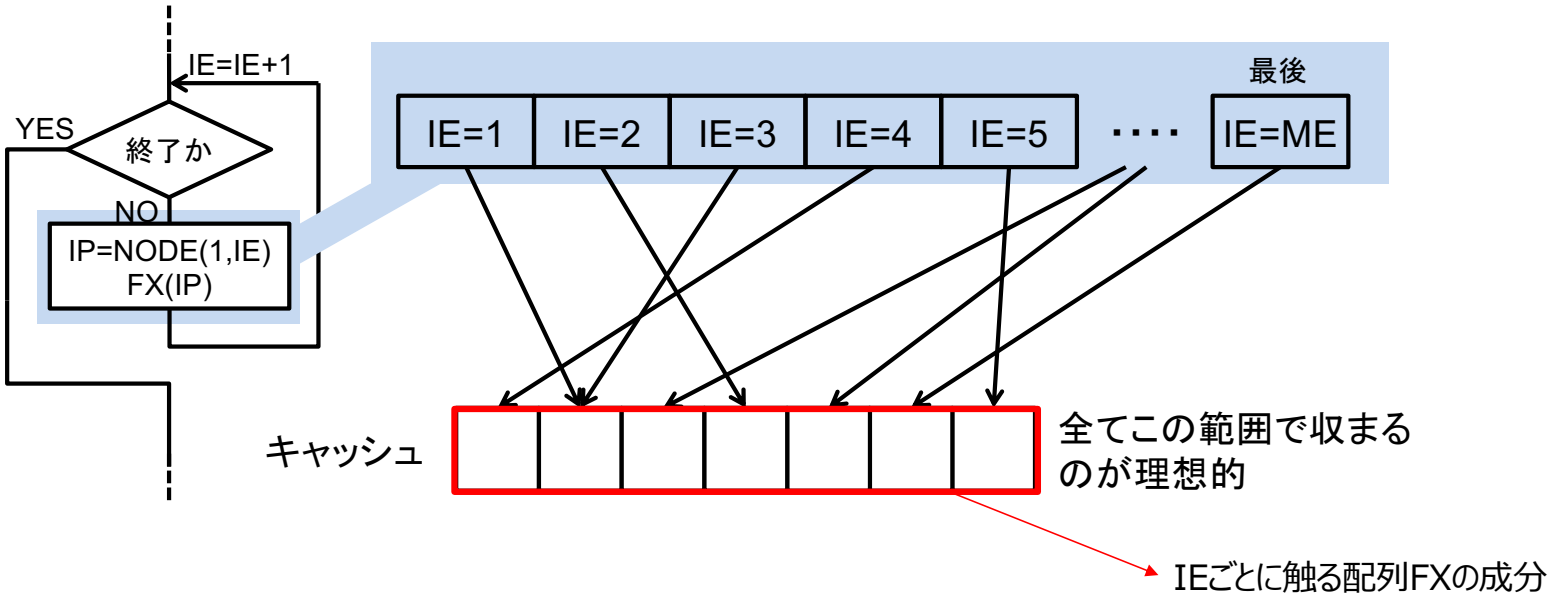


性能評価　要求バイト数の算出

waveZ

配列FX,FY,FZ

- 配列NODEの値に依存するリストアクセス故にシーケンシャルではない
- 再利用性がある
- リストによって変化してしまい一概には算出できない



まず限界性能の算出に当たり，メモリプレッシャーはないと仮定する

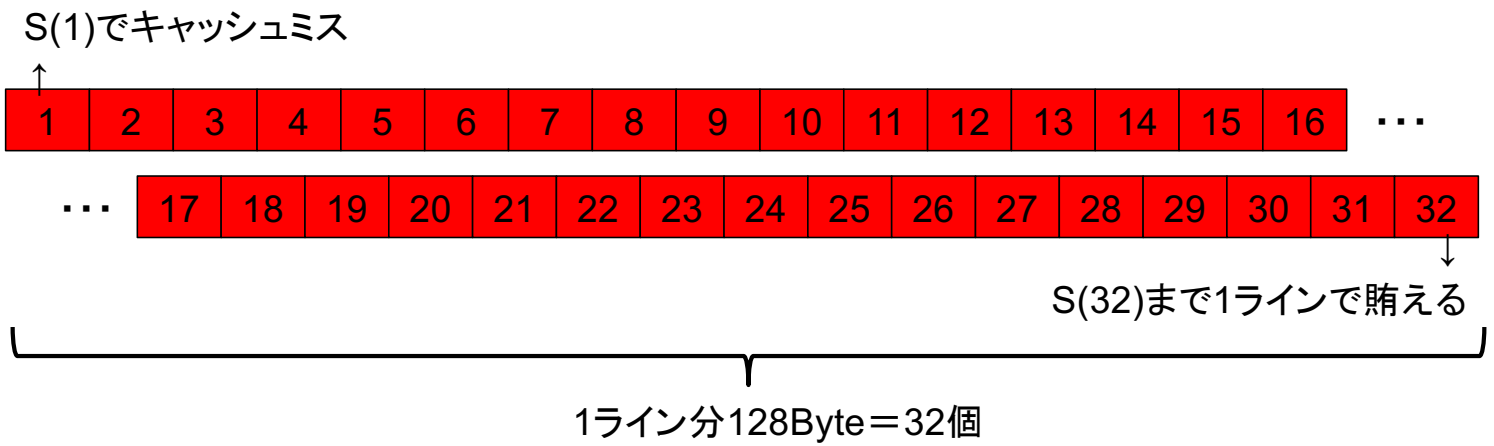
性能評価　要求バイト数の算出

waveZ

配列S

- 再内のループ変数IEに従い連続アクセス
- 再利用性はない
- 1回目のアクセスでキャッシュミスし1ライン分128Byteがメモリから転送される
- ループ1回転で4Byte、128Byteでは32回転を賄える
- メモリプレッシャーは32回転につき128Byte

```
REAL*4  S(NE)

DO  IE=IES,IEE
     ・・・
    SWRK=S(IE)
     ・・・
ENDDO
```



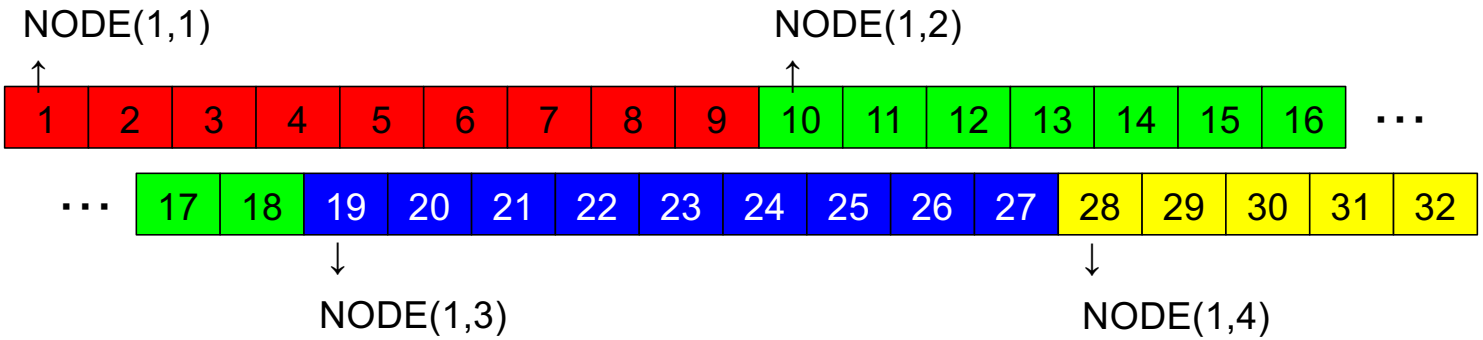
性能評価　要求バイト数の算出

waveZ

配列NODE

- 再内のループ変数IEに従い規則的にアクセス
- 再利用性はない
- 1回目のアクセスでキャッシュミスし1ライン分128Byteがメモリから転送される
- 第1次元のサイズが9でありループ1回転で36Byte、128Byteでは3.56回転を賄える
- 32回転で9回ミスしメモリプレッシャーは32回転につき1152Byte

```
INTEGER*4 NODE(9,NE)

DO IE=IES,IEE
    IP1=NODE(1,IE)
    IP2=NODE(2,IE)
    IP3=NODE(3,IE)
    IP4=NODE(4,IE)
ENDDO
```



性能評価　要求バイト数の算出

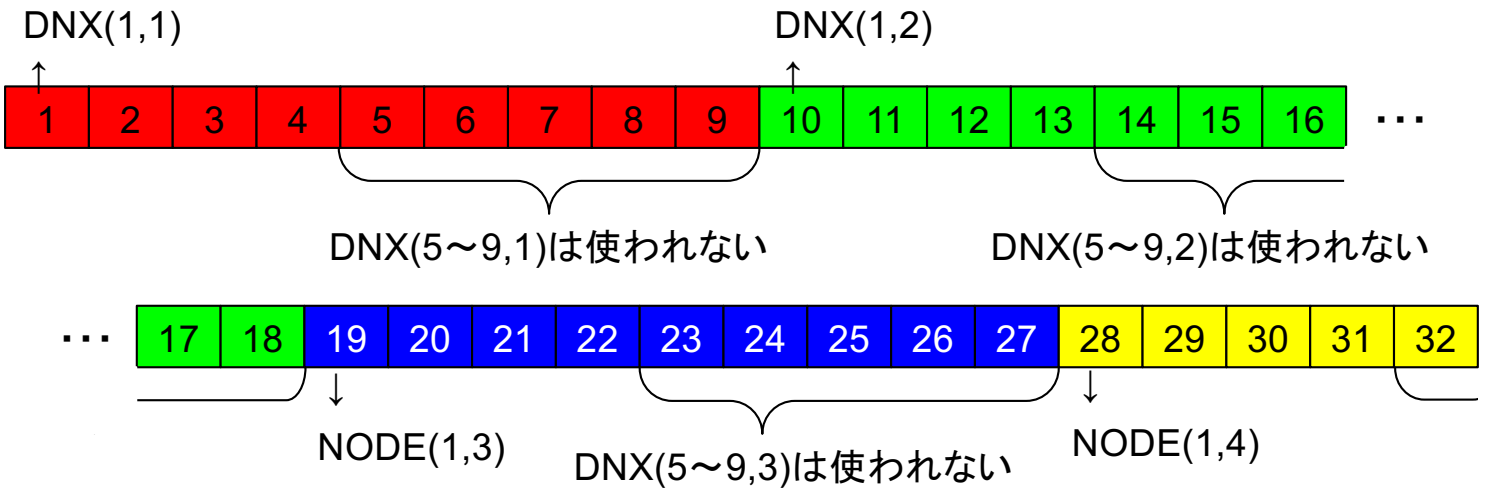
waveZ

配列DNX, DNY, DNZ

- 第1次元のサイズが9であり配列NODEと同じ性質
- メモリプレッシャーは32回転につきそれぞれ1152Byte

```
INTEGER*4 NODE(9,NE)

DO IE=IES,IEE
    ・・DNX(1,IE)
    ・・DNX(2,IE)
    ・・DNX(3,IE)
    ・・DNX(4,IE)
ENDDO
```

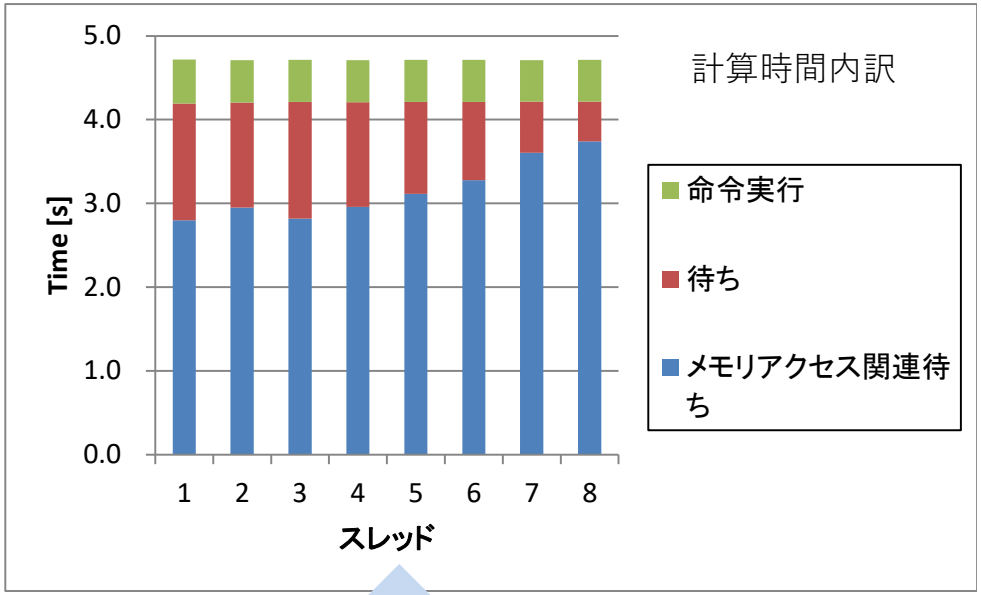
使われないがメモリ帯域を使う分を考慮する必要



性能評価

waveZ

- 要求バイト
  128＋1152＋1152×3＝4736Byte
- 浮動小数点演算数
  32回転×24FLOP＝768Flop
- B/F値は 4736/768=6.17
- STREAMベンチマークによる「京」の実効メモリバンド幅は46.6GB/s
- 「京」の実効B/F値は46.6/128=0.36
- メモリバンド幅がネックとなり実効上の性能上限は128GFLOPSの0.36/6.17=5.83%

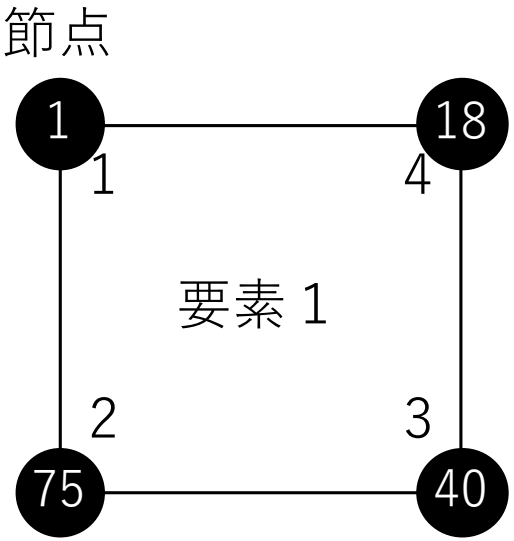


- 82万個の4面体要素
- マルチスレッドで1.6%
- メモリスループットは10.29GB/s
- メモリアクセス関連待ちが70%程度
- L1Dキャッシュミス率21%
- **キャッシュ利用効率が悪い**

キャッシュの利用効率改善

waveZ

- 配列FX, FY, FZへのアクセスには再利用性がある
- 常にオンキャッシュであると仮定(理想的)
- 実際の入力データでは再利用性はあるものの，幾何的に近い節点に，近い番号がつけられているとは限らない




```
 1=NODE(1,IE)
18=NODE(2,IE)
40=NODE(3,IE)
75=NODE(4,IE)
 ・・・
FX( 1)=FX( 1)-SWRK*DNX(1,IE)
FX(18)=FX(18)-SWRK*DNX(2,IE)
FX(40)=FX(40)-SWRK*DNX(3,IE)
FX(75)=FX(75)-SWRK*DNX(4,IE)
```

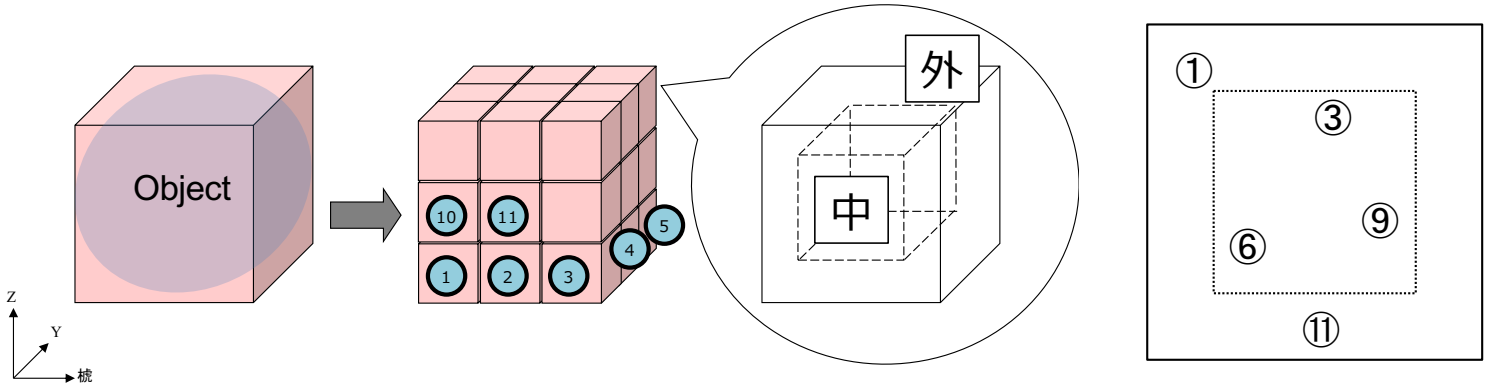
参照	ADDRESS	LINE
FX(1)	0	0
FX(75)	296	2
FX(40)	156	1
FX(18)	68	0

互いに遠い節点を参照(跳び)
キャッシュの利用効効率が悪い

幾何的に近い節点には近い番号をつける必要

キャッシュの利用効率改善

- 幾何的に近い節点には近い番号をつける
- 行列・ベクトル席の前処理として，バンド幅の縮小や，直接法を用いる際のFill-inの抑制を目的として行や列の並びを変更する手法
  - Minimum Degree法，Nested Dissection法，グラフ理論を応用した方法など
- 行列を作らずに要素毎に計算する実装なため行や列の並べ替え操作は，節点の番号を入れ替えることに相当

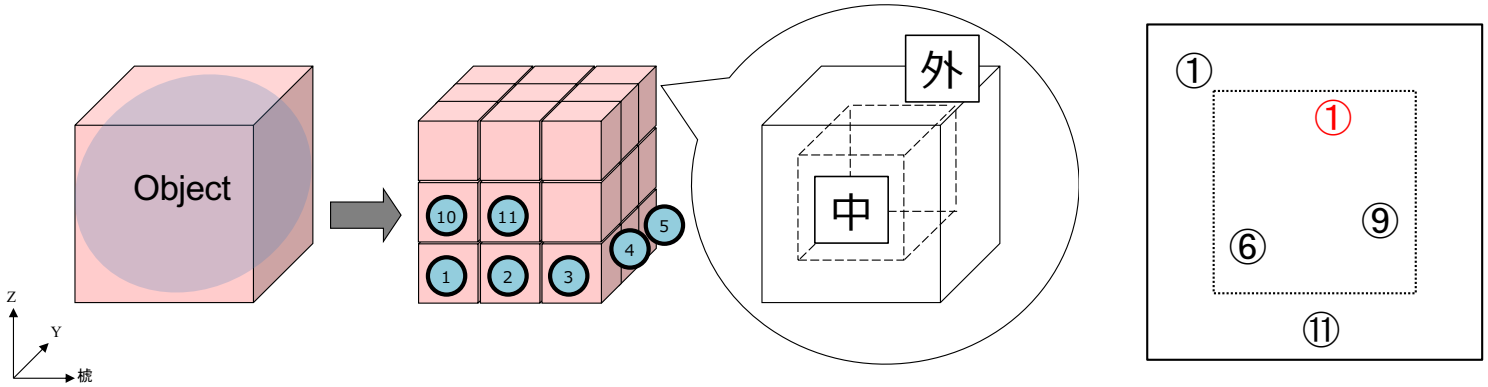


Axial-Aligned Bounding Box（AABB）

キャッシュの利用効率改善

waveZ

- 幾何的に近い節点には近い番号をつける
- 行列・ベクトル席の前処理として，バンド幅の縮小や，直接法を用いる際のFill-inの抑制を目的として行や列の並びを変更する手法
  - Minimum Degree法，Nested Dissection法，グラフ理論を応用した方法など
- 行列を作らずに要素毎に計算する実装なため行や列の並べ替え操作は，節点の番号を入れ替えることに相当

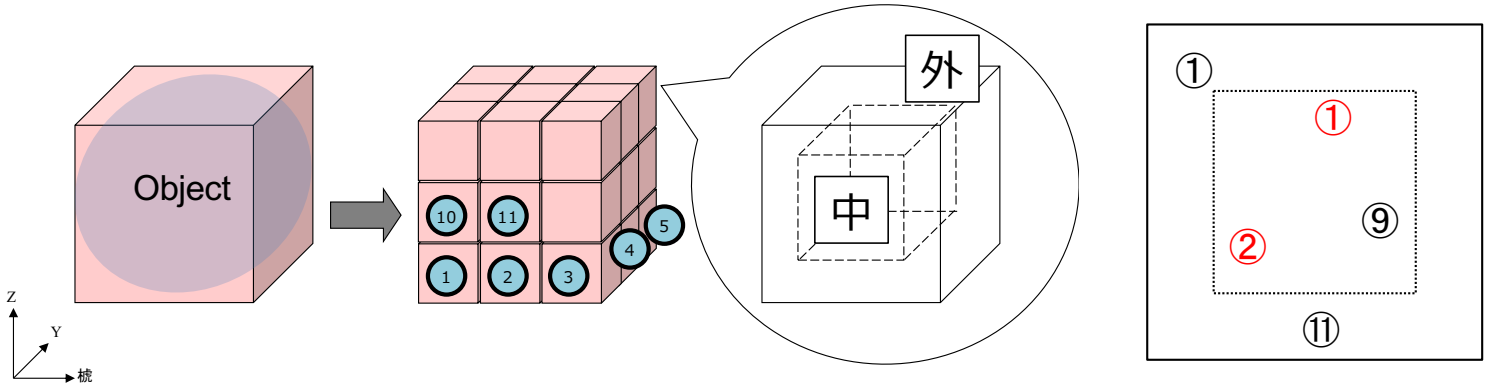


Axial-Aligned Bounding Box（AABB）

キャッシュの利用効率改善

waveZ

- 幾何的に近い節点には近い番号をつける
- 行列・ベクトル席の前処理として，バンド幅の縮小や，直接法を用いる際のFill-inの抑制を目的として行や列の並びを変更する手法
  - Minimum Degree法，Nested Dissection法，グラフ理論を応用した方法など
- 行列を作らずに要素毎に計算する実装なため行や列の並べ替え操作は，節点の番号を入れ替えることに相当

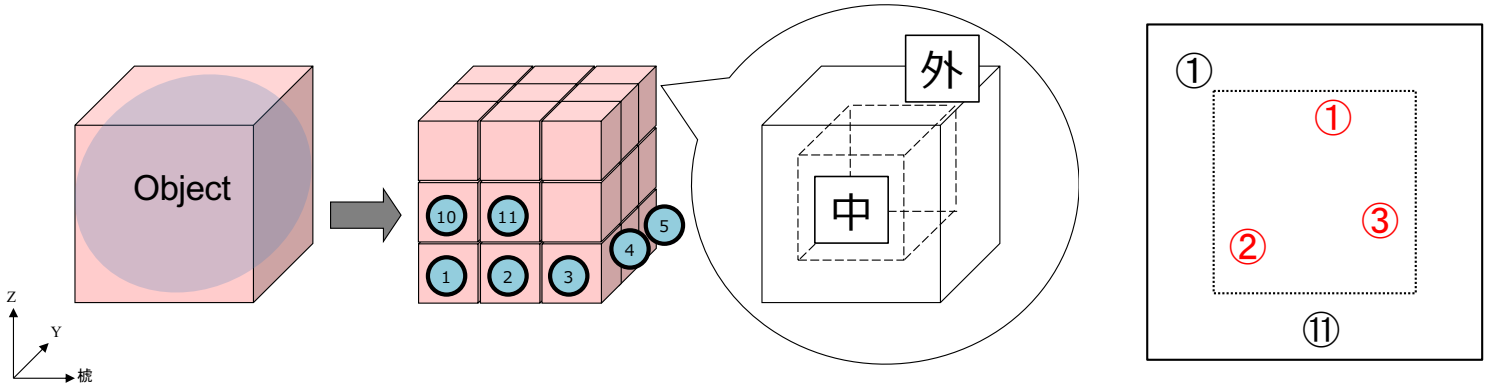


Axial-Aligned Bounding Box（AABB）

キャッシュの利用効率改善

waveZ

- 幾何的に近い節点には近い番号をつける
- 行列・ベクトル席の前処理として，バンド幅の縮小や，直接法を用いる際のFill-inの抑制を目的として行や列の並びを変更する手法
  - Minimum Degree法，Nested Dissection法，グラフ理論を応用した方法など
- 行列を作らずに要素毎に計算する実装なため行や列の並べ替え操作は，節点の番号を入れ替えることに相当

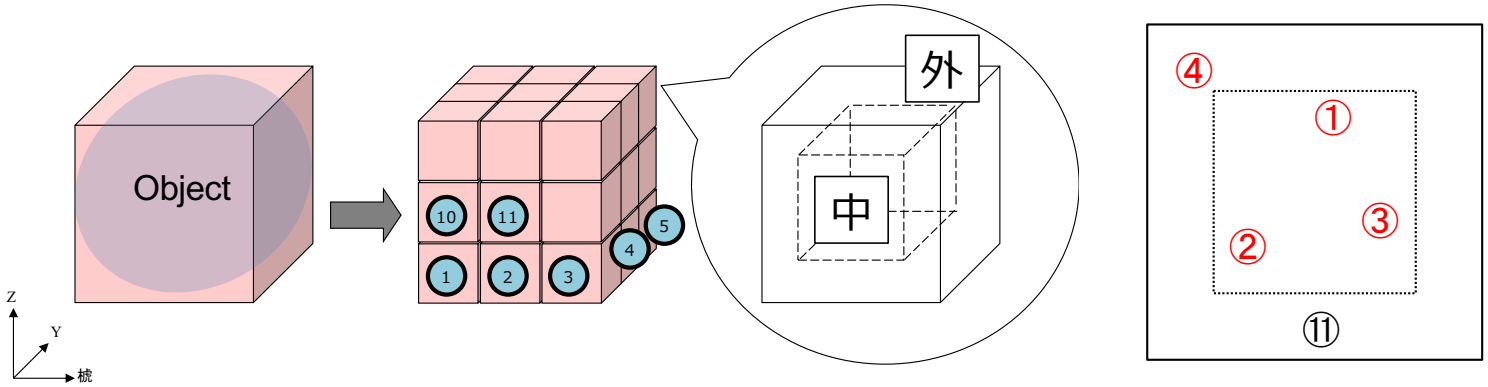


Axial-Aligned Bounding Box（AABB）

キャッシュの利用効率改善

waveZ

- 幾何的に近い節点には近い番号をつける
- 行列・ベクトル席の前処理として，バンド幅の縮小や，直接法を用いる際のFill-inの抑制を目的として行や列の並びを変更する手法
  - Minimum Degree法，Nested Dissection法，グラフ理論を応用した方法など
- 行列を作らずに要素毎に計算する実装なため行や列の並べ替え操作は，節点の番号を入れ替えることに相当

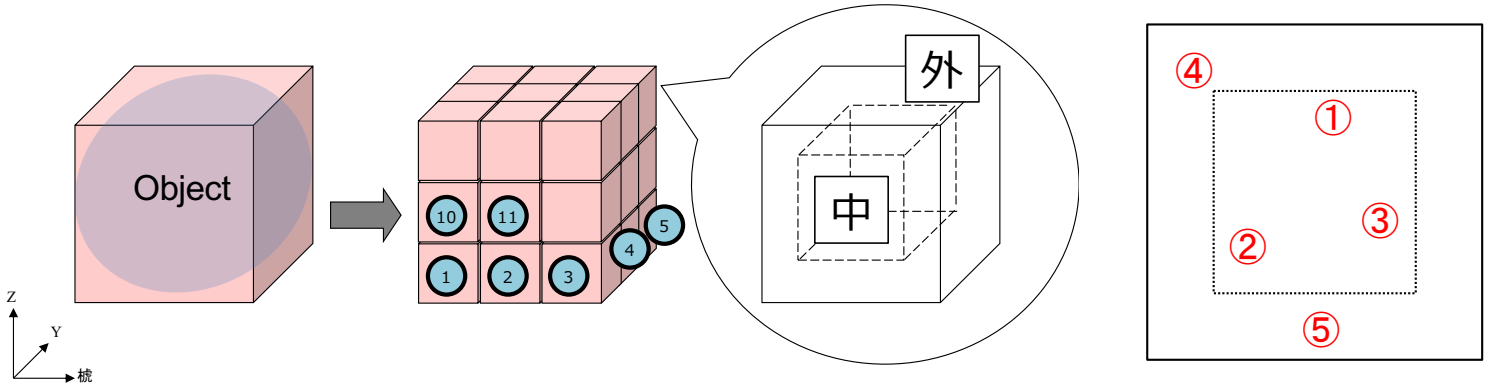


Axial-Aligned Bounding Box（AABB）

キャッシュの利用効率改善

waveZ

- 幾何的に近い節点には近い番号をつける
- 行列・ベクトル席の前処理として，バンド幅の縮小や，直接法を用いる際のFill-inの抑制を目的として行や列の並びを変更する手法
  - Minimum Degree法, Nested Dissection法, グラフ理論を応用した方法など
- 行列を作らずに要素毎に計算する実装なため行や列の並べ替え操作は，節点の番号を入れ替えることに相当

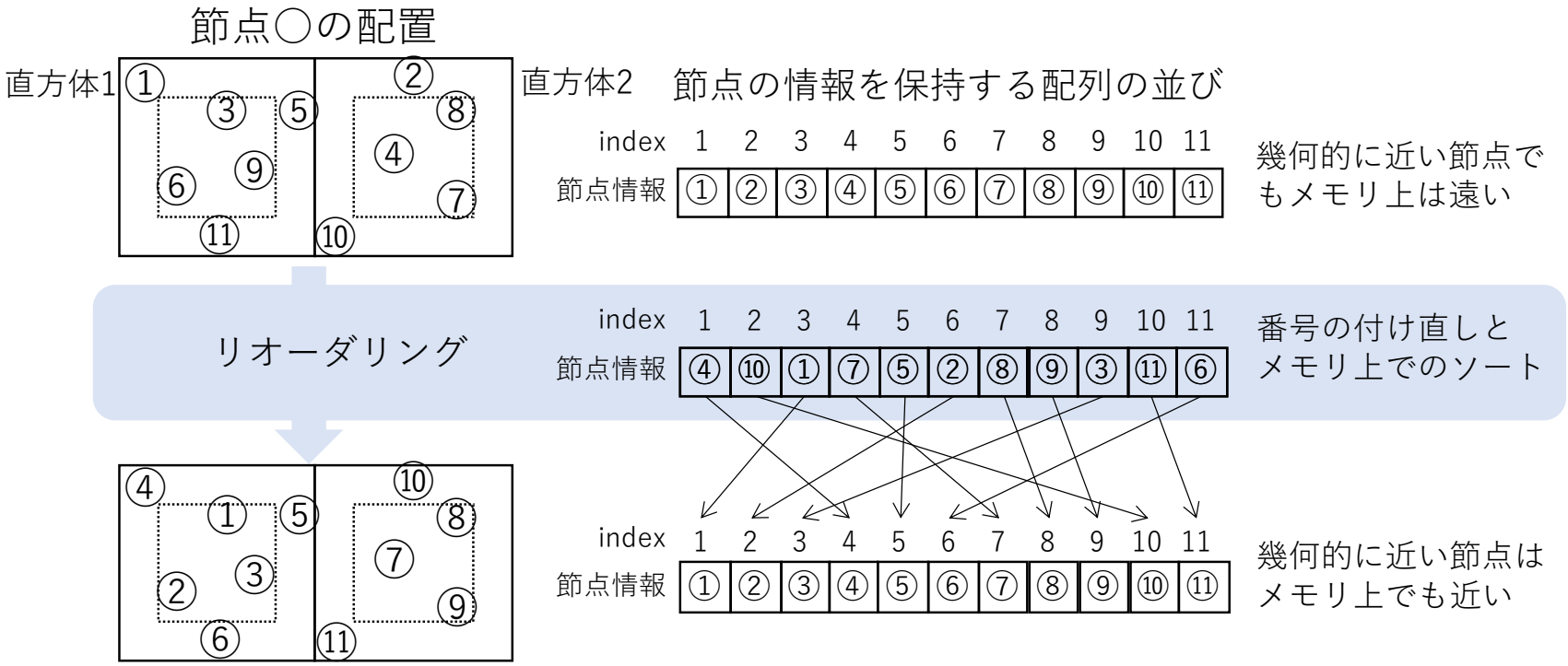


Axial-Aligned Bounding Box（AABB）

キャッシュの利用効率改善

waveZ

- 幾何的に近い節点に近い番号がつけられた
- 幾何的に近い節点の情報が，メモリ上でも近い位置になるようにメモリ上でソートしておく



Appendix) リオーダーの効果

waveZ

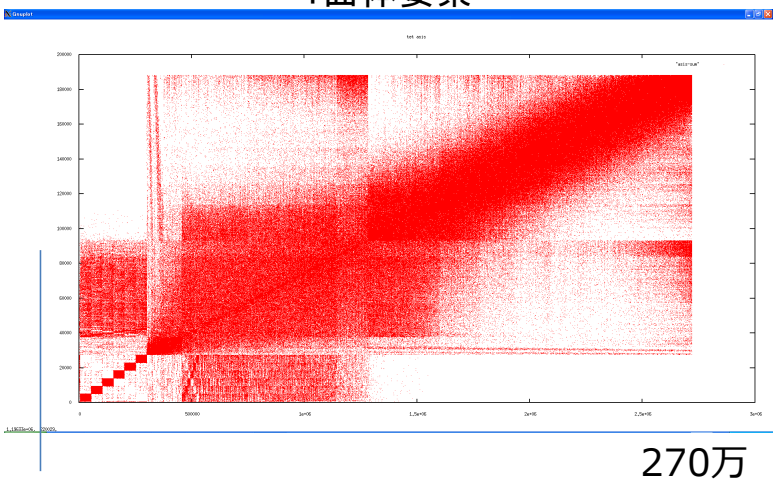
同じアプリの別部分

図は、横軸がループの回転、縦軸が、そのループで
アクセスする配列インデックスを示したもの

```
DO  I=1,N
   IDX=LIST(I)
   V=ARRAY(IDX)
   ・・・
ENDDO
```

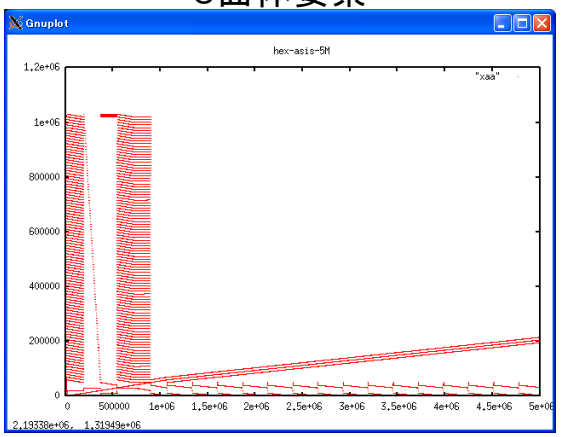
Before

4面体要素



270万

6面体要素



2700万

※一部抜粋

Appendix) リオーダーの効果

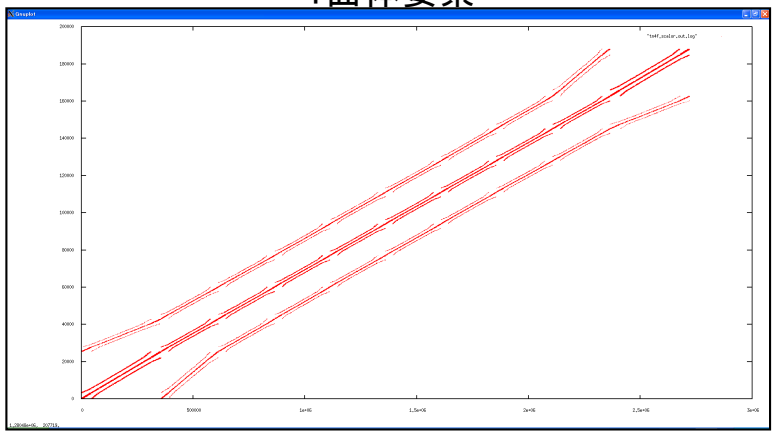
同じアプリの別部分

図は、横軸がループの回転、縦軸が、そのループで
アクセスする配列インデックスを示したもの

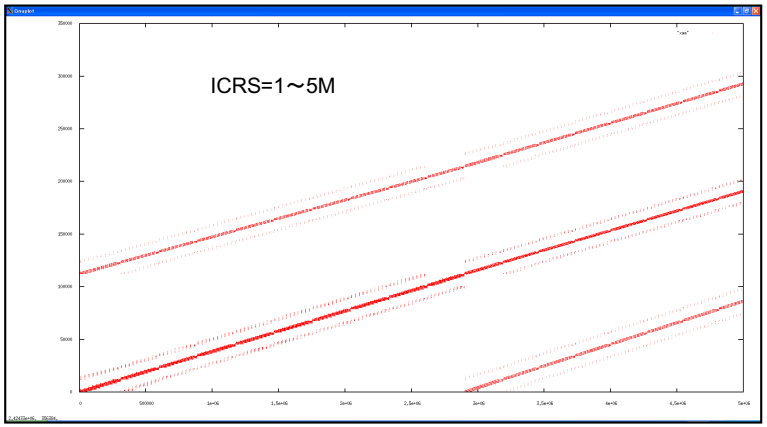
```
DO  I=1,N
   IDX=LIST(I)
   V=ARRAY(IDX)
   ・・・
ENDDO
```

After

4面体要素



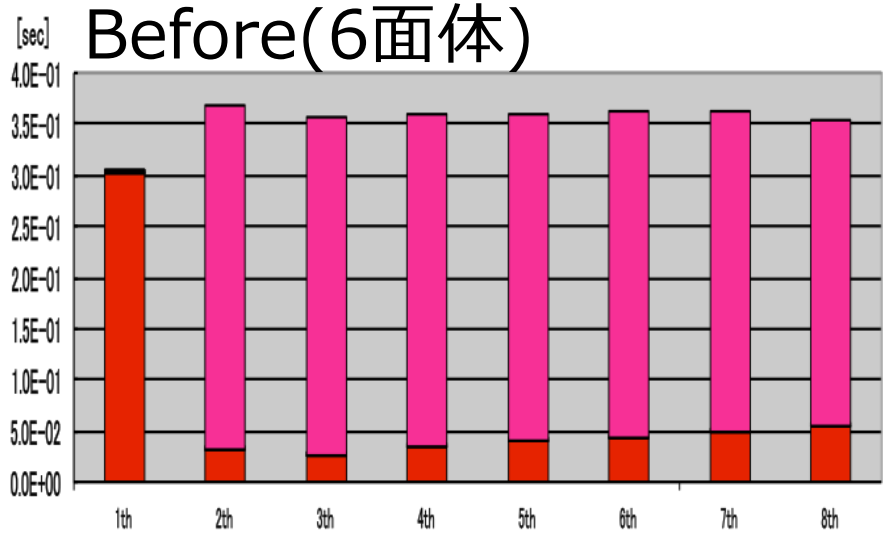
6面体要素

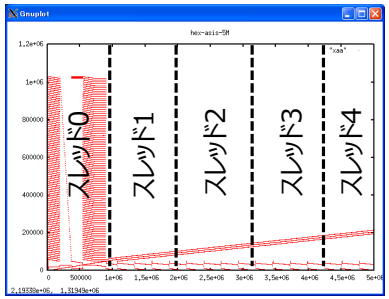


Appendix) リオーダーの効果

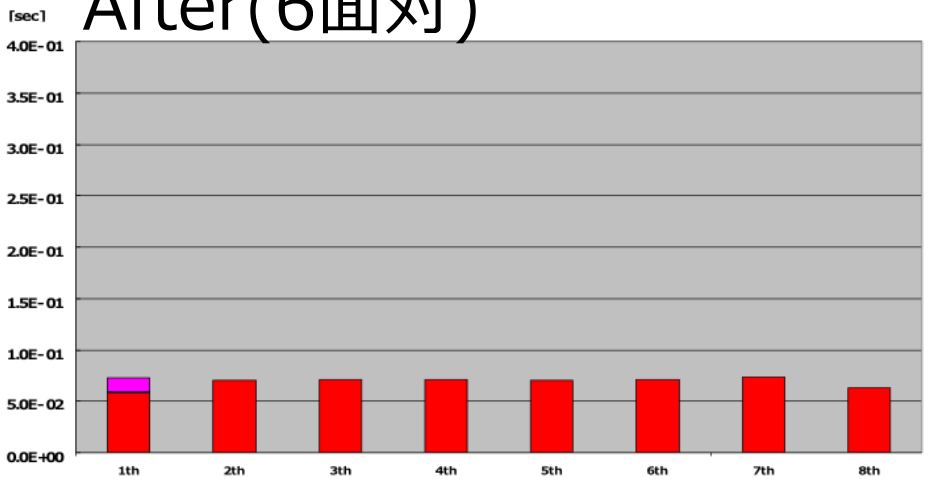
同じアプリの別部分

Before(6面体)





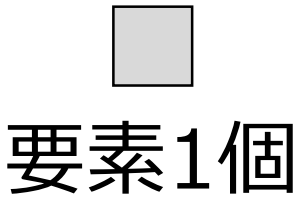
After(6面対)



スレッド並列化の方法の工夫

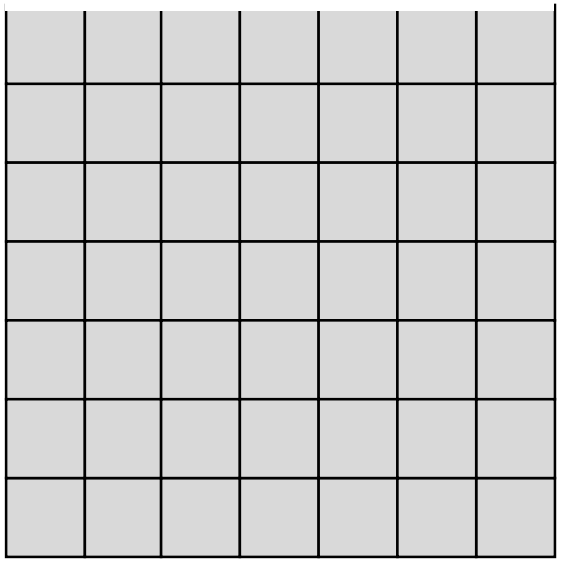
waveZ

- 要素のカラーリング領域全体で行っているので，最内ループで中には幾何的に遠い要素が出現する
- 参照する節点がメモリ上でも遠い位置あり不利

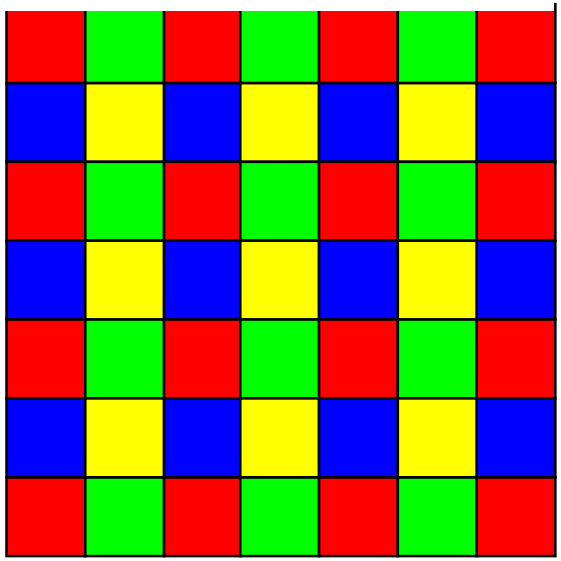


要素1個

メッシュ



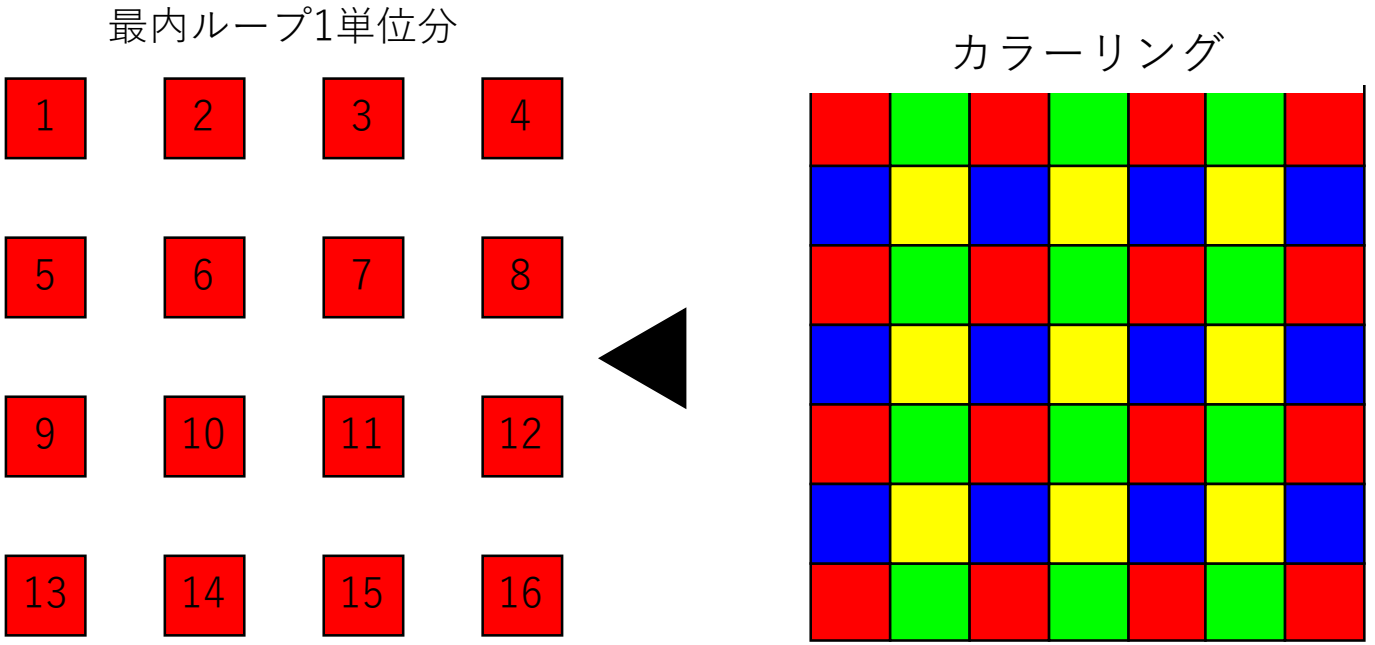
カラーリング



スレッド並列化の方法の工夫

waveZ

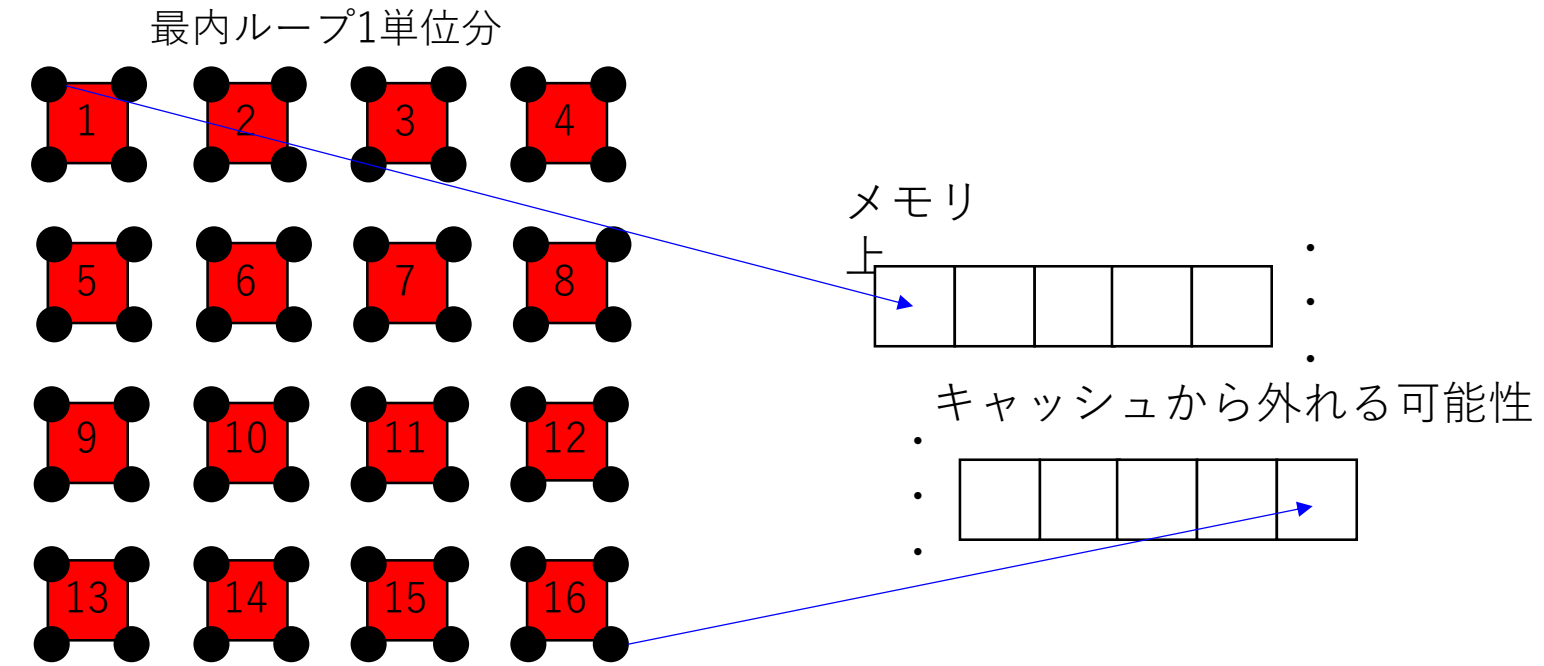
- 要素のカラーリング領域全体で行っているので，最内ループで中には幾何的に遠い要素が出現する
- 参照する節点がメモリ上でも遠い位置あり不利



スレッド並列化の方法の工夫

waveZ

- 要素のカラーリング領域全体で行っているので，最内ループで中には幾何的に遠い要素が出現する
- 参照する節点がメモリ上でも遠い位置あり不利

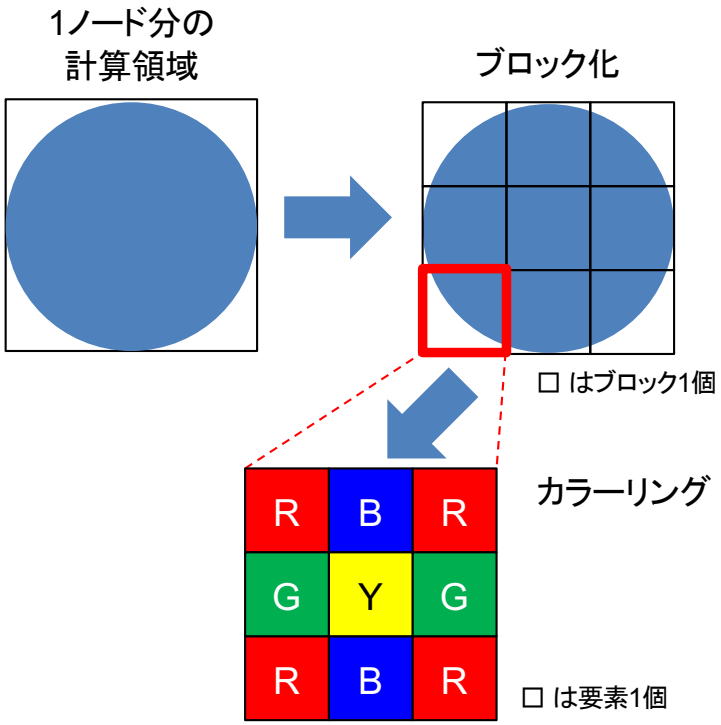


カラー内に含まれる要素を，幾何的に近いものにする必要

スレッド並列化の方法の工夫

waveZ

- カラー内に含まれる要素を，幾何的に近いものにする
- ブロックで動くループが追加




```
DO IBLOCK=1,NBLOCK(1)
   DO ICOLOR=1,NCOLOR(IBLOCK,1)
      IES=LLOOP(IBLOCK,ICOLOR  ,1)+1
      IEE=LLOOP(IBLOCK,ICOLOR+1,1)
      DO IE=IES,IEE・・・・・・・ここで並列
         IP1=NODE(1,IE)
         IP2=NODE(2,IE)
         IP3=NODE(3,IE)
         IP4=NODE(4,IE)
         SWRK=S(IE)
         FX(IP1)=FX(IP1)-SWRK*DNX(1,IE)
         FX(IP2)=FX(IP2)-SWRK*DNX(2,IE)
         FX(IP3)=FX(IP3)-SWRK*DNX(3,IE)
         FX(IP4)=FX(IP4)-SWRK*DNX(4,IE)

         FY(IP1)=FY(IP1)-SWRK*DNY(1,IE)
         FY(IP2)=FY(IP2)-SWRK*DNY(2,IE)
         FY(IP3)=FY(IP3)-SWRK*DNY(3,IE)
         FY(IP4)=FY(IP4)-SWRK*DNY(4,IE)

         FZ(IP1)=FZ(IP1)-SWRK*DNZ(1,IE)
         FZ(IP2)=FZ(IP2)-SWRK*DNZ(2,IE)
         FZ(IP3)=FZ(IP3)-SWRK*DNZ(3,IE)
         FZ(IP4)=FZ(IP4)-SWRK*DNZ(4,IE)
      ENDDO
   ENDDO
ENDDO
```

スレッド並列化の方法の工夫

waveZ

ブロックの大きさ(含む要素数)について検討

1要素当たりの必要メモリ量

```
IP1=NODE(1,IE)
IP2=NODE(2,IE)
IP3=NODE(3,IE)
IP4=NODE(4,IE)
```
NODE(1～9, IE)
36バイト

```
SWRK=S(IE)
```
S(IE)→4バイト

```
FX(IP1)=FX(IP1)-SWRK*DNX(1,IE)
FX(IP2)=FX(IP2)-SWRK*DNX(2,IE)
FX(IP3)=FX(IP3)-SWRK*DNX(3,IE)
FX(IP4)=FX(IP4)-SWRK*DNX(4,IE)
```
DNX(1-9,IE)
36バイト

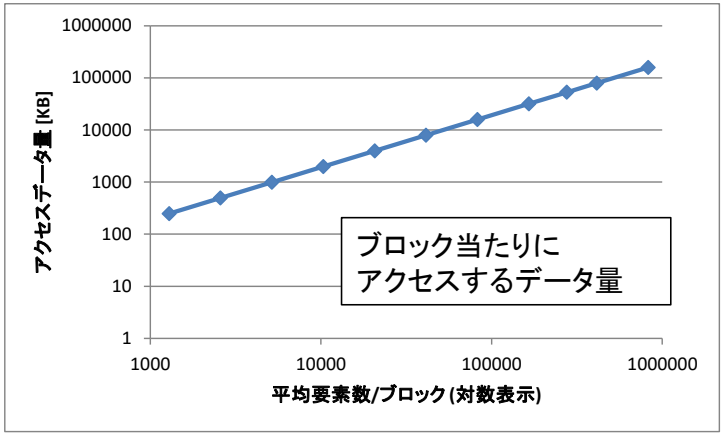
```
FY(IP1)=FY(IP1)-SWRK*DNY(1,IE)
FY(IP2)=FY(IP2)-SWRK*DNY(2,IE)
FY(IP3)=FY(IP3)-SWRK*DNY(3,IE)
FY(IP4)=FY(IP4)-SWRK*DNY(4,IE)
```
DNY(1-9,IE)
36バイト

```
FZ(IP1)=FZ(IP1)-SWRK*DNZ(1,IE)
FZ(IP2)=FZ(IP2)-SWRK*DNZ(2,IE)
FZ(IP3)=FZ(IP3)-SWRK*DNZ(3,IE)
FZ(IP4)=FZ(IP4)-SWRK*DNZ(4,IE)
```
DNZ(1-9,IE)
36バイト

FX(IP1～IP4)
FY(IP1～IP4)
FZ(IP1～IP4)
48バイト

196バイト/要素

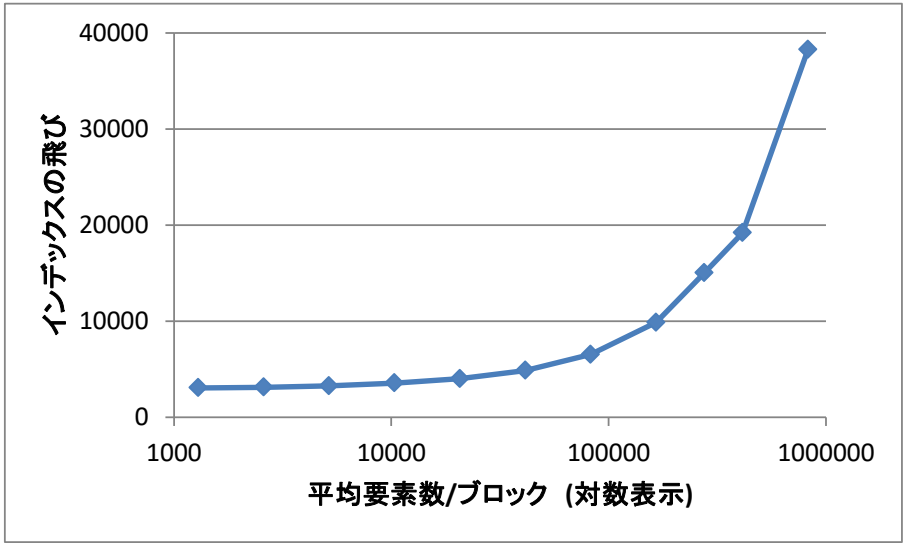
ケースNo	ブロック分割数	平均要素数/ブロック	アクセスデータ量 [KB]	1スレッド当りメモリ [KB]	on L1	on L2
1	640	1291	247.1	30.9	○	○
2	320	2583	494.4	61.8	×	○
3	160	5166	988.8	123.6	×	○
4	80	10333	1977.8	247.2	×	○
5	40	20666	3955.6	494.5	×	○
6	20	41332	7911.2	988.9	×	×
7	10	82665	15822.6	1977.8	×	×
8	5	165331	31645.4	3955.7	×	×
9	3	275552	52742.4	6592.8	×	×
10	2	413328	79113.6	9889.2	×	×
11	1	826656	158227.1	19778.4	×	×

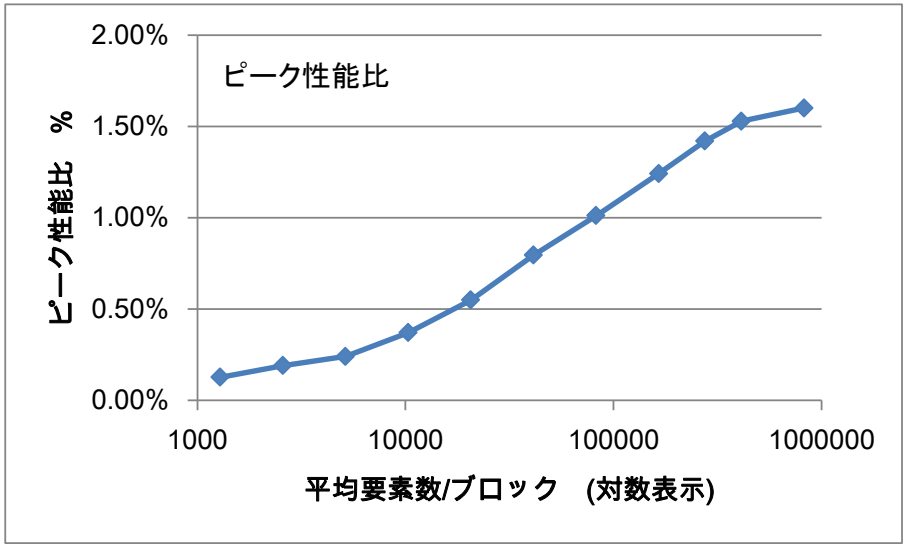


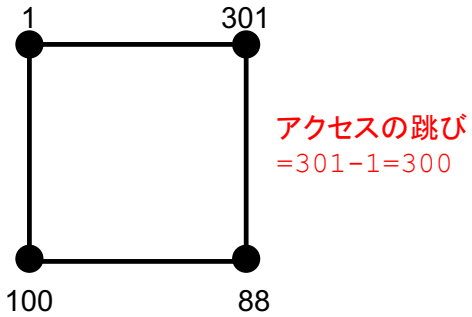
スレッド並列化の方法の工夫

waveZ

ブロックの大きさ(含む要素数)について検討





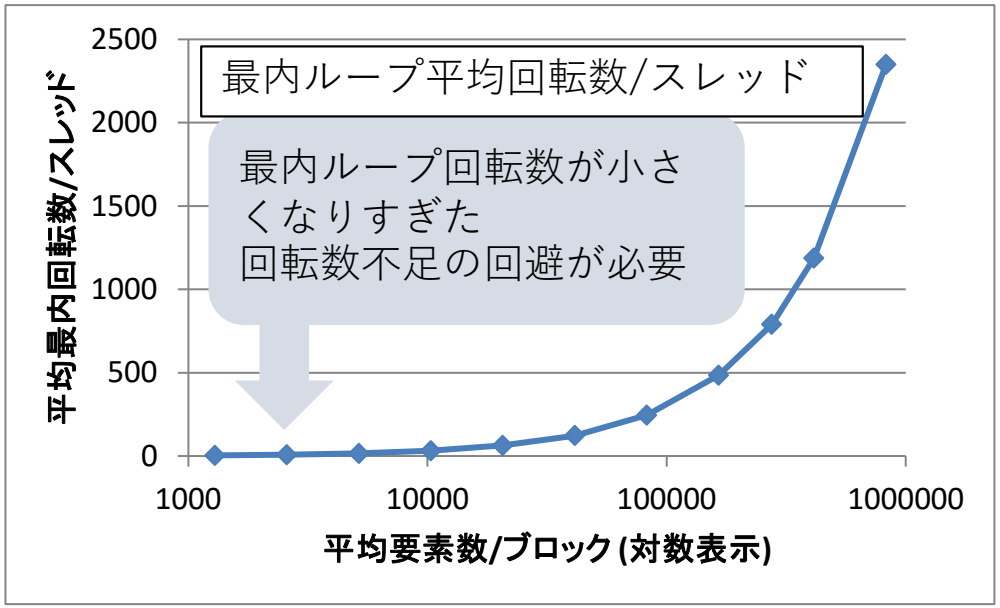


スレッド並列化の方法の工夫

waveZ

ブロックの大きさ(含む要素数)について検討

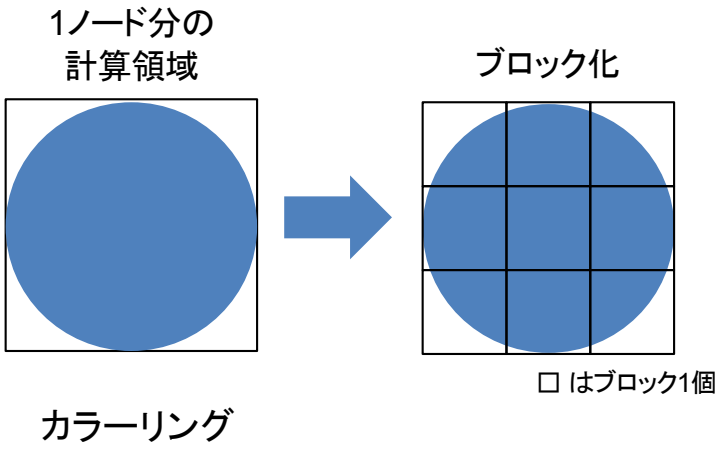
ケースNo	ブロック分割数	平均要素数/ブロック	平均カラー数/ブロック	平均要素数/カラー	平均最内回転数/スレッド
1	640	1291	37.4	34.0	4
2	320	2583	38.7	66.0	8
3	160	5166	39.7	130.0	16
4	80	10333	41.0	252.0	32
5	40	20666	41.5	498.0	62
6	20	41332	42.1	981.0	123
7	10	82665	42.2	1958.0	245
8	5	165331	42.6	3881.0	485
9	3	275552	43.7	6310.0	789
10	2	413328	43.5	9501.0	1188
11	1	826656	44.0	18787.0	2348



スレッド並列化の方法の工夫

waveZ

- カラー内に含まれる要素を，幾何的に近いものにする
- ブロックとカラーの関係を反転

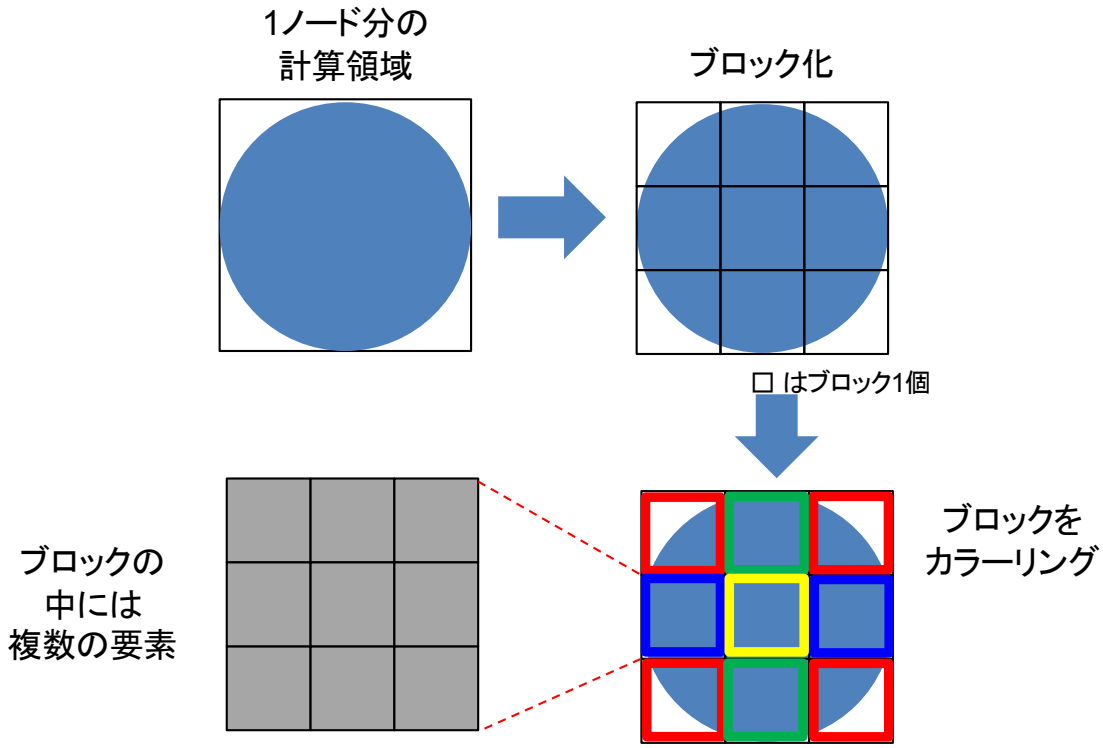



```
DO ICOLOR=1,NCOLOR(1)
   DO IBLOCK=1,NBLOCK(IBLOCK,1)
      IES=LLOOP(IBLOCK,ICOLOR   ,1)+1
      IEE=LLOOP(IBLOCK,ICOLOR+1,1)
      DO IE=IES,IEE
         IP1=NODE(1,IE)
         IP2=NODE(2,IE)
         IP3=NODE(3,IE)
         IP4=NODE(4,IE)
         SWRK=S(IE)
         FX(IP1)=FX(IP1)-SWRK*DNX(1,IE)
         FX(IP2)=FX(IP2)-SWRK*DNX(2,IE)
         FX(IP3)=FX(IP3)-SWRK*DNX(3,IE)
         FX(IP4)=FX(IP4)-SWRK*DNX(4,IE)

         FY(IP1)=FY(IP1)-SWRK*DNY(1,IE)
         FY(IP2)=FY(IP2)-SWRK*DNY(2,IE)
         FY(IP3)=FY(IP3)-SWRK*DNY(3,IE)
         FY(IP4)=FY(IP4)-SWRK*DNY(4,IE)

         FZ(IP1)=FZ(IP1)-SWRK*DNZ(1,IE)
         FZ(IP2)=FZ(IP2)-SWRK*DNZ(2,IE)
         FZ(IP3)=FZ(IP3)-SWRK*DNZ(3,IE)
         FZ(IP4)=FZ(IP4)-SWRK*DNZ(4,IE)
      ENDDO
   ENDDO
ENDDO
```

スレッド並列化の方法の工夫

waveZ

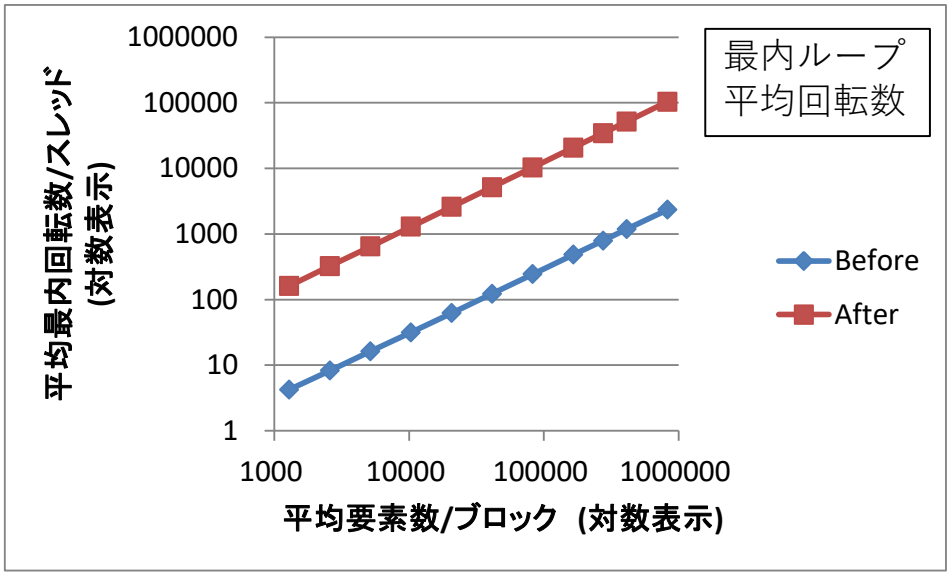
- カラー内に含まれる要素を，幾何的に近いものにする
- ブロックとカラーの関係を反転

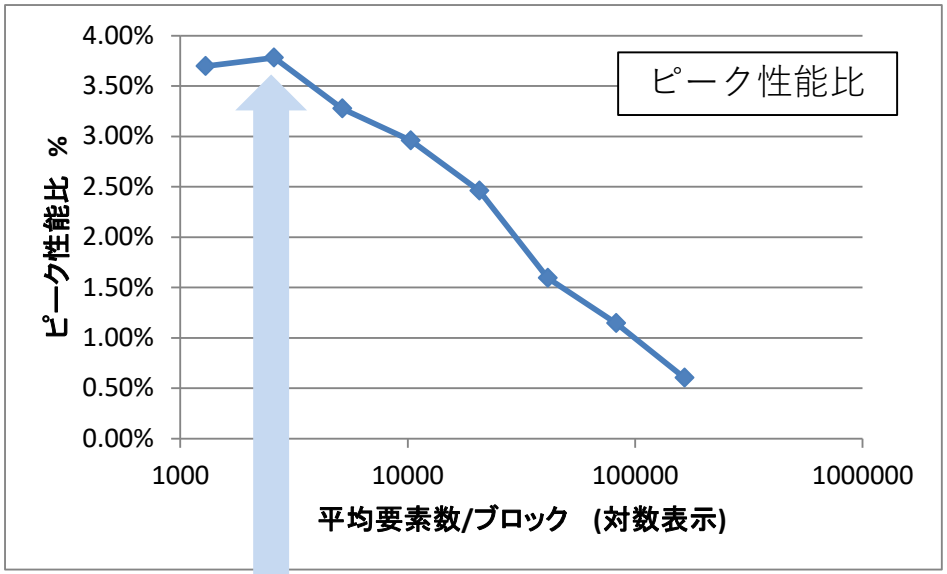



```
DO ICOLOR=1,NCOLOR(1)
   DO IBLOCK=1,NBLOCK(IBLOCK,1)  ここで並列
      IES=LLOOP(IBLOCK,ICOLOR    ,1)+1
      IEE=LLOOP(IBLOCK,ICOLOR+1,1)
      DO IE=IES,IEE
         IP1=NODE(1,IE)
         IP2=NODE(2,IE)
         IP3=NODE(3,IE)
         IP4=NODE(4,IE)
         SWRK=S(IE)
         FX(IP1)=FX(IP1)-SWRK*DNX(1,IE)
         FX(IP2)=FX(IP2)-SWRK*DNX(2,IE)
         FX(IP3)=FX(IP3)-SWRK*DNX(3,IE)
         FX(IP4)=FX(IP4)-SWRK*DNX(4,IE)

         FY(IP1)=FY(IP1)-SWRK*DNY(1,IE)
         FY(IP2)=FY(IP2)-SWRK*DNY(2,IE)
         FY(IP3)=FY(IP3)-SWRK*DNY(3,IE)
         FY(IP4)=FY(IP4)-SWRK*DNY(4,IE)

         FZ(IP1)=FZ(IP1)-SWRK*DNZ(1,IE)
         FZ(IP2)=FZ(IP2)-SWRK*DNZ(2,IE)
         FZ(IP3)=FZ(IP3)-SWRK*DNZ(3,IE)
         FZ(IP4)=FZ(IP4)-SWRK*DNZ(4,IE)
      ENDDO
   ENDDO
ENDDO
```

スレッド並列化の方法の工夫

waveZ





- 最内ループの回転数が38～44倍程度まで増大
- ブロックサイズ小の側で性能向上
- 最良の結果は平均最内回転数/スレッドが323回転

- 平均最内回転数/スレッド=323回転
- ピーク性能比3.78%
- L1Dミス率4%
- メモリスループット32GB/s

スレッド並列化の方法の工夫

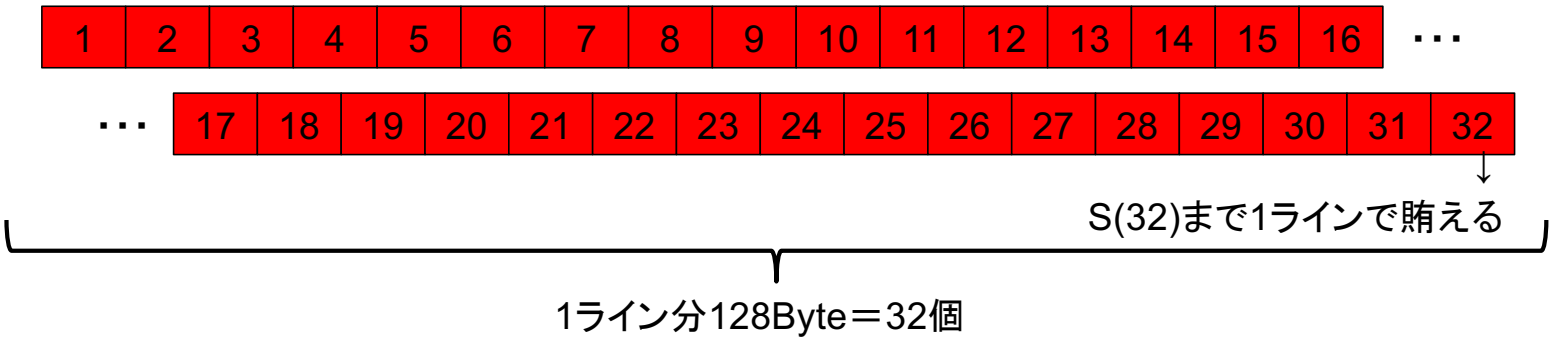
waveZ

ここまでのチューニングにより

	オリジナル	改善後	理想値
ピーク性能比(%)	1.6	3.8	5.83
メモリスループット(GB/s)	10.3	32	46
L1Dキャッシュミス率(%)	21.0	3.78	3.13

もう少し改善の余地があるのでは？

4バイト変数の配列S(N)



32回に1回ミスするので1/32=3.125%が理論的なミス率
(ただし全てシーケンシャルな場合なので注意)

スレッド並列化の方法の工夫

waveZ

配列融合によるストリーム削減

オリジナル

```
FX(IP1)=FX(IP1)+SWRK*DNX(1,IE)
FY(IP1)=FY(IP1)+SWRK*DNY(1,IE)
FZ(IP1)=FZ(IP1)+SWRK*DNZ(1,IE)
```

パターン4

```
FXYZ(1,IP1)=FXYZ(1,IP1)+SWRK*DNXYZ(1,1,IE)
FXYZ(2,IP1)=FXYZ(2,IP1)+SWRK*DNXYZ(2,1,IE)
FXYZ(3,IP1)=FXYZ(3,IP1)+SWRK*DNXYZ(3,1,IE)
```

パターンNo	内容
1	DNX, DNY, DNZをDNXYZに融合
2	FX, FY, FZをFXYZに融合
3	パターン2＋演算順序変更
4	パターン3＋パターン1

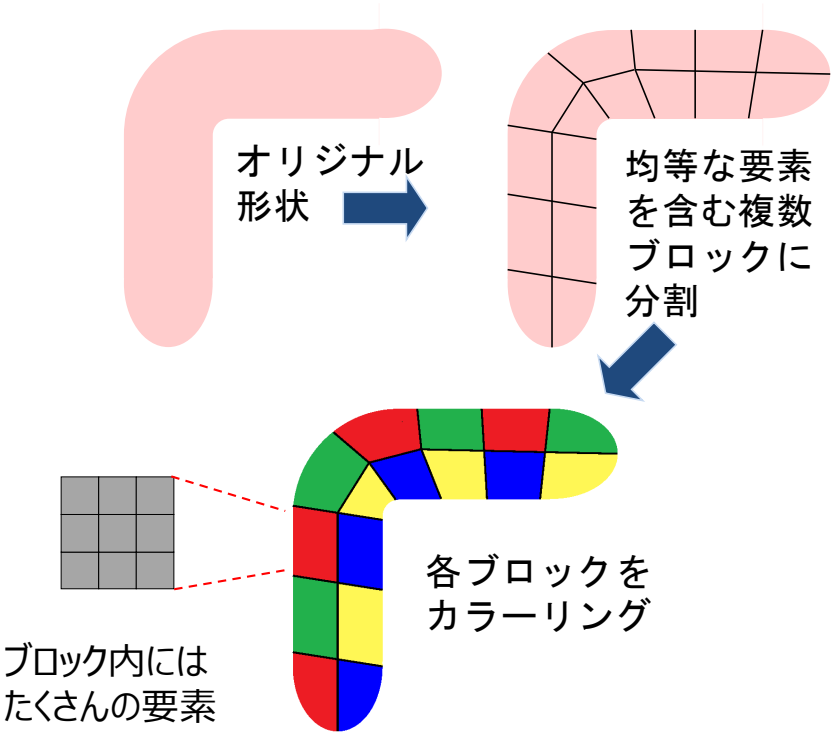
要求バイトが変わらないため、理論性能は変化せず

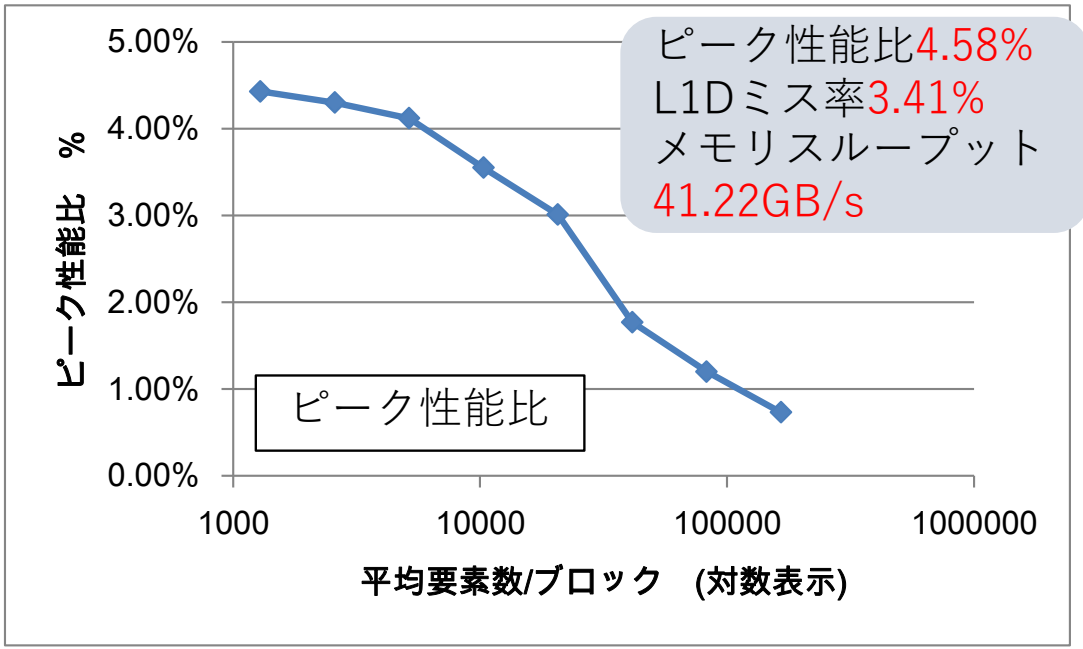
パターンNo	ピーク性能比%	L1Dキャッシュミス率%	メモリスループットGB/s
1	3.95	3.98	35.70
2	4.05	3.69	35.91
3	4.05	3.69	35.88
4	4.41	3.37	38.80

スレッド並列化の方法の工夫

waveZ

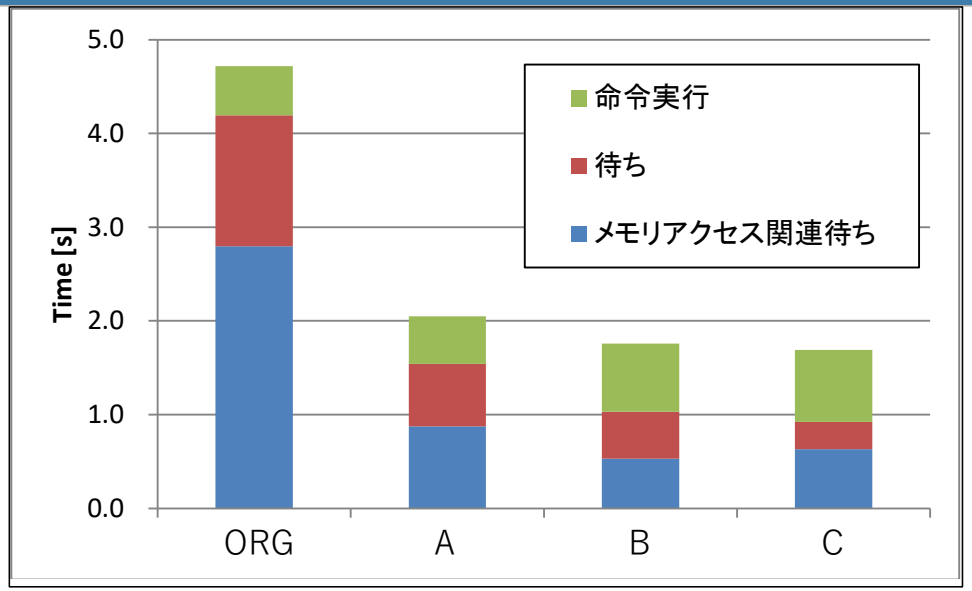
- 矩形分割ではブロック間に要素数のインバランスが生ずる可能性
- 要素のブロック化にMETISを用いて様々な形状にも対応



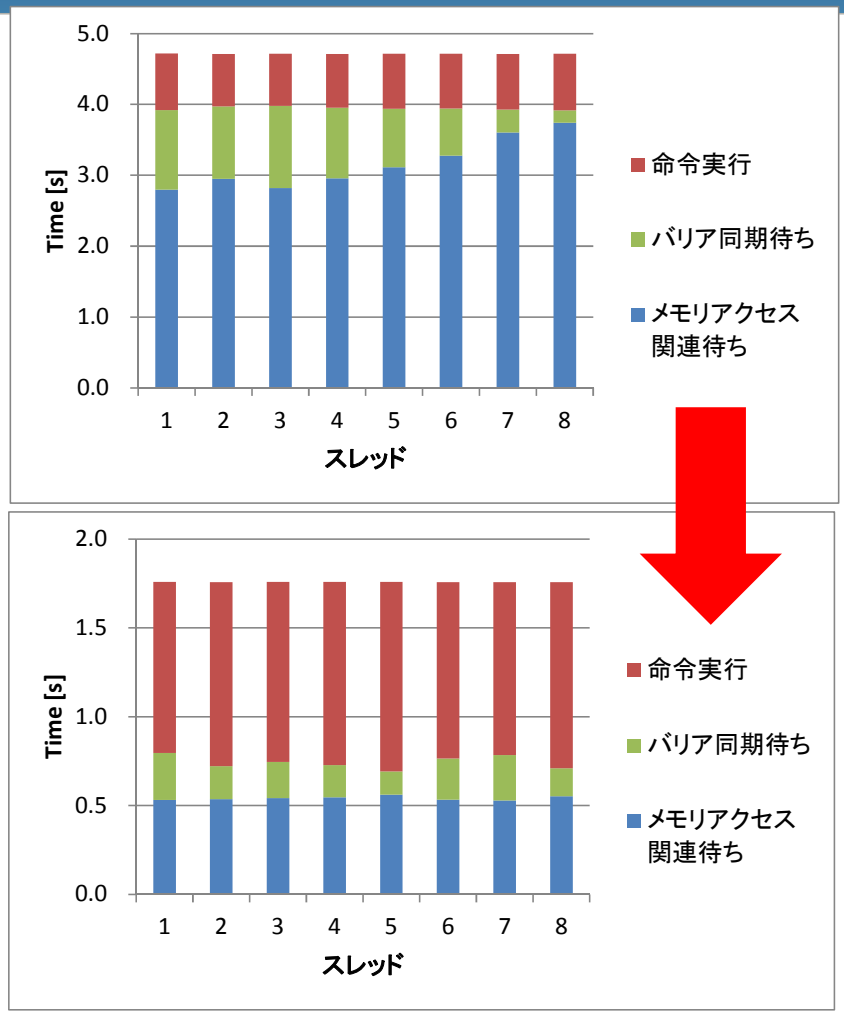


まとめ

waveZ



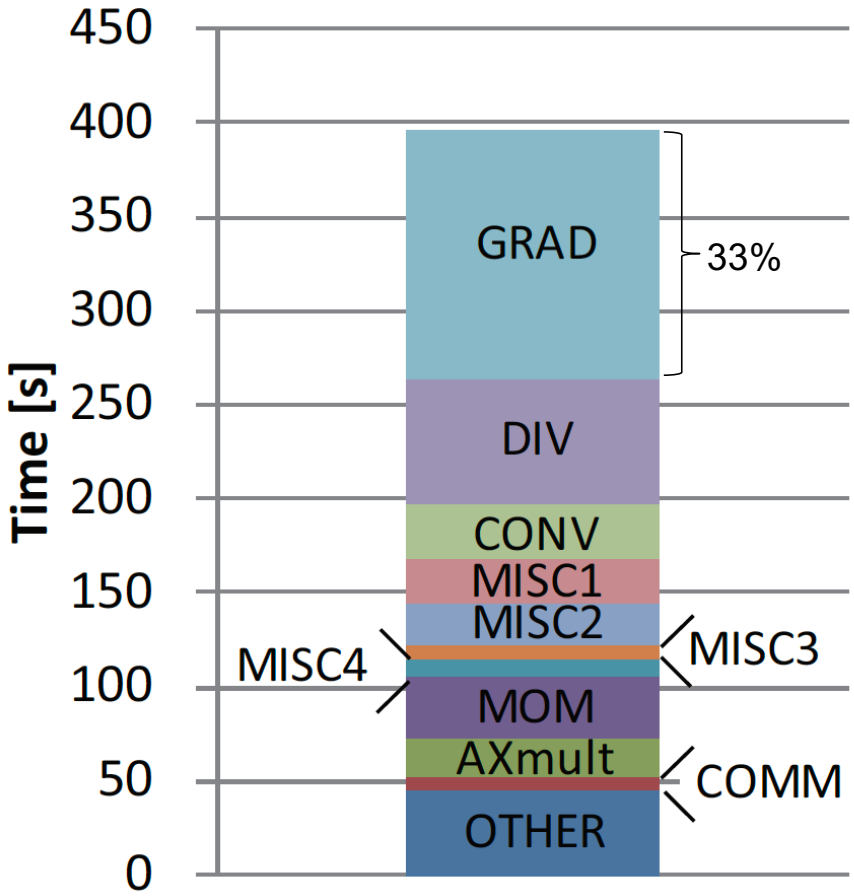
ケース名	内容
ORG	オリジナル
A	節点リナンバー，要素ブロック，外側カラー化
B	配列融合
C	ブロック化手法改善



実アプリ

ストアのシーケンシャル化

チューニング対象ルーチン



- メインループ（時間積分ループ）の内訳

- 入力データは単純な6面体要素の形状で，1プロセスあたり100万要素
- 全体では24プロセスで，京で1ノード1プロセス，1プロセス8スレッド実行

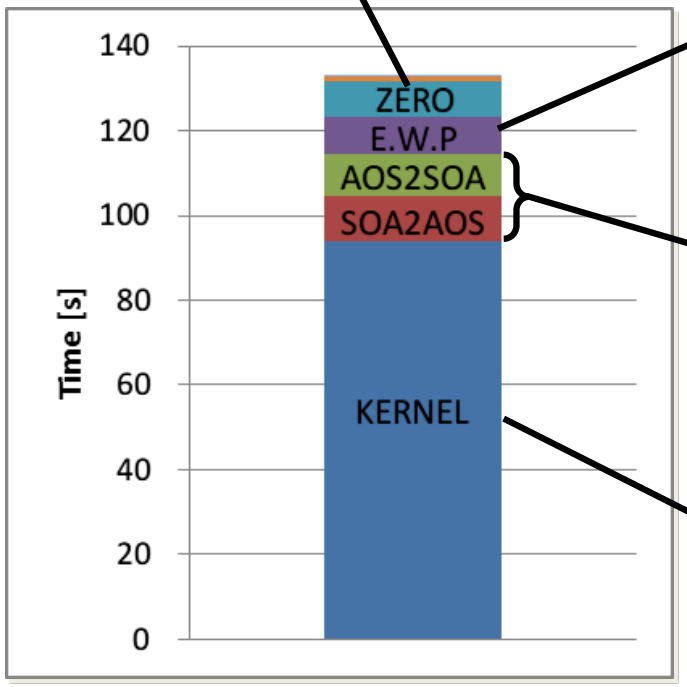
- GRAD, DIV) 質量保存（ポアソン方程式）関係
- MOM, AXmult) 運動量関係
- CONV) データの変換
- COMM) 通信
- MISC) 色々雑多
- OTHER, 個別に扱えない細かいものの合計

一番重たいGRADを対象とする

チューニング対象カーネル

waveZ

Initialize array to zero



Ingredient of routine GRAD

Element Wise Product of vector
(Hadamard Product)

$A=\{a_1, a_2, \ldots, a_n\}$
$B=\{b_1, b_2, \ldots, b_n\}$
$C=\{c_1, c_2, \ldots, c_n\}$

$$c_i = a_i b_i$$

Data Structure Conversion

$\vec{v}_1 = \{x_1, y_1, z_1\}$
$\vec{v}_2 = \{x_2, y_2, z_2\}$
$\vdots$
$\vec{v}_N = \{x_N, y_N, z_N\}$

There are *N* vectors

Memory Space

Structure of Array (SOA)

x_1 x_2 ··· x_N y_1 y_2 ··· y_N z_1 z_2 ··· z_N

Array of Structure (AOS)

x_1 y_1 z_1 x_2 y_2 z_2 ····· x_N y_N z_N

Main Calculation

カーネル処理内容

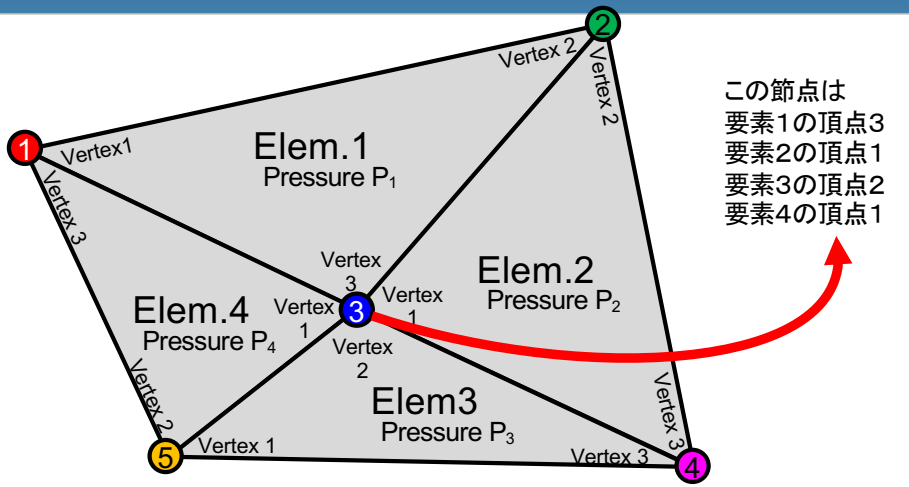
waveZ

前ページの事例の最終状態(のすこし違うVer)

```
 1 DO iCOL=1,iNUM_COLOR
 2   DO iBLK=1,iNUM_BLOCK(iCOL)     Multi-Thread
 3     iHEAD=iHEAD_ELEM(iBLK,iCOL)
 4     iTAIL=iTAIL_ELEM(iBLK,iCOL)
 5     DO iE=iHEAD, iTIAL
 6       iN1=iNODE(1,iE)
 7       iN2=iNODE(2,iE)
...      ...
13       iN8=iNODE(8,iE)
14       fP=fPRES(iE)
15
16       fDP(1,iN1)+=fP*fDN(1,1,iE)
17       fDP(2,iN1)+=fP*fDN(2,1,iE)
18       fDP(3,iN1)+=fP*fDN(3,1,iE)
19
20       fDP(1,iN2)+=fP*fDN(1,2,iE)
21       fDP(2,iN2)+=fP*fDN(2,2,iE)
22       fDP(3,iN2)+=fP*fDN(3,2,iE)
...      ...
44       fDP(1,iN8)+=fP*fDN(1,8,iE)
45       fDP(2,iN8)+=fP*fDN(2,8,iE)
46       fDP(3,iN8)+=fP*fDN(3,8,iE)
47     ENDDO
48   ENDDO
49 ENDDO
```

$\frac{\partial N_{iE,1}}{\partial x}$

$\frac{\partial N_{iE,8}}{\partial z}$



$N_{i,j}$ は有限要素法の形状関数で，要素iの頂点jごと定義されている，
たとえば，$N_{1,1}$ と $N_{4,3}$ は一見同じ位置にあるが，違うもの

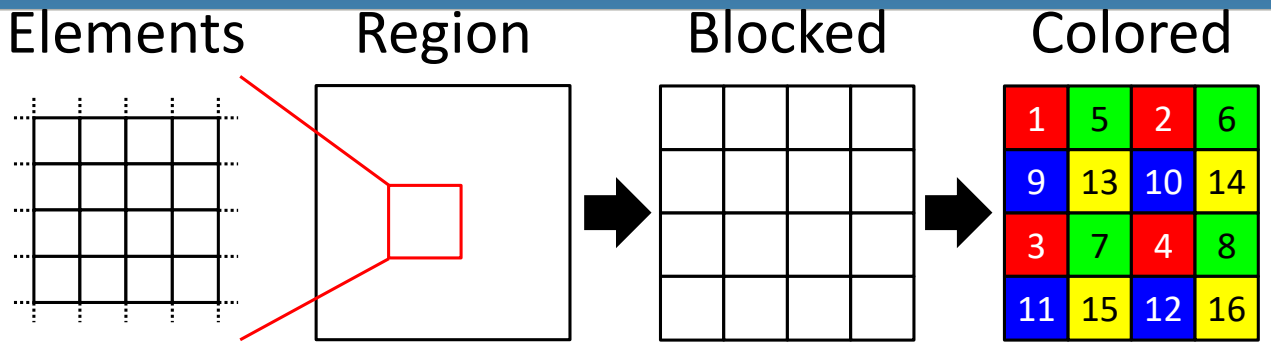
中央の節点における圧力勾配は以下の計算となる

$$\nabla P = \begin{cases} \frac{\partial N_{1,3}}{\partial x} P_1 + \frac{\partial N_{2,1}}{\partial x} P_2 + \frac{\partial N_{3,2}}{\partial x} P_3 + \frac{\partial N_{4,1}}{\partial x} P_4 \\ \frac{\partial N_{1,3}}{\partial y} P_1 + \frac{\partial N_{2,1}}{\partial y} P_2 + \frac{\partial N_{3,2}}{\partial y} P_3 + \frac{\partial N_{4,1}}{\partial y} P_4 \end{cases}$$

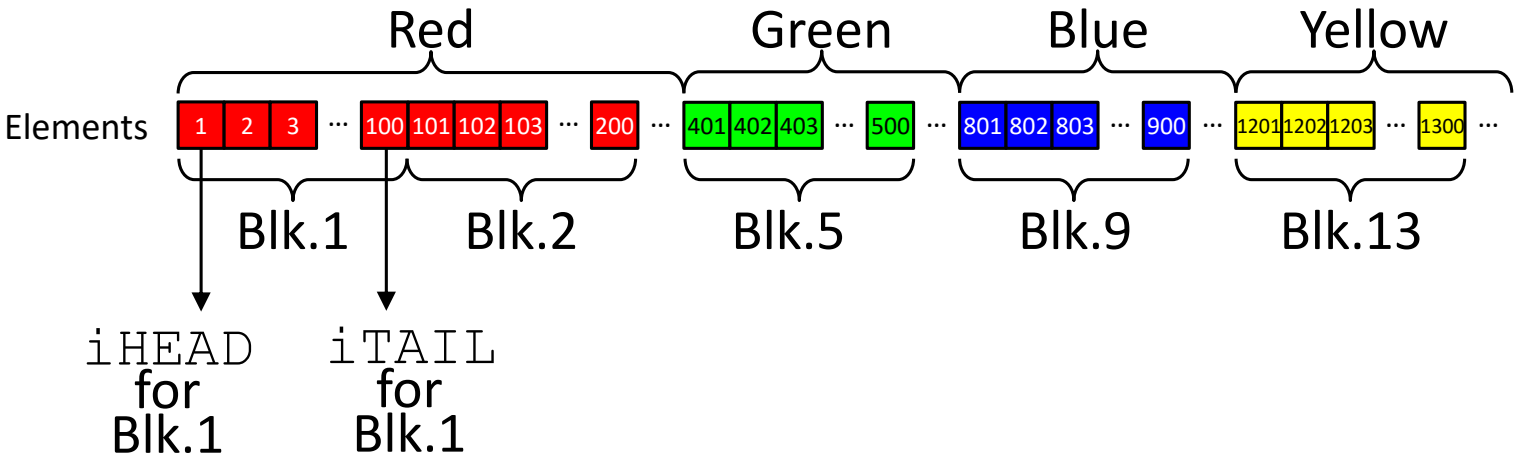
44

データ構造

waveZ



Elements were sorted and renumbered in data



性能評価

waveZ

Runnig Time	91.5	sec
Performance	6.6	GFLOPS
Ratio to 128GFLOPS	5.2	%
Memory throughput	26.8	GB/S

Peak value is 46GB/s by STREAM bench.

前ページの事例と同じように，B/Fから理想的な性能を求めると京では13% (16.7GFLOPS/NODE)出るとの見積

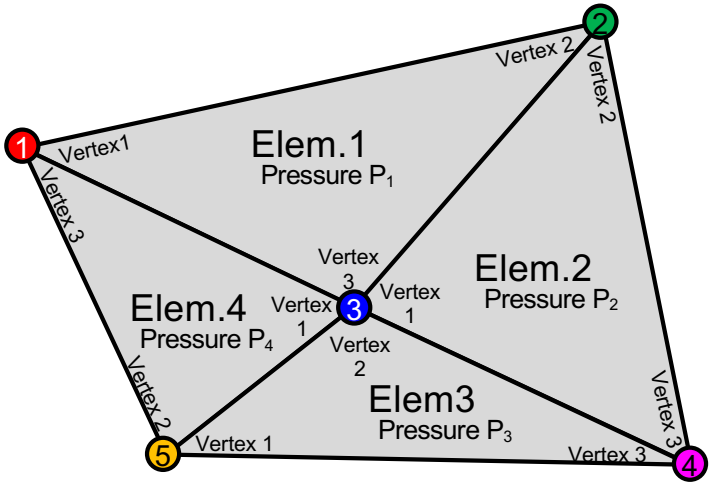
見積方法)

最内ループ1回転，つまり1要素当たり132Byte使う．一方演算量は48FLOPなので B/F=132/48=2.75

実効のメモリバンド幅はストリームベンチマークから46GB．演算性能は128GFLOPS/ノードなので，B/Fの上限はB/F=46/128=0.36

その比から 0.36/2.75=13%

何がボトルネックか？



プロファイラで実行時間内訳をみてみると

- メモリアクセス待ち時間が多かった
- ストア命令がたくさん出ていた

ストアが多かった理由

- ひとつの節点を複数の要素で共有していて，違うタイミングで同じ節点に何度もストアされる

節点への足しこみ(A=A+B)があるので

- SIMDとソフトウェアパイプラインが出来ない

①For Elem.1

Vertex 1
$$\nabla P_1 += \left\{ \frac{\partial N_{1,1}}{\partial x} P_1, \frac{\partial N_{1,1}}{\partial y} P_1 \right\}$$

Vertex 2
$$\nabla P_2 += \left\{ \frac{\partial N_{1,2}}{\partial x} P_1, \frac{\partial N_{1,2}}{\partial y} P_1 \right\}$$

Vertex 3
$$\nabla P_3 += \left\{ \frac{\partial N_{1,3}}{\partial x} P_1, \frac{\partial N_{1,3}}{\partial y} P_1 \right\}$$

②For Elem.2

$$\nabla P_3 += \left\{ \frac{\partial N_{2,1}}{\partial x} P_2, \frac{\partial N_{2,1}}{\partial y} P_2 \right\} \quad \nabla P_2 += \left\{ \frac{\partial N_{2,2}}{\partial x} P_2, \frac{\partial N_{2,2}}{\partial y} P_2 \right\}$$

$$\nabla P_4 += \left\{ \frac{\partial N_{2,3}}{\partial x} P_2, \frac{\partial N_{2,3}}{\partial y} P_2 \right\}$$

③For Elem.3

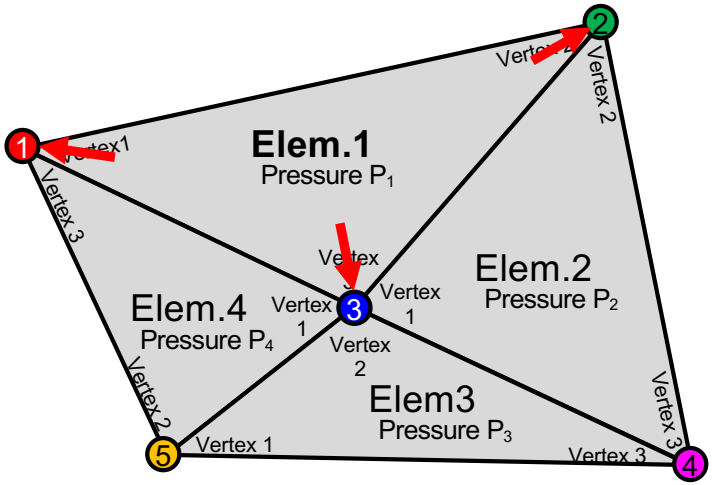
・・・略・・・

④For Elem.4

$$\nabla P_3 += \left\{ \frac{\partial N_{4,1}}{\partial x} P_4, \frac{\partial N_{4,1}}{\partial y} P_4 \right\} \quad \nabla P_5 += \left\{ \frac{\partial N_{4,2}}{\partial x} P_4, \frac{\partial N_{4,2}}{\partial y} P_4 \right\}$$

$$\nabla P_1 += \left\{ \frac{\partial N_{4,3}}{\partial x} P_4, \frac{\partial N_{4,3}}{\partial y} P_4 \right\}$$

何がボトルネックか？



プロファイラで実行時間内訳をみてみると

- メモリアクセス待ち時間が多かった
- ストア命令がたくさん出ていた

ストアが多かった理由

- ひとつの節点を複数の要素で共有していて，違うタイミングで同じ節点に何度もストアされる

節点への足しこみ(A=A+B)があるので

- SIMDとソフトウェアパイプラインが出来ない

①For **Elem.1**

Vertex 1
$$\nabla P_1 += \left\{ \frac{\partial N_{1,1}}{\partial x} P_1, \frac{\partial N_{1,1}}{\partial y} P_1 \right\}$$

Vertex 2
$$\nabla P_2 += \left\{ \frac{\partial N_{1,2}}{\partial x} P_1, \frac{\partial N_{1,2}}{\partial y} P_1 \right\}$$

Vertex 3
$$\nabla P_3 += \left\{ \frac{\partial N_{1,3}}{\partial x} P_1, \frac{\partial N_{1,3}}{\partial y} P_1 \right\}$$

②For Elem.2

$$\nabla P_3 += \left\{ \frac{\partial N_{2,1}}{\partial x} P_2, \frac{\partial N_{2,1}}{\partial y} P_2 \right\} \quad \nabla P_2 += \left\{ \frac{\partial N_{2,2}}{\partial x} P_2, \frac{\partial N_{2,2}}{\partial y} P_2 \right\}$$

$$\nabla P_4 += \left\{ \frac{\partial N_{2,3}}{\partial x} P_2, \frac{\partial N_{2,3}}{\partial y} P_2 \right\}$$

③For Elem.3

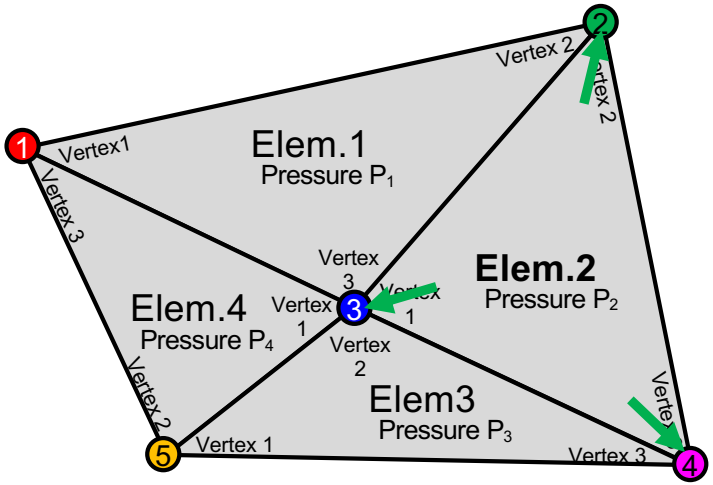
・・・略・・・

④For Elem.4

$$\nabla P_3 += \left\{ \frac{\partial N_{4,1}}{\partial x} P_4, \frac{\partial N_{4,1}}{\partial y} P_4 \right\} \quad \nabla P_5 += \left\{ \frac{\partial N_{4,2}}{\partial x} P_4, \frac{\partial N_{4,2}}{\partial y} P_4 \right\}$$

$$\nabla P_1 += \left\{ \frac{\partial N_{4,3}}{\partial x} P_4, \frac{\partial N_{4,3}}{\partial y} P_4 \right\}$$

何がボトルネックか？



プロファイラで実行時間内訳をみてみると

- メモリアクセス待ち時間が多かった
- ストア命令がたくさん出ていた

ストアが多かった理由

- ひとつの節点を複数の要素で共有していて，違うタイミングで同じ節点に何度もストアされる

節点への足しこみ(A=A+B)があるので

- SIMDとソフトウェアパイプラインが出来ない

①For Elem.1

Vertex 1 $\nabla P_1 += \left\{ \frac{\partial N_{1,1}}{\partial x} P_1, \frac{\partial N_{1,1}}{\partial y} P_1 \right\}$

Vertex 2 $\nabla P_2 += \left\{ \frac{\partial N_{1,2}}{\partial x} P_1, \frac{\partial N_{1,2}}{\partial y} P_1 \right\}$

Vertex 3 $\nabla P_3 += \left\{ \frac{\partial N_{1,3}}{\partial x} P_1, \frac{\partial N_{1,3}}{\partial y} P_1 \right\}$

②For **Elem.2**

$\nabla P_3 += \left\{ \frac{\partial N_{2,1}}{\partial x} P_2, \frac{\partial N_{2,1}}{\partial y} P_2 \right\}$ $\nabla P_2 += \left\{ \frac{\partial N_{2,2}}{\partial x} P_2, \frac{\partial N_{2,2}}{\partial y} P_2 \right\}$

$\nabla P_4 += \left\{ \frac{\partial N_{2,3}}{\partial x} P_2, \frac{\partial N_{2,3}}{\partial y} P_2 \right\}$

③For Elem.3

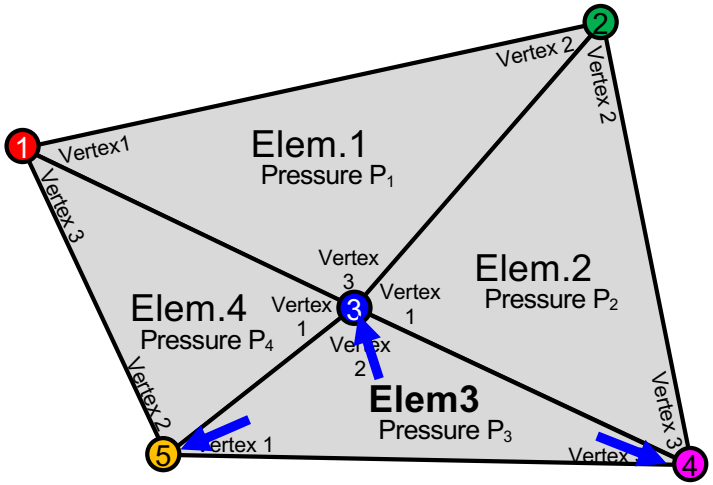
・・・略・・・

④For Elem.4

$\nabla P_3 += \left\{ \frac{\partial N_{4,1}}{\partial x} P_4, \frac{\partial N_{4,1}}{\partial y} P_4 \right\}$ $\nabla P_5 += \left\{ \frac{\partial N_{4,2}}{\partial x} P_4, \frac{\partial N_{4,2}}{\partial y} P_4 \right\}$

$\nabla P_1 += \left\{ \frac{\partial N_{4,3}}{\partial x} P_4, \frac{\partial N_{4,3}}{\partial y} P_4 \right\}$

何がボトルネックか？



プロファイラで実行時間内訳をみてみると

- メモリアクセス待ち時間が多かった
- ストア命令がたくさん出ていた

ストアが多かった理由

- ひとつの節点を複数の要素で共有していて，違うタイミングで同じ節点に何度もストアされる

節点への足しこみ(A=A+B)があるので

- SIMDとソフトウェアパイプラインが出来ない

①For Elem.1

Vertex 1
$$\nabla P_1 += \left\{ \frac{\partial N_{1,1}}{\partial x} P_1, \frac{\partial N_{1,1}}{\partial y} P_1 \right\}$$

Vertex 2
$$\nabla P_2 += \left\{ \frac{\partial N_{1,2}}{\partial x} P_1, \frac{\partial N_{1,2}}{\partial y} P_1 \right\}$$

Vertex 3
$$\nabla P_3 += \left\{ \frac{\partial N_{1,3}}{\partial x} P_1, \frac{\partial N_{1,3}}{\partial y} P_1 \right\}$$

②For Elem.2

$$\nabla P_3 += \left\{ \frac{\partial N_{2,1}}{\partial x} P_2, \frac{\partial N_{2,1}}{\partial y} P_2 \right\} \quad \nabla P_2 += \left\{ \frac{\partial N_{2,2}}{\partial x} P_2, \frac{\partial N_{2,2}}{\partial y} P_2 \right\}$$

$$\nabla P_4 += \left\{ \frac{\partial N_{2,3}}{\partial x} P_2, \frac{\partial N_{2,3}}{\partial y} P_2 \right\}$$

③For **Elem.3**

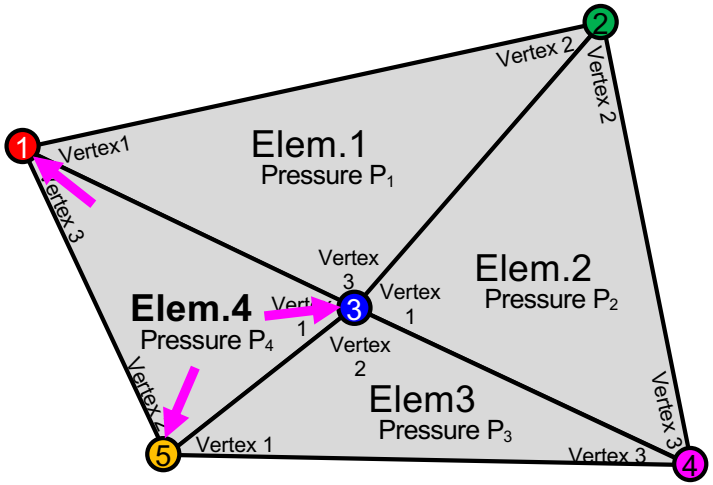
・・・略・・・

④For Elem.4

$$\nabla P_3 += \left\{ \frac{\partial N_{4,1}}{\partial x} P_4, \frac{\partial N_{4,1}}{\partial y} P_4 \right\} \quad \nabla P_5 += \left\{ \frac{\partial N_{4,2}}{\partial x} P_4, \frac{\partial N_{4,2}}{\partial y} P_4 \right\}$$

$$\nabla P_1 += \left\{ \frac{\partial N_{4,3}}{\partial x} P_4, \frac{\partial N_{4,3}}{\partial y} P_4 \right\}$$

何がボトルネックか？



プロファイラで実行時間内訳をみてみると

- メモリアクセス待ち時間が多かった
- ストア命令がたくさん出ていた

ストアが多かった理由

- ひとつの節点を複数の要素で共有していて，違うタイミングで同じ節点に何度もストアされる

節点への足しこみ(A=A+B)があるので

- SIMDとソフトウェアパイプラインが出来ない

①For Elem.1

Vertex 1
$$\nabla P_1 += \left\{ \frac{\partial N_{1,1}}{\partial x} P_1, \frac{\partial N_{1,1}}{\partial y} P_1 \right\}$$

Vertex 2
$$\nabla P_2 += \left\{ \frac{\partial N_{1,2}}{\partial x} P_1, \frac{\partial N_{1,2}}{\partial y} P_1 \right\}$$

Vertex 3
$$\nabla P_3 += \left\{ \frac{\partial N_{1,3}}{\partial x} P_1, \frac{\partial N_{1,3}}{\partial y} P_1 \right\}$$

②For Elem.2

$$\nabla P_3 += \left\{ \frac{\partial N_{2,1}}{\partial x} P_2, \frac{\partial N_{2,1}}{\partial y} P_2 \right\} \quad \nabla P_2 += \left\{ \frac{\partial N_{2,2}}{\partial x} P_2, \frac{\partial N_{2,2}}{\partial y} P_2 \right\}$$

$$\nabla P_4 += \left\{ \frac{\partial N_{2,3}}{\partial x} P_2, \frac{\partial N_{2,3}}{\partial y} P_2 \right\}$$

③For Elem.3

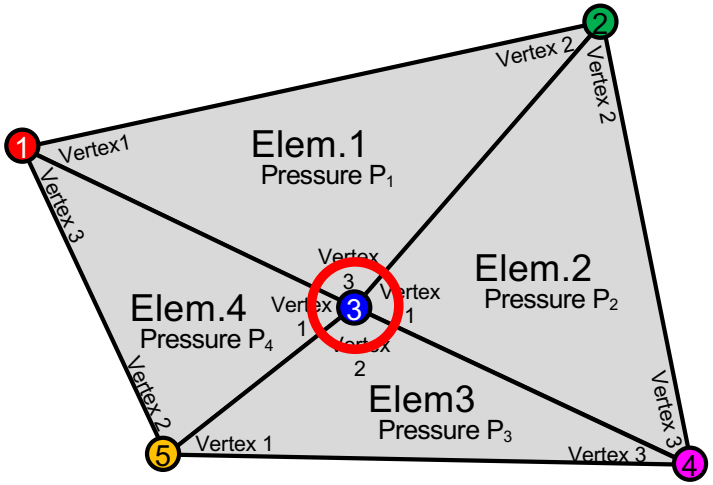
・・・略・・・

④For **Elem.4**

$$\nabla P_3 += \left\{ \frac{\partial N_{4,1}}{\partial x} P_4, \frac{\partial N_{4,1}}{\partial y} P_4 \right\} \quad \nabla P_5 += \left\{ \frac{\partial N_{4,2}}{\partial x} P_4, \frac{\partial N_{4,2}}{\partial y} P_4 \right\}$$

$$\nabla P_1 += \left\{ \frac{\partial N_{4,3}}{\partial x} P_4, \frac{\partial N_{4,3}}{\partial y} P_4 \right\}$$

何がボトルネックか？



プロファイラで実行時間内訳をみてみると

- メモリアクセス待ち時間が多かった
- ストア命令がたくさん出ていた

ストアが多かった理由

- ひとつの節点を複数の要素で共有していて，違うタイミングで同じ節点に何度もストアされる

節点への足しこみ(A=A+B)があるので

- SIMDとソフトウェアパイプラインが出来ない

①For Elem.1

Vertex 1
$$\nabla P_1 += \left\{ \frac{\partial N_{1,1}}{\partial x} P_1, \frac{\partial N_{1,1}}{\partial y} P_1 \right\}$$

Vertex 2
$$\nabla P_2 += \left\{ \frac{\partial N_{1,2}}{\partial x} P_1, \frac{\partial N_{1,2}}{\partial y} P_1 \right\}$$

Vertex 3
$$\nabla P_3 += \left\{ \frac{\partial N_{1,3}}{\partial x} P_1, \frac{\partial N_{1,3}}{\partial y} P_1 \right\}$$

②For Elem.2

$$\nabla P_3 += \left\{ \frac{\partial N_{2,1}}{\partial x} P_2, \frac{\partial N_{2,1}}{\partial y} P_2 \right\}$$

$$\nabla P_2 += \left\{ \frac{\partial N_{2,2}}{\partial x} P_2, \frac{\partial N_{2,2}}{\partial y} P_2 \right\}$$

$$\nabla P_4 += \left\{ \frac{\partial N_{2,3}}{\partial x} P_2, \frac{\partial N_{2,3}}{\partial y} P_2 \right\}$$

③For Elem.3

・・・略・・・

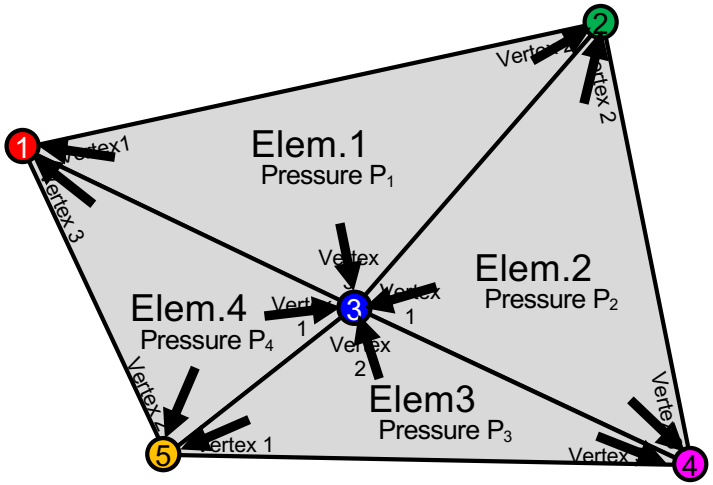
④For Elem.4

$$\nabla P_3 += \left\{ \frac{\partial N_{4,1}}{\partial x} P_4, \frac{\partial N_{4,1}}{\partial y} P_4 \right\}$$

$$\nabla P_5 += \left\{ \frac{\partial N_{4,2}}{\partial x} P_4, \frac{\partial N_{4,2}}{\partial y} P_4 \right\}$$

$$\nabla P_1 += \left\{ \frac{\partial N_{4,3}}{\partial x} P_4, \frac{\partial N_{4,3}}{\partial y} P_4 \right\}$$

ストアを連続，かつ1回だけにする



処理の流れを変更する
要素を順番に巡って，要素からの計算結果を，
要素が使っている節点に**足しこむ(A=A+B)**
↓
節点を順番に巡って，その節点を使っている
要素からの計算結果をまとめて節点に**代入(A=B)**

①節点1を処理

要素1と4の寄与
各ストア1回

$$\nabla P_1 = \begin{Bmatrix} \frac{\partial N_{1,1}}{\partial x} P_1 + \frac{\partial N_{4,3}}{\partial x} P_4 \\ \frac{\partial N_{1,1}}{\partial y} P_1 + \frac{\partial N_{4,3}}{\partial y} P_4 \end{Bmatrix}$$

②節点2を処理

要素1と2の寄与

$$\nabla P_2 = \begin{Bmatrix} \frac{\partial N_{1,2}}{\partial x} P_1 + \frac{\partial N_{2,2}}{\partial x} P_2 \\ \frac{\partial N_{1,2}}{\partial y} P_1 + \frac{\partial N_{2,2}}{\partial y} P_2 \end{Bmatrix}$$

③節点3を処理

要素1と2と3と4の寄与

$$\nabla P_3 = \begin{Bmatrix} \frac{\partial N_{1,3}}{\partial x} P_1 + \frac{\partial N_{2,1}}{\partial x} P_2 + \frac{\partial N_{3,2}}{\partial x} P_3 + \frac{\partial N_{4,1}}{\partial x} P_4 \\ \frac{\partial N_{1,3}}{\partial y} P_1 + \frac{\partial N_{2,1}}{\partial y} P_2 + \frac{\partial N_{3,2}}{\partial y} P_3 + \frac{\partial N_{4,1}}{\partial y} P_4 \end{Bmatrix}$$

④節点4を処理

要素2と3の寄与

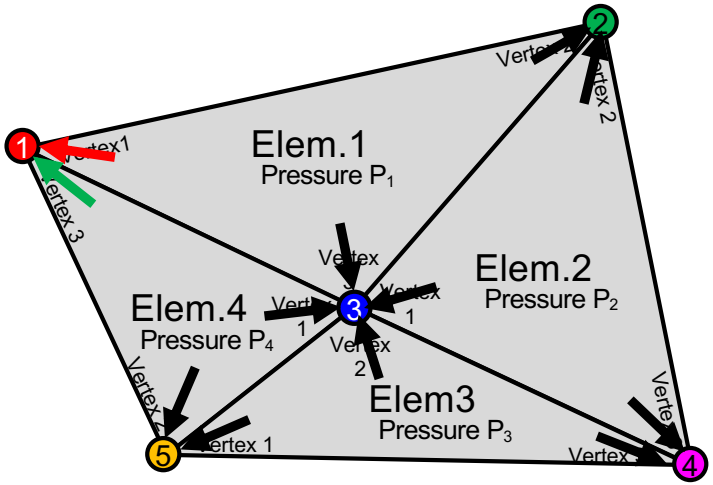
$$\nabla P_4 = \begin{Bmatrix} \frac{\partial N_{2,3}}{\partial x} P_2 + \frac{\partial N_{3,3}}{\partial x} P_3 \\ \frac{\partial N_{2,3}}{\partial y} P_3 + \frac{\partial N_{3,3}}{\partial y} P_3 \end{Bmatrix}$$

⑤節点5を処理

要素3と4の寄与

$$\nabla P_5 = \begin{Bmatrix} \frac{\partial N_{3,1}}{\partial x} P_3 + \frac{\partial N_{4,2}}{\partial x} P_4 \\ \frac{\partial N_{3,1}}{\partial y} P_3 + \frac{\partial N_{4,2}}{\partial y} P_4 \end{Bmatrix}$$

ストアを連続，かつ1回だけにする



処理の流れを変更する
要素を順番に巡って，要素からの計算結果を，
要素が使っている節点に**足しこむ(A=A+B)**
↓
節点を順番に巡って，その節点を使っている
要素からの計算結果をまとめて要素に**代入(A=B)**

①節点1を処理

要素1と4の寄与

$$\nabla P_1 = \begin{Bmatrix} \frac{\partial N_{1,1}}{\partial x} P_1 + \frac{\partial N_{4,3}}{\partial x} P_4 \\ \frac{\partial N_{1,1}}{\partial y} P_1 + \frac{\partial N_{4,3}}{\partial y} P_4 \end{Bmatrix}$$

各ストア1回

②節点2を処理

要素1と2の寄与

$$\nabla P_2 = \begin{Bmatrix} \frac{\partial N_{1,2}}{\partial x} P_1 + \frac{\partial N_{2,2}}{\partial x} P_2 \\ \frac{\partial N_{1,2}}{\partial y} P_1 + \frac{\partial N_{2,2}}{\partial y} P_2 \end{Bmatrix}$$

③節点3を処理

要素1と2と3と4の寄与

$$\nabla P_3 = \begin{Bmatrix} \frac{\partial N_{1,3}}{\partial x} P_1 + \frac{\partial N_{2,1}}{\partial x} P_2 + \frac{\partial N_{3,2}}{\partial x} P_3 + \frac{\partial N_{4,1}}{\partial x} P_4 \\ \frac{\partial N_{1,3}}{\partial y} P_1 + \frac{\partial N_{2,1}}{\partial y} P_2 + \frac{\partial N_{3,2}}{\partial y} P_3 + \frac{\partial N_{4,1}}{\partial y} P_4 \end{Bmatrix}$$

④節点4を処理

要素2と3の寄与

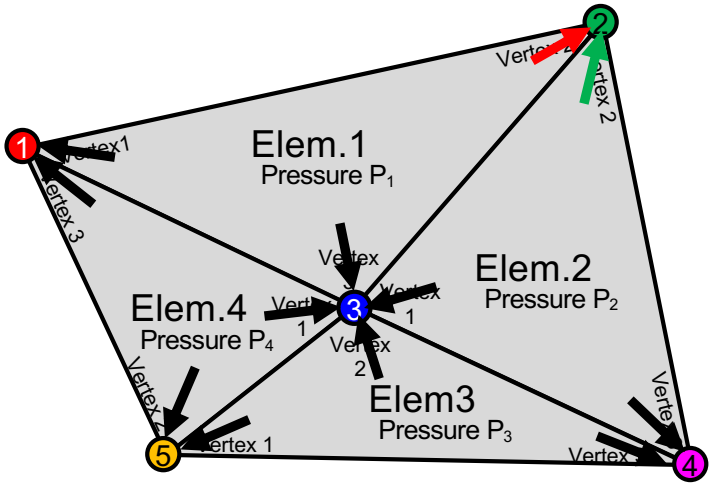
$$\nabla P_4 = \begin{Bmatrix} \frac{\partial N_{2,3}}{\partial x} P_2 + \frac{\partial N_{3,3}}{\partial x} P_3 \\ \frac{\partial N_{2,3}}{\partial y} P_3 + \frac{\partial N_{3,3}}{\partial y} P_3 \end{Bmatrix}$$

⑤節点5を処理

要素3と4の寄与

$$\nabla P_5 = \begin{Bmatrix} \frac{\partial N_{3,1}}{\partial x} P_3 + \frac{\partial N_{4,2}}{\partial x} P_4 \\ \frac{\partial N_{3,1}}{\partial y} P_3 + \frac{\partial N_{4,2}}{\partial y} P_4 \end{Bmatrix}$$

ストアを連続，かつ1回だけにする



処理の流れを変更する
要素を順番に巡って，要素からの計算結果を，
要素が使っている節点に**足しこむ(A=A+B)**
↓
節点を順番に巡って，その節点を使っている
要素からの計算結果をまとめて要素に**代入(A=B)**

①節点1を処理

$$\nabla P_1 = \begin{Bmatrix} \text{要素1と4の寄与} \\ \text{各ストア1回} \end{Bmatrix} \begin{Bmatrix} \frac{\partial N_{1,1}}{\partial x} P_1 + \frac{\partial N_{4,3}}{\partial x} P_4 \\ \frac{\partial N_{1,1}}{\partial y} P_1 + \frac{\partial N_{4,3}}{\partial y} P_4 \end{Bmatrix}$$

②節点2を処理

$$\nabla P_2 = \begin{Bmatrix} \text{要素1と2の寄与} \end{Bmatrix} \begin{Bmatrix} \frac{\partial N_{1,2}}{\partial x} P_1 + \frac{\partial N_{2,2}}{\partial x} P_2 \\ \frac{\partial N_{1,2}}{\partial y} P_1 + \frac{\partial N_{2,2}}{\partial y} P_2 \end{Bmatrix}$$

③節点3を処理

$$\nabla P_3 = \begin{Bmatrix} \text{要素1と2と3と4の寄与} \end{Bmatrix} \begin{Bmatrix} \frac{\partial N_{1,3}}{\partial x} P_1 + \frac{\partial N_{2,1}}{\partial x} P_2 + \frac{\partial N_{3,2}}{\partial x} P_3 + \frac{\partial N_{4,1}}{\partial x} P_4 \\ \frac{\partial N_{1,3}}{\partial y} P_1 + \frac{\partial N_{2,1}}{\partial y} P_2 + \frac{\partial N_{3,2}}{\partial y} P_3 + \frac{\partial N_{4,1}}{\partial y} P_4 \end{Bmatrix}$$

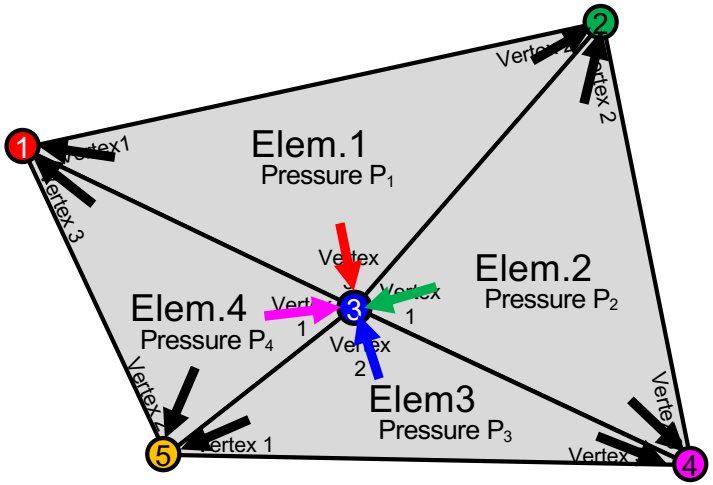
④節点4を処理

$$\nabla P_4 = \begin{Bmatrix} \text{要素2と3の寄与} \end{Bmatrix} \begin{Bmatrix} \frac{\partial N_{2,3}}{\partial x} P_2 + \frac{\partial N_{3,3}}{\partial x} P_3 \\ \frac{\partial N_{2,3}}{\partial y} P_3 + \frac{\partial N_{3,3}}{\partial y} P_3 \end{Bmatrix}$$

⑤節点5を処理

$$\nabla P_5 = \begin{Bmatrix} \text{要素3と4の寄与} \end{Bmatrix} \begin{Bmatrix} \frac{\partial N_{3,1}}{\partial x} P_3 + \frac{\partial N_{4,2}}{\partial x} P_4 \\ \frac{\partial N_{3,1}}{\partial y} P_3 + \frac{\partial N_{4,2}}{\partial y} P_4 \end{Bmatrix}$$

ストアを連続，かつ1回だけにする



処理の流れを変更する
要素を順番に巡って，要素からの計算結果を，
要素が使っている節点に**足しこむ(A=A+B)**
↓
節点を順番に巡って，その節点を使っている
要素からの計算結果をまとめて要素に**代入(A=B)**

①節点1を処理

$$\nabla P_1 = \begin{Bmatrix} \text{要素1と4の寄与} \\ \text{各ストア1回} \end{Bmatrix} \begin{Bmatrix} \frac{\partial N_{1,1}}{\partial x} P_1 + \frac{\partial N_{4,3}}{\partial x} P_4 \\ \frac{\partial N_{1,1}}{\partial y} P_1 + \frac{\partial N_{4,3}}{\partial y} P_4 \end{Bmatrix}$$

②節点2を処理

$$\nabla P_2 = \begin{Bmatrix} \text{要素1と2の寄与} \end{Bmatrix} \begin{Bmatrix} \frac{\partial N_{1,2}}{\partial x} P_1 + \frac{\partial N_{2,2}}{\partial x} P_2 \\ \frac{\partial N_{1,2}}{\partial y} P_1 + \frac{\partial N_{2,2}}{\partial y} P_2 \end{Bmatrix}$$

③節点3を処理

$$\nabla P_3 = \begin{Bmatrix} \text{要素1と2と3と4の寄与} \end{Bmatrix} \begin{Bmatrix} \frac{\partial N_{1,3}}{\partial x} P_1 + \frac{\partial N_{2,1}}{\partial x} P_2 + \frac{\partial N_{3,2}}{\partial x} P_3 + \frac{\partial N_{4,1}}{\partial x} P_4 \\ \frac{\partial N_{1,3}}{\partial y} P_1 + \frac{\partial N_{2,1}}{\partial y} P_2 + \frac{\partial N_{3,2}}{\partial y} P_3 + \frac{\partial N_{4,1}}{\partial y} P_4 \end{Bmatrix}$$

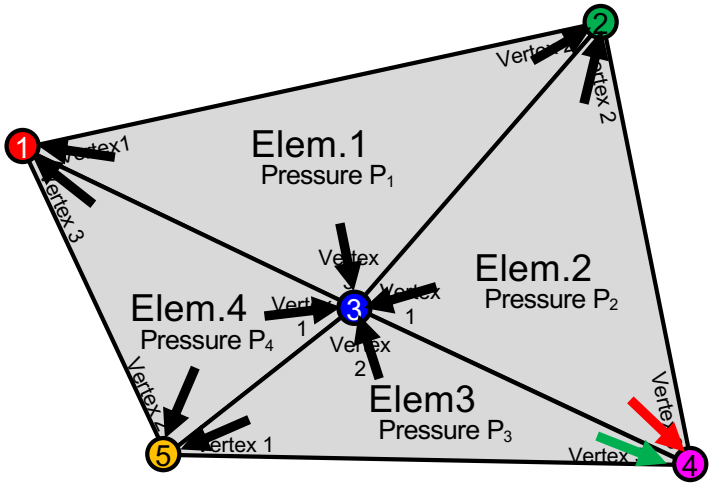
④節点4を処理

$$\nabla P_4 = \begin{Bmatrix} \text{要素2と3の寄与} \end{Bmatrix} \begin{Bmatrix} \frac{\partial N_{2,3}}{\partial x} P_2 + \frac{\partial N_{3,3}}{\partial x} P_3 \\ \frac{\partial N_{2,3}}{\partial y} P_3 + \frac{\partial N_{3,3}}{\partial y} P_3 \end{Bmatrix}$$

⑤節点5を処理

$$\nabla P_5 = \begin{Bmatrix} \text{要素3と4の寄与} \end{Bmatrix} \begin{Bmatrix} \frac{\partial N_{3,1}}{\partial x} P_3 + \frac{\partial N_{4,2}}{\partial x} P_4 \\ \frac{\partial N_{3,1}}{\partial y} P_3 + \frac{\partial N_{4,2}}{\partial y} P_4 \end{Bmatrix}$$

ストアを連続，かつ1回だけにする



処理の流れを変更する
要素を順番に巡って，要素からの計算結果を，
要素が使っている節点に**足しこむ(A=A+B)**
↓
節点を順番に巡って，その節点を使っている
要素からの計算結果をまとめて要素に**代入(A=B)**

①節点1を処理

$$\nabla P_1 = \begin{Bmatrix} \frac{\partial N_{1,1}}{\partial x} P_1 + \frac{\partial N_{4,3}}{\partial x} P_4 \\ \frac{\partial N_{1,1}}{\partial y} P_1 + \frac{\partial N_{4,3}}{\partial y} P_4 \end{Bmatrix}$$

要素1と4の寄与
各ストア1回

②節点2を処理

$$\nabla P_2 = \begin{Bmatrix} \frac{\partial N_{1,2}}{\partial x} P_1 + \frac{\partial N_{2,2}}{\partial x} P_2 \\ \frac{\partial N_{1,2}}{\partial y} P_1 + \frac{\partial N_{2,2}}{\partial y} P_2 \end{Bmatrix}$$

要素1と2の寄与

③節点3を処理

$$\nabla P_3 = \begin{Bmatrix} \frac{\partial N_{1,3}}{\partial x} P_1 + \frac{\partial N_{2,1}}{\partial x} P_2 + \frac{\partial N_{3,2}}{\partial x} P_3 + \frac{\partial N_{4,1}}{\partial x} P_4 \\ \frac{\partial N_{1,3}}{\partial y} P_1 + \frac{\partial N_{2,1}}{\partial y} P_2 + \frac{\partial N_{3,2}}{\partial y} P_3 + \frac{\partial N_{4,1}}{\partial y} P_4 \end{Bmatrix}$$

要素1と2と3と4の寄与

④節点4を処理

$$\nabla P_4 = \begin{Bmatrix} \frac{\partial N_{2,3}}{\partial x} P_2 + \frac{\partial N_{3,3}}{\partial x} P_3 \\ \frac{\partial N_{2,3}}{\partial y} P_2 + \frac{\partial N_{3,3}}{\partial y} P_3 \end{Bmatrix}$$

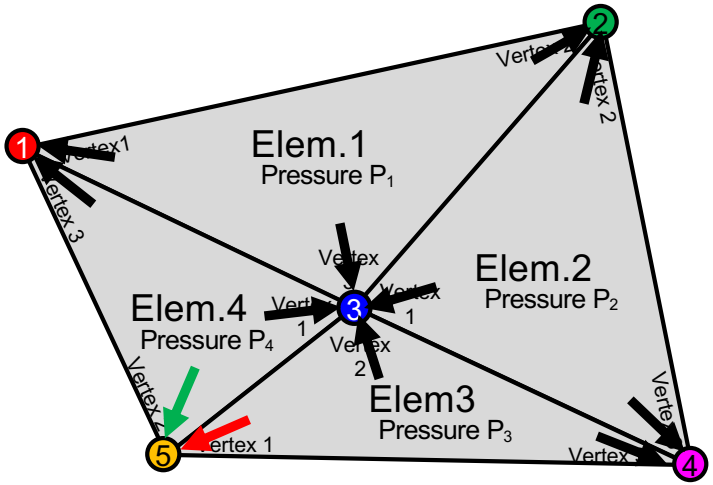
要素2と3の寄与

⑤節点5を処理

$$\nabla P_5 = \begin{Bmatrix} \frac{\partial N_{3,1}}{\partial x} P_3 + \frac{\partial N_{4,2}}{\partial x} P_4 \\ \frac{\partial N_{3,1}}{\partial y} P_3 + \frac{\partial N_{4,2}}{\partial y} P_4 \end{Bmatrix}$$

要素3と4の寄与

ストアを連続，かつ1回だけにする



処理の流れを変更する
要素を順番に巡って，要素からの計算結果を，
要素が使っている節点に**足しこむ(A=A+B)**
↓
節点を順番に巡って，その節点を使っている
要素からの計算結果をまとめて要素に**代入(A=B)**

①節点1を処理

要素1と4の寄与
$$\nabla P_1 = \begin{Bmatrix} \frac{\partial N_{1,1}}{\partial x} P_1 + \frac{\partial N_{4,3}}{\partial x} P_4 \\ \frac{\partial N_{1,1}}{\partial y} P_1 + \frac{\partial N_{4,3}}{\partial y} P_4 \end{Bmatrix}$$
各ストア1回

②節点2を処理

要素1と2の寄与
$$\nabla P_2 = \begin{Bmatrix} \frac{\partial N_{1,2}}{\partial x} P_1 + \frac{\partial N_{2,2}}{\partial x} P_2 \\ \frac{\partial N_{1,2}}{\partial y} P_1 + \frac{\partial N_{2,2}}{\partial y} P_2 \end{Bmatrix}$$

③節点3を処理

要素1と2と3と4の寄与
$$\nabla P_3 = \begin{Bmatrix} \frac{\partial N_{1,3}}{\partial x} P_1 + \frac{\partial N_{2,1}}{\partial x} P_2 + \frac{\partial N_{3,2}}{\partial x} P_3 + \frac{\partial N_{4,1}}{\partial x} P_4 \\ \frac{\partial N_{1,3}}{\partial y} P_1 + \frac{\partial N_{2,1}}{\partial y} P_2 + \frac{\partial N_{3,2}}{\partial y} P_3 + \frac{\partial N_{4,1}}{\partial y} P_4 \end{Bmatrix}$$

④節点4を処理

要素2と3の寄与
$$\nabla P_4 = \begin{Bmatrix} \frac{\partial N_{2,3}}{\partial x} P_2 + \frac{\partial N_{3,3}}{\partial x} P_3 \\ \frac{\partial N_{2,3}}{\partial y} P_3 + \frac{\partial N_{3,3}}{\partial y} P_3 \end{Bmatrix}$$

⑤節点5を処理

要素3と4の寄与
$$\nabla P_5 = \begin{Bmatrix} \frac{\partial N_{3,1}}{\partial x} P_3 + \frac{\partial N_{4,2}}{\partial x} P_4 \\ \frac{\partial N_{3,1}}{\partial y} P_3 + \frac{\partial N_{4,2}}{\partial y} P_4 \end{Bmatrix}$$

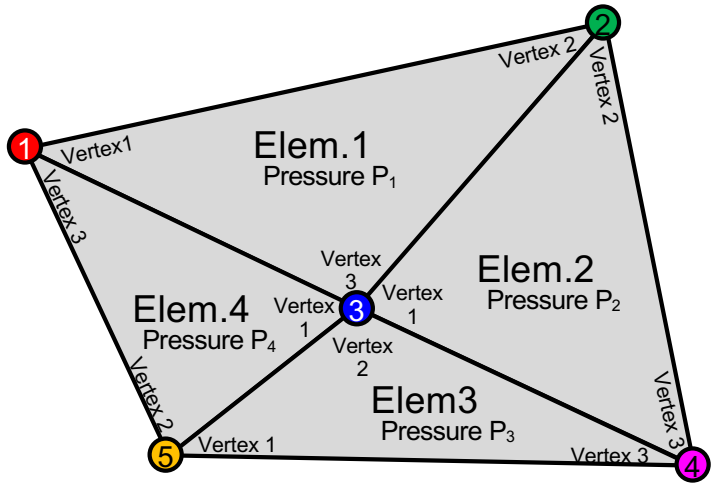
ソース変更

waveZ

ループ構造を変更した

要素→節点と辿るのではなく節点のループを回して直接節点を参照する

```
 1 DO iN=1,iNUM_NODE    Multi-Thread
 2    fTMPX=0.0
 3    fTMPY=0.0
 4    fTMPZ=0.0
 5    DO iI=1,8   節点iNを使っている上位要素の数だけ回る
 6       iE=iELEM(iI,iN)  上位要素番号
 7       iV=iERT(iI,iN)    要素iEから見た節点iNは頂点番号
 8       fP=fPRES(iE)
 9
10       fTMPX += fP * fDN(1, iV, iE)
11       fTMPY += fP * fDN(2, iV, iE)
12       fTMPZ += fP * fDN(3, iV, iE)
13    ENDDO
14    fDP(1,iN) = fTMPX
15    fDP(2,iN) = fTMPY
16    fDP(3,iN) = fTMPZ
17 ENDDO
```



ケース1) 処理の流れだけ変更

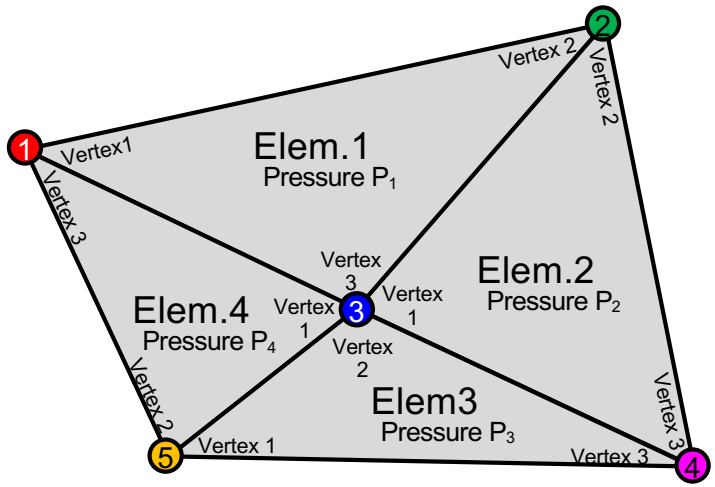
ソース変更 もう1ケース

waveZ

ループ構造を変更した

要素→節点と辿るのではなく節点のループを回して直接節点を参照する

```
 1 DO iN=1,iNUM_NODE    Multi-Thread
 2    fTMPX=0.0
 3    fTMPY=0.0
 4    fTMPZ=0.0
 5    DO iI=1,8
 6       iE=iELEM(iI,iN)
 7       fP=fPRES(iE)
 8
 9       fTMPX += fP * fDN2(1, iI, iN)
10       fTMPY += fP * fDN2(2, iI, iN)
11       fTMPZ += fP * fDN2(3, iI, iN)
12    ENDDO
13    fDP(1,iN) = fTMPX
14    fDP(2,iN) = fTMPY
15    fDP(3,iN) = fTMPZ
16 ENDDO
```



$\frac{\partial N_{iE,1}}{\partial x}$ $\frac{\partial N_{iE,8}}{\partial z}$ などの値は節点が持つようにした

ケース2) 配列fDN→fDN2と変えた

改善効果

waveZ

リスト値
iE=iELEM(iI, iN)
iV=iVERT(iI, iN)
iI, iNの値によらず同じ値とした
↓
キャッシュミスしなくなり，遅い原因がキャッシュミスなのか否かが明らかになる

測定結果

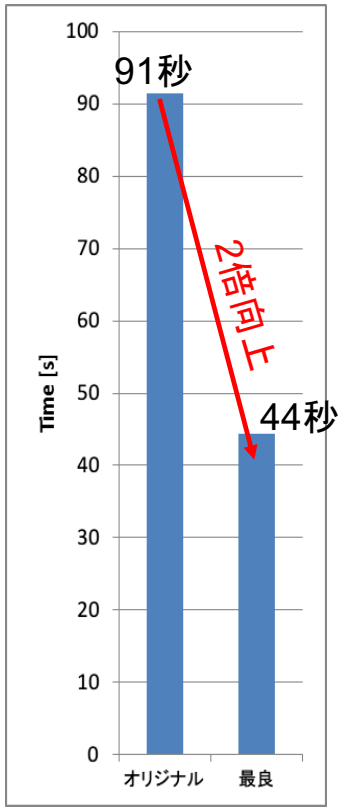
ケース	1		2		単位
パターン	A	B	A	B	
実行時間	123	135	44.3	44.2	秒
性能	5.6	5.2	15.8	15.9	GFLOPS
ピーク比	4.4	4.1	12.4	12.4	%
メモリスループット	23.2	23.2	47.2	47.2	GB/S

ケース2はほぼ理論的な性能12%に到達．一方でケース1は低い(メモリスループットも)

人工的なリスト値(全て同じ値)での測定結果

ケース	1		2		単位
パターン	A	B	A	B	
実行時間	37.4	30.2	40.9	41.7	秒
性能	18.9	23.3	17.1	16.9	GFLOPS
ピーク比	14.8	18.2	13.4	13.2	%
メモリスループット	30.9	38	48.4	47.7	GB/S

リスト値(配列iELEM,iVERT)の値を全て同一値にしてリストの性質の善し悪しの影響を排除した結果，1の低性能はリストの影響と判明．2は元々リストアクセスされる配列が少ないので，リスト値の品質の影響は小さい

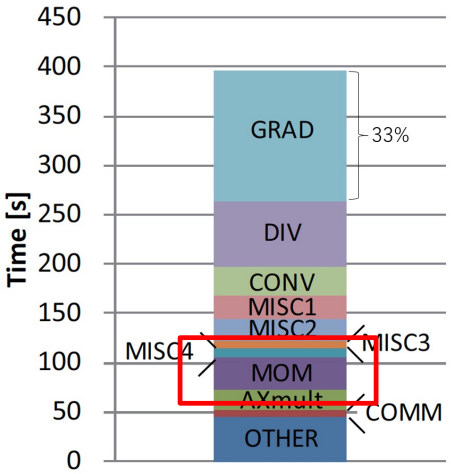


※K.Kumahata, K.Minami, Y.Yamade, C.Kato, “Performance improvement of the general-purpose CFD code Front/Flow/blue on the K computer”, HPC Asia 2018, Tokyo, Japan, (2018) https://doi.org/10.1145/3149457.3149470

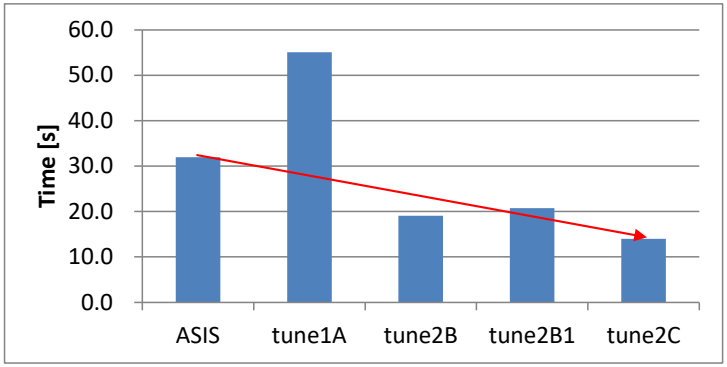
同じ改善を別カーネルへ

waveZ

アプリ中に同じ改善が適応可能なカーネルがあり，それにも適用し，実行時間半減



CASE	実行時間[秒]	向上率 1以上なら高速化	チューニング内容
ASIS	32.0	1.00	オリジナル(要素で回すループ)
tune1A	55.1	0.58	節点で回すループに変更(配列アクセスはランダム)
tune2B	19.1	1.67	tune1Aを配列を並び替えて配列アクセスを連続アクセス化
tune2B1	20.7	1.54	tune2Bで最内ループ回転数固定化
tune2C	14.0	2.29	tune2B1で最内ループ内処理のアンロール実装を再度ループ化



ソース概要

```
DO IE=1, 全ヘキサ要素
  DO I=1, ヘキサ要素の8頂点
    IP=頂点Iのグローバル節点番号
    演算(長いので略)
    節点IPへのストア
  ENDDO
ENDDO
```

配列へのストアは連続に、かつA＝A＋Bの形にしない

実アプリ

ストリーム削減・セクタキャッシュ・プリフェッチ

チューニング対象カーネル

waveZ

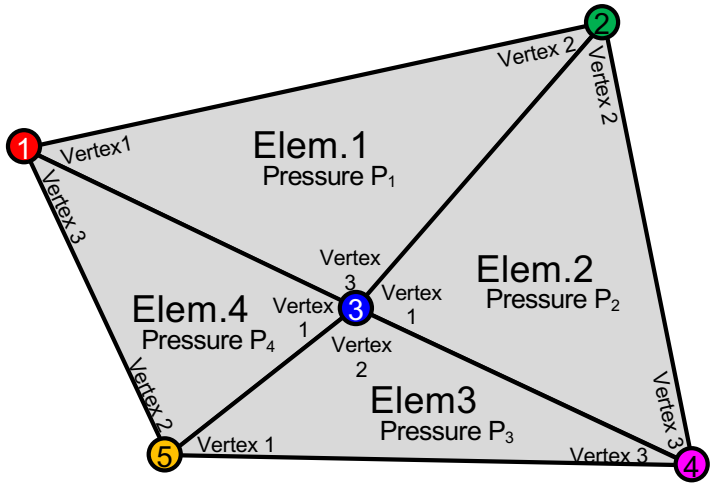
DIVのソース

```
 1 DO iE=1,iNUM_ELEM                    Multi-Thread
 2   fLP(iE)=fDNX(1,iE)*fDP(1,iNODE(1,iE))+
 3           fDNX(2,iE)*fDP(1,iNODE(2,iE))+
...          ...
 9           fDNX(8,iE)*fDP(1,iNODE(8,iE))+
10
11           fDNY(1,iE)*fDP(2,iNODE(1,iE))+
12           fDNY(2,iE)*fDP(2,iNODE(2,iE))+
...          ...
18           fDNY(8,iE)*fDP(2,iNODE(8,iE))+
19
20           fDNZ(1,iE)*fDP(3,iNODE(1,iE))+
21           fDNZ(2,iE)*fDP(3,iNODE(2,iE))+
...          ...
27           fDNZ(8,iE)*fDP(3,iNODE(8,iE))
28 ENDDO
```

処理内容

要素が頂点として参照する節点がもつ(前の事例GRAD求められた)圧力勾配ベクトルの発散を求める

$$\nabla \cdot \nabla p = \sum_{i=1}^{M} [\frac{\partial N_i}{\partial x}(\nabla p)_x^i + \frac{\partial N_i}{\partial y}(\nabla p)_y^i + \frac{\partial N_i}{\partial z}(\nabla p)_z^i]$$



※) 絵では3頂点の要素だが実際は8頂点の要素

チューニング対象カーネル

waveZ

DIVソース

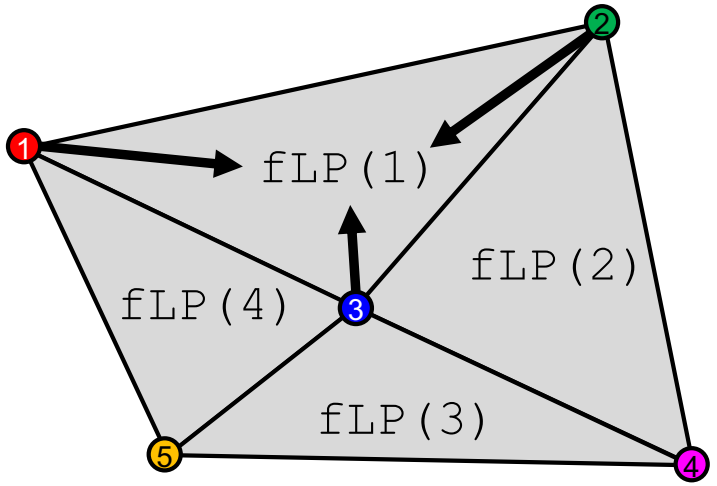
```
 1| DO iE=1,iNUM_ELEM                    Multi-Thread
 2|   fLP(iE)=fDNX(1,iE)*fDP(1,iNODE(1,iE))+
 3|            fDNX(2,iE)*fDP(1,iNODE(2,iE))+
...|            ...
 9|            fDNX(8,iE)*fDP(1,iNODE(8,iE))+
10|
11|            fDNY(1,iE)*fDP(2,iNODE(1,iE))+
12|            fDNY(2,iE)*fDP(2,iNODE(2,iE))+
...|            ...
18|            fDNY(8,iE)*fDP(2,iNODE(8,iE))+
19|
20|            fDNZ(1,iE)*fDP(3,iNODE(1,iE))+
21|            fDNZ(2,iE)*fDP(3,iNODE(2,iE))+
...|            ...
27|            fDNZ(8,iE)*fDP(3,iNODE(8,iE))
28| ENDDO
```

処理内容

要素が頂点として参照する節点がもつ(前の事例GRAD求められた)圧力勾配ベクトルの発散を求める

要素の頂点数

$$\underbrace{\nabla \cdot \nabla p}_{\text{fLP}} = \sum_{i=1}^{\overline{M}} [\underbrace{\frac{\partial N_i}{\partial x}}_{\text{fDNX}} \underbrace{(\nabla p)_x^i}_{\text{fDP}} + \underbrace{\frac{\partial N_i}{\partial y}}_{\text{fDNY}} \underbrace{(\nabla p)_y^i}_{\text{fDP}} + \underbrace{\frac{\partial N_i}{\partial z}}_{\text{fDNZ}} \underbrace{(\nabla p)_z^i}_{\text{fDP}}]$$



※) 絵では3頂点の要素だが実際は8頂点の要素

性能評価

waveZ

DIVソース

```
 1 DO iE=1,iNUM_ELEM                    Multi-Thread
 2   fLP(iE)=fDNX(1,iE)*fDP(1,iNODE(1,iE))+
 3           fDNX(2,iE)*fDP(1,iNODE(2,iE))+
...          ...
 9           fDNX(8,iE)*fDP(1,iNODE(8,iE))+
10
11           fDNY(1,iE)*fDP(2,iNODE(1,iE))+
12           fDNY(2,iE)*fDP(2,iNODE(2,iE))+
...          ...
18           fDNY(8,iE)*fDP(2,iNODE(8,iE))+
19
20           fDNZ(1,iE)*fDP(3,iNODE(1,iE))+
21           fDNZ(2,iE)*fDP(3,iNODE(2,iE))+
...          ...
27           fDNZ(8,iE)*fDP(3,iNODE(8,iE))
28 ENDDO
```

最適化状況
ストアの向きがGRADとは逆なのでデータ依存性なしでSIMD化，ソフトウェアパイプライン化は適応されている

結果

実行時間	57.4	秒
性能	10.3	GFLOPS
ピーク比	8.0	%
メモリスループット	36.8	GB/S

B/Fから求まる理想値は13%なので遅い

改善手法1

waveZ

DIVソース

```
 1 DO iE=1,iNUM_ELEM                    Multi-Thread
 2   fLP(iE)=fDNX(1,iE)*fDP(1,iNODE(1,iE))+
 3           fDNX(2,iE)*fDP(1,iNODE(2,iE))+
・・・        ・・・
 9           fDNX(8,iE)*fDP(1,iNODE(8,iE))+
10
11           fDNY(1,iE)*fDP(2,iNODE(1,iE))+
12           fDNY(2,iE)*fDP(2,iNODE(2,iE))+
・・・        ・・・
18           fDNY(8,iE)*fDP(2,iNODE(8,iE))+
19
20           fDNZ(1,iE)*fDP(3,iNODE(1,iE))+
21           fDNZ(2,iE)*fDP(3,iNODE(2,iE))+
・・・        ・・・
27           fDNZ(8,iE)*fDP(3,iNODE(8,iE))
28 ENDDO
```

改善手法

1. 配列のサイズをコンパイラに教える

上位ルーチンでは

```
real*4 :: fDNX(8、NE)
```

↓ 呼び出し

```
call DIV(・・・,fDNX,8,NE,・・・)
```

DIVルーチンでは

```
subroutine DIV(・・・,fDNX,SZ,NE,・・・)
real*4 :: fDNX(SZ,NE)
```

となっており、コンパイラは配列fDNXの1次元目の大きさが可変という前提で命令を作るが、実際には固定なので、コンパイラに教えてみる

改善手法2

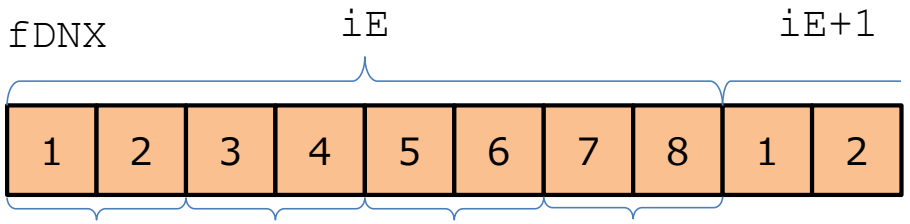
waveZ

DIVソース

```
  1 DO iE=1,iNUM_ELEM                    Multi-Thread
  2   fLP(iE)=fDNX(1,iE)*fDP(1,iNODE(1,iE))+
  3           fDNX(2,iE)*fDP(1,iNODE(2,iE))+
...           ...
  9           fDNX(8,iE)*fDP(1,iNODE(8,iE))+
 10
 11           fDNY(1,iE)*fDP(2,iNODE(1,iE))+
 12           fDNY(2,iE)*fDP(2,iNODE(2,iE))+
...           ...
 18           fDNY(8,iE)*fDP(2,iNODE(8,iE))+
 19
 20           fDNZ(1,iE)*fDP(3,iNODE(1,iE))+
 21           fDNZ(2,iE)*fDP(3,iNODE(2,iE))+
...           ...
 27           fDNZ(8,iE)*fDP(3,iNODE(8,iE))
 28 ENDDO
```

改善手法

2. SIMDロードを適応するため，アンロール実装をループに戻す



ひとつのロード命令でまとめてロードできるのだが、fDNX(1,iE)のような書き方だとやってくれないので、fDNX(j, iE)のように書けるように変更(リループ)

改善手法2

waveZ

```
DO iE=1,iNUM_ELEM          スレッド並列、SWP
   DO J=1,8                SIMD, FULLUNROLL
     tmp = tmp + fDNX(J,iE)* fDP(1, iNODE(J,iE))
               + fDNY(J,iE)* fDP(2, iNODE(J,iE))
               + fDNZ(J,iE)* fDP(3, iNODE(J,iE))
  ENDDO
   fLP(iE) = tmp
ENDDO
```

配列の1次元目がループ変数になったため、SIMDロードになるはず

改善手法3

waveZ

FLD3X2ソース

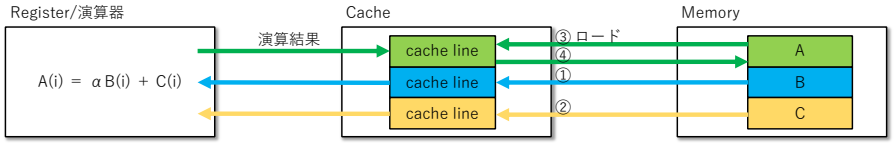
```
DO iE=1,iNUM_ELEM                    スレッド並列、SWP
   DO J=1,8                          SIMD, FULLUNROLL
     tmp = tmp + fDNXYZ(J,1,iE)* fDP(1, iNODE(J,iE))
               + fDNXYZ(J,2,iE)* fDP(2, iNODE(J,iE))
               + fDNXYZ(J,3,iE)* fDP(3, iNODE(J,iE))
  ENDDO
   fLP(iE) = tmp
ENDDO
```

改善手法

3. 配列fDNX, fDNY, fDNZをfDNXYZにまとめてストリーム削減

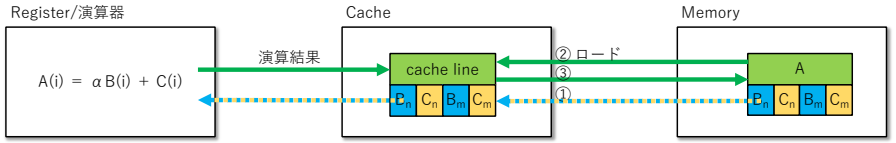
```
do I=i, N
   A(i) = A(i) + B(i) * C(i)
enddo
```

Aは書き込みだけでなく
参照もされている場合




```
do I=i, N
   A(i) = A(i) + D(1,i) * D(2,i)
enddo                    B(i)     C(i)
```

D 

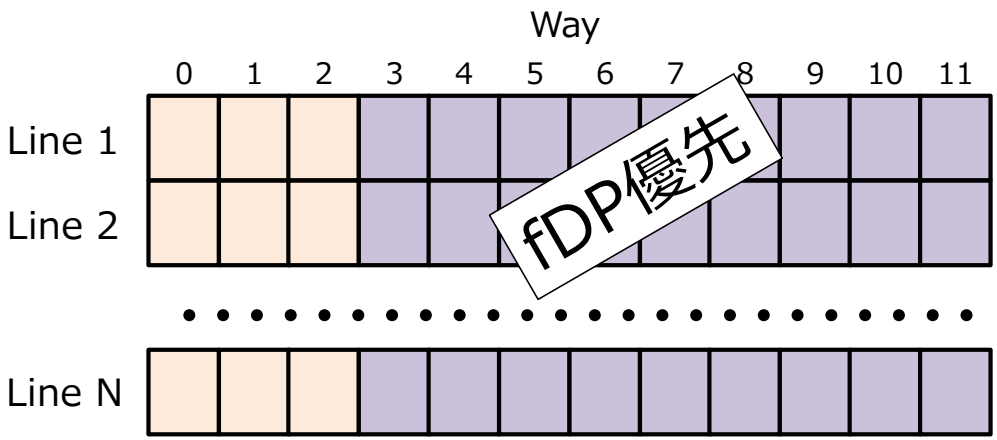


改善手法4

waveZ

FLD3X2ソース

```
!ocl cache_sector_size(3,9)
!ocl cache_subsector_assign(fDP)
DO iE=1,iNUM_ELEM                    スレッド並列、SWP
   DO J=1,8                          SIMD, FULLUNROLL
     tmp = tmp + fDNXYZ(J,iE)* fDP(1, iNODE(J,iE))
               + fDNXYZ(J,iE)* fDP(2, iNODE(J,iE))
               + fDNXYZ(J,iE)* fDP(3, iNODE(J,iE))
  ENDDO
   fLP(iE) = tmp
ENDDO
```

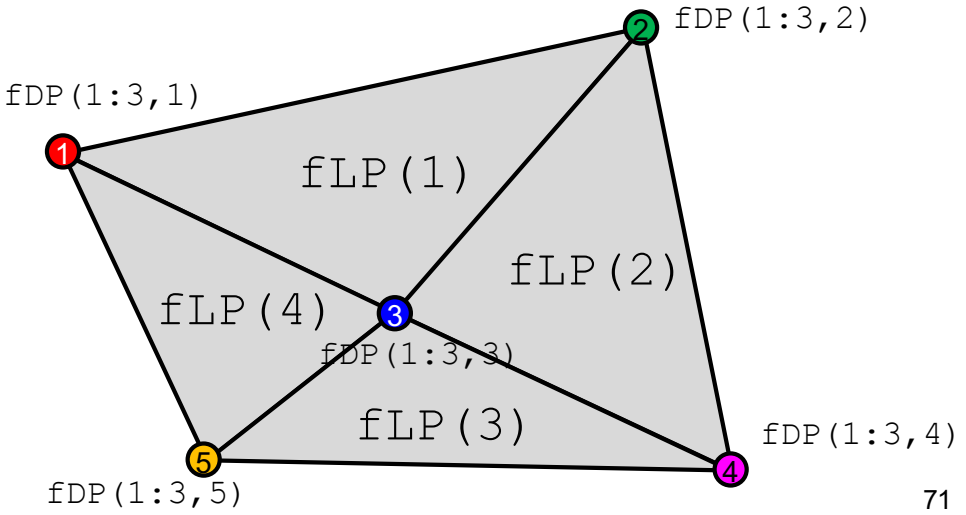


改善手法

4. セクタキャッシュを利用

配列fDPは繰り返し利用されると分かっているので優先的にキャッシュに乗っていて欲しい

L2は12ウェイあるので、3:9に分けて、9の側はfDPを優先にする



改善効果

waveZ

オリジナル

実行時間	57.4	秒
性能	10.3	GFLOPS
ピーク比	8.0	%
メモリスループット	36.8	GB/S

改善手法

1. 配列のサイズをコンパイラに教える
2. 1＋SIMDロードを適応するため，アンロール実装をループにする
3. 2＋配列`fDNX`，`fDNY`，`fDNZ`を`fDNXYZ`にまとめてストリーム削減
4. 3＋配列を「京」の機能であるセクタキャッシュに載せる

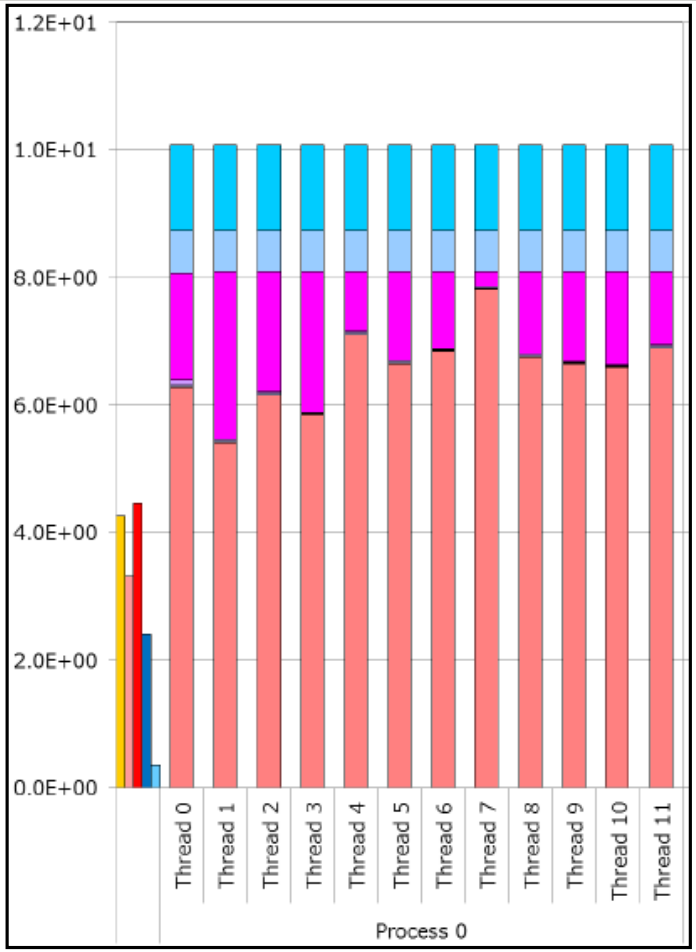
	1	2	3	4	
実行時間	52.6	49.9	49	47.5	秒
性能	11.3	12.4	12.6	13.0	GFLOPS
ピーク比	8.8	9.7	9.8	10.2	%
メモリスループット	40.3	42.4	43.2	42.5	GB/S

ソフトウェアプリフェッチ@富岳試作機

waveZ

```
!ocl cache_sector_size(3,9)
!ocl cache_subsector_assign(fDP)
DO iE=1,iNUM_ELEM                    スレッド並列、SWP
   DO J=1,8                          SIMD, FULLUNROLL
      tmp = tmp + fDNXYZ(J,1,iE)* fDP(1,iNODE(J,iE))
                + fDNXYZ(J,2,iE)* fDP(2,iNODE(J,iE))
                + fDNXYZ(J,3,iE)* fDP(3,iNODE(J,iE))
  ENDDO
   fLP(iE) = tmp
ENDDO
```

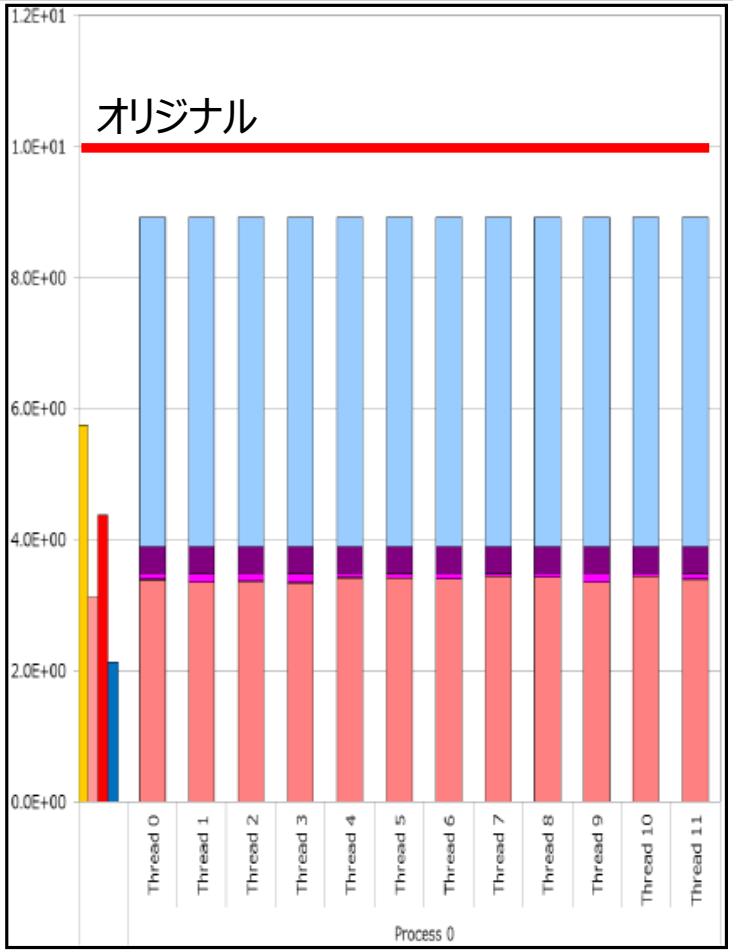
富岳試作機で測定
演算量は全てのスレッドで一定のはずが、バリア同期が発生している
↓
入力メッシュデータに依存して、スレッドごとのキャッシュミス率が異なってしまっている



ソフトウェアプリフェッチ@A64FX

waveZ

```
!ocl cache_sector_size(3,9)
!ocl cache_subsector_assign(fDP)
DO iE=1,iNUM_ELEM                    スレッド並列、SWP

!ocl prefetch_read(fDP(1,NODE(1,iE+32)),level=1)
!ocl prefetch_read(fDP(1,NODE(2,iE+32)),level=1)
!ocl prefetch_read(fDP(1,NODE(3,iE+32)),level=1)
!ocl prefetch_read(fDP(1,NODE(4,iE+32)),level=1)
!ocl prefetch_read(fDP(1,NODE(5,iE+32)),level=1)
!ocl prefetch_read(fDP(1,NODE(6,iE+32)),level=1)
!ocl prefetch_read(fDP(1,NODE(7,iE+32)),level=1)
!ocl prefetch_read(fDP(1,NODE(8,iE+32)),level=1)
DO J=1,8                             SIMD, FULLUNROLL
     tmp = tmp + fDNXYZ(J,1,iE)* fDP(1,iNODE(J,iE))
               + fDNXYZ(J,2,iE)* fDP(2,iNODE(J,iE))
               + fDNXYZ(J,3,iE)* fDP(3,iNODE(J,iE))
  ENDDO
   fLP(iE) = tmp
ENDDO
```



配列fDPのプリフェッチをパラメータスタディ
スレッドごとの打ち分けが均等になり、1秒縮んだ

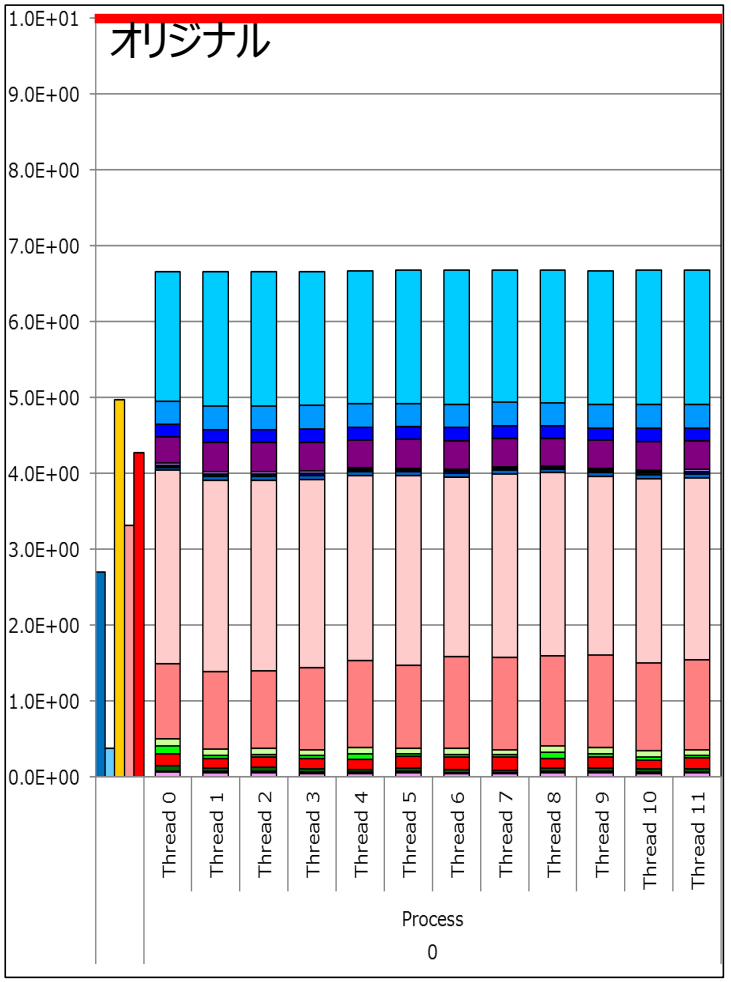
ソフトウェアプリフェッチ@A64FX

waveZ

```
!ocl cache_sector_size(3,9)
!ocl cache_subsector_assign(fDP)
DO iE=1,iNUM_ELEM                    スレッド並列、SWP

   DO J=1,8                          SIMD, FULLUNROLL
!ocl prefetch_read(fDP(1,NODE(J,IE+64), level=1)
     tmp = tmp + fDNXYZ(J,1,iE)* fDP(1,iNODE(J,iE))
               + fDNXYZ(J,2,iE)* fDP(2,iNODE(J,iE))
               + fDNXYZ(J,3,iE)* fDP(3,iNODE(J,iE))
  ENDDO
   fLP(iE) = tmp
ENDDO
```

プリフェッチの指示行をSIMDが有効になっている
ループ内へと変更し、プリフェッチもSIMD化

10.1秒 → 9.2秒 → 6.7秒



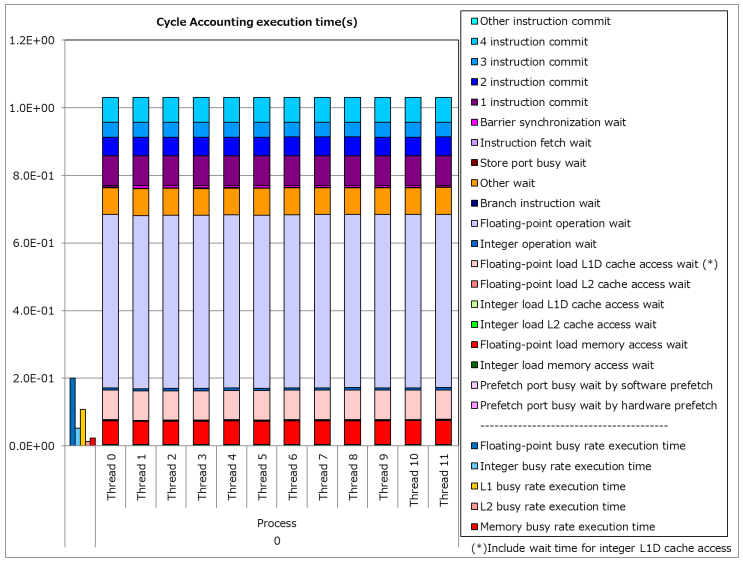
その他のTips

なるべく長いループでSIMDに

waveZ

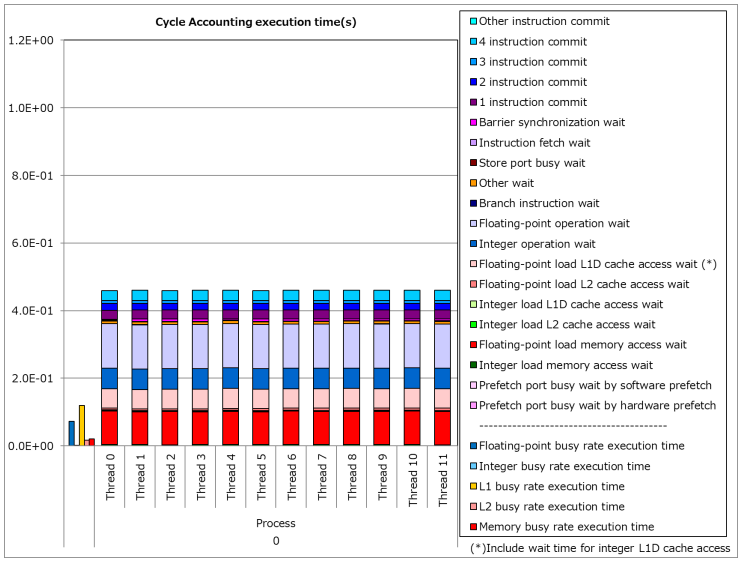
SIMD化は**最内ループ**に適応される

```
(1,2) do idx=1, max(=191700)        SWP
(3-1)   do ichem=1, spc_num(=8)     8SIMD
(3-2)   do j=1,3                    FULL UNROLL
(3-3)   do j=1, spc_num(=8)         8SIMD
```



1.03秒，3.33E+10 FLOP, SIMD率40.0%

```
(1,2) do idx=1, max(=191700)        8SIMD
(3-1)   do ichem=1, spc_num(=8)     FULL UNROLL
(3-2)   do j=1,3                    FULL UNROLL
(3-3)   do j=1, spc_num(=8)         FULL UNROLL
```

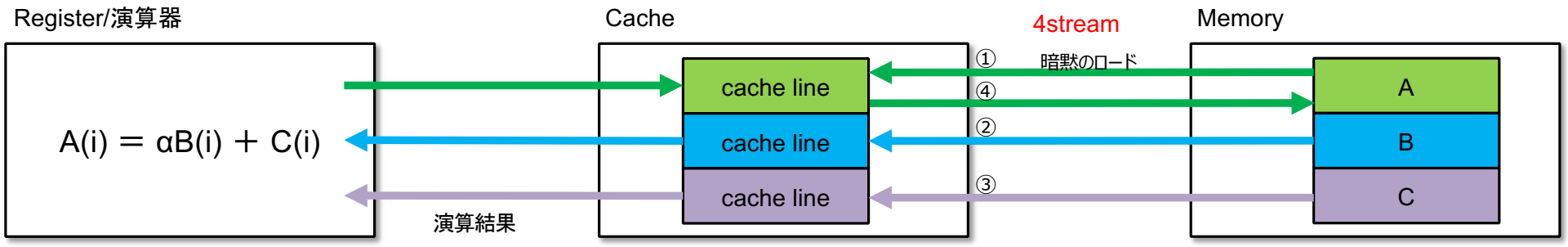


0.46秒，2.35E+10FLOP，SIMD率90.4%

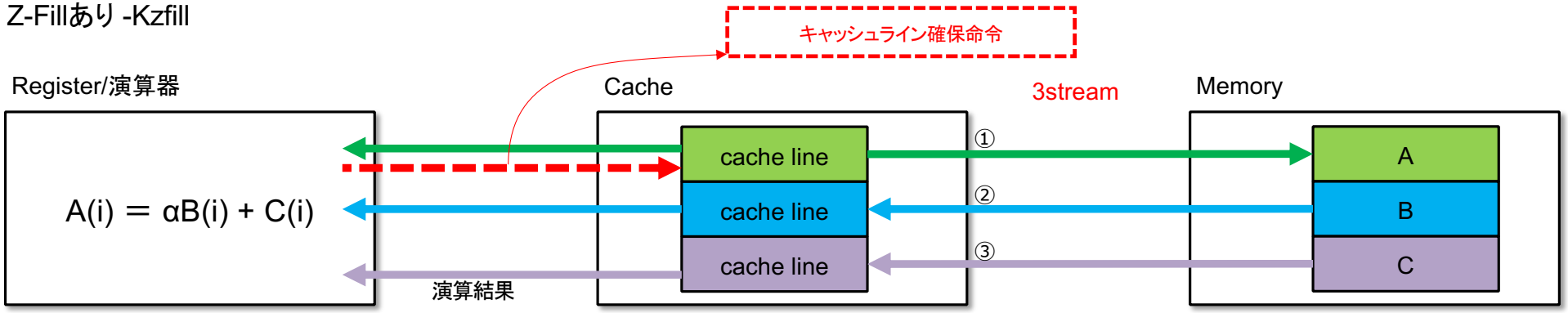
Z-Fillの効果

waveZ

Z-Fillなし



Z-Fillあり -Kzfill



Z-Fillの効果

waveZ

疑似コード

```
integer*8 :: I, SZ, ITR, NITR
parameter(SZ=104857600)
parameter(NITR=100)
real*8 :: A(SZ), B(SZ), C(SZ), alpha

start = gettime()
call FAPP_START()
do ITR=1, NITR
   do I=1, SZ
      A(I)=alpha * B(I) + C(I)
   enddo
enddo
call FAPP_STOP()
dt = start - gettime()
```

本体部分

Z-Fillにより暗黙のAのロードがなくなり
ストリームが削減し性能改善

配列ひとつのサイズ MB	case	GB	dt [s]	GB/s
800	withoutzfill	335.544	1.616	207.657
	withoutzfill_numactl	335.544	1.618	207.343
	withzfill	251.658	1.236	203.634
	withzfill_numactl	251.658	1.237	203.382
80	withoutzfill	335.544	1.601	209.577
	withoutzfill_numactl	335.544	1.623	206.767
	withzfill	251.658	1.242	202.633
	withzfill_numactl	251.658	1.241	202.788
8	withoutzfill	335.544	1.463	229.422
	withoutzfill_numactl	335.544	1.674	200.500
	withzfill	251.658	1.279	196.833
	withzfill_numactl	251.658	1.284	195.948

その他

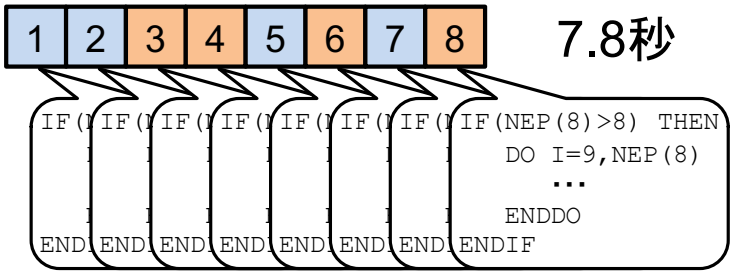
waveZ

ifを含むループはリストベクトル化

オリジナルは，全ての節点で
判定とループを実行

上位要素数8までの節点
上位要素数8以上の節点

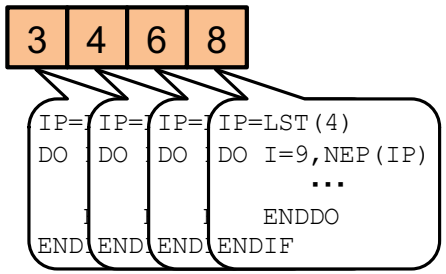
節点リスト



7.8秒

リストベクトル化
ifに合格するものだけの集合

上位要素数が8以上の節点リスト



0.0?秒

リストベクトル化後は
IFなしで，実行可能
↓
最内ループは必ず実行
され，SIMD化・SWP化
されている

配列、特に多次元配列の初期化を、配列の形に合わせたようなループでやらない

```
REAL A(N1, N2, NE)

DO IE=1, NE                マルチスレッド
  DO I=1, N2
    DO J=1, N1                        SIMD
      A(J,I,IE)=Const
    ENDDO
  ENDDO
ENDDO
```

2.2秒

C言語のmemsetを呼ぶラッパー関数を
用意
配列を8等分し，8スレッドから呼ぶ

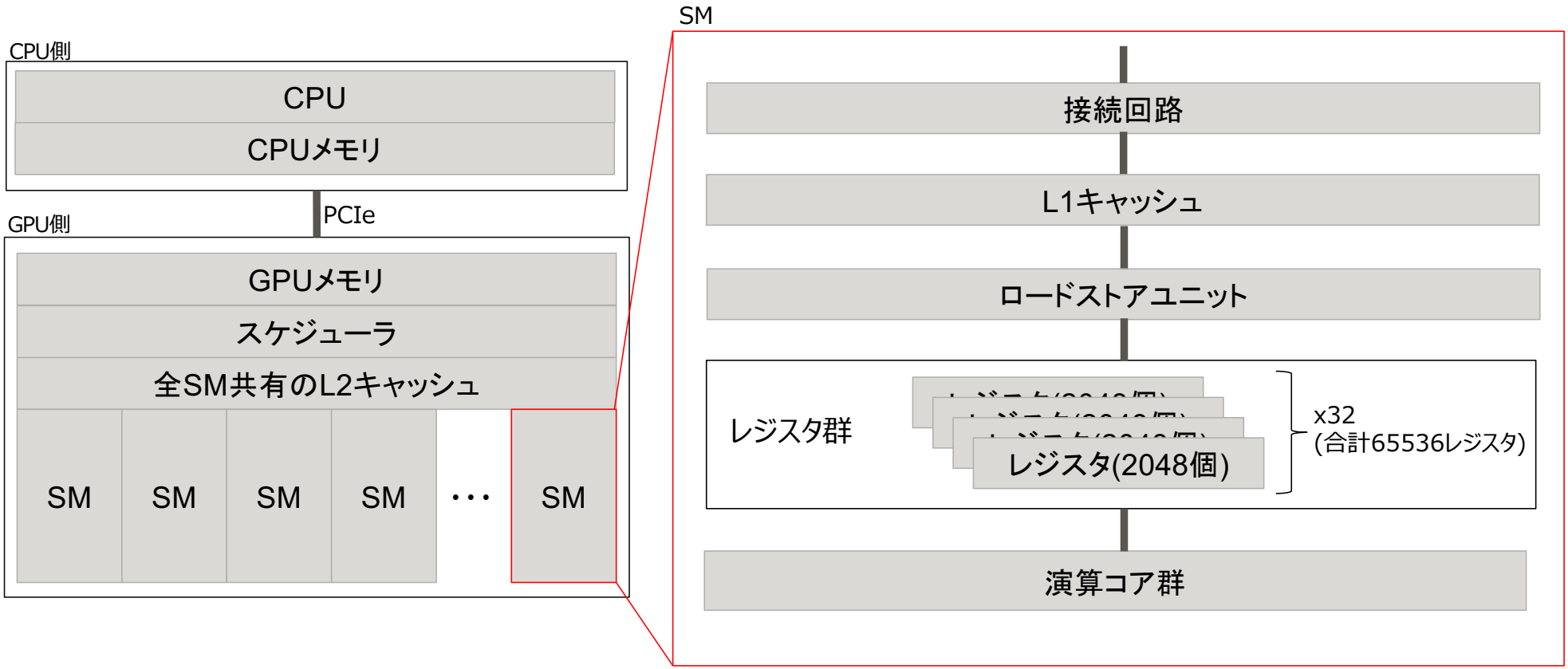
```
REAL A(N1, N2, NE)

SZ=(N1*N2*NE) / 8
DO I=8                     マルチスレッド
  ST=1+SZ*(I-1)
  call memset_wrap(A(P),Const,SZ)
ENDDO
```

0.6秒

GPUの話

GPUの大まかな構造

waveZ



※サブユニットの階層は省略

実行形態

waveZ

GPUで多数のスレッド(たとえば128)で実行したい処理

```
for(int i=0; i<128; i++){
    A[i]=B[i]+C(i)
}
```

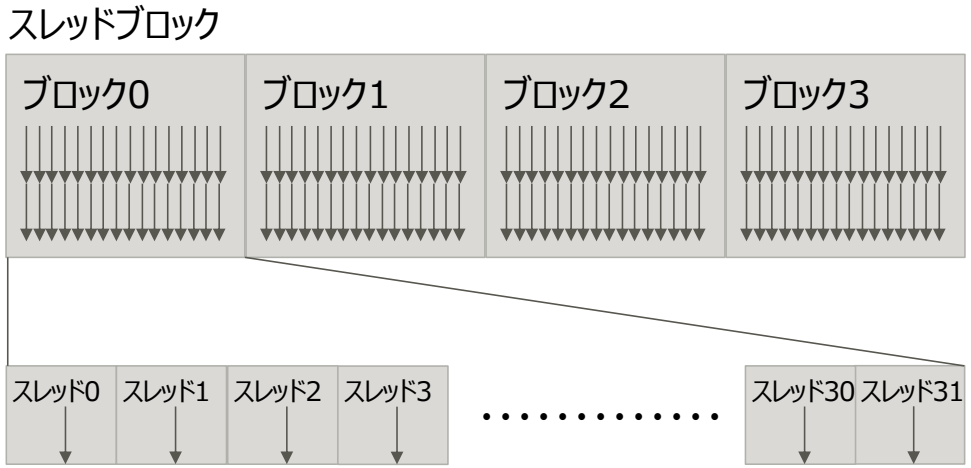
これが一つのスレッドやる処理

GPUカーネル化イメージ

```
void kernel(double *A, double* B, double *C)
{
    i = blockIdx * blockDim + threadIdx
    A[i] = B[i] + C[i]
}
```

実行

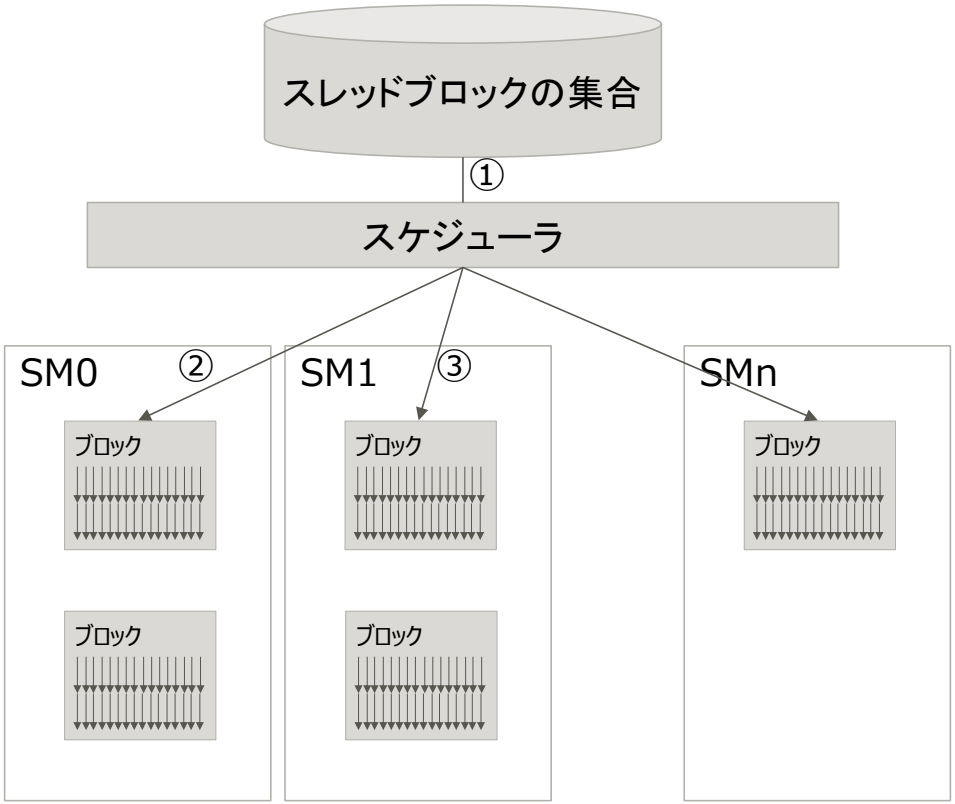
```
kernel<<<4,32>>>(A, B, C)
```



実行の流れ 1/N

waveZ

- 生成されたスレッドの集合体(＝スレッドブロック)の管理
  - スケジューラが行う



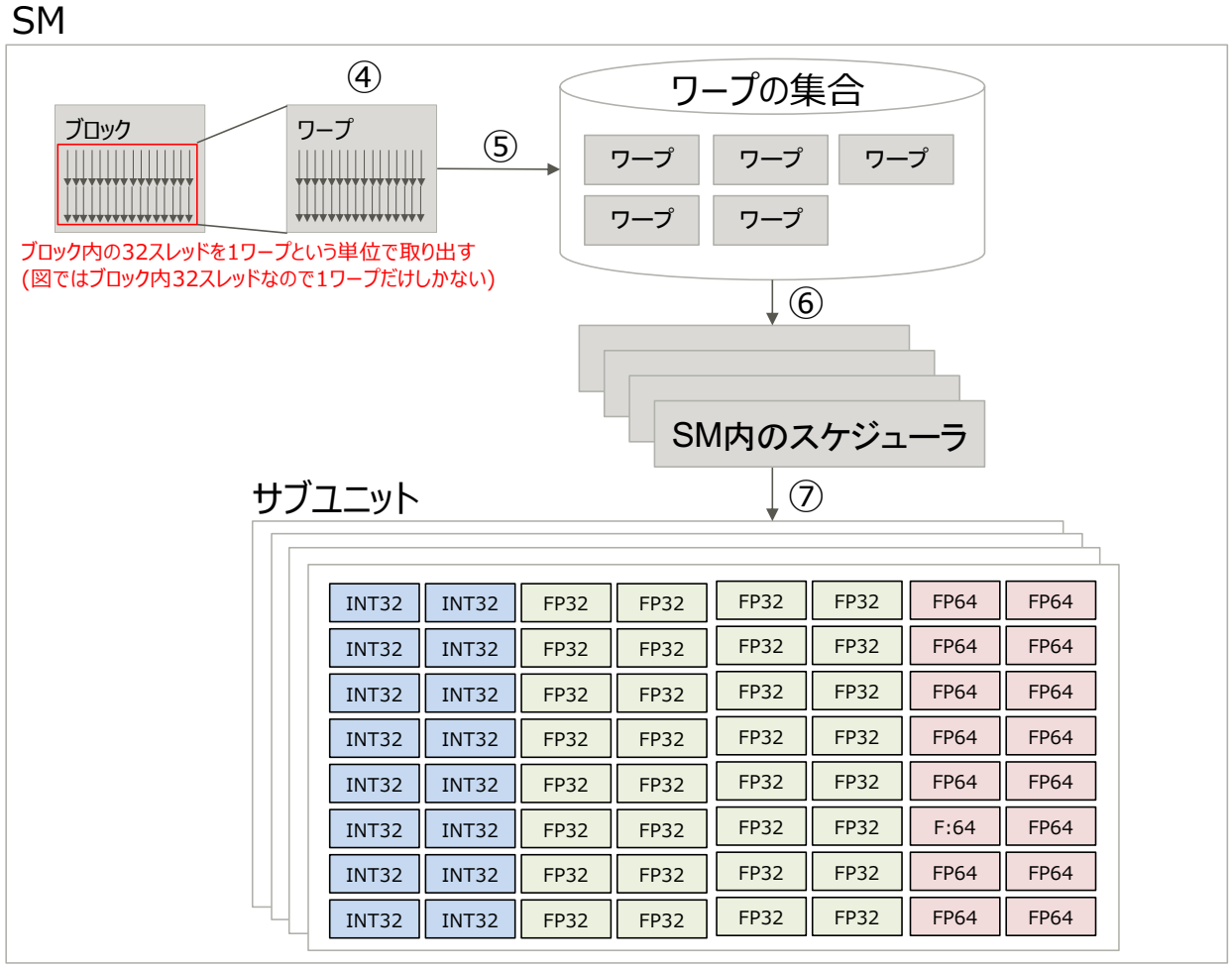
①スレッドブロックの集合体から一つスレッドブロックを取り出し

②SM0に割り付ける

③次のスレッドブロックを次のSM(図ではSM1)に割り付ける
総スレッドブロック数がSMの数より多ければまた先頭のSMから繰り返す

実行の流れ 2/N

waveZ



④スレッドブロック内のスレッドをワープ単位で取り出し

⑤ワープをワープの集合(バッファ)に入れる

⑥SM内のスケジューラがそれぞれ，ワープの集合に入っているワープ見て今実行できるものを選択
「入力オペランドが揃い」「演算器が利用可能」と書いてある

⑦選択したワープから命令を取り出して演算コアに割り当て実行

実行中ワープの実行が終わるとワーププールに空きが出来るので，新たにワープの集合に入れる
全てのワープが終わると，またスレッドブロックが割り当てられる

実行の流れ 3/N

waveZ

■平等に
今ワープの集合に入っている実行可能なワープの命令を順番に処理する(ワープの集合に入っている全ワープを少しずつ進める)

■貪欲に
さっき実行したワープの命令をまた実行する．ただしメモリ待ち，演算の依存待ちなどで待ち時間が発生したら別のワープを実行する

どちらのポリシーでも一度選択されたワープの命令が最後まで実行される訳ではない．これでレイテンシを隠蔽し何もしていない時間を極力減らす

これを繰り返すがその際のワープ選択にはいくつかのロジックがあると推察されている(左枠)

④スレッドブロック内のスレッドをワープ単位で取り出し

⑤ワープをワープの集合(バッファ)に入れる

⑥SM内のスケジューラがそれぞれ，ワープの集合に入っているワープ見て今実行できるものを選択
「入力オペランドが揃い」「演算器が利用可能」と書いてある

⑦選択したワープから命令を取り出して演算コアに割り当て実行

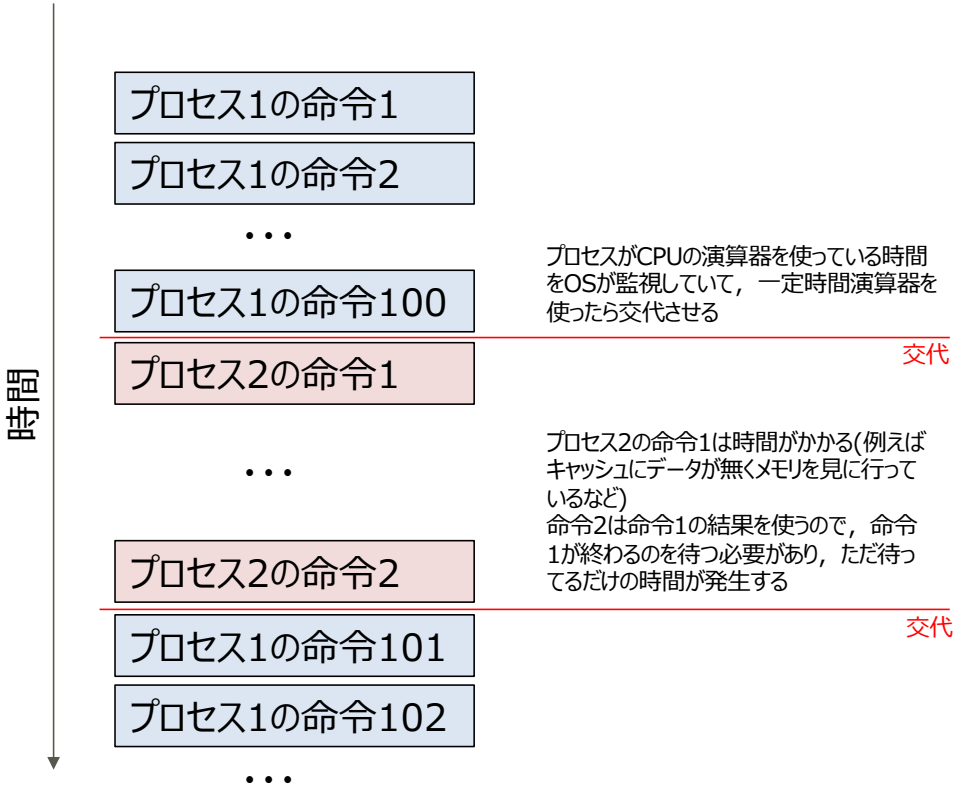
実行中ワープの実行が終わるとワーププールに空きが出来るので，新たにワープの集合に入れる

余裕ができたら，またスレッドブロックが割り当てられる

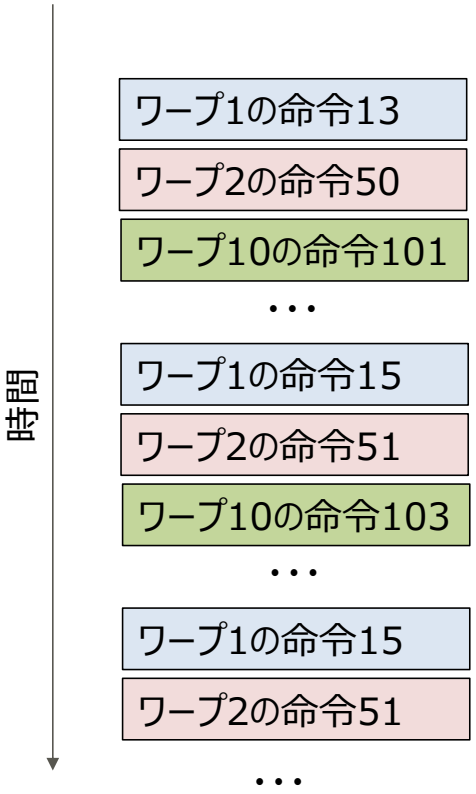
実行命令の切り替え(CPUとの違い)

waveZ

CPU(コア)



GPU(演算器)



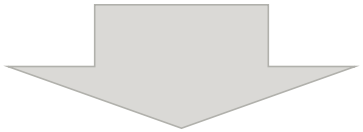
ワープスケジューラが今ワーププールにあるワープの命令を見て
今実行できる命令を演算器に割り当てる

ワーププールに入っているワープが少ないと，今実行できる命令の候補が少なく，演算器に割り当てられず，何もしていない時間が発生してしまう

条件分岐への対応

waveZ

- 実行分岐
  - グラフィックスでは来たデータに全て同じ処理してやればよい
  - 科学技術計算では実行の途中の結果に応じて続く処理が変わるのは良くあること
  - ワープ中の全スレッドが同じ命令を実行するので，ワープ中の一部のスレッドだけが分岐A，残りのスレッドが分岐Bの命令を実行することはできない



- プレディケートの導入
  - ifの判定によりどっちに流れるかを制御

ifがあるときの動作

waveZ

個々の命令実行が分岐の際にどのように動いているのかを考える

else側を実行することになったスレッドはelse側命令列にジャンプしてthen側と同時に動いているのではなく

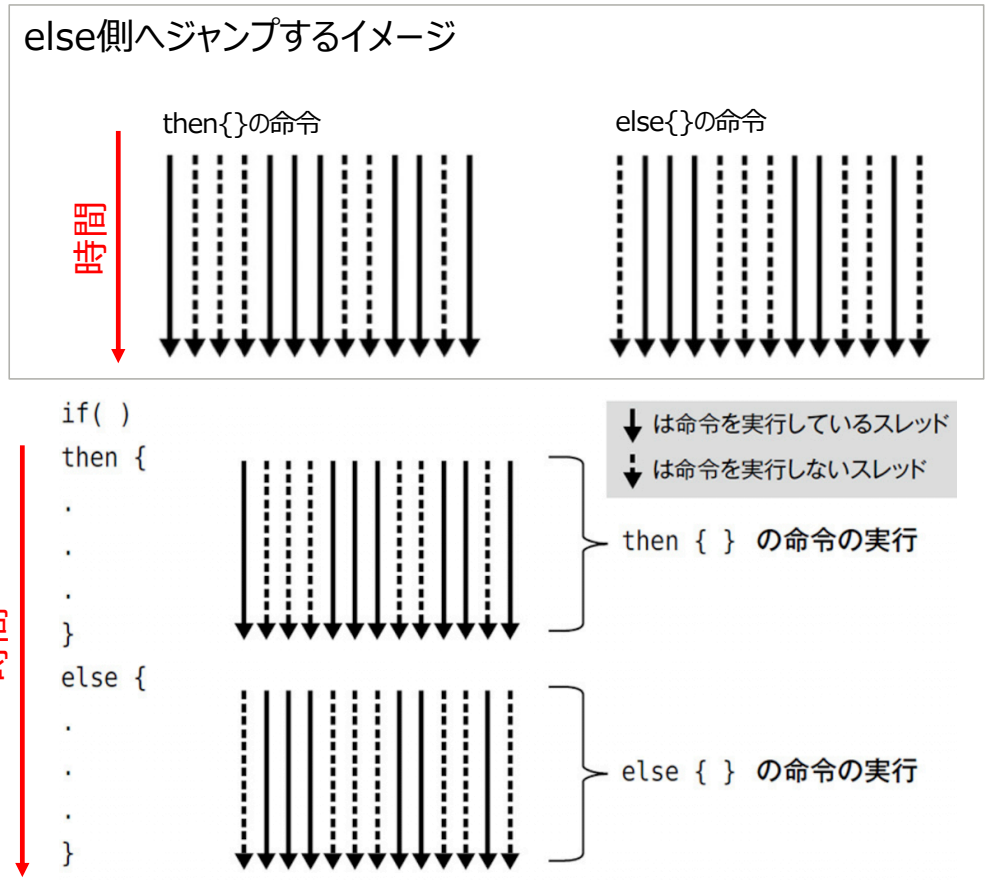
他のスレッドがthen側を実行している間は何もしない

実は，一緒にthen側を実行するが，エラー(参照先アドレスがないとか0割発生などの例外)が出ても無視し，計算結果をストア先に書かない

then側実行が終わるとその次にelse側を実行するステップになるのでelse側を実行する

今度はthen側のスレッドが何もしない

if(とelse)があると全スレッドがif(とelse)の命令を実行するので時間がかかるので避けたい



実行時間は thenの実行分とelseの実行分の両方かかる

まとめ

waveZ

- これまでに実アプリケーションの単体性能チューニングを行った事例を紹介しました
  - キャッシュ利用効率の改善・スレッド並列化の方法の工夫
  - ストアのシーケンシャル化
  - ストリーム削減・セクタキャッシュ・プリフェッチ
- その他細かいTips
- GPUの話


waveZ