大量のSPILL発生をプログラム修正により抑止

本資料では「富岳」におけるSPILL軽減による実行性能の高速化の事例を紹介します。基本プロファイラfippを用いてループの実行時間を測定し、プログラムを修正することで実行時間を改善、高速化を実現しています。

2023年 03月
高度情報科学技術研究機構 (RIST)

もくじ

性能確認環境
- 対象システム　：「富岳」
- 言語環境　：　lang/tcsds-1.2.30a
- 言語種別　：　C++
- 翻訳時オプション：-Kfast,ocl,openmp,optmsg=2 -Nlst=t
- 対象ソフトウェア：　ユーザプログラム
- 支援期間：　2021年11月 ～ 2021年12月

プログラム(original)コードの翻訳情報からの分析

original　lstファイル

```
389             /*****↓
390                       for( int jb = 0; jb < BLOCK*BLOCK*BLOCK/16; jb++ ){↓
391                       #pragma omp simd↓
392                       for( int jxx = 0; jxx < 16; jxx++ ){↓
393             *****/↓
394                       #pragma omp simd↓
                 <<< Loop-information Start >>>↓
                 <<<  [OPTIMIZATION]↓
                 <<<    SIMD(VL: 8)↓
                 <<<    PREFETCH(HARD) Expected by compiler :↓
                 <<<      (unknown)↓
                 <<<    PREFETCH(SOFT) : 22↓
                 <<<     SEQUENTIAL : 22↓
                 <<<      (unknown): 22↓
                 <<<    SPILLS :↓
                 <<<      GENERAL    : SPILL 0   FILL 47↓
                 <<<      SIMD&FP    : SPILL 0   FILL 0↓
                 <<<      SCALABLE   : SPILL 174   FILL 406↓
                 <<<      PREDICATE : SPILL 0   FILL 0↓
                 <<< Loop-information  End >>>↓
395           v           for( int j = 0; j < BLOCK*BLOCK*BLOCK; j++ ){↓
396           v               const int jx = j % BLOCK;↓
397           v               const int jy = ( j % ( BLOCK*BLOCK ) ) / BLO
398           v               const int jz = j / ( BLOCK*BLOCK );↓
399               ↓
400                           //-- streaming ↓
401           v               real f[Q] = {};↓
402           v               f[0 ] = lbm_val.f[0 ][ OFFNEI(jx+1, jy+1, jz+1) ];
403           v               f[1 ] = lbm_val.f[1 ][ OFFNEI(jx+0, jy+1, jz+1) ];
404           v               f[2 ] = lbm_val.f[2 ][ OFFNEI(jx-1, jy+1, jz+1) ];
```

翻訳情報をlstファイルに出力するにはC言語使用手引書2.2.2.6参照

大量のSPILLが発生

→ 考えられる原因

1. ループボディが大きい
2. ループ内に多数の変数、配列が存在それにより、データのロード・ストアが発生

考えられる原因(1, 2)を解決するため
次ページ以降 v1～v8 の対策を順に行う

j-loopのリロール（1重ループ化） (v1)

original

```
for( int jb = 0; jb < BLOCK*BLOCK*BLOCK/16; jb++ ){
#pragma omp simd
for( int jxx = 0; jxx < 16; jxx++ ){
    const int j = jxx + jb*16;
    const int jx = j % BLOCK;
    const int jy = ( j % ( BLOCK*BLOCK ) ) / BLOCK;
    const int jz = j / ( BLOCK*BLOCK );

    //-- streaming
    real f[Q] = {};
    f[0 ] = lbm_val.f[0 ][ OFFNEI(jx+1, jy+1, jz+1) ]
    f[1 ] = lbm_val.f[1 ][ OFFNEI(jx+0, jy+1, jz+1) ]
```

originalのコードでは、ループ長が16と短いため、SIMD演算器の性能を活かすことができていない。

v1

```
#pragma omp simd
for( int j = 0; j < BLOCK*BLOCK*BLOCK; j++ ){
    const int jx = j % BLOCK;
    const int jy = ( j % ( BLOCK*BLOCK ) ) / BLOCK;
    const int jz = j / ( BLOCK*BLOCK );

    //-- streaming
    real f[Q] = {};
    f[0 ] = lbm_val.f[0 ][ OFFNEI(jx+1, jy+1, jz+1) ]
    f[1 ] = lbm_val.f[1 ][ OFFNEI(jx+0, jy+1, jz+1) ]
    .......
```

経過時間

original	190.391 [sec]
v1	183.765 [sec]

ループ分割　その1 (v2)

Section 1

```
//-- streaming
real f[Q] = {};
f[0 ] = lbm_val.f[0 ][ OFFNEI(jx+1, jy+1, jz+1) ];
f[1 ] = lbm_val.f[1 ][ OFFNEI(jx+0, jy+1, jz+1) ];
.....

//-- forward central moment
.....
k101 = um* wm*f[0 ] + -uc* wm*f[1 ] + .....
k011 = vm* wm*f[0 ] +  vm* wm*f[1 ] + .....
.....
```

```
//-- forward cumulant
real c000, c200, c020, c002, c110, c101, c011;
c000 = k000;
c200 = k200;
c020 = k020;
c002 = k002;
......
//-- cumulant collision
......
c020 = c200 - cs200_020;
c002 = c200 - cs200_002;
```

ループ分割位置1
⇒Section 2

```
//-- backward cumulant
real ks[3][3][3];
ks[0][0][0] = c000;
ks[1][0][0] = -k100;
ks[0][1][0] = -k010;
ks[0][0][1] = -k001;
........

//-- store
lbm_val.fn[0 ][ offset*BLOCK_CUBE + j ] = f[0 ];
lbm_val.fn[1 ][ offset*BLOCK_CUBE + j ] = f[1 ];
.....
```

ループ分割位置2
⇒　Section 3

ワーク配列の使用:

```
real rhoc_j[BLOCK*BLOCK*BLOCK];
real   ir_j[BLOCK*BLOCK*BLOCK];
real   uc_j[BLOCK*BLOCK*BLOCK];
real   vc_j[BLOCK*BLOCK*BLOCK];
real   wc_j[BLOCK*BLOCK*BLOCK];

real k101[BLOCK*BLOCK*BLOCK];
....
real k222[BLOCK*BLOCK*BLOCK];
```

各ループの計算内容
Section1: fのデータロード、
　　k000 ... k222の計算
Section2: c000 ... c222の計算
Section3: テンソルksの計算
　　lvm_val.fnへのデータストア

1ループ内のロード・ストアを減らすループ分割位置(Section)数については数十パターンの測定を行い最善をみつけた

v1（ループ分割前）	183.765 [sec]
v2（ループ分割後）	157.857 [sec]

基本プロファイラ　測定結果 (for v2)

関数 stream_collision_cumの箇所のみを測定

```
Loops profile (Total thread cost basis)

*****************************************************************************************
    Application - loops
      Application and Process outputs the total value of the cost of each thread.
      Procedure outputs the total value of the loop cost of each thread.
*****************************************************************************************

          Cost        %  Operation (s)       Barrier        %    Nest    Kind    Exec    Start      End
-----------------------------------------------------------------------------------------------------
       1303645  100.0000     13157.2402         40280   3.0898      --      --
-----------------------------------------------------------------------------------------------------
        984563   75.5239      9937.3311             0   0.0000       2   FOR     OpenMP      415      522  stream_collision_cum(LE
        215759   16.5504      2177.2732             0   0.0000       2   FOR     OpenMP      578      684  stream_collision_cum(LE
         91539    7.0218       923.7029         40188   3.0827       1   FOR     OpenMP      376      686  stream_collision_cum(LE
          9253    0.7098        93.4102             0   0.0000       2   FOR     OpenMP      525      575  stream_collision_cum(LE
           383    0.0294         3.8636             0   0.0000       2   FOR     OpenMP      385      390  stream_collision_cum(LE
```

Section 1が主なホットスポット

v2の415行のforループの最適化情報

（※　Section3: 578-684）

```
            #pragma omp simd
    <<< Loop-information Start >>>
    <<<  [OPTIMIZATION]
    <<<    SIMD(VL: 8)
    <<<    PREFETCH(HARD) Expected by compiler :
    <<<      rhoc_j, ir_j, uc_j, vc_j, wc_j, k000, k100
    <<<      k010, k001, k200, k020, k002, k110, k101, k011
    <<<    SPILLS :
    <<<      GENERAL   : SPILL 13  FILL 23
    <<<      SIMD&FP   : SPILL 0  FILL 0
    <<<      SCALABLE  : SPILL 80  FILL 181
    <<<      PREDICATE : SPILL 0  FILL 0
    <<< Loop-information  End >>>
v             for( int j = 0; j < BLOCK*BLOCK*BLOCK; j++ ){
v                 const int jx = j % BLOCK;
v                 const int jy = ( j % ( BLOCK*BLOCK ) ) / BLOCK;
v                 const int jz = j / ( BLOCK*BLOCK );

                  //-- streaming
v                 real f[Q] = {};
v                 f[0 ] = lbm_val.f[0 ][ OFFNEI(jx+1, jy+1, jz+1) ];
v                 f[1 ] = lbm_val.f[1 ][ OFFNEI(jx+0, jy+1, jz+1) ];
v                 f[2 ] = lbm_val.f[2 ][ OFFNEI(jx-1, jy+1, jz+1) ];
```

lbm_val.f[.][.]のデータロードについて、
ワーク配列f_jを用意して、ループ分割を行う。
（各jごとにまとめて、f[0] ... f[25]を計算）

配列成分が不連続

ループ分割　その2 (v3)

```
real f_j[Q][BLOCK*BLOCK*BLOCK];

//-- streaming
// #pragma omp simd
for( int j = 0; j < BLOCK*BLOCK*BLOCK; j++ ){
    const int jx = j % BLOCK;
    const int jy = ( j % ( BLOCK*BLOCK ) ) / BLOCK;
    const int jz = j / ( BLOCK*BLOCK );

    f_j[0 ][j] = lbm_val.f[0 ][ OFFNEI(jx+1, jy+1, jz+1) ];
}
// #pragma omp simd
for( int j = 0; j < BLOCK*BLOCK*BLOCK; j++ ){
    const int jx = j % BLOCK;
    const int jy = ( j % ( BLOCK*BLOCK ) ) / BLOCK;
    const int jz = j / ( BLOCK*BLOCK );

    f_j[1 ][j] = lbm_val.f[1 ][ OFFNEI(jx+0, jy+1, jz+1) ];
}
....
#pragma omp simd
for( int j = 0; j < BLOCK*BLOCK*BLOCK; j++ ){
    real f[Q] = {};

    f[0 ] = f_j[0 ][j];
    f[1 ] = f_j[1 ][j];
    ....
```

#pragma omp simd をコメント**(無効)**にすることで最適化促進

SWP化、loop融合

```
p                   //-- streaming
                    //  #pragma omp simd
                     <<< Loop-information Start >>>
                     <<<  [OPTIMIZATION]
                     <<<    FUSED(lines: 417,425)
                     <<<    SIMD(VL: 16)
                     <<<    SOFTWARE PIPELINING(IPC: 2.00, ITR: 96, MVE: 2, POL: S)
                     <<<    PREFETCH(HARD) Expected by compiler :
                     <<<      f_j
                     <<< Loop-information  End >>>
p       v             for( int j = 0; j < BLOCK*BLOCK*BLOCK; j++ ){
p       v                 const int jx = j % BLOCK;
p       v                 const int jy = ( j % ( BLOCK*BLOCK ) ) / BLOCK;
p       v                 const int jz = j / ( BLOCK*BLOCK );

p       v                 f_j[0 ][j] = lbm_val.f[0 ][ OFFNEI(jx+1, jy+1, jz+1) ];
p       v             }
                    //  #pragma omp simd
                     <<< Loop-information Start >>>
                     <<<  [OPTIMIZATION]
                     <<<    FUSED
                     <<<    PREFETCH(HARD) Expected by compiler :
                     <<<      (unknown)
                     <<< Loop-information  End >>>
p       v             for( int j = 0; j < BLOCK*BLOCK*BLOCK; j++ ){
p       v                 const int jx = j % BLOCK;
p       v                 const int jy = ( j % ( BLOCK*BLOCK ) ) / BLOCK;
p       v                 const int jz = j / ( BLOCK*BLOCK );

p       v                 f_j[1 ][j] = lbm_val.f[1 ][ OFFNEI(jx+0, jy+1, jz+1) ];
p       v             }
                     <<< Loop-information Start >>>
                     <<<  [OPTIMIZATION]
                     <<<    SIMD(VL: 8)
                     <<<    PREFETCH(HARD) Expected by compiler :
                     <<<      rhoc_j, ir_j, uc_j, vc_j, wc_j, k000, k100
                     <<<      k010, k001, k200, k020, k002, k110, k101, k011
                     <<<    PREFETCH(SOFT) : 54
                     <<<     SEQUENTIAL : 54
                     <<<      f_j: 54
                     <<<    SPILLS :
                     <<<      GENERAL   : SPILL 0  FILL 0
                     <<<      SIMD&FP   : SPILL 0  FILL 0
                     <<<      SCALABLE  : SPILL 49  FILL 139
                     <<<      PREDICATE : SPILL 0  FILL 0
                     <<< Loop-information  End >>>
        v             for( int j = 0; j < BLOCK*BLOCK*BLOCK; j++ ){
        v                 real f[Q] = {};

        v                 f[0 ] = f_j[0 ][j];
        v                 f[1 ] = f_j[1 ][j];
        v                 f[2 ] = f_j[2 ][j];
```

V2（**section 1 のループ分割前**）	157.857 [sec]
#pragma omp simd	149.206 [sec]
#pragma omp simd **(無効)**	131.899 [sec]

ループ分割　その3 (v4)

ワーク配列f_jの使用により、Section 1をさらに分割することが可能

```
    .....
    f[25] = f_j[25][j];
    f[26] = f_j[26][j];

    k000[j] = f[0 ] + ..... + f[26];

    rhoc_j[j] = ( (f[0]+f[26]) + ..... + (f[12]+f[14]) ) + f[13];
    .....
    wc_j[j]   = (( (f[26]-f[0]) + ..... + (f[22]-f[4])) * ir_j[j];
}
```

```
#pragma omp simd
for( int j = 0; j < BLOCK*BLOCK*BLOCK; j++ ){
    real f[Q] = {};

    f[0 ] = f_j[0 ][j];
    f[1 ] = f_j[1 ][j];
    .....

    //-- forward central moment
    real uc, vc, wc;
    real up, um, vp, vm, wp, wm;
    uc = uc_j[j];
    up = R(1.)-uc_j[j];
    um = R(-1.)-uc_j[j];
    .....
    // k000[j] = f[0 ] + f[1 ] + .....
    k100[j] = um*f[0 ] + -uc*f[1 ] + .....
    k010[j] = vm*f[0 ] +  vm*f[1 ] + .....
    .....
```

```
                //-- macro
//                real rhoc, ir, uc, vc, wc;
            #pragma omp simd
 <<< Loop-information Start >>>
 <<<  [OPTIMIZATION]
 <<<    SIMD(VL: 8)
 <<<    PREFETCH(HARD) Expected by compiler :
 <<<      k000, rhoc_j, ir_j, uc_j, vc_j, wc_j
 <<<    PREFETCH(SOFT) : 54
 <<<     SEQUENTIAL : 54
 <<<      f_j: 54
 <<<    SPILLS :
 <<<      GENERAL   : SPILL 0  FILL 0
 <<<      SIMD&FP   : SPILL 0  FILL 0
 <<<      SCALABLE  : SPILL 1  FILL 1
 <<<      PREDICATE : SPILL 0  FILL 0
 <<< Loop-information  End >>>
v              for( int j = 0; j < BLOCK*BLOCK*BLOCK; j++ ){
v                  real f[Q] = {};

v                  f[0 ] = f_j[0 ][j];
v                  f[1 ] = f_j[1 ][j];
```

```
            #pragma omp simd
 <<< Loop-information Start >>>
 <<<  [OPTIMIZATION]
 <<<    SIMD(VL: 16)
 <<<    PREFETCH(HARD) Expected by compiler :
 <<<      uc_j, vc_j, wc_j, k100, k010, k001, k200, k020
 <<<      k002, k110, k101, k011
 <<<    PREFETCH(SOFT) : 54
 <<<     SEQUENTIAL : 54
 <<<      f_j: 54
 <<<    SPILLS :
 <<<      GENERAL   : SPILL 0  FILL 0
 <<<      SIMD&FP   : SPILL 0  FILL 0
 <<<      SCALABLE  : SPILL 41  FILL 107
 <<<      PREDICATE : SPILL 0  FILL 0
 <<< Loop-information  End >>>
v              for( int j = 0; j < BLOCK*BLOCK*BLOCK; j++ ){
v                  real f[Q] = {};

v                  f[0 ] = f_j[0 ][j];
v                  f[1 ] = f_j[1 ][j];
v                  f[2 ] = f_j[2 ][j];
```

ループ分割により軽減可能（p10-p12）

	v3	v4
col_stream	131.899 [sec]	113.255 [sec]

コードレベルでの命令並びの最適化 (v5)

Section 3

```
f[0 ] = R(0.5) * ( KMMG(0)*(wc*wc-wc) + KMMG(1)*(R(2.)*wc-R(1.)) + KMMG(2) );
.....
f[9 ] = KMMG(0)*(R(1.)-wc*wc) - R(2.)*wc*KMMG(1) - KMMG(2);
.....
f[18] = R(0.5) * ( KMMG(0)*(wc*wc+wc) + KMMG(1)*(R(2.)*wc+R(1.)) + KMMG(2) );

//-- store
lbm_val.fn[0 ][ offset*BLOCK_CUBE + j ] = f[0 ];
lbm_val.fn[1 ][ offset*BLOCK_CUBE + j ] = f[1 ];
lbm_val.fn[2 ][ offset*BLOCK_CUBE + j ] = f[2 ];
.....
```

```
          #pragma omp simd
<<< Loop-information Start >>>
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 16)
<<<    PREFETCH(HARD) Expected by compiler :
<<<      uc_j, vc_j, wc_j, k000, k100, k010, k001, k200
<<<      k020, k002, k110, k101, k011, ir_j, (unknown)
<<<    PREFETCH(SOFT) : 50
<<<     SEQUENTIAL : 50
<<<      (unknown): 50
<<<    SPILLS :
<<<      GENERAL    : SPILL 0  FILL 15
<<<      SIMD&FP    : SPILL 0  FILL 0
<<<      SCALABLE   : SPILL 91  FILL 165
<<<      PREDICATE  : SPILL 0  FILL 0
<<< Loop-information  End >>>
          for( int j = 0; j < BLOCK*BLOCK*BLOCK; j++ ){
              real f[0];
```

使用するマクロごとに式をまとめる

```
//-- backward central moment
// using macro KMMG
f[0 ] = R(0.5) * ( KMMG(0)*(wc*wc-wc) + KMMG(1)*(R(2.)*wc-R(1.)) + KMMG(2) );
lbm_val.fn[0 ][ offset*BLOCK_CUBE + j ] = f[0 ];

f[9 ] = KMMG(0)*(R(1.)-wc*wc) - R(2.)*wc*KMMG(1) - KMMG(2);
lbm_val.fn[9 ][ offset*BLOCK_CUBE + j ] = f[9 ];

f[18] = R(0.5) * ( KMMG(0)*(wc*wc+wc) + KMMG(1)*(R(2.)*wc+R(1.)) + KMMG(2) );
lbm_val.fn[18][ offset*BLOCK_CUBE + j ] = f[18];

// using macro KCMG
f[1 ] = R(0.5) * ( KCMG(0)*(wc*wc-wc) + KCMG(1)*(R(2.)*wc-R(1.)) + KCMG(2) );
lbm_val.fn[1 ][ offset*BLOCK_CUBE + j ] = f[1 ];
......
```

```
          #pragma omp simd
<<< Loop-information Start >>>
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 16)
<<<    PREFETCH(HARD) Expected by compiler :
<<<      uc_j, vc_j, wc_j, k000, k100, k010, k001, k200
<<<      k020, k002, k110, k101, k011, ir_j, (unknown)
<<<    PREFETCH(SOFT) : 50
<<<     SEQUENTIAL : 50
<<<      (unknown): 50
<<<    SPILLS :
<<<      GENERAL    : SPILL 0  FILL 9
<<<      SIMD&FP    : SPILL 0  FILL 0
<<<      SCALABLE   : SPILL 60  FILL 125
<<<      PREDICATE  : SPILL 0  FILL 0
<<< Loop-information  End >>>
          for( int j = 0; j < BLOCK*BLOCK*BLOCK; j++ ){
```

	v4	v5
col_stream	113.255 [sec]	107.298 [sec]

関数形式マクロの配列への置き換え(v6)

Section 3

v5

```
f[0 ] = R(0.5) * ( KMMG(0)*(wc*wc-wc) + KMMG(1)*(R(2.)*wc-R(1.)) + KMMG(2) );
lbm_val.fn[0 ][ offset*BLOCK_CUBE + j ] = f[0 ];

f[9 ] = KMMG(0)*(R(1.)-wc*wc) - R(2.)*wc*KMMG(1) - KMMG(2);
lbm_val.fn[1 ][ offset*BLOCK_CUBE + j ] = f[1 ];

f[18] = R(0.5) * ( KMMG(0)*(wc*wc+wc) + KMMG(1)*(R(2.)*wc+R(1.)) + KMMG(2) );
lbm_val.fn[2 ][ offset*BLOCK_CUBE + j ] = f[2 ];
.....
```

```
<<<     SEQUENTIAL : 50
<<<       (unknown): 50
<<<     SPILLS :
<<<       GENERAL    : SPILL 0  FILL 9
<<<       SIMD&FP    : SPILL 0  FILL 0
<<<       SCALABLE   : SPILL 60  FILL 125
<<<       PREDICATE  : SPILL 0  FILL 0
<<< Loop-information  End >>>
          for( int j = 0; j < BLOCK*BLOCK*BLOCK; j++ ){
```

マクロが二重に展開される
→変数が増加しSPILLが発生

マクロ関数の引数は0, 1, 2のみ
→ 配列として準備しておくことで、SPILLを抑制し、また計算の効率化

v6

```
/* the part using macro KMMG */
real kmmg[3];
kmmg[0] = KMMG(0);
kmmg[1] = KMMG(1);
kmmg[2] = KMMG(2);
f[0 ] = R(0.5) * ( kmmg[0]*(wc*wc-wc) + kmmg[1]*(R(2.)*wc-R(1.)) + kmmg[2] );
lbm_val.fn[0 ][ offset*BLOCK_CUBE + j ] = f[0 ];

f[9 ] = kmmg[0]*(R(1.)-wc*wc) - R(2.)*wc*kmmg[1] - kmmg[2];
lbm_val.fn[9 ][ offset*BLOCK_CUBE + j ] = f[9 ];

f[18] = R(0.5) * ( kmmg[0]*(wc*wc+wc) + kmmg[1]*(R(2.)*wc+R(1.)) + kmmg[2] );
lbm_val.fn[18][ offset*BLOCK_CUBE + j ] = f[18];

/* the part using macro KCMG */
real kcmg[3];
.....
```

```
          #pragma omp simd
<<< Loop-information Start >>>
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 16)
<<<    PREFETCH(HARD) Expected by compiler :
<<<      uc_j, vc_j, wc_j, k000, k100, k010, k001, k200
<<<      k020, k002, k110, k101, k011, ir_j, (unknown)
<<<    PREFETCH(SOFT) : 50
<<<      SEQUENTIAL : 50
<<<        (unknown): 50
<<<    SPILLS :
<<<      GENERAL    : SPILL 0  FILL 11
<<<      SIMD&FP    : SPILL 0  FILL 0
<<<      SCALABLE   : SPILL 29  FILL 61
<<<      PREDICATE  : SPILL 0  FILL 0
<<< Loop-information  End >>>
          for( int j = 0; j < BLOCK*BLOCK*BLOCK; j++ ){
              real f[Q];
```

	v5	v6
col_stream	107.298 [sec]	103.394 [sec]

計算の効率化およびループ分割 (v7)

Section 1後半 （k001 ... k200 を計算するループ内）

```
k100[j] = um*f[0 ] + -uc*f[1 ] + up*f[2 ] + um*f[3 ] + -uc*f[4 ] + up*f[5 ] + um*f[6 ] + -uc*f[7 ] + up*f[8 ]
        + um*f[9 ] + -uc*f[10] + up*f[11] + um*f[12] + -uc*f[13] + up*f[14] + um*f[15] + -uc*f[16] + up*f[17]
        + um*f[18] + -uc*f[19] + up*f[20] + um*f[21] + -uc*f[22] + up*f[23] + um*f[24] + -uc*f[25] + up*f[26];
⇒
K100[j] = um * (f[0 ] + f[3 ] + f[6 ] + f[9 ] + f[12] + f[15] + f[18] + f[21] + f[24])
        + ......
```

```
k101[j] = um* wm*f[0 ] + -uc* wm*f[1 ] + up* wm*f[2 ] + um* wm*f[3 ] + -uc* wm*f[4 ] + up* wm*f[5 ]
        + um* wm*f[6 ] + -uc* wm*f[7 ] + up* wm*f[8 ] + um*-wc*f[9 ] + -uc*-wc*f[10] + up*-wc*f[11]
        + um*-wc*f[12] + -uc*-wc*f[13] + up*-wc*f[14] + um*-wc*f[15] + -uc*-wc*f[16] + up*-wc*f[17]
        + um* wp*f[18] + -uc* wp*f[19] + up* wp*f[20] + um* wp*f[21] + -uc* wp*f[22] + up* wp*f[23]
        + um* wp*f[24] + -uc* wp*f[25] + up* wp*f[26];
⇒
K101[j] = um * ((wm * (f[0] + f[3] + f[6]) - wc * (f[9] + f[12] + f[15]) + wp * (f[18] + f[21] + f[24]))
        + ......
```

```
real f036[9];

f036[0] = f[0 ] + f[3 ] + f[6 ];
f036[1] = f[9 ] + f[12] + f[15];
f036[2] = f[18] + f[21] + f[24];

f036[3] = f[1 ] + f[4 ] + f[7 ];
f036[4] = f[10] + f[13] + f[16];
f036[5] = f[19] + f[22] + f[25];

f036[6] = f[2 ] + f[5 ] + f[8 ];
f036[7] = f[11] + f[14] + f[17];
f036[8] = f[20] + f[23] + f[26];
```

```
k100[j] = um * (f036[0] + f036[1] + f036[2])
        - uc * (f036[3] + f036[4] + f036[5])
        + up * (f036[6] + f036[7] + f036[8]);

k200[j] = um * um * (f036[0] + f036[1] + f036[2])
        + uc * uc * (f036[3] + f036[4] + f036[5])
        + up * up * (f036[6] + f036[7] + f036[8]);

k101[j] = um * (wm * f036[0] - wc * f036[1] + wp * f036[2])
        - uc * (wm * f036[3] - wc * f036[4] + wp * f036[5])
        + up * (wm * f036[6] - wc * f036[7] + wp * f036[8]);
```

計算の効率化およびループ分割 (v7)

```
real f012[9];
f012[0] = f[0 ] + f[1 ] + f[2 ];
f012[1] = f[3 ] + f[4 ] + f[5 ];
f012[2] = f[6 ] + f[7 ] + f[8 ];
f012[3] = f[9 ] + f[10] + f[11];
f012[4] = f[12] + f[13] + f[14];
f012[5] = f[15] + f[16] + f[17];
f012[6] = f[18] + f[19] + f[20];
f012[7] = f[21] + f[22] + f[23];
f012[8] = f[24] + f[25] + f[26];
```

```
k001[j] = wm * (f012[0] + f012[1] + f012[2])
        - wc * (f012[3] + f012[4] + f012[5])
        + wp * (f012[6] + f012[7] + f012[8]);

k002[j] = wm * wm * (f012[0] + f012[1] + f012[2])
        + wc * wc * (f012[3] + f012[4] + f012[5])
        + wp * wp * (f012[6] + f012[7] + f012[8]);

k010[j] = vm * f012[0] - vc * f012[1] + vp * f012[2]
        + vm * f012[3] - vc * f012[4] + vp * f012[5]
        + vm * f012[6] - vc * f012[7] + vp * f012[8];

......
```

Section 1後半のループ（k001 ... k200 を計算）をさらに3分割

1. k100, k200, k101を計算するループ（一時配列 f036 を使用）
   → SPILL/FILLの解消、SIMD化+SWP化
2. k001, k002, k010, k011, k020を計算するループ （一時配列f012を使用）
   → SIMD化
3. k110を計算するループ
   → SPILL/FILLの解消、SIMD化＋SWP化

	v6	v7
col_stream	103.394 [sec]	93.163 [sec]

計算の効率化 (v8)

Section 1前半 （ir，vc，uc，wc を計算するループ内）

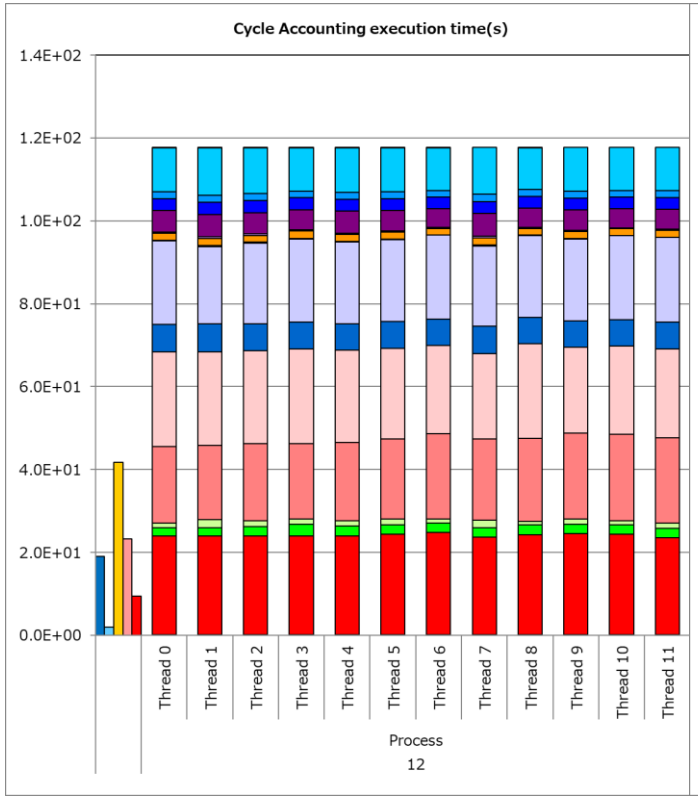
```
ir_j[j] = R(1.) / ( rhoc_j[j] + 1.e-20 );

uc_j[j]   = (( (f[26]-f[0]) + (f[2]-f[24])
            +   (f[20]-f[6]) + (f[8]-f[18]) )
            + ( (f[23]-f[3]) + (f[5]-f[21])
            + (  f[17]-f[9]) + (f[11]-f[15]) )
            + (  f[14]-f[12])) * ir_j[j];

vc_j[j]   = (( (f[26]-f[0]) + (f[24]-f[2])
          +  (  f[6]-f[20]) + (f[8]-f[18]) )
          + ( (f[25]-f[1]) + (f[7]-f[19])
          +   (f[17]-f[9]) + (f[15]-f[11]) )
          + (f[16]-f[10])) * ir_j[j];

wc_j[j]   = (( (f[26]-f[0]) + (f[24]-f[2])
             + (f[20]-f[6]) + (f[18]-f[8]) )
            + ( (f[25]-f[1]) + (f[23]-f[3])
            + (  f[21]-f[5]) + (f[19]-f[7]) )
            + (f[22]-f[4])) * ir_j[j];
```

```
real f0m26 = f[0 ] - f[26];
real f2m24 = f[2 ] - f[24];
real f6m20 = f[6 ] - f[20];
real f8m18 = f[8 ] - f[18];

uc_j[j] = -f0m26 + f2m24 - f6m20 + f8m18;
vc_j[j] = -f0m26 - f2m24 + f6m20 + f8m18;
wc_j[j] = -f0m26 - f2m24 - f6m20 - f8m18;

real f1m25  = f[1 ] - f[25];
real f3m23  = f[3 ] - f[23];
real f5m21  = f[5 ] - f[21];
real f7m19  = f[7 ] - f[19];
real f9m17  = f[9 ] - f[17];
real f11m15 = f[11] - f[15];

real f4m22  = f[4 ] - f[22];
real f12m14 = f[12] - f[14];
real f10m16 = f[10] - f[16];

uc_j[j] += (-f3m23 + f5m21 - f9m17 + f11m15) - f12m14;
vc_j[j] += (-f1m25 + f7m19 - f9m17 - f11m15) - f10m16;
wc_j[j] += (-f1m25 - f3m23 - f5m21 -  f7m19)  - f4m22;

ir_j[j] = R(1.) / ( rhoc_j[j] + 1.e-20 );

uc_j[j] *= ir_j[j];
vc_j[j] *= ir_j[j];
wc_j[j] *= ir_j[j];
```

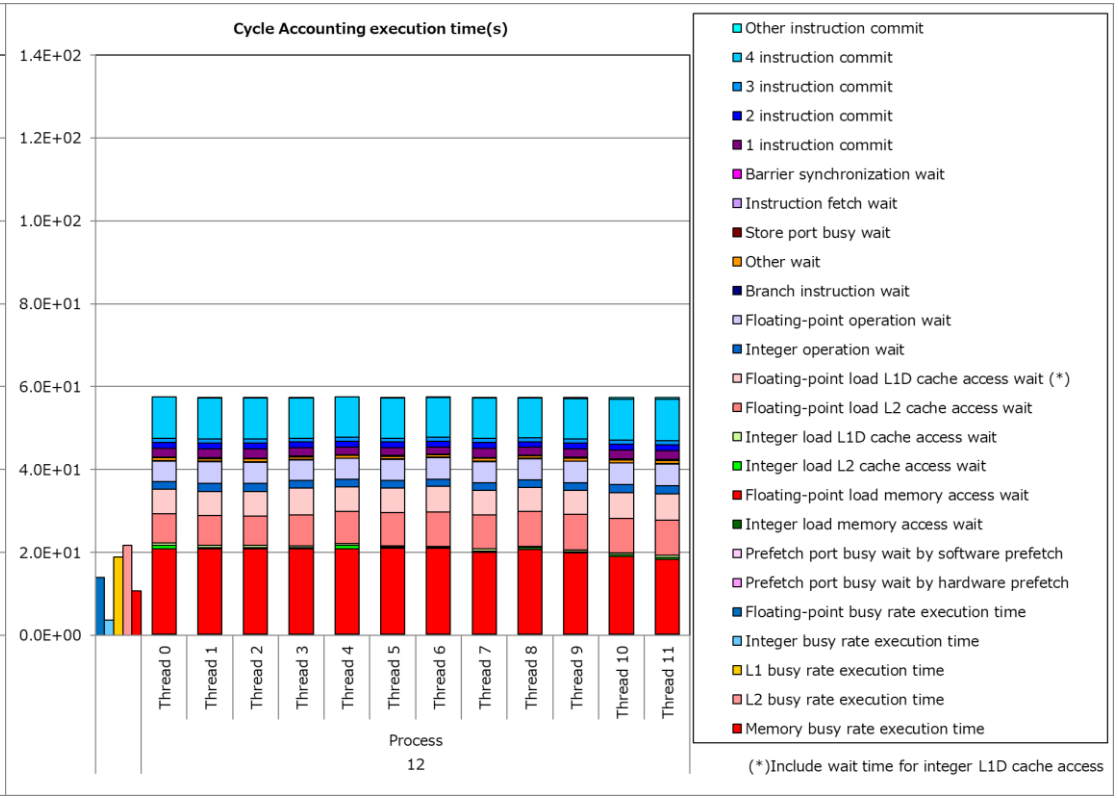
	v7	v8
col_stream	93.163 [sec]	91.842 [sec]

結果 (CPU性能解析レポート)

original版



v8版



Floating-point operation wait
Floating-point load L1D cache access wait
Floating-point load L2 cache access wait　が減少

	original	v8
col_stream	190.391 [sec]	91.842 [sec]


RIST