

第185回 お試しアカウント付き 並列プログラミング講習会 「Wisteria実践」演習

東京大学情報基盤センター
埴 敏博
星野 哲也 三木 洋平

ユーザアカウント

- 使用システム： Wisteria/BDEC-01 (Wisteria)
 - `$ ssh USERNAME@wisteria.cc.u-tokyo.ac.jp`
- 本講習会でのユーザ名
 - 利用者番号： `tABCDE` (ABCDEは, 適宜書き換えてください)
 - 利用グループ： `gt00`
- 利用期限
 - `10/15 9:00`まで有効
- 注:本講習会関連の質問はhanawa[at]cc.u-tokyo.ac.jpまで
 - Slackで質問していただくほうがありがたいです
 - (講習会アカウントでは) 公式の相談対応システムは使わないでください

バッチキューの設定のしかた

- Wisteriaでのバッチ処理は、富士通のバッチシステムで管理されています。
- 以下、主要コマンドを説明します。
 - ジョブの投入：`pjsub <ジョブスクリプトファイル名>`
 - 自分が投入したジョブの状況確認：`pjstat`
 - 投入ジョブの削除：`pjdel <ジョブID>`
 - バッチキューの状態を見る：`pjstat --rsc`
 - バッチキューの詳細構成を見る：`pjstat --rsc -x`
 - 投げられているジョブ数を見る：`pjstat -b`
 - 過去の投入履歴を見る：`pjstat -H`
 - 同時に投入できる数／実行できる数を見る：`pjstat --limit`

pjstat --rsc の実行画面例

```
$ pjstat --rsc
SYSTEM: Odyssey
RSCGRP
lecture-o
tutorial-o
```

```
STATUS
[ENABLE,START]
[ENABLE,START]
```

```
NODE
96
2x12x16
```

```
SYSTEM: Aquarius
RSCGRP
Lecture-a
tutorial-a
```

```
STATUS
[ENABLE,START]
[DISABLE,STOP]
```

```
NODE GPU
7 56
2 16
```

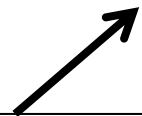
ノードの
利用可能数



GPUの
利用可能数
(Aquarius)



現在使えるか



使える
キュー名
(リソース
グループ)



pjstat --rsc -x の実行画面例

```
$ pjstat --rsc -x
```

```
SYSTEM: Odyssey
```

RSCGRP	STATUS	MIN_NODE	MAX_NODE	MAX_ELAPSE	REMAIN_ELAPSE	MEM(GiB)	PROJECT
lecture-o	[ENABLE,START]	1	12	00:15:00	00:15:00	28	gt68
tutorial-o	[DISABLE,STOP]	1	12	00:15:00	--:--:--	28	gt68

```
SYSTEM: Aquarius
```

RSCGRP	STATUS	MIN_NODE	MAX_NODE	AVAIL_GPU	MAX_ELAPSE	REMAIN_ELAPSE	MEM(GiB)	PROJECT
lecture-a	[ENABLE,START]	1	1	1,2,4	00:15:00	00:15:00	448	gt68
tutorial-a	[DISABLE,STOP]	1	1	1,2,4	00:15:00	--:--:--	448	gt68

↑
使える
キュー名
(リソース
グループ)

↑
現在
使えるか

↑
ノードの
実行情報

↑
課金情報 (財布)
実習では1つのみ

pjstat --rsc -b の実行画面例

```
$ pjstat --rsc -b  
SYSTEM: Odyssey
```

RSCGRP	STATUS	TOTAL	RUNNING	QUEUED	HOLD	OTHER	NODE
lecture-o	[ENABLE,START]	0	0	0	0	0	96
tutorial-o	[DISABLE,STOP]	0	0	0	0	0	2x12x16

```
SYSTEM: Aquarius
```

RSCGRP	STATUS	TOTAL	RUNNING	QUEUED	HOLD	OTHER	NODE
lecture-a	[ENABLE,START]	0	0	0	0	0	7
tutorial-a	[DISABLE,STOP]	0	0	0	0	0	2

↑
使えるキュー名
(リソースグループ)

↑
現在
使えるか

↑
ジョブの
総数

↑
実行してい
るジョブの
数

↑
待たされてい
るジョブの数

↑
ノードの
利用可能数

moduleの指定

- コンパイラ・ライブラリ等の環境をセットアップ
 - Odyssey向け富士通コンパイラ、MPIを使用
`$ module load odyssey`
 - Aquarius向けgcc、cuda, OpenMPI(CUDA対応)を使用
`$ module load aquarius cuda omp-cuda`
`$ module load nvidia cuda omp-cuda`
(omp-cudaに対応のものがないためworkaround)
- 困った時は
`$ module purge`

moduleの一覧

- 現在利用中の環境に追加できるものを確認

```
$ module avail
```

```
---/work/opt/local/modules/modulefiles/LN/aquarius/mpi/gcc/8.3.1/mpi-cuda/4.1.1-11.4 -----  
gromacs/2021.2  lammps/29Oct2020
```

...

- 全ての環境を確認

```
$ show_module
```

ApplicationName	ModuleName	Node	BaseCompiler/MPI
Archiconda	archiconda3/0.2.3	odyssey	-
Arm Forge	forge/21.0.2	aquarius	-
...			
GROMACS	gromacs/2021.2	aquarius	gcc/8.3.1/mpi-cuda/4.1.1-11.2

サンプルプログラムの取得 (1/2)

- 実行してもらおうコマンドは \$ 以降に青字で記載しています
 - ターミナルへの入力が終わったら 「Enter」 キーを押してください
- 1. Lustreファイルシステムに移動
 - \$ cd /work/gt00/tABCDE # 下線部は自分のIDに変更
- 2. /work/gt00/share/Wisteria にあるサンプルファイルをコピー
 - \$ cp /work/gt00/share/Wisteria/*.gz .
* と . (ドット) の間に半角スペース
- 3. サンプルファイルを展開
 - \$ tar xvfz Samples-wo.tar.gz

サンプルプログラムの取得 (2/2)

4. Samples-wo ディレクトリに入る

```
$ cd Samples-wo
```

5. 自分の使いたい言語のディレクトリに入る

```
$ cd C # C言語を使用する場合
```

```
$ cd F # Fortranを使用する場合
```

6. サンプルプログラムがあることを確認

```
$ ls
```

TIPS (タブ補完)

- ターミナル上では[Tab]キーを入力してタブ補完を効かせながら入力すると良い
 - キー入力数が減るのでお得 (自動的にtypoも減る)
 - 自動補完できる部分だけを入力するので, とりあえず[Tab]を入力した上で補ってあげれば良い
 - Windowsとは違い, 候補を順番に表示するようなことはない
- 先ほど入力してもらったコマンド群の場合:
 1. `$ cd /wo[Tab]/gt00/tABCDE`
 2. `$ cp /wo[Tab]/gt00/z30105/W[Tab]/* .`
 3. `$ tar xvfz S[Tab]`
 4. `$ cd S[Tab]`

Streamプログラムをコンパイルしよう (1/2)

1. Samples-wo フォルダに入る
`$ cd Samples-wo`
2. C言語 : `$ cd C`
Fortran90言語 : `$ cd F`
3. アプリ名のフォルダに入る
`$ cd stream`

Streamプログラムをコンパイルしよう (2/2)

6. module loadしてmake する

```
$ module load odyssey
```

```
$ make
```

7. 実行ファイル(stream)ができていることを確認する

```
$ ls
```

プログラムを実行しよう

1. streamフォルダ中で以下を実行する
`$ pjsub stream.bash`
2. 自分の導入されたジョブを確認する
`$ pjstat`
3. 実行が終了すると、以下のファイルが生成される
`stream.bash.XXXXXX.err`
`stream.bash.XXXXXX.out` (XXXXXXは数字)
4. 上記の標準出力ファイルの中身を見してみる
`$ cat stream.bash.XXXXXX.out`
5. エラー出力は以下のファイルにあるので念のため確認
`$ cat stream.bash.XXXXXX.err`

通信性能

OSU Micro Benchmarks (Odyssey用)

- オハイオ州立大学 D.K. Pandaのグループが開発, Version 6.0
- インストール場所
/work/share/benchmarks/Odyssey/osu-micro-benchmarks-6.0
- 実行ジョブスクリプト
/work/share/benchmarks/Odyssey/osu-micro-benchmarks-6.0/run/*.sh
- 実行の方法
 1. ディレクトリを作り、ジョブスクリプトをコピー

```
$ mkdir omb-6.0-odyssey  
$ cd omb-6.0-odyssey  
$ cp /work/share/benchmarks/Odyssey/osu-micro-benchmarks-6.0/run/*.sh .
```
 2. ジョブ実行

```
$ pjsub osu_latency.sh
```

Tea Leaf (1/2)

- 熱伝導のベンチマーク
 - mantevoプロジェクトのミニアプリの一つ
 - <http://uk-mac.github.io/TeaLeaf/>

- 展開

```
$ tar xvfz TeaLeaf-wo.tar.gz  
$ cd TeaLeaf_ref-1.3
```

- Wisteria-0向けバイナリのコンパイル

```
$ make COMPILER=FJ MPI_COMPILER=mpifrtpx C_MPI_COMPILER=mpifccpx
```


Tea Leaf (2/2)

性能は以下の順に高くなるはず
フラットOMP << ハイブリッドMPI
< ハイブリッド+形状指定

OpenMP+MPIハイブリッド
(フラットOMP)

```
$ pjsub tea_leaf.sh
```

OpenMP+MPIハイブリッド

```
$ pjsub tea_leaf2.sh
```

OpenMP+MPIハイブリッド+ノード形状指定

```
$ pjsub tea_leaf3.sh
```

- ノード4x3形状、かつノード内4プロセス → 8x6の並びで実行
- 実行時間を比較: (それぞれに行う)

```
$ grep wall tea_leaf.sh.XXXX.err | tail -1
```

7点ステンスル

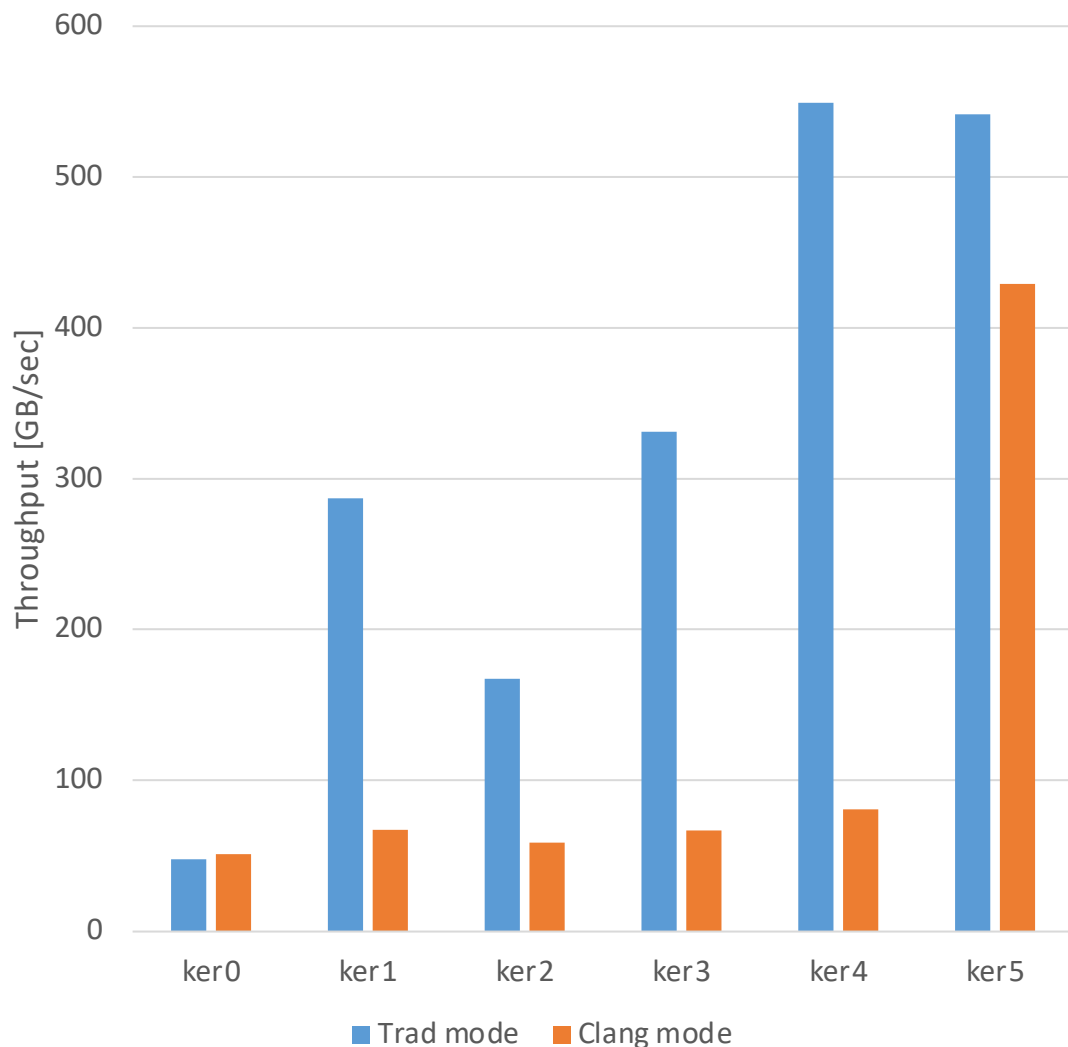
- 拡散方程式のカーネル
 - ✓ 自身と前後左右上下の7点を参照して値を更新
- ディリクレ境界条件
 - ✓ 境界では自身の値を参照
- 倍精度
- サイズ：256*384*384
 - ✓ A64FXが48コアのため、 $48*8 = 384$
- Byte/Flop = 16/13
 - ✓ 各ステップで読み込んだデータは全てキャッシュに乗ると仮定して性能評価
- ソースコード
 - ✓ /work/gt00/share/wisteria_trial.tar.gz

```
1 do {
2 #pragma omp parallel for
3   for (int z = 0; z < nz; z++) {
4     for (int y = 0; y < ny; y++) {
5       for (int x = 0; x < nx; x++) {
6         int c = x + y * nx + z * nx * ny;
7         int w = (x == 0) ? c : c - 1;
8         int e = (x == nx-1) ? c : c + 1;
9         int n = (y == 0) ? c : c - nx;
10        int s = (y == ny-1) ? c : c + nx;
11        int b = (z == 0) ? c : c - nx * ny;
12        int t = (z == nz-1) ? c : c + nx * ny;
13        f2_t[c] = cc * f1_t[c]
14                + cw * f1_t[w] + ce * f1_t[e]
15                + cs * f1_t[s] + cn * f1_t[n]
16                + cb * f1_t[b] + ct * f1_t[t];
17      }
18    }
19  }
20  double *tmp = f1_t;
21  f1_t = f2_t;
22  f2_t = tmp;
23  time += dt;
24 } while (time + 0.5*dt < 0.1);
```

性能評価：Odyssey 1 node

C言語版

ソース：wisteria_trial/Odyssey/C



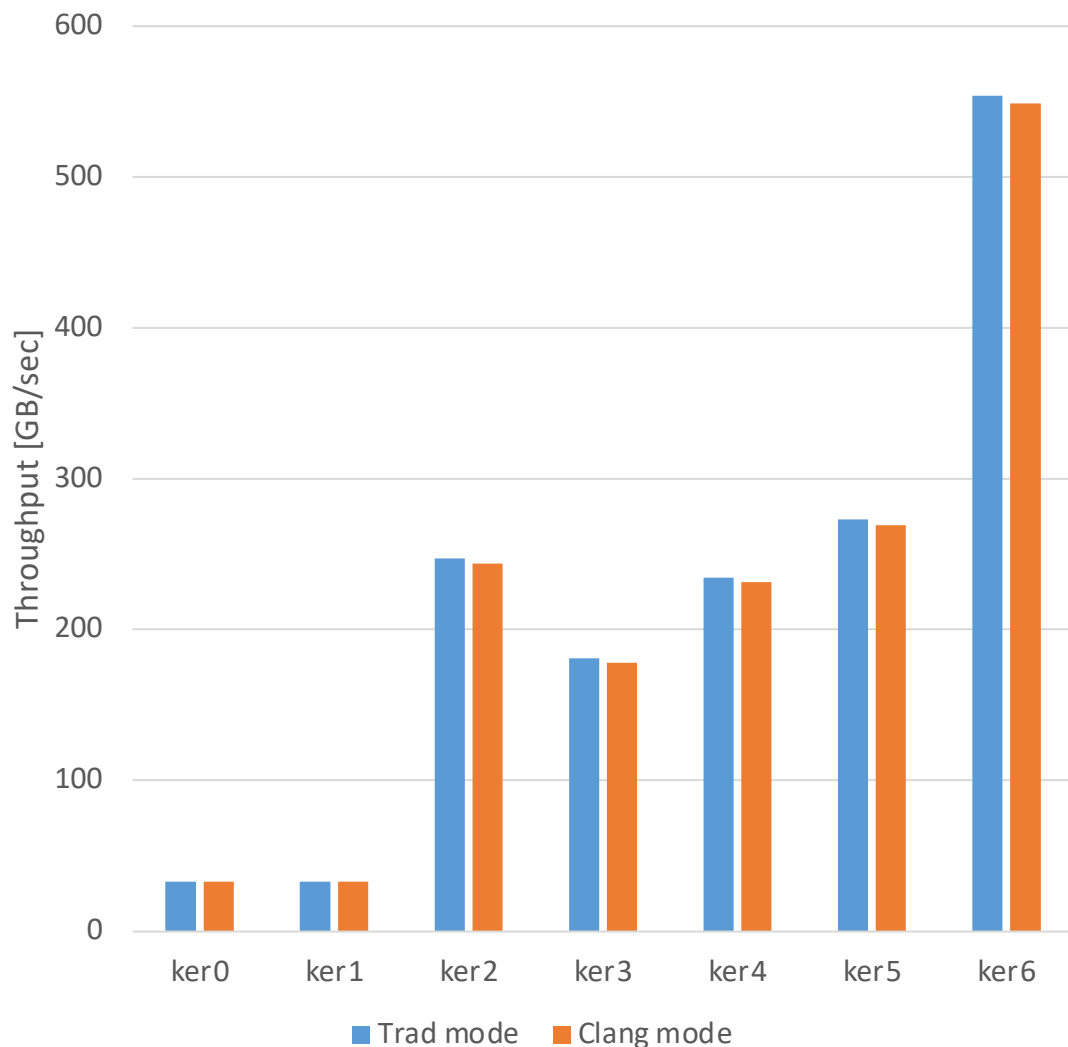
	全最適化	新規最適化
ker0	naive	ナイーブな実装
ker1	firstTouch	ファーストタッチ適用
ker2	firstTouch_yDim	Yループに #pragma omp for nowait を適用
ker3	firstTouch_yzDim	ZループをCMG分割、YループをCMG内スレッド分割
ker4	firstTouch_yzDim_peeling	ループピーリング適用
ker5	firstTouch_yzDim_peeling_intrinsic	intrinsic実装

- この問題では全体的にTrad modeが速い
- clang modeでは、Intrinsicsを書くなどの処置をしないとSIMD化されない
 - ✓ 今後のcompiler updateに期待

性能評価：Odyssey 1 node

Fortran版

ソース：wisteria_trial/Odyssey/F



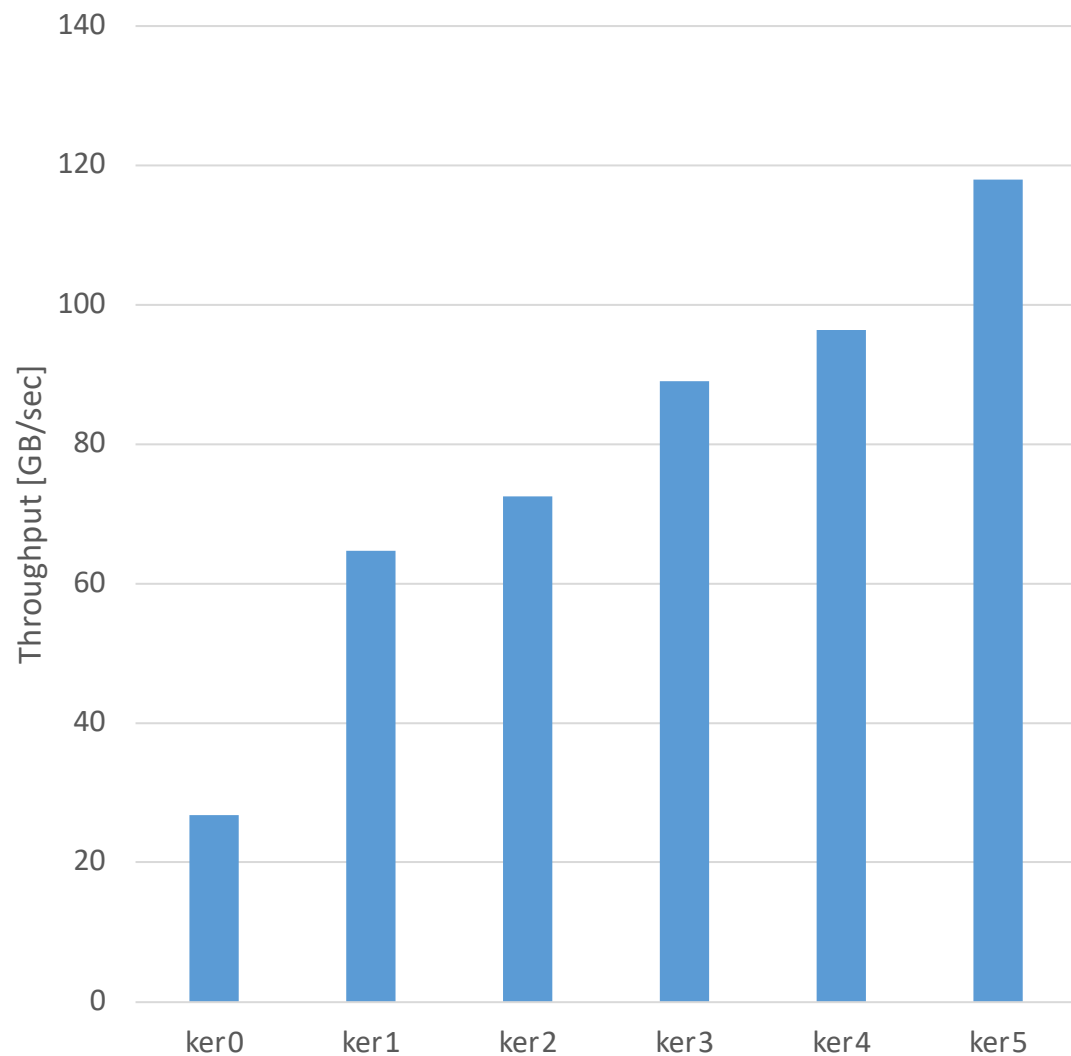
	全最適化	新規最適化
ker0	naive	ナイーブな実装
ker1	firstTouch	ファーストタッチ適用
ker2	firstTouch_noPointer	並列領域からPointer属性を排除
ker3	firstTouch_noPointer_yDim	Yループに !\$omp do / !\$omp end do nowait を適用
ker4	firstTouch_noPointer_yzDim	ZループをCMG分割、YループをCMG内スレッド分割
ker5	firstTouch_noPointer_yzDim_peeling	ループピーリング適用
ker6	firstTouch_noPointer_yzDim_peeling_contiguous	配列にcontiguous属性を追加

- Fortran版ではTrad mode, Clang modeでほとんど差はない
- **Pointer属性を使う場合には注意が必要**
 - ✓ コンパイラの最適化がほとんど効かない
 - ✓ 左の結果は、FirstTouchの意味が無いわけではなく、Pointerのせいで効果が見えていないだけ

性能評価：Aquarius 1 CPU socket

C言語版

ソース：wisteria_trial/Aquarius-CPU/C



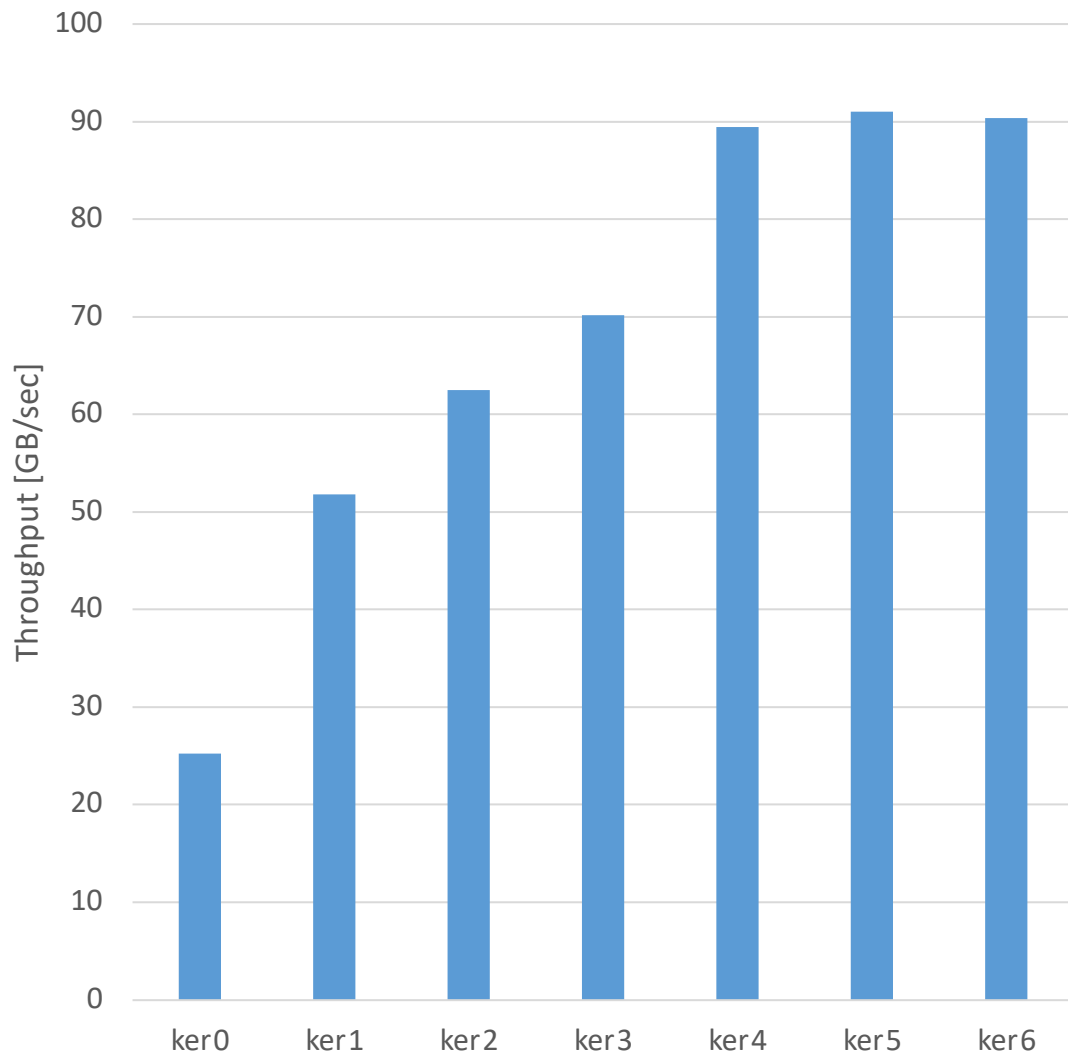
	全最適化	新規最適化
ker0	naive	ナイーブな実装
ker1	firstTouch	ファーストタッチ適用
ker2	firstTouch_yDim	Yループに #pragma omp for nowait を適用
ker3	firstTouch_yzDim	ZループをCMG分割、YループをCMG内スレッド分割
ker4	firstTouch_yzDim_peeling	ループピーリング適用
ker5	firstTouch_yzDim_peeling_intrinsic	intrinsic実装

- Intel Xeon Platinum 8360Y (Ice Lake)は、socket内で2 numa構成となっているため、A64FXのCMG向け最適化が同様に有効
- numactl --cpubind=0,1 --membind=0,1 ./a.outのように numactl を設定している
- 講習会で使える lecture-a キューは CPU を共有しているため、遅くなる可能性がある

性能評価：Aquarius 1 CPU socket

Fortran版

ソース：wisteria_trial/Aquarius-CPU/F



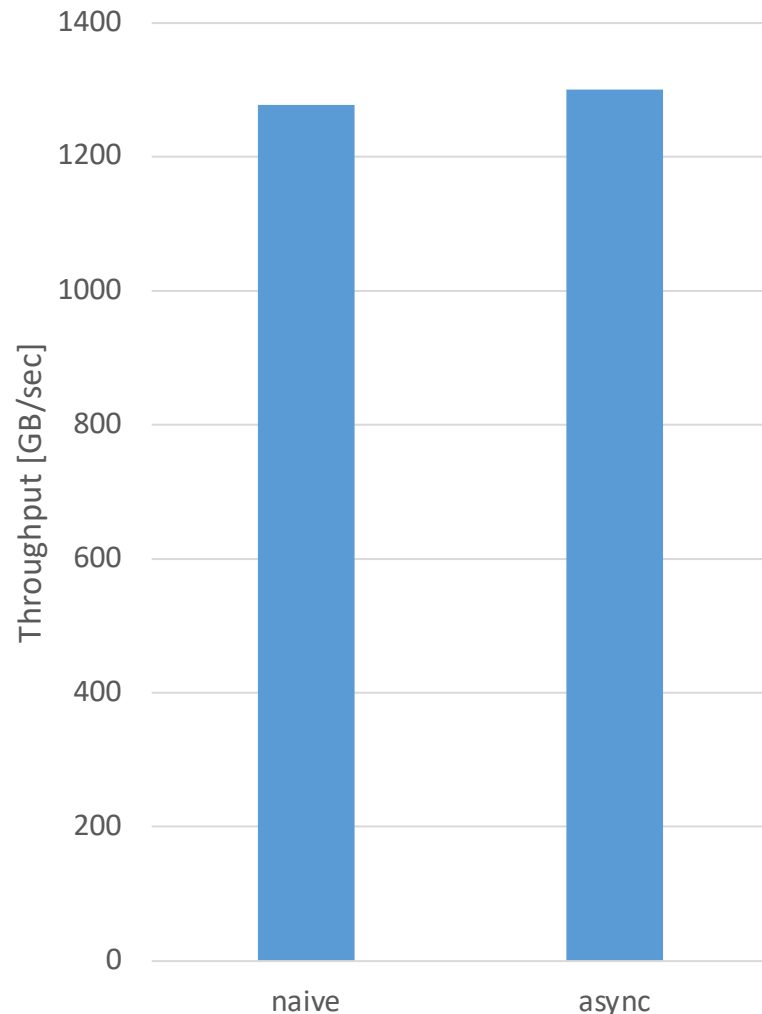
	全最適化	新規最適化
ker0	naive	ナイーブな実装
ker1	firstTouch	ファーストタッチ適用
ker2	firstTouch_noPointer	並列領域からPointer属性を排除
ker3	firstTouch_noPointer_yDim	Yループに !\$omp do / !\$omp end do nowait を適用
ker4	firstTouch_noPointer_zyDim	ZループをCMG分割、YループをCMG内スレッド分割
ker5	firstTouch_noPointer_zyDim_peeling	ループピーリング適用
ker6	firstTouch_noPointer_zyDim_peeling_contiguous	配列にcontiguous属性を追加

- Odysseyと比較すると、pointer属性の影響は小さい

性能評価：Aquarius 1 GPU

C言語版

ソース：wisteria_trial/Aquarius-GPU/C



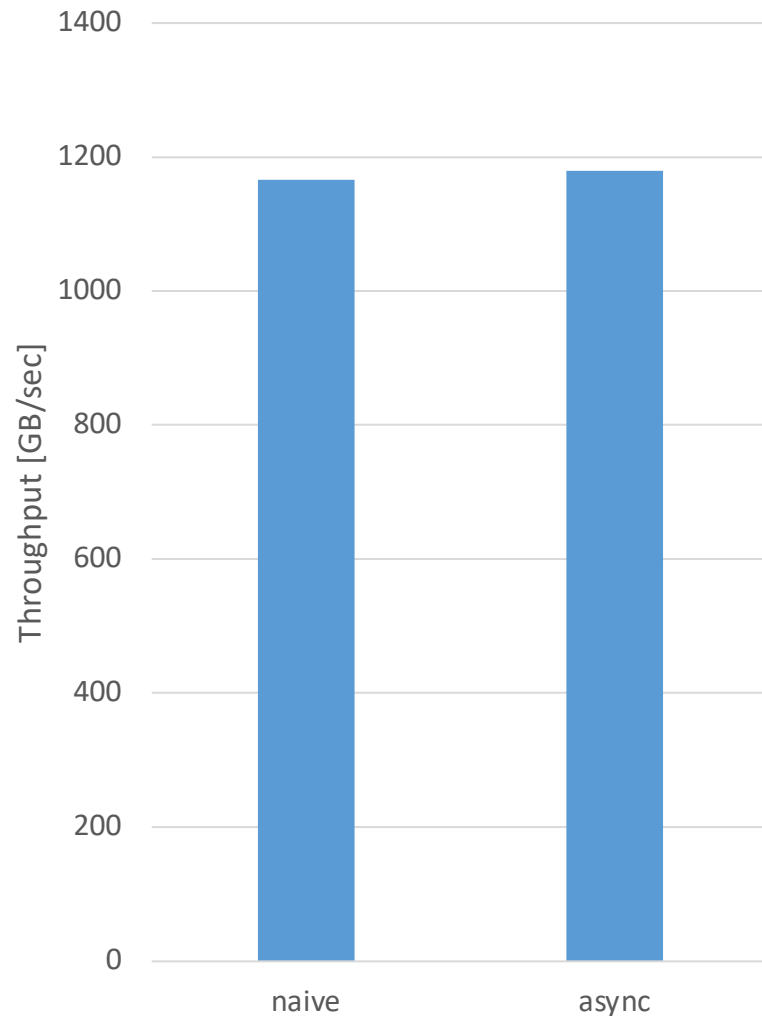
```
#pragma acc kernels present(f2_t, f1_t) async(0)
#pragma acc loop independent
    for (z = 0; z < nz; z++) {
#pragma acc loop independent
    for (y = 0; y < ny; y++) {
#pragma acc loop independent
    for (x = 0; x < nx; x++) {
        c = x + y * nx + z * nx * ny;
        w = (x == 0) ? c : c - 1;
        e = (x == nx-1) ? c : c + 1;
        n = (y == 0) ? c : c - nx;
        s = (y == ny-1) ? c : c + nx;
        b = (z == 0) ? c : c - nx * ny;
        t = (z == nz-1) ? c : c + nx * ny;
        f2_t[c] = cc * f1_t[c] + cw * f1_t[w] + ce * f1_t[e]
            + cs * f1_t[s] + cn * f1_t[n] + cb * f1_t[b] + ct * f1_t[t];
    }
    }
}
```

- 計測ミスを疑うほど速い
- 上記 OpenACC 実装の時点でほぼ理想値。これ以上最適化の余地がない。（計測ミスが無ければ）
 - ✓ ミスが見つかったらご連絡ください -> hoshino @ cc.u-tokyo.ac.jp

性能評価：Aquarius 1 GPU

Fortran版

ソース：wisteria_trial/Aquarius-GPU/F



```
!$acc kernels present(f1,f2) async(0)
!$acc loop independent
do k = 1, nz
  !$acc loop independent
  do j = 1, ny
    !$acc loop independent
    do i = 1, nx

      w = -1; e = 1; n = -1;
      s = 1; b = -1; t = 1;
      if(i == 1) w = 0
      if(i == nx) e = 0
      if(j == 1) n = 0
      if(j == ny) s = 0
      if(k == 1) b = 0
      if(k == nz) t = 0
      f2(i, j, k) = cc * f1(i, j, k) + cw * f1(i+w, j, k) &
        + ce * f1(i+e, j, k) + cs * f1(i, j+s, k) + cn * f1(i, j+n, k) &
        + cb * f1(i, j, k+b) + ct * f1(i, j, k+t)

    end do
  end do
end do
!$acc end kernels
```

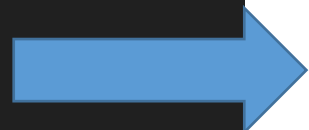
- C言語版程ではないがかなり速い

First Touch

- 1プロセスで13スレッド以上(2 CMG以上)使う場合**必須**
 - ✓ 配列のメモリへの確保は、malloc時ではなくプログラム中最初に触ったタイミングで行われる。0番スレッドのみで初期化する場合、配列はCMG 0番に近いメモリ上に確保されるため、**その他 CMG からアクセスすると遅い**。=> First touchで回避。
 - ✓ export XOS_MMM_L_PAGING_POLICY=demand:demand:demand の設定も合わせて**必須**。この環境変数により、自スレッドに一番近いメモリ上に配列が確保される。

```
diffusion_ker00.c
void init_ker00(REAL *buff1, const int nx, const int ny, const int nz,
               const REAL kx, const REAL ky, const REAL kz,
               const REAL dx, const REAL dy, const REAL dz,
               const REAL kappa, const REAL time) {

    REAL ax, ay, az;
    int jz, jy, jx;
    ax = exp(-kappa*time*(kx*kx));
    ay = exp(-kappa*time*(ky*ky));
    az = exp(-kappa*time*(kz*kz));
    for (jz = 0; jz < nz; jz++) {
        for (jy = 0; jy < ny; jy++) {
            for (jx = 0; jx < nx; jx++) {
                int j = jz*nx*ny + jy*nx + jx;
            }
        }
    }
```



```
diffusion_ker01.c
void init_ker01(REAL *buff1, const int nx, const int ny, const int nz,
               const REAL kx, const REAL ky, const REAL kz,
               const REAL dx, const REAL dy, const REAL dz,
               const REAL kappa, const REAL time) {

    REAL ax, ay, az;
    int jz, jy, jx;
    ax = exp(-kappa*time*(kx*kx));
    ay = exp(-kappa*time*(ky*ky));
    az = exp(-kappa*time*(kz*kz));
    #pragma omp parallel for private(jx, jy, jz)
    for (jz = 0; jz < nz; jz++) {
        for (jy = 0; jy < ny; jy++) {
            for (jx = 0; jx < nx; jx++) {
                int j = jz*nx*ny + jy*nx + jx;
            }
        }
    }
```

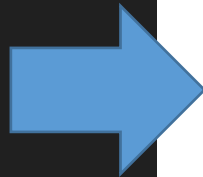
- 並列領域 (diffusion関数) で扱う配列に最初に触る箇所 (init関数) を**並列領域と同じように並列化する**。

Pointer 回避 (Fortranのみ)

- 富士通Fortranコンパイラはかなり保守的なようで、**ポインタ配列があるとほとんどの最適化を諦める**。C pointerでもrestrictは付けた方がいい。
- 回避方法例
 - ✓ allocatable 配列を使う。
 - ✓ 関数呼び出しにより、見かけ上pointer属性でない配列を作る。(下図)

```
diffusion_ker01.f90
subroutine diffusion_ker01(f1, f2, nx, ny, nz, ce, cw, cn, cs, ct,
REAL, pointer, contiguous, dimension(:, :, :), intent(inout) :: f1, f2
REAL, pointer, contiguous, dimension(:, :, :)) :: ftmp
    !!!中略!!!
do while (time + 0.5*dt < 0.1)
    !$omp parallel do private(i, j, k, w, e, n, s, b, t)
    do k = 1, nz
    do j = 1, ny
    do i = 1, nx
        !!!中略!!!
    end do
    end do
end do
!$omp end parallel do
ftmp => f1
f1 => f2
f2 => ftmp

time = time + dt
count = count + 1
end do
```



```
diffusion_ker02.f90
subroutine diffusion_ker02(f1, f2, nx, ny, nz, ce, cw, cn, cs, ct, cb, cc, dt, time
REAL, pointer, contiguous, dimension(:, :, :), intent(inout) :: f1, f2
REAL, pointer, contiguous, dimension(:, :, :)) :: ftmp
    !!!中略!!!
do while (time + 0.5*dt < 0.1)
    call diffusion_ker02_parallel(f1, f2, nx, ny, nz, ce, cw, cn, cs, ct, cb, cc)
    ftmp => f1
    f1 => f2
    f2 => ftmp

subroutine diffusion_ker02_parallel(f1, f2, nx, ny, nz, ce, cw, cn, cs, ct,
REAL, dimension(:, :, :), intent(inout) :: f1, f2
    !!!中略!!!
    !$omp parallel do private(i, j, k, w, e, n, s, b, t)
    do k = 1, nz
    do j = 1, ny
    do i = 1, nx
```

- 並列領域を関数化し、**pointer属性を落とす**

ループ分割手法変更

- 7点テンシルの計算は、一つ隣のデータにアクセスするため、並列化すると冗長なメモリアクセスが発生する。=> **共有キャッシュの利用で効率化**

diffusion_ker01.c

Z軸分割

```
#pragma omp parallel for private(x, y, z, c, w, e, n, s, b, t)
  for (z = 0; z < nz; z++) {
    for (y = 0; y < ny; y++) {
      for (x = 0; x < nx; x++) {
```

- 最も一般的な分割方法
- コア間のメモリアクセスの距離が遠くなるので、キャッシュ効率は悪い

diffusion_ker02.c

Y軸分割

```
#pragma omp parallel private(x, y, z, c, w, e, n, s, b, t)
  for (z = 0; z < nz; z++) {
#pragma omp for nowait
    for (y = 0; y < ny; y++) {
      for (x = 0; x < nx; x++) {
```

- nowaitが必須**
- キャッシュの利用効率が上がる
- CMG間のデータ共有が発生するため、複数CMGを使う場合には遅くなる**

diffusion_ker03.c

Y-Z軸分割

```
#pragma omp parallel
{
  !!!中略!!!

  int tid = omp_get_thread_num();
  int nth = omp_get_num_threads();
  int tz = tid/THREAD_BLOCK_SIZE;
  int ty = tid%THREAD_BLOCK_SIZE;
  int ychunk = (ny-1)/THREAD_BLOCK_SIZE + 1;
  int zchunk = nz/((nth-1)/THREAD_BLOCK_SIZE+1);
  do {
    for (z = tz*zchunk; z < MIN((tz+1)*zchunk, nz); z++) {
      for (y = ty*ychunk; y < MIN((ty+1)*ychunk, ny); y++) {
        for (x = 0; x < nx; x++) {
```

- 右図のイメージ
- CMG間のメモリアクセスの距離を離しつつ、CMG内のキャッシュ効率が高まる
- 4 MPI process + 12 OpenMP thread でも良い

#pragma omp for (CMG分割)

```
for (z = 0; z < nz; z++){
```

#pragma omp for nowait (Thread分割)

```
for(y = 0; y < ny; y++){
```

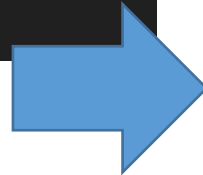
※富士通コンパイラではこのようなOMP指示文の使い方はできません

ループピーリングによる分岐削減

- 7点テンシルの計算では、境界部分の計算のために分岐が必要となる
 - ✓ ループピーリング（皮むき）により最内ループから分岐を排除する。
 - ✓ A64FXは分岐に弱いようなので、効果的。

diffusion_ker03.c

```
for (z = tz*zchunk; z < MIN((tz+1)*zchunk, nz); z++) {
  for (y = ty*ychunk; y < MIN((ty+1)*ychunk, ny); y++) {
    for (x = 0; x < nx; x++) {
      c = x + y * nx + z * nx * ny;
      w = (x == 0) ? c : c - 1;
      e = (x == nx-1) ? c : c + 1;
      n = (y == 0) ? c : c - nx;
      s = (y == ny-1) ? c : c + nx;
      b = (z == 0) ? c : c - nx * ny;
      t = (z == nz-1) ? c : c + nx * ny;
      f2_t[c] = cc * f1_t[c] + cw * f1_t[w] + ce * f1_t[e]
        + cs * f1_t[s] + cn * f1_t[n] + cb * f1_t[b] + ct * f1_t[t];
    }
  }
}
```



diffusion_ker04.c

```
for (z = tz*zchunk; z < MIN((tz+1)*zchunk, nz); z++) {
  b = (z == 0) ? 0 : - nx * ny;
  t = (z == nz-1) ? 0 : nx * ny;
  for (y = ty*ychunk; y < MIN((ty+1)*ychunk, ny); y++) {
    n = (y == 0) ? 0 : - nx;
    s = (y == ny-1) ? 0 : nx;
    c = y * nx + z * nx * ny;
    f2_t[c] = cc * f1_t[c] + cw * f1_t[c] + ce * f1_t[c+1]
      + cs * f1_t[c+s] + cn * f1_t[c+n] + cb * f1_t[c+b] + ct * f1_t[c+t];
    c++;
    for (x = 1; x < nx-1; x++) {
      f2_t[c] = cc * f1_t[c] + cw * f1_t[c-1] + ce * f1_t[c+1]
        + cs * f1_t[c+s] + cn * f1_t[c+n] + cb * f1_t[c+b] + ct * f1_t[c+t];
      c++;
    }
    f2_t[c] = cc * f1_t[c] + cw * f1_t[c-1] + ce * f1_t[c]
      + cs * f1_t[c+s] + cn * f1_t[c+n] + cb * f1_t[c+b] + ct * f1_t[c+t];
  }
}
```

- $x = 0$, $0 < x < nx-1$, $x = nx-1$ に処理を分けることで、 $0 < x < nx-1$ の区間では分岐処理なく実行可能

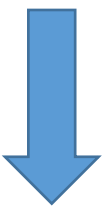
contiguous属性の付与（Fortranのみ）

diffusion_ker05.f90 のプロファイル結果

- contiguous（メモリ上に連続的に確保されている）かどうかわからない配列へのアクセスでは、**効率の悪いgather命令が使われてしまう。**

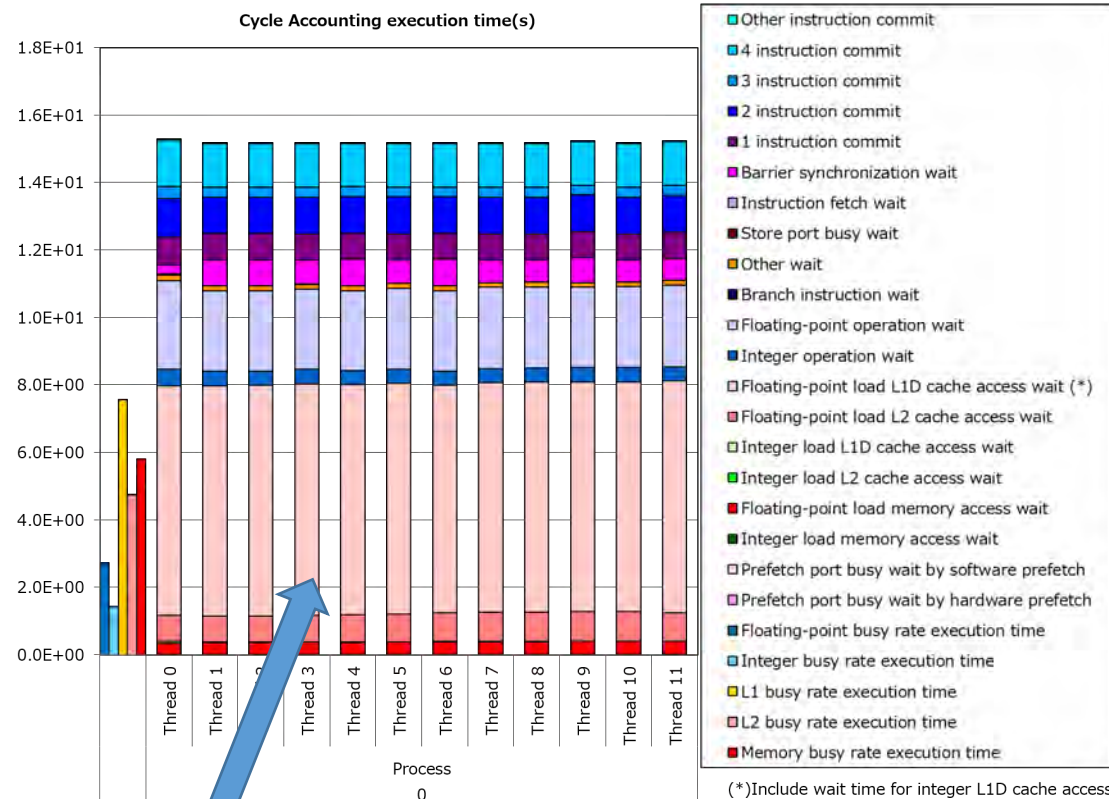
✓ contiguous属性の付与により回避可能

```
diffusion_ker05.f90
subroutine diffusion_ker05_parallel(f1, f2, nx, ny
REAL, dimension(:, :, :), intent(inout) :: f1, f2
```



```
diffusion_ker06.f90
subroutine diffusion_ker06_parallel(f1, f2, nx, ny, nz, ce, c
REAL, dimension(:, :, :), contiguous, intent(inout) :: f1, f2
```

- 関数化された並列領域。**pointer属性を落とすした上で、contiguous属性を付与。**



L1キャッシュアクセス待ちが発生している

Non-contiguous gather load instruction が大量に発生している

	Load instruction				F r e q u e n c y
	Single vector contiguous load instruction	Multiple vector contiguous structure load instruction	Non-contiguous gather load instruction	Broadcast load instruction	
	9.55E+03	0.00E+00	4.52E+09	1.12E+06	2.51E+06
	0.00E+00	0.00E+00	4.51E+09	8.89E+04	6.40E+01
	0.00E+00	0.00E+00	4.51E+09	8.89E+04	6.40E+01
	5.00E+00	0.00E+00	4.51E+09	8.89E+04	6.40E+01
	0.00E+00	0.00E+00	4.51E+09	8.89E+04	6.40E+01
	5.00E+00	0.00E+00	4.51E+09	8.89E+04	6.40E+01
	5.00E+00	0.00E+00	4.51E+09	8.89E+04	6.40E+01
	5.00E+00	0.00E+00	4.51E+09	8.89E+04	6.40E+01
	0.00E+00	0.00E+00	4.51E+09	8.89E+04	6.40E+01

A64FX**必須**最適化まとめ

- 並列領域内のFortran Pointer徹底排除(-Knoaliasではダメ)
 - ✓ ついでに適用可能な配列にはcontiguous属性を付与する
 - ✓ Fortranほどではないが、Cのpointerもrestrictをつけた方が良い
- 自動SIMD化のための最内ループ内の整理
 - ✓ 自作関数を展開してベタ書き（Fortran組込み関数はOK, コンパイラの機能によるインライン展開はNG）
 - ✓ ループ外に出せる分岐処理の全排除
 - ✓ サイズ不明でフルアンローリングできない繰り返し処理（配列代入など）の排除
- CMG間通信ペナルティ排除(1 プロセス/nodeのとき)
 - ✓ `export XOS_MMM_L_PAGING_POLICY=demand:demand:demand`
 - ✓ First touch
 - ✓ スレッドの領域分割方針の変更、4プロセス/nodeへ変更
- メモリ律速なカーネルでは可能な限りのデータ連続配置

通信性能

OSU Micro Benchmarks (Aquarius用)

- オハイオ州立大学 D.K. Pandaのグループが開発, Version 6.0
- インストール場所
/work/share/benchmarks/Aquarius/osu-micro-benchmarks-6.0
/work/share/benchmarks/Aquarius/osu-micro-benchmarks-6.0-mpi414
(自前ビルドOpenMPI-4.1.4: /work/share/openmpi-4.1.4 使用)
- 実行ジョブスクリプト
/work/share/benchmarks/Odyssey/osu-micro-benchmarks-6.0/run/*.sh
/work/share/benchmarks/Odyssey/osu-micro-benchmarks-6.0-mpi414/run/*.sh
- 実行の方法 (osu-micro-benchmarks-6.0)
 1. ディレクトリを作り、ジョブスクリプトをコピー
\$ mkdir omb-6.0-aquarius
\$ cd omb-6.0-aquarius
\$ cp /work/share/benchmarks/Aquarius/osu-micro-benchmarks-6.0/run/*.sh .
 2. ジョブ実行
\$ pjsub osu_latency.sh

Tea Leaf CUDA版

- 熱伝導のベンチマークのCUDA版
 - mantevoプロジェクトのミニアプリの一つ
 - <http://uk-mac.github.io/TeaLeaf/>
- 展開

```
$ tar xvfz TeaLeaf_wa.tar.gz
$ cd TeaLeaf_CUDA-0ecb1a474304c4ea916503fe6ef594f5d3f6703d/
```
- Wisteria-A向けバイナリのコンパイル

```
$ module purge; module load aquarius cuda omp-cuda
$ make COMPILER=GNU
```
- ジョブ実行

```
$ pjsub tea_leaf.sh
```


Singularity

- Tensorflowは、ライブラリの依存性が強く、環境構築が困難
 - ➔ Singularity経由でDockerイメージを利用
 - ファイルサイズが大きいため、みなさんは実行はしないでください：
`$ singularity build tf-image.file docker://tensorflow/tensorflow:latest-gpu`
 - <https://hub.docker.com/r/tensorflow/tensorflow> の latest-gpu タグからダウンロード
- `singularity exec /work/gt00/share/tf-image.file python`
スクリプト名
tf-image.file 内のpythonを使って動作
 - ホストのOSはRedHat 8, コンテナでは Ubuntu 18.04.5
 - ホストのCUDAは11.2, コンテナでは 11.0のように違う環境で実行できる

MNIST学習の実行 (Keras+Tensorflow)

- Tensorflow 2.5.0 (KerasはTensorflowに含まれる)

```
$ singularity exec /work/gt00/share/tf-image.file python -c "import tensorflow as tf; print(tf.__version__, tf.keras.__version__)"  
2.5.0 2.5.0
```

- 以下を実行 (コンパイルは不要！)

```
$ pjsub keras-tf-mnist.bash
```

- 実行が終了したら、以下を実行する

```
$ cat keras-tf-mnist.log.XXXX
```

MNIST by Keras+Tensorflow

- <https://www.tensorflow.org/tutorials/images/cnn?hl=ja>

- 結果はこんな感じ

- Epoch: データセットで学習した回数
- n/1875: 反復回数 (60000サンプル/32バッチ=1875)
- Loss: (学習用画像の) 正解との誤差
- Accuracy: テスト画像での正解率

```
... Successfully opened dynamic library libcudart.so.11.0
Epoch 1/5
1875/1875 [=====...=====] - 9s 4ms/step - loss: 0.1511 - accuracy: 0.9539
...
Epoch 5/5
1875/1875 [=====...=====] - 8s 4ms/step - loss: 0.0219 - accuracy: 0.9935
313/313 - 0s - loss: 0.0316 - accuracy: 0.9914
0.9914000034332275
```