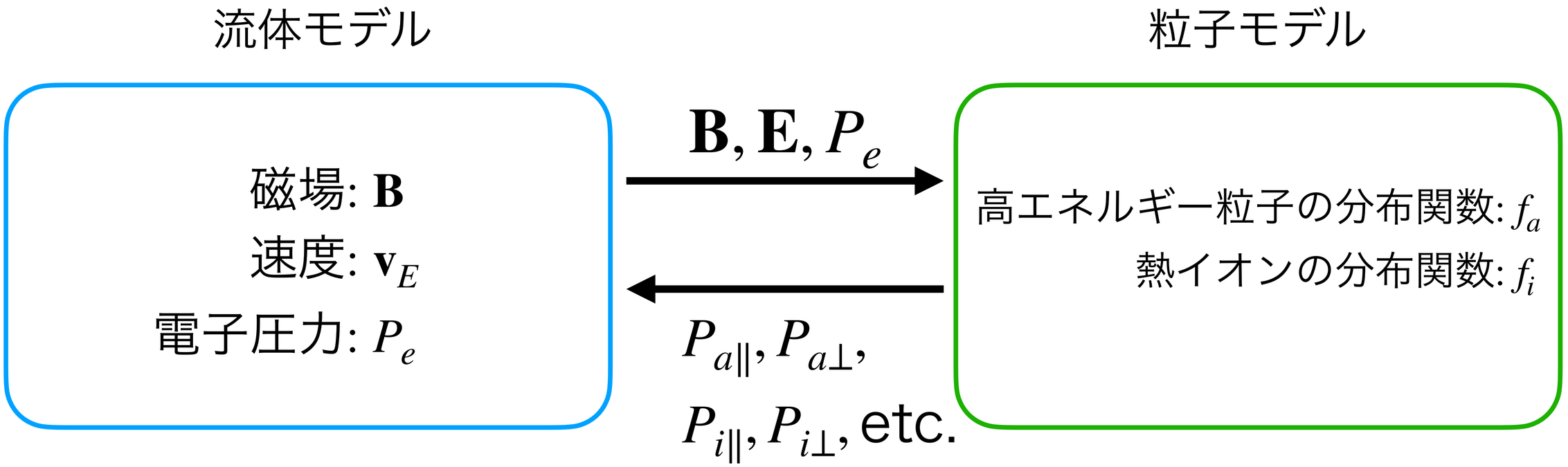
プラズマシミュレータシンポジウム2025
2025年9月11日

MEGAの移植・最適化報告

佐藤雅彦（核融合科学研究所）

MEGAコードで用いている計算モデル

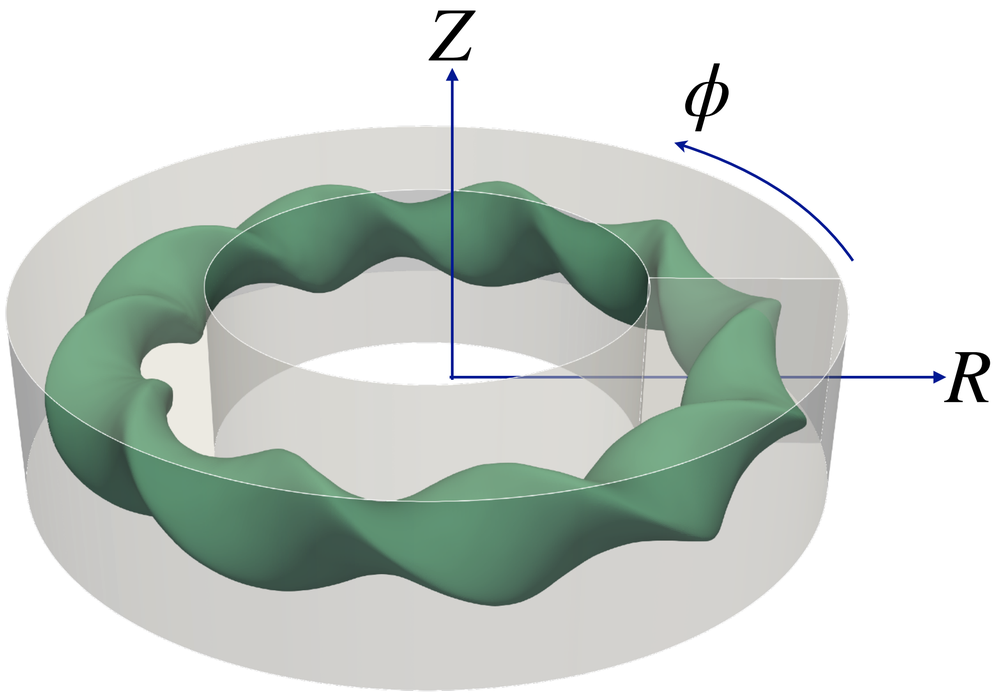


MEGAコードの計算スキーム（流体部分）

- 4次精度の有限差分法による空間方向の離散化

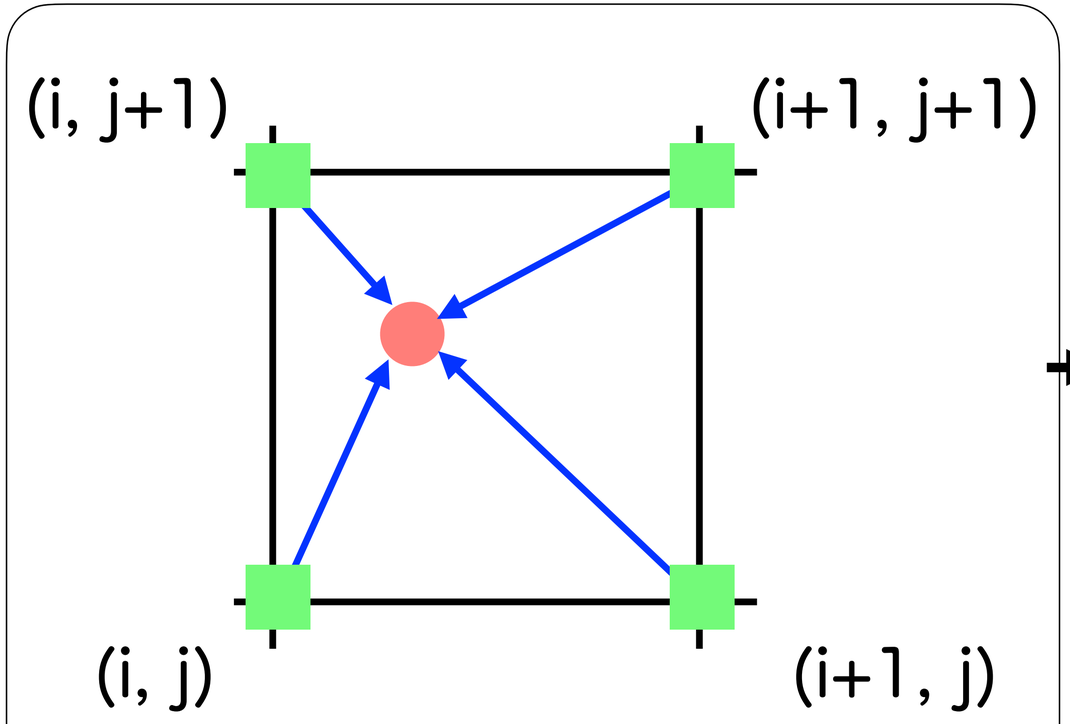
$$\frac{\partial f}{\partial r} \rightarrow \frac{-f_{i+2} + 8f_{i+1} - 8f_{i-1} + f_{i-2}}{12\Delta r}$$

- 4次精度のルンゲクッタ法による時間積分



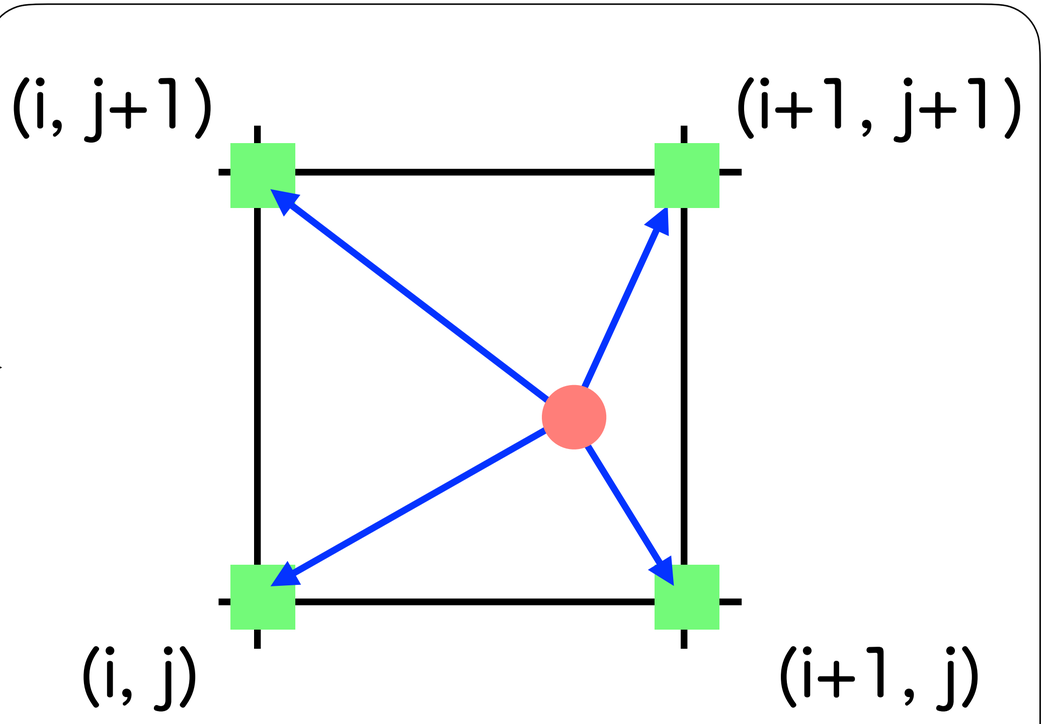
MEGAコードの計算スキーム（粒子部分）

PIC法



粒子が存在するセルの周囲に配置された格子点の場の値を用いて、粒子位置における場の値を算出する。

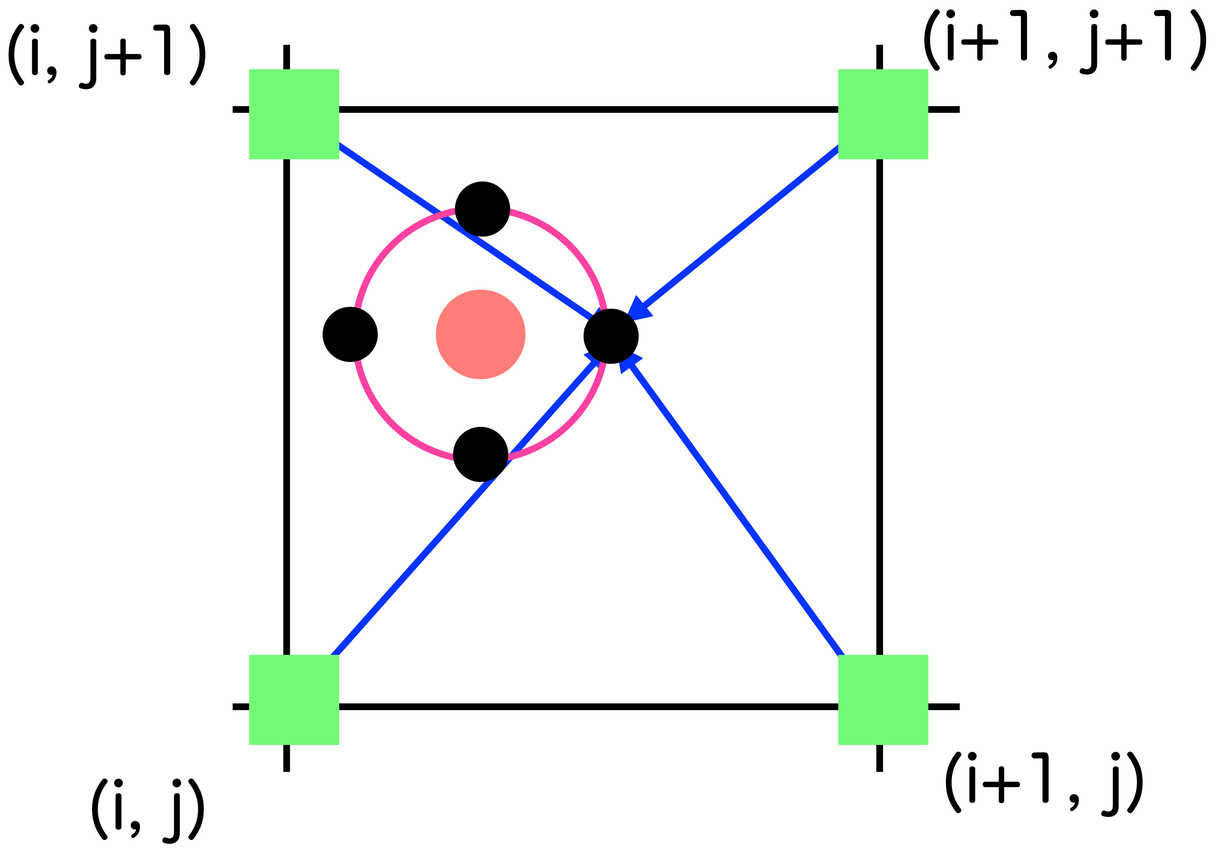
粒子のドリフト速度を求め、粒子を動かす。



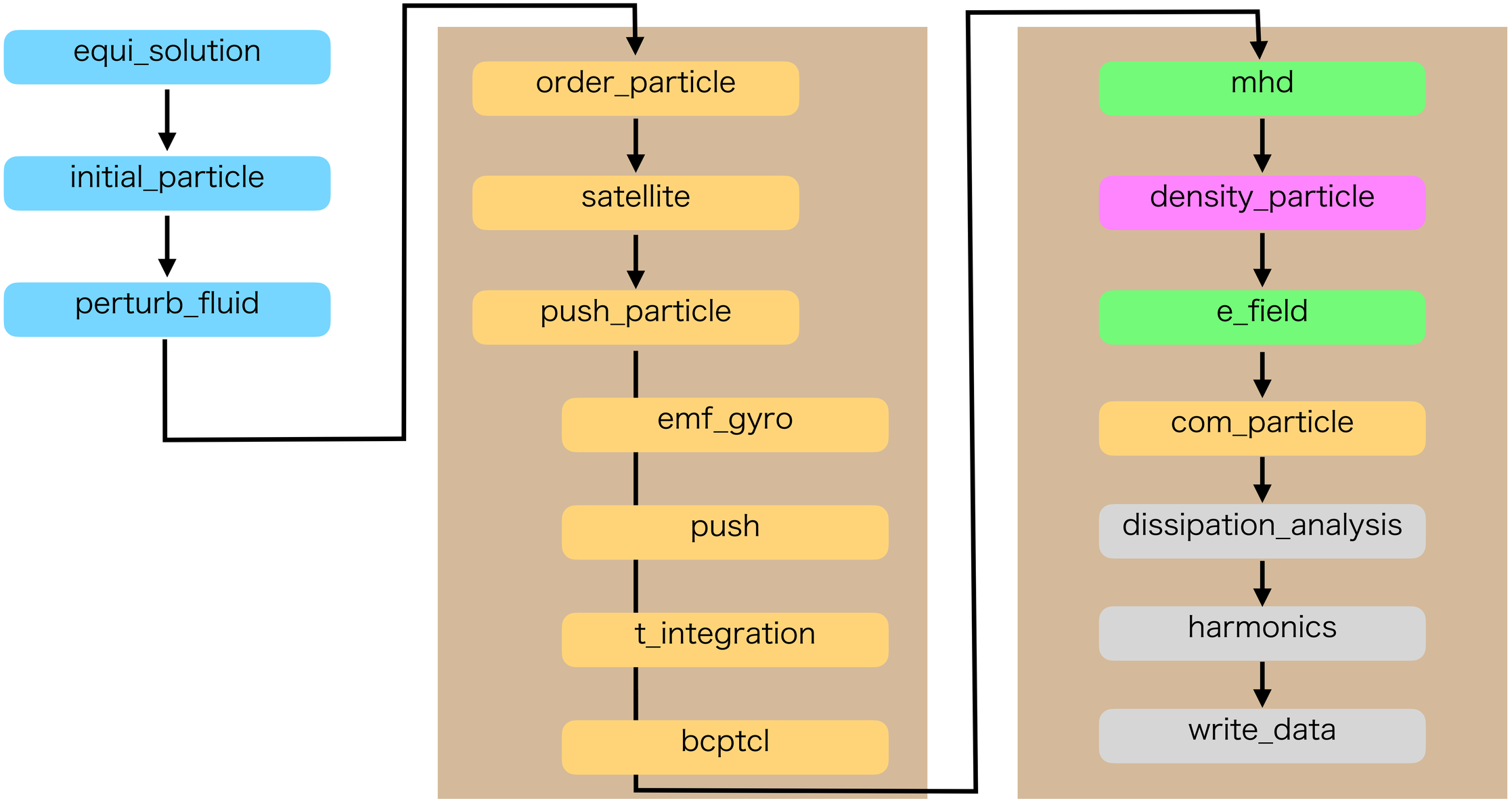
粒子分布関数の速度モーメントから圧力などを算出し、それを粒子の存在するセル周辺の格子点に分配する。

MEGAコードの計算スキーム（粒子部分）

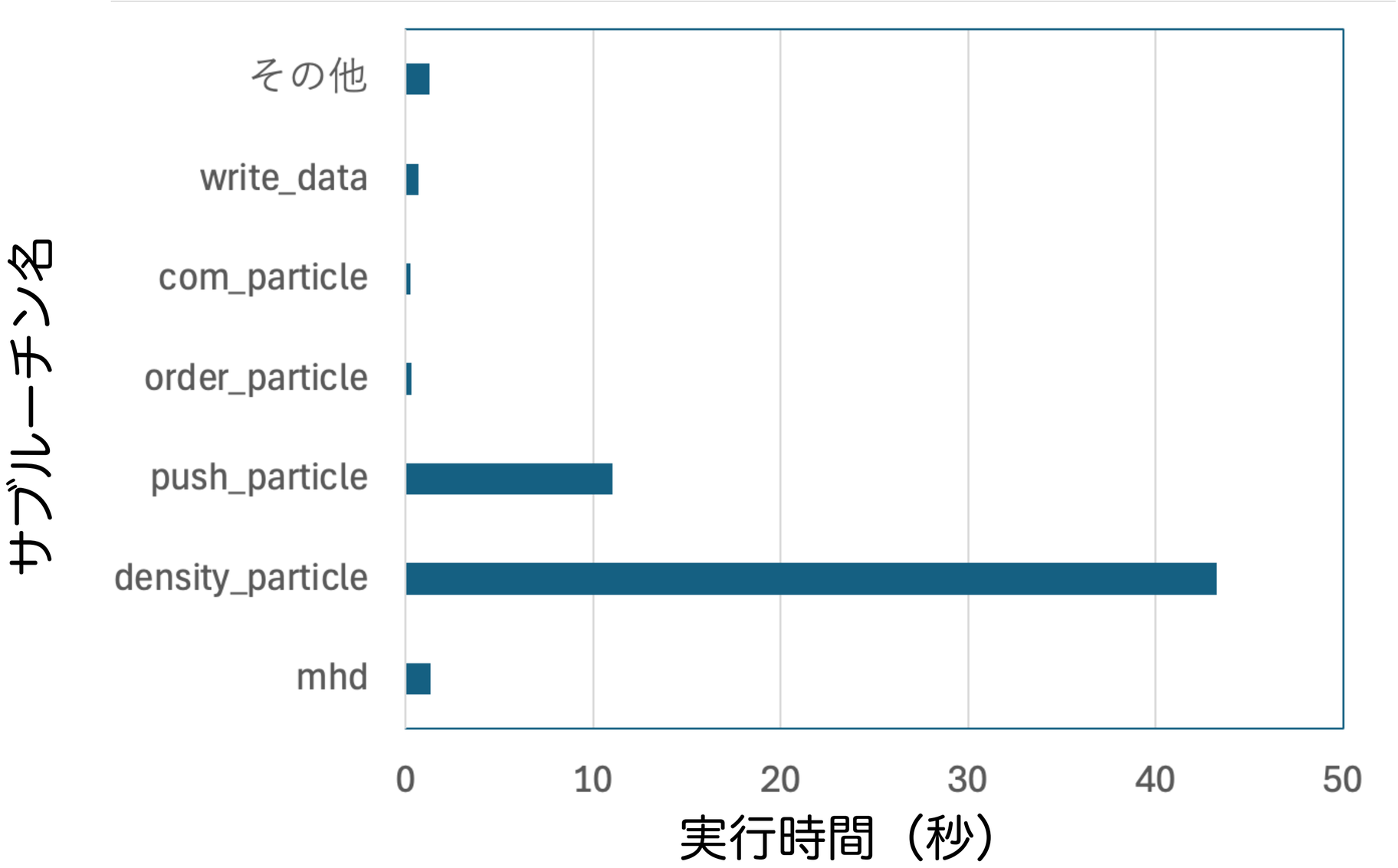
有限ラーマー半径効果



MEGAコードのソースプログラムの構成



各サブルーチンの計算負荷（SX-Aurorra）



GPU最適化の性能目標

スペック比較（GPU/CPU/VE）

① 新PS・サブシステムB：AMD/MI300A(GPU)

② 新PS・サブシステムA：Intel/Xeon 6980P(GNR)

③ 旧PS　　：NEC/SX-Aurora TSUBASA(Type10AE)

	①MI300A	②Xeon（GNR）	性能比（①/②）	③Aurora	性能比（①/③）
FP64 vector演算性能［TFLOPS］	61.3	8.19	7.48	2.43	25.33
FP64 matrix演算性能［TFLOPS］	122.6	---		---	
メモリバンド幅［TB/s］	5.3	0.844	6.28	1.35	3.93
TDP［W］	760	500		300	

- GPU数とVE数が同じになるようにして、性能を比較。
- SX-Auroraよりも3.93倍の高速化を目標とする。

  → 3.50倍の高速化を達成。

GPU最適化の進め方

- OpenMP target指示行によるGPU化
- MPI 1プロセスが1 APUを使用する実装
- ユニファイドメモリ（CPUとGPUのメモリが統合されている）を前提としたGPU化

  ➡ Host-device間のメモリコピーを明示的に制御せずにGPU化を進める（XNACK機能を使用）

GPU最適化の例（１）

- 基本的に、ブロックのループをOpenMP target指示行で並列化
- “teams distribute”でblock並列、“parallel do”でthread並列を有効

```
!$omp target teams distribute parallel do
      do n = nsta, nend
        ivect = n - nsta + 1

        ijk_a(ivect,1)=max(1,min(lr  -1,int((gc(1,n)-ma_mi_r)*dr1  ) + kr  ))
        ijk_a(ivect,2)=max(1,min(lz  -1,int(gc(2,n)          *dz1  ) + kz  ))
        ijk_a(ivect,3)=max(1,min(lphi-1,int(gc(3,n)          *dphi1) + kphi))

        ar1  = max(0.0d0, min(1.0d0, (gc(1,n) - ma_mi_r)*dr1 -  dble(ijk_a(ivect,1) - kr)  ) )
        ar   = 1.0d0 - ar1
        az1  = max(0.0d0, min(1.0d0, gc(2,n)*dz1 - dble(ijk_a(ivect,2) - kz) ) )
        az   = 1.0d0 - az1
        aphi1 = max(0.0d0, min(1.0d0, gc(3,n)*dphi1 - dble(ijk_a(ivect,3) - kphi) ) )
        aphi = 1.0d0 - aphi1
```

GPU最適化の例（２）

・外側ループも並列可能である多重ループについては、collapse指示句を用いてループを結合し、一つの大きなループに展開して並列化を行う。

```
!$omp target teams distribute parallel do collapse(2)
      do kfl = kfl_start, nflp

      do n = nsta, nend
        ivect = n - nsta + 1

        ia=ijk_a(ivect,1)
        ja=ijk_a(ivect,2)
        ka=ijk_a(ivect,3)

        flp(kfl,n) = fld(kfl, ia, ja,   ka  )*aaa(ivect,1) + fld(kfl, ia+1,ja,   ka  )*aaa(ivect,2) &
                   + fld(kfl, ia, ja+1,ka  )*aaa(ivect,3) + fld(kfl, ia+1,ja+1,ka  )*aaa(ivect,4) &
                   + fld(kfl, ia, ja,  ka+1)*aaa(ivect,5) + fld(kfl, ia+1,ja,  ka+1)*aaa(ivect,6) &
                   + fld(kfl, ia, ja+1,ka+1)*aaa(ivect,7) + fld(kfl, ia+1,ja+1,ka+1)*aaa(ivect,8)
        end do

      end do
```

GPU最適化の例（３）

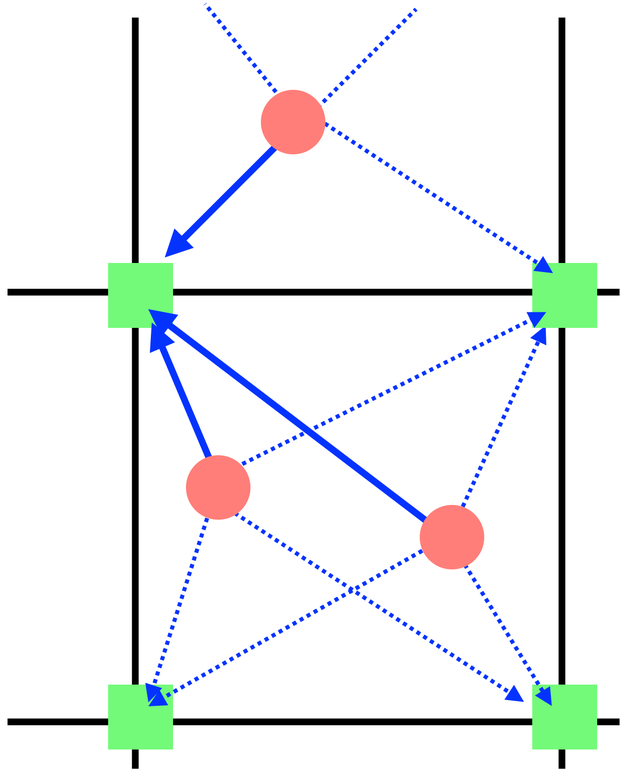
サブルーチン　density_gyro

- SX-Auroraでは「止まり木法」を適用
- 「止まり木法」の適用をやめて、read-modify-writeが衝突する可能性がある処理をatmoic指定

SX-Aurora：43秒
GPU　　：40秒（最適化前：165秒）

- 「止まり木法」によるatomic処理の衝突を緩和することで高速化

➡ 40秒から16秒に。



粒子分布関数の速度モーメントから圧力などを算出し、それを粒子の存在するセル周辺の格子点に分配する。

GPU最適化の例（４）

サブルーチン　density_gyro

・冗長な次元を1次元目として、block内のアクセスを局在化

[修正前]

```
      integer,parameter::nwork=8 ★ ワーク配列数は8 （2～16まで試行し最速）
      real(8),dimension(lr,lz,lphi,nwork)::dns_,mom_,ppara_,pperp_ ★ 冗長化されるのは最後の次元
[...]
!$omp  target teams distribute parallel do
!$omp& private(n,nc,ia,ja,ka,ar1,ar,az1,az,aphi1,aphi,aaa1,aaa2,aaa3,aaa4,aaa5,aaa6,aaa7,aaa8)
!$omp& private(p0,p1,p2,mu1,ib)
       do n = 1, marker_num_gyro
        ib = mod(n-1,nwork)+1
        nc =(n-1)/ngyro + 1
[...]
!$omp atomic update
        dns_(ia,  ja,  ka  ,ib) = dns_(ia,  ja,  ka  ,ib) + aaa1*p0
```

[修正後]

```
      integer,parameter::nwork=32 ★ ワーク配列数は32（8～64まで試行し最速）
      real(8),dimension(nwork,lr,lz,lphi)::dns_,mom_,ppara_,pperp_ ★ 冗長化されるのは最初の次元
[...]
!$omp  target teams distribute parallel do
!$omp& private(n,nc,ia,ja,ka,ar1,ar,az1,az,aphi1,aphi,aaa1,aaa2,aaa3,aaa4,aaa5,aaa6,aaa7,aaa8)
!$omp& private(p0,p1,p2,mu1,ib)
       do n = 1, marker_num_gyro
        ib = mod(n-1,nwork)+1
        nc =(n-1)/ngyro + 1
[...]
!$omp atomic update
        dns_(ib,ia,  ja,  ka  ) = dns_(ib,ia,  ja,  ka  ) + aaa1*p0
```

GPU最適化の例（５）

- OpenMP reduction指示行の対象に配列を指定できないため、reductionの対象となる配列についてはスカラ変数へのたしこみに変更。

修正前

```
!$omp parallel do reduction(+:de_trans_local)
      do k = lphistart, lphiend
      do j = lzstart, lzend
      do i = lrstart, lrend
        de_trans_local(1) = de_trans_local(1) &
       + grr(i,j,k)*( &
            c_ar(i,j,k)*eid_r(i,j,k) &
          + c_az(i,j,k)*eid_z(i,j,k) &
          + c_aphi(i,j,k)*eid_phi(i,j,k) &
                    )
[…]
      de_trans = de_trans_local*dr*dz*dphi*dt
```

修正後

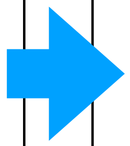
```
!$omp target teams distribute parallel do private(k,j,i) collapse(3) reduction(+:t1,t2,t3,t4,t5)
      do k = lphistart, lphiend
      do j = lzstart, lzend
      do i = lrstart, lrend
        t1 = t1 &
       + grr(i,j,k)*( &
            c_ar(i,j,k)*eid_r(i,j,k) &
          + c_az(i,j,k)*eid_z(i,j,k) &
          + c_aphi(i,j,k)*eid_phi(i,j,k) &
                    )
[…]
      de_trans(1) = t1*dr*dz*dphi*dt
```

GPU最適化の例（６）

- 配列構文をOpenMP target指示行の対象とするとエラーが発生するため、
  配列構文を通常のループに置き換えた上でGPU化を実施。

[修正前]

```
!$omp workshare
        fld(1,:,:,:) = er(:,:,:)
        fld(2,:,:,:) = ez(:,:,:)
        fld(3,:,:,:) = ephi(:,:,:)
        fld(4,:,:,:) = epara(:,:,:)
        fld(5,:,:,:) = br(:,:,:) - br0(:,:,:)
        fld(6,:,:,:) = bz(:,:,:) - bz0(:,:,:)
        fld(7,:,:,:) = bphi(:,:,:) - bphi0(:,:,:)
!$omp end workshare
```



[修正後]

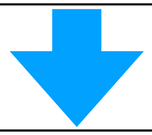
```
!$omp target teams distribute parallel do collapse(3)
        do k=1,lphi
        do j=1,lz
        do i=1,lr
        fld(1,i,j,k) = er(i,j,k)
        fld(2,i,j,k) = ez(i,j,k)
        fld(3,i,j,k) = ephi(i,j,k)
        fld(4,i,j,k) = epara(i,j,k)
        fld(5,i,j,k) = br(i,j,k) - br0(i,j,k)
        fld(6,i,j,k) = bz(i,j,k) - bz0(i,j,k)
        fld(7,i,j,k) = bphi(i,j,k) - bphi0(i,j,k)
        end do
        end do
        end do
```

GPU最適化の例（7）

・OpenMP target指示行の対象に配列を指定できないため、reductionが生じるループを並列化の対象外とし、ループ入れ替えを適用。

修正前

```
!$omp parallel do private(n1,n2) reduction(+:work) !2024-12-16, correction suggested by Jialei Wang
do k = lphistart, lphiend ※ k方向でreduction(縮約)
  do n = 0, lphi_n
    n1 = 2*n + 1
    n2 = 2*n + 2
    do i = 1, lrz
      do j = 1, imulti
        work(j,i,1,n1) = work(j,i,1,n1) + aaa(j,i,1,k)*cos_phi(n,k)
        work(j,i,1,n2) = work(j,i,1,n2) + aaa(j,i,1,k)*sin_phi(n,k)
```



修正後

```
!$omp target teams distribute parallel do collapse(3) private(n,i,j,n1,n2,k)
do n = 0, lphi_n
  do i = 1, lrz
    do j = 1, imulti
      n1 = 2*n + 1 ※ 密なループにするためjループ内に移動
      n2 = 2*n + 2
      do k = lphistart, lphiend ※ k方向はreduction、並列化していない
        work(j,i,1,n1) = work(j,i,1,n1) + aaa(j,i,1,k)*cos_phi(n,k)
        work(j,i,1,n2) = work(j,i,1,n2) + aaa(j,i,1,k)*sin_phi(n,k)
```

GPU最適化作業において問題となった箇所

- Fortranの組み込み数学関数erfc（相補誤差関数）のGPU版が未実装

  ➡ netlibからソースコードをコピーし、それを加工して使用。

- GPU版の実数のmod関数が存在しないため、mod関数の定義式に従って書き換え。

  ➡ mod(a, b) → a - int(a / b)*b

- 乱数生成処理をAMD GPU向けのhipRANDライブラリで行うように修正。（次ページ）

  ➡ 80 APUで実行した際、初期処理の実行時間が54.3秒から10.8秒に短縮。

乱数生成処理の修正

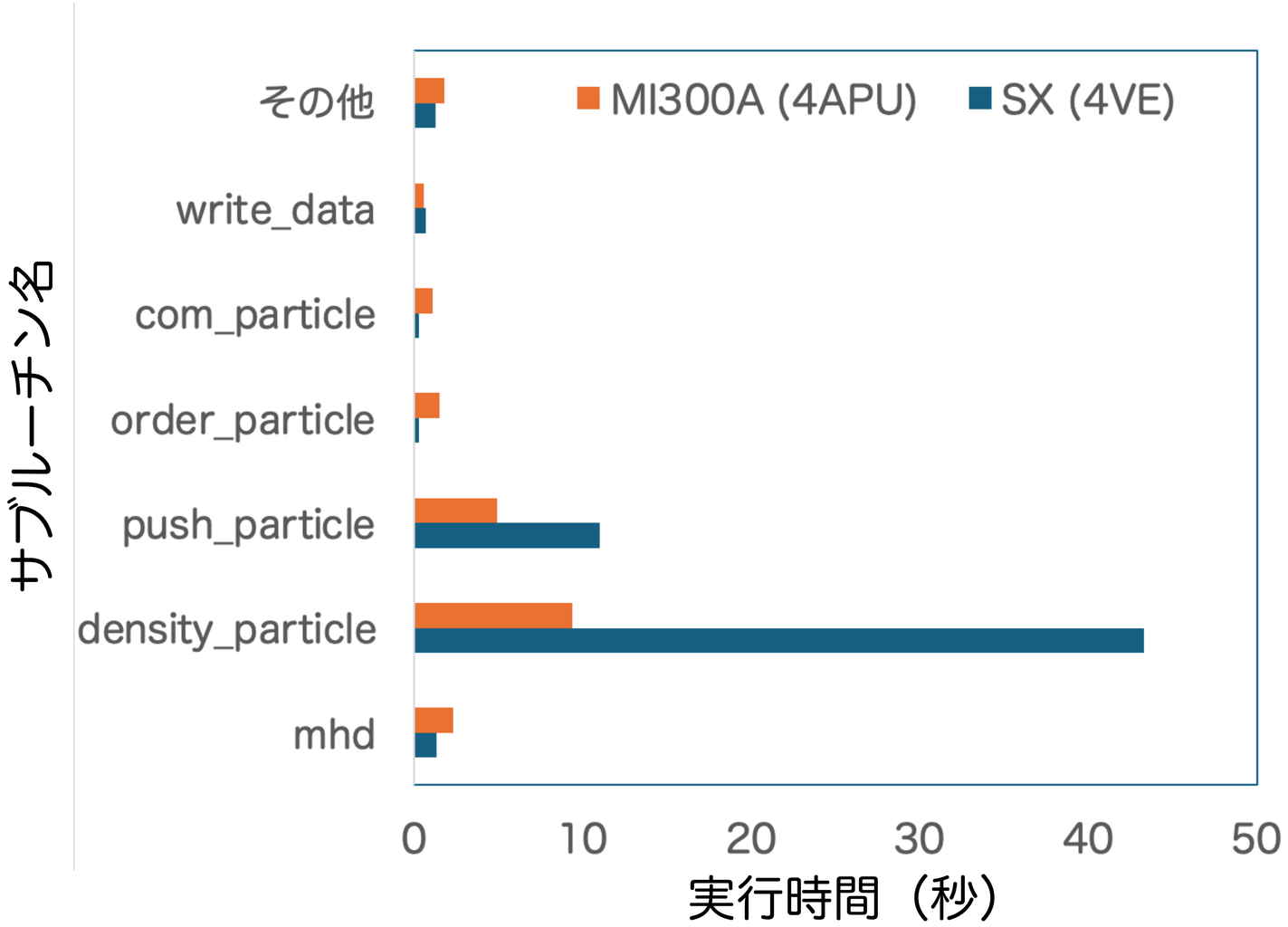
- C言語によるラッパ関数を作成（右図）
- サブルーチンransuuからこれらを適宜呼び出す（下図）

```
[mega_opt1a.f90]
[…]
subroutine ransuu(nrandom, rand_no)
[…]
#elif AMDGPU // プリプロセサマクロAMDGPU指定時
      if (iflag==0) then
        call hiprand_init()  // 初期化
        iflag = 1
      end if

      allocate(rand_no_f95(nrandom))

      do k_rank = 0, my_rank
        call hiprand_gen(nrandom, rand_no_f95) // 乱数生成
      end do

!$omp target teams distribute parallel do private(i)
      do i = 1, nrandom
        rand_no(i) = rand_no_f95(i)
      end do

      do k_rank = my_rank + 1, nprocess-1
        call hiprand_gen(nrandom, rand_no_f95) // 乱数生成
      end do
[…]
```

```
[hiprandwrapper.cu]

include <stdio.h>
#include <stdlib.h>
#include <hiprand/hiprand.h>

hiprandGenerator_t gen;

extern “C” void hipdevicesynchronize_(){ // GPUとCPUの同期を行う
   hipDeviceSynchronize();                // HIP関数の呼び出し
}

extern “C” void hiprand_init_(){ // 初期化
    hiprandCreateGenerator(&gen, HIPRAND_RNG_PSEUDO_DEFAULT);
    hipDeviceSynchronize(); // GPUとCPUの同期、必須

    hiprandSetPseudoRandomGeneratorSeed(gen, 1234ULL); // seedは固定
    hipDeviceSynchronize(); // GPUとCPUの同期、必須
}

extern “C” void hiprand_gen_(int *n_, double *a){ // 乱数生成
    hiprandGenerateUniformDouble(gen, a, *n_);
    hipDeviceSynchronize(); // GPUとCPUの同期、必須
}

extern “C” void hiprand_finalize_(){  // 終端化
    hiprandDestroyGenerator(gen);
    hipDeviceSynchronize(); // GPUとCPUの同期、必須
}
```

性能評価（１）

メッシュ数：128x16x128, 粒子数：16個/メッシュ



SX (4VE)：58.1秒

2.7倍の高速化

MI300A (4APU)：21.6秒

（最適化前：670.3秒）

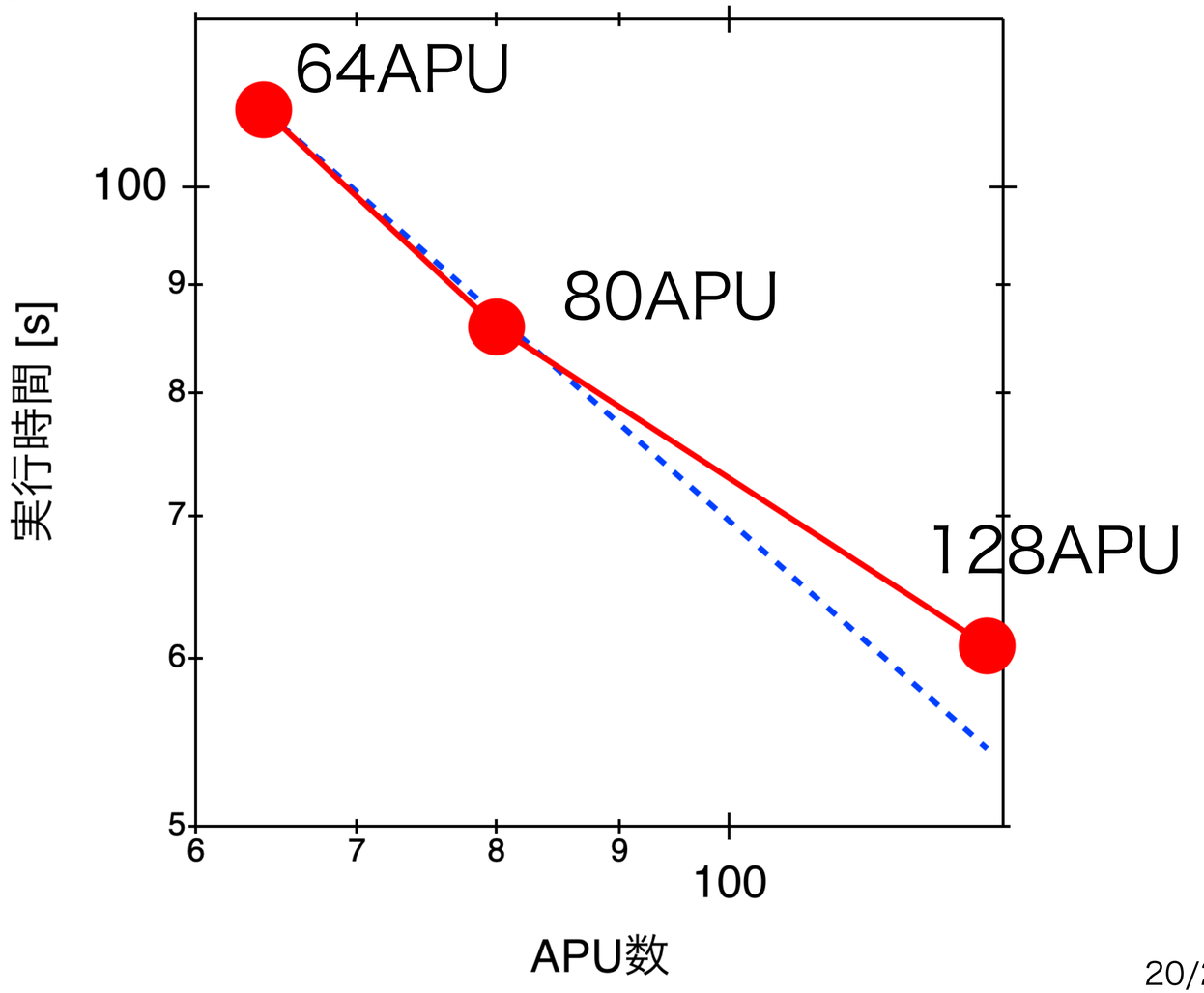
性能評価（2）

メッシュ数：256x320x256, 粒子数：16個/メッシュ
（メッシュ数を80倍）

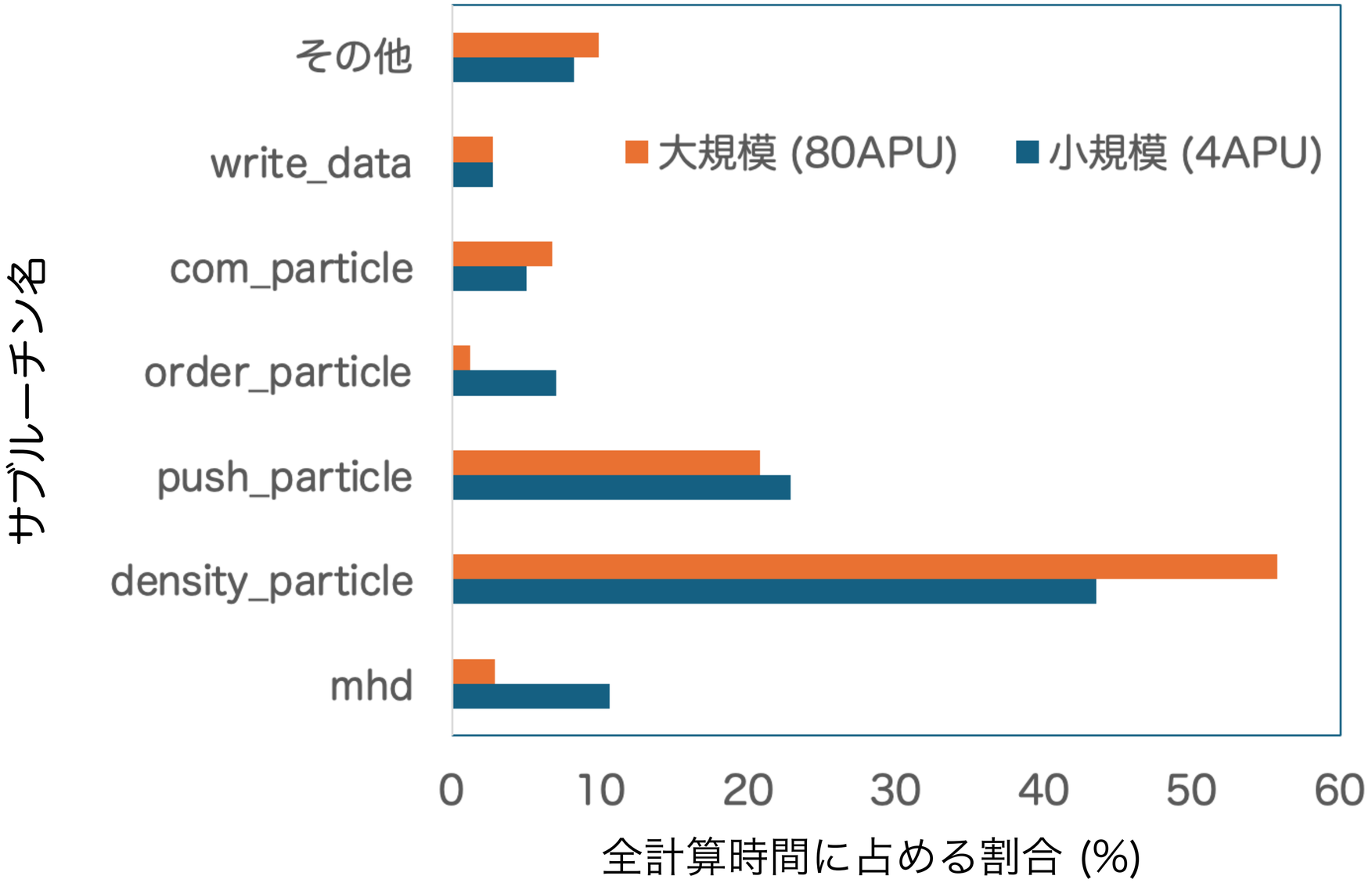
SX (80VE) : 301.1秒

↓ 3.5倍の高速化

MI300A (80APU) : 85.9秒



性能評価（３）


その他
write_data
com_particle
order_particle
push_particle
density_particle
mhd
大規模 (80APU)
小規模 (4APU)
サブルーチン名
0
10
20
30
40
50
60
全計算時間に占める割合 (%)

問題点

- 少ないAPU数（32, 40APU）でエラーとなる。
  - メモリ量としては十分と思われるが、メモリ不足と思われる挙動を示す。
  - GPU化前のコードでCPU実行しても同様。
  - コンパイラ（amdflang）の問題？

まとめ

- OpenMP target指示句によるGPU化。
  - 一番計算負荷が大きい、イオンの分布関数の速度モーメントから圧力等を計算するサブルーチンには、「止まり木法」が有効であった。
- SX-Auroraと比較して、約3.5倍の高速化を達成。
  - 目標値：約3.9倍（メモリ性能比）
- 少ないAPU数でエラーとなる問題あり。