2026年度GPU講習会（3日目）

三木 洋平
（東京大学 情報基盤センター）

2026年度GPU講習会@国立天文台三鷹キャンパス

3日目(8/20)の予定

- 9:00--9:45　　前日の復習
- 10:00--11:00　　指示文を使用したGPU化1　　[講義・演習]
- 11:15--12:15　　指示文を使用したGPU化2　　[演習]
- 13:30--14:30　　指示文＋CUDA混合コードの実装　　[講義・演習]
- 14:45--15:45　　MPIを使用したマルチGPUコードの実装　　[講義]
- 16:00--17:00　　最近＋近未来のGPUコンピューティング　　[講義]

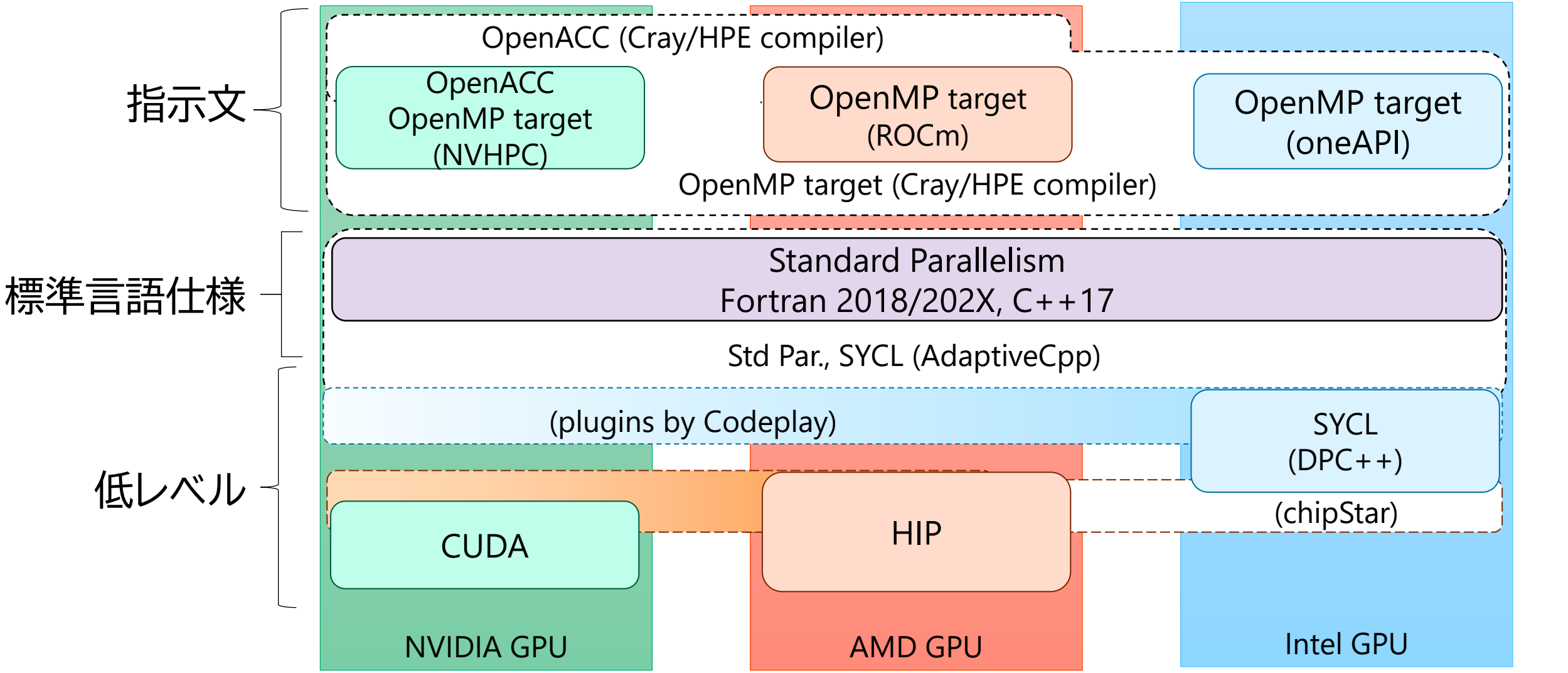
- 最終日は皆さんが自身のコードをGPU化する際に知っておくと便利な情報を提供することを目的にします

3日目(8/20)の予定

- 9:00--9:45　前日の復習
- 10:00--11:00　指示文を使用したGPU化1　[講義・演習]
- 11:15--12:15　指示文を使用したGPU化2　[演習]
- 13:30--14:30　指示文＋CUDA混合コードの実装　[講義・演習]
- 14:45--15:45　MPIを使用したマルチGPUコードの実装　[講義]
- 16:00--17:00　最近＋近未来のGPUコンピューティング　[講義]

GPU向けのプログラミング環境

2024年2月のPCCC AI/HPC OSS活用WSでの講演資料
(https://www.pccluster.org/ja/event/data/240205_pccc_wsAI-HPC-OSS_06_hanawa-miki.pdf)



GPUプログラミング手法の比較表

- Fortran の「OK」は動作するという意味. 性能については別問題

手法	ベース言語など	強み	弱み	Fortran対応	対象GPUベンダー
CUDA C++	C++	詳細な最適化可能 最新機能が使える	記述量が多い GPU専用コード	CUDA Fortran	NVIDIA限定
HIP	C++	詳細な最適化可能 ほぼCUDA C++	記述量が多い GPU専用コード	hipfort	AMD, NVIDIA （Intel: chipStar?）
SYCL	C++	詳細な最適化可能 ラムダ式	記述量が多い ラムダ式	N/A	Intel, NVIDIA, AMD
OpenACC	指示文	移植コスト低 CPUコードと共通化可能	CUDAより遅い （メモリ律速ならそれなり？）	OK	ほぼNVIDIA限定 （HPE Cray コンパイラはAMDも対応）
OpenMP （target指示文）	指示文	移植コスト低 CPUコードと共通化可能	CUDAより遅い （メモリ律速ならそれなり？）	OK	NVIDIA, AMD, Intel
言語の標準規格	C++17以降 Fortran 2008	CPUでも同じコードが動く	多くの制約あり CUDAより遅い	Fortran 2008	NVIDIA, AMD, Intel

大まかな考え方の違い

- 簡易GPU化(OpenACC, OpenMP target, C++17, …)
  - 並列化可能なループであることをコンパイラに伝える
    (自明な並列度があるループであるので, 簡単にGPU化できるループ)
  - 「どうGPU化するか」はコンパイラ任せ
  - なるべくコンパイラの自由度を増やしてあげる方が速くなる傾向
  - 初心者・初級者向け
  - 既存のCPUコードのGPU化という意味ではこちらの方が低コスト
- 自力でのGPU化(CUDA, HIP, SYCL, …)
  - 「どうGPU化するか」を自分で考え, 自分で実装
  - どういう演算命令を使うか, など自分で何でもできる
  - OpenACC, OpenMP target, C++17ではGPU化が難しいループであってもGPU化できる
    - 例:重力ツリーコードの完全GPU実装は指示文では手に負えないが, CUDAなどでは実装可能
  - 中・上級者向け. 指示文でできなかった時にこちらの採用を検討

指示文ベースでGPU化したい場合の選択肢

- OpenACC
  - GPU向けのメジャーな指示文
  - PGIがNVIDIAに買収された結果, NVIDIA色が強くなってしまった
  - AMD, Intelは(きっと)サポートしない
    - HPE Crayコンパイラであれば, AMD GPU向けのOpenACCもサポート
    - IntelはOpenACCからOpenMP targetへの変換ツールを開発中
    - ➔GPUベンダー(AMD, Intel)による直接支援が受けられない
- OpenMPのtarget指示文
  - OpenMP 4.0以降でアクセラレータへのオフロードがサポート
  - OpenMP 5.0で loop 指示節が追加, OpenACC的実装も可能に
  - NVIDIA, AMD, Intel 全てのGPU向けにサポートされる
  - 現時点ではOpenACCの全ての機能に対応できていない
    - 非同期実行の(細やかな)制御など
- 実装時には, 使う指示文についても選択する必要がある

GPU向け指示文統合マクロSolomon

- OpenACCとOpenMP両方に対応した指示文の追加例(右側)
  - 場合分け(ACC for GPU, OMP for CPUなど)がかなり煩雑
  - `$ nvc++ -acc=multicore -mp=gpu …` も(見かけないが)実は可能
- プリプロセッサマクロを用いてインターフェースを統合するライブラリを開発
  - https://github.com/ymiki-repo/solomon で公開
  - Miki & Hanawa (2024, IEEE Access)
  - 手品の種: `_Pragma()`形式で指示文を記述
  - 対応しているバックエンド:
    - OpenACC, OpenMP target, OpenMP
- やる気が出るのはどちらの手法?
  - 通常の(煩雑な)実装方法➡
  - ⬇ Solomon を用いて簡易化した手法

```
OFFLOAD(AS_INDEPENDENT, NUM_THREADS(NTHREADS))
for(int i = 0; i < N; i++){
```

```
#ifdef OFFLOAD_BY_OPENACC
#ifdef OFFLOAD_BY_OPENACC_PARALLEL
#pragma acc parallel vector_length(NTHREADS)
#pragma acc loop independent
#else//OFFLOAD_BY_OPENACC_PARALLEL
#pragma acc kernels vector_length(NTHREADS)
#pragma acc loop independent
#endif//OFFLOAD_BY_OPENACC_PARALLEL
#endif//OFFLOAD_BY_OPENACC
#ifdef OFFLOAD_BY_OPENMP_TARGET
#ifdef OFFLOAD_BY_OPENMP_TARGET_DISTRIBUTE
#pragma omp target teams distribute parallel for simd
thread_limit(NTHREADS)
#else//OFFLOAD_BY_OPENMP_TARGET_DISTRIBUTE
#pragma omp target teams loop thread_limit(NTHREADS)
#endif//OFFLOAD_BY_OPENMP_TARGET_DISTRIBUTE
#endif//OFFLOAD_BY_OPENMP_TARGET
  for(int i = 0; i < N; i++){
```

GPUプログラミングに関する資料

- HAIRDESC GPUプログラミング教材
  - https://hairdesc.jp/gpumaterial/
- 「UTokyo N-Ways to GPU Programming Bootcamp」講習会資料
  - https://www.cc.u-tokyo.ac.jp/events/lectures/207/
  - 講習会の録画も公開されています
  - ISO標準言語, OpenACC, CUDA
- 「GPUプログラミング入門」講習会資料
  - https://www.cc.u-tokyo.ac.jp/events/lectures/251/
  - OpenACC
- 「指示文とMPIによるマルチGPUプログラミング入門」講習会資料
  - https://www.cc.u-tokyo.ac.jp/events/lectures/257/
  - 指示文(OpenACC, OpenMP target, Solomon)+MPI
- GPU移行に関するポータルサイト
  - https://jcahpc.github.io/gpu_porting/

指示文でのGPU化方針

1. まずは演算部分のGPU実装のみに注力
   - (マルチコアCPU向けの)OpenMP実装されていれば,
     `#pragma omp parallel for` をGPU向けの指示文に置き換えていく
   - メモリ上のデータがCPU側にあるか, GPU側にあるかは一旦忘れて実装
   - GPU上のメモリ確保, CPU-GPU 間のデータ転送を処理系に全てお任せ
     - NVIDIA, AMDともにこうしたGPUプログラミングサポート機能を提供している
2. (処理系任せのCPU-GPU間のデータ転送性能に満足できない場合)
   CPU・GPUメモリを明示する指示文を使ってコードをアップデート
   - 例えばNVIDIA GPU向けのManaged Memory では, 一旦メモリを読んで, ページフォルトがあればハードウェアレベルでページ単位で転送という仕組み
   - 必要なデータ転送は自分で指示した方が必然的に速くなる
   - GPUDirect系の機能を活用する際には,
     GPU上にデータが置かれていることを保証できる実装手法を使いたくなる

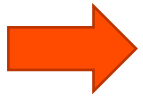
指示文でのGPU化方針

1. まずは演算部分のGPU実装のみに注力
   - (マルチコアCPU向けの)OpenMP実装されていれば,
     `#pragma omp parallel for` をGPU向けの指示文に置き換えていく
   - メモリ上のデータがCPU側にあるか, GPU側にあるかは一旦忘れて実装
   - GPU上のメモリ確保, CPU-GPU 間のデータ転送を処理系に全てお任せ
     - NVIDIA, AMDともにこうしたGPUプログラミングサポート機能を提供している
2. (処理系任せのCPU-GPU間のデータ転送性能に満足できない場合)
   CPU・GPUメモリを明示する指示文を使ってコードをアップデート
   - 例えばNVIDIA GPU向けのManaged Memory では, 一旦メモリを読んで, ページフォルトがあればハードウェアレベルでページ単位で転送という仕組み
   - 必要なデータ転送は自分で指示した方が必然的に速くなる
   - GPUDirect系の機能を活用する際には,
     GPU上にデータが置かれていることを保証できる実装手法を使いたくなる

OpenACCでのループ並列化(kernels構文)

- #pragma omp parallel for を置き換えていく
  - スレッド並列化可能なループであることの目印
- #pragma acc kernels はGPUで並列化するかコンパイラが最終判断
  - GPU化はするが逐次実行になることもある
    (並列化できない場合, 性能が出ないと予想される場合)
- #pragma acc kernels だけでもGPUで並列化してくれるが,
  #pragma acc loop もつけることが多い(最適化などの追加指示に使う)

```
#pragma omp parallel for
  for(int i = 0; i < N; i++){
    // main computation
  }
```

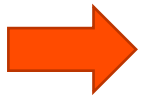


```
#pragma acc kernels
  for(int i = 0; i < N; i++){
    // main computation
  }

#pragma acc kernels
#pragma acc loop
  for(int i = 0; i < N; i++){
    // main computation
  }
```

OpenACCでのループ並列化(parallel構文)

- #pragma omp parallel for を置き換えていく
  - スレッド並列化可能なループであることの目印
- #pragma acc parallel はGPU化可能だとユーザーが保証(自己責任)
  - 対象リージョンはGPU化される
  - 元々#pragma omp parallel forがついていたループであれば,
    スレッド並列によるGPU化が可能と考えて問題ない
  - kernelsではGPU化されなかったが, 本来GPU化できるはずという時に使う
- #pragma acc parallel 単体ではGPU化領域の指定のみなので,
  #pragma acc loop もつけて並列化対象のループを指示する

```
#pragma omp parallel for
  for(int i = 0; i < N; i++){
    // main computation
  }
```



```
#pragma acc parallel
#pragma acc loop
  for (int i = 0; i < N; i++) {
    // main computation
  }
```

OpenACCでの最適化などの追加指示例

- parallel/kernels 構文での違いは特にない
- 各処理が独立に実行できることを保証
  - loop指示文にindependentを追加
  - コンパイラが並列性を十分に見きれなかった場合

```
#pragma acc parallel
#pragma acc loop independent
  for (int i = 0; i < N; i++) {
    // main computation
  }
```

- 多重ループの結合
  - loop指示文にcollapse(n)を追加
  - 大外ループだけでは並列度が足りない場合
  - Tightly-nested loopでないといけない

```
#pragma acc parallel
#pragma acc loop collapse(3)
  for (int i = 0; i < Nx; i++) {
    for (int j = 0; j < Ny; j++) {
      for (int k = 0; k < Nz; k++) {
        // main computation
      }
    }
  }
```

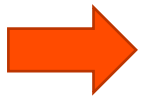
- リダクション演算
  - loop指示文にreduction(…)を追加
  - CPU向けOpenMPと同じ

```
#pragma acc parallel
#pragma acc loop reduction(+ : sum)
  for (int i = 0; i < N; i++) {
    // main computation
    sum += val[i];
  }
```

OpenMPでのループ並列化(distribute指示文版)

- #pragma omp parallel for を置き換えていく
  - スレッド並列化可能なループであることの目印
- #pragma omp target teams distribute parallel for に置き換え
  - target teams distribute を挿入
  - target: デバイス(GPU)上で処理したいということを指示
  - teams: 複数のスレッドからなるグループ(チーム)を作成し, 処理をチーム間で分担
  - distribute: teamsで生成した複数のチームに対して, 処理を分配
    - スレッドの生成・並列化処理については parallel for が担当

```
#pragma omp parallel for
  for(int i = 0; i < N; i++){
    // main computation
  }
```



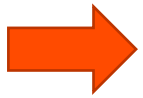
```
#pragma omp target teams distribute parallel for
  for(int i = 0; i < N; i++){
    // main computation
  }
```

- ループ内依存性などで並列化すべきでない処理についてもGPU内並列化するので, GPU内並列化して良いかはユーザーが慎重に判断する
  - ユーザー側の責任という点は, マルチコアCPU向けのOpenMP並列と全く同じ

OpenMPでのループ並列化(loop指示文版)

- `#pragma omp parallel for` を置き換えていく
  - スレッド並列化可能なループであることの目印
- `#pragma omp target teams loop`に置き換え
  - (先述の)`distribute`よりも, コンパイラに多くを委ねる実装法
    - `distribute parallel for` と書くかわりに, 単に `loop` と書くだけなので簡潔
  - OpenMP 5.0 で導入された記法

```
#pragma omp parallel for
  for(int i = 0; i < N; i++){
    // main computation
  }
```



```
#pragma omp target teams loop
  for(int i = 0; i < N; i++){
    // main computation
  }
```

- ループ内依存性などで並列化すべきでない処理についてもGPU内並列化するので, GPU内並列化して良いかはユーザーが慎重に判断する
  - ユーザー側の責任という点は, マルチコアCPU向けのOpenMP並列と全く同じ
- (個人的な経験として)AMD MI300A上ではCPU上で動いてしまう模様
  - Sirius 向けのGPU化という意味では, こちらは使わないほうが良いかも

OpenMPでの最適化などの追加指示例

- 各処理が独立に実行できることを保証
  - simdを追加(仕様上はdistributeの場合のみ使える)
    - NVIDIA HPC SDKでは, loop指示文にも追加できた

```
#pragma omp target teams distribute parallel for simd
  for(int i = 0; i < N; i++){
    // main computation
  }
```

- 多重ループの結合
  - collapse(n)を追加

```
#pragma omp target teams distribute parallel for collapse(3)
  for(int i = 0; i < Nx; i++){
    for(int j = 0; j < Ny; j++){
      for(int k = 0; k < Nz; k++){
        // main computation
      }
    }
  }
```

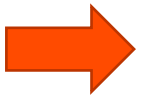
- リダクション演算
  - reduction(…)を追加
  - CPU向けOpenMPと同じ

```
#pragma omp target teams distribute parallel for reduction(+:sum)
  for(int i = 0; i < N; i++){
    // main computation
    sum += val[i];
  }
```

Solomonでのループ並列化

- `#pragma omp parallel for` を置き換えていく
  - スレッド並列化可能なループであることの目印
- `OFFLOAD()` に置き換え
  - `solomon.hpp` をインクルードしておく
  - OpenACC的記法, OpenMP的記法もあるが, 簡易記法の方がきっと直感的
    - `PRAGMA_ACC_KERNELS_LOOP()`, `PRAGMA_ACC_PARALLEL_LOOP()`
    - `PRAGMA_OMP_TARGET_TEAMS_DISTRIBUTE_PARALLEL_FOR()`, `PRAGMA_OMP_TARGET_TEAMS_LOOP()`
  - 実際に使われる指示文は OpenACC/OpenMP targetになるため, 制約は同じ
  - OpenACC/OpenMP targetが使えない環境ではCPU向けのOpenMPが動作: `OFFLOAD()` は `#pragma omp parallel for` に置き換わる
    - CPU向けのOpenMP並列を入れたつもりで, 実は同時にGPU化も対応, とできるかも (たぶん誰も試したことのない方式だと思いますが)

```
#pragma omp parallel for
  for(int i = 0; i < N; i++){
    // main computation
  }
```



```
OFFLOAD()
  for(int i = 0; i < N; i++){
    // main computation
  }
```

Solomonでの最適化などの追加指示例

- 各処理が独立に実行できることを保証
  - AS_INDEPENDENTを追加
    - ACC_CLAUSE_INDEPENDENT, OMP_TARGET_CLAUSE_SIMD でも良い(各記法を混ぜても良い)

```
OFFLOAD(AS_INDEPENDENT)
  for(int i = 0; i < N; i++){
    // main computation
  }
```

- 多重ループの結合
  - COLLAPSE(n)を追加

```
OFFLOAD(COLLAPSE(3))
  for(int i = 0; i < Nx; i++){
    for(int j = 0; j < Ny; j++){
      for(int k = 0; k < Nz; k++){
        // main computation
      }
    }
  }
```

- リダクション演算
  - REDUCTION(…)を追加
  - CPU向けOpenMPと同じ

```
OFFLOAD(REDUCTION(+:sum))
  for(int i = 0; i < N; i++){
    // main computation
    sum += val[i];
  }
```

Solomon 使用にあたっての注意点, 推奨事項など

- 指示節などはカンマ区切りで入力する
  - 各指示節が適用可能かを(Solomon側で)判定し, OKなもののみを有効化する実装
  - OpenMP的記法をOpenACCバックエンドで動かすために実装した機能
    - `_Pragma("omp target teams loop collapse(3) thread_limit(128)")`
      ➔ `_Pragma("acc kernels vector_length(128)") _Pragma("acc loop collapse(3)")`
    - 指示節の適切な振り分けは, Solomon 側で対応すべき機能の一つ
- OpenACCでもOFFLOAD(…), PRAGMA_ACC_[KERNELS PARALLEL]_LOOP(…) の使用を推奨
  - PRAGMA_ACC_[KERNELS PARALLEL](…)とPRAGMA_ACC_LOOP(…)に分けて書くと, OpenMP target への適切な変換が困難になる(LOOP側に入力した指示節が渡らなくなる)
- PRAGMA_OMP_TARGET_DATA(…) は非推奨
  - OpenACC における data, host_data と OpenMP target の data が対応
  - バックエンドを OpenACC にした時にエラーとなる場合がある
  - 推奨:PRAGMA_ACC_[DATA HOST_DATA](…), DATA_ACCESS_BY_[DEVICE HOST](…)

ブロックあたりのスレッド数を示唆する方法

- いくつかのスレッドの塊をGPU内の演算コアグループ(SM/CU/EU)に割り当てるが, このときのスレッド数は代表的な性能最適化パラメータの一つ
  - 無指定であれば, コンパイラがデフォルト値を使ってくれる(指示文の場合)
  - NVIDIA GPUの場合は32の倍数にしておく
  - AMD GPUの場合は64の倍数にしておく
- OpenACC

```
#pragma acc parallel vector_length(128)
#pragma acc loop
  for (int i = 0; i < N; i++) {
    // main computation
  }
```

```
#pragma acc parallel
#pragma acc loop vector(128)
  for (int i = 0; i < N; i++) {
    // main computation
  }
```

- OpenMP target

```
#pragma omp target teams distribute parallel for thread_limit(128)
  for(int i = 0; i < N; i++){
    // main computation
  }
```

- Solomon

```
OFFLOAD(NUM_THREADS(128))
  for(int i = 0; i < N; i++){
    // main computation
  }
```

指示文でのGPU化方針

1. まずは演算部分のGPU実装のみに注力
   - （マルチコアCPU向けの）OpenMP実装されていれば,
     `#pragma omp parallel for` をGPU向けの指示文に置き換えていく
   - メモリ上のデータがCPU側にあるか, GPU側にあるかは一旦忘れて実装
   - GPU上のメモリ確保, CPU-GPU 間のデータ転送を処理系に全てお任せ
     - NVIDIA, AMDともにこうしたGPUプログラミングサポート機能を提供している
2. （処理系任せのCPU-GPU間のデータ転送性能に満足できない場合）
   CPU・GPUメモリを明示する指示文を使ってコードをアップデート
   - 例えばNVIDIA GPU向けのManaged Memory では, 一旦メモリを読んで, ページフォルトがあればハードウェアレベルでページ単位で転送という仕組み
   - 必要なデータ転送は自分で指示した方が必然的に速くなる
   - GPUDirect系の機能を活用する際には,
     GPU上にデータが置かれていることを保証できる実装手法を使いたくなる

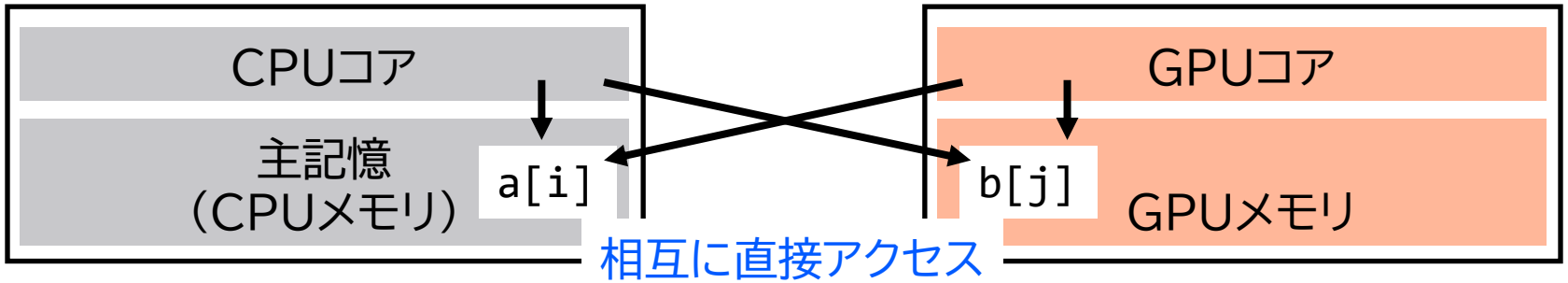
CPUとGPUのメモリ空間(NVIDIA)

- CPUとGPUは, 物理的に独立したメモリを持っていることが多い
  - GPU上で処理するならば, 扱うデータもGPU側に置いておく方が高速
  - CPU-GPU間のデータ転送コストがボトルネックになることも
    (NVIDIA GH200などではCPU-GPU間の帯域が太いため, 状況を緩和)

- (NVIDIA GPUの場合の)GPUメモリの使い方は3通り
  - Unified Memory: CPUとGPUどちらかのメモリにデータを置き,
    CPU・GPU双方から直接アクセス
  - Managed Memory: CPUとGPU両方のメモリにデータを置き,
    システムが必要に応じて自動同期
  - Separate: ユーザーが明示的にメモリ管理(GPUコンピューティング最初期からの手法であり, 上記2手法は利便性向上のため後に追加された使い方)

Unified Memory（NVIDIA）

- CPUとGPUどちらかのメモリにデータを置き，
  CPU･GPU双方から直接アクセス
  - NVIDIA GH200/GB200などの機種でサポートされている
  - `nvc/nvc++`では，`-gpu=mem:unified:nomanagedalloc`を指定
    - `:nomanagedalloc` なしでは次ページのmanagedが使われることもあるので，追記を推奨
- `malloc/new` などで動的に確保したメモリに対して，
  - CPU･GPUのはじめにアクセスした側のメモリにデータを配置（ファーストタッチ）
  - 以降のアクセスパターンについても監視され，GPU側に置いた方が高速化されると判断されると（自動的に）データがGPU側に置かれる（auto migration）
  - 下図では，a，bともに`malloc`で確保し，aはCPUから，bはGPUから1st touch



Managed Memory（NVIDIA）

- CPUとGPU両方のメモリにデータを置いておき，システムが必要に応じて自動的にデータを同期
  - 一旦メモリを読み，ページフォルトがあれば（自分の側に最新のデータがなければ）ページ単位でデータを転送するという仕組み（配列丸ごとの自動コピーではない）
    - データを少し読んで必要に応じて細切れにして転送するため，便利だが最高性能にはならない
    - MPI通信においては，CPUメモリを経由してしまう
  - 全てのNVIDIA GPUでサポートされている
  - `nvc/nvc++`では，`-gpu=mem:managed`を指定
- `malloc/new` などで動的に確保したメモリを，CPU・GPU両側に配置
  - 下図では，aは`malloc`で動的に確保している（宣言・確保方法はCPUコードと同じ）

Separate(NVIDIA)

- ユーザーによる明示的なメモリ管理(OpenACCでの書き方)
- malloc/new などでCPU上に動的確保したメモリに対して, 各処理を記述
  - GPU側にもマッピング
  - CPUからGPUへのデータ転送
  - GPUからCPUへのデータ転送
  - メモリ解放

```
double *a;
 a = (double*)malloc(sizeof(double) * size_t(N));
#pragma acc enter data create(a[0:N])
#pragma acc update device(a[0:N])
#pragma acc update host(a[0:N])
#pragma acc exit data delete(a[0:N])
 free(a);
```

CPUとGPUのメモリ空間(AMD)

- CPUとGPUは, 物理的に独立したメモリを持っていることが多い
  - 例外:AMD MI300AではCPUとGPUが物理的に同じメモリにアクセス
  - GPU上で処理するならば, 扱うデータもGPU側に置いておく方が高速
  - CPU-GPU間のデータ転送コストがボトルネックになることも

- (AMD GPUの場合の)GPUメモリの使い方は2通り
  - ランタイムにメモリ管理をお任せ(XNACK使用)
    - AMD MI300AのようにCPUとGPUが物理的に同じメモリを共有する場合もこちら
    - NVIDIA GPUのUnified/Managed Memoryに対応するプログラミング手法
  - ユーザーが明示的にメモリ管理
    - GPUコンピューティング最初期からの手法であり,
      上記手法は利便性向上・NVIDIAとの互換性などのため後に追加された使い方

ランタイムにメモリ管理をお任せ（AMD）

- 実際の挙動はプラットフォーム依存だが, CPUとGPU両側からメモリにアクセスできる
  - AMD MI300AのようなAPUでは, CPUとGPUが物理的に同じメモリを共有しているため, メモリ管理はそもそも気にしなくて良い（APUプログラミングモデル）
  - AMD MI300XのようにGPUが独立したメモリを持っていても, ランタイムが適切にメモリ管理してくれる（NVIDIA GPUのManaged Memoryと同様）
  - 全てのAMD GPUでサポートされている
  - XNACKを有効化して動作させる必要がある
- `malloc/new` などで動的に確保したメモリを, CPU・GPU両側に配置
  - 下図では, aは`malloc`で動的に確保している（宣言・確保方法はCPUコードと同じ）

OpenMP targetで明示的なデータ管理を行う方法

- NVIDIA/AMD GPU共通の内容
- CPU上で動的に確保されたメモリをGPU側でもマッピングし，CPU⇔GPU間のデータ転送を記述
  - GPU側メモリのマッピング
  - CPUからGPUへのデータ転送
  - GPUからCPUへのデータ転送
  - メモリ解放

```
double *a;
 a = (double*)malloc(sizeof(double) * size_t(N));
#pragma omp target enter data map(alloc: a[0:N])

#pragma omp target update to(a[0:N])

#pragma omp target update from(a[0:N])

#pragma omp target exit data map(delete: a[0:N])
 free(a);
```

Solomonで明示的なデータ管理を行う方法

- OpenACC/OpenMP的記法もあるが, ここでは簡易記法のみを紹介
- CPU上で動的に確保されたメモリをGPU側でもマッピングし, CPU⇔GPU間のデータ転送を記述
  - GPU側メモリのマッピング

```
double *a;
 a = (double*)malloc(sizeof(double) * size_t(N));
MALLOC_ON_DEVICE(a[0:N])
```

  - CPUからGPUへのデータ転送

```
MEMCPY_H2D(a[0:N])
```

  - GPUからCPUへのデータ転送

```
MEMCPY_D2H(a[0:N])
```

  - メモリ解放

```
FREE_FROM_DEVICE(a[0:N])
 free(a);
```

OpenACCコードのコンパイル方法など

- NVIDIA HPC SDKを使用
  （下記はC++向け. C言語の場合はnvcを使用）
- Unified Memory 使用時のコンパイル方法
  - `$ nvc++ -acc=gpu -gpu=cc80,mem:unified:nomanagedalloc ¥ -Minfo=accel,opt`
  - リンク時にも `-acc=gpu -gpu=mem:unified:nomanagedalloc` を指定
  - `-gpu=cc80`は, 用いるGPUの世代(80はAmpere(A100など))を指定
- Managed Memory 使用時のコンパイル方法
  - `$ nvc++ -acc=gpu -gpu=cc80,mem:managed -Minfo=accel,opt`
  - リンク時にも `-acc=gpu -gpu=mem:managed` を指定
- Separate 使用時のコンパイル方法(OpenACCのデフォルト)
  - `$ nvc++ -acc=gpu -gpu=cc80 -Minfo=accel,opt`
  - リンク時にも `-acc=gpu` を指定

OpenMPコードのコンパイル方法(NVIDIA)

- NVIDIA GPU向け(NVIDIA HPC SDKを使用)
- `$ nvc++ -mp=gpu -gpu=cc80`
  - リンク時にも `-mp=gpu` を指定する
  - `-mp=gpu` ,ターゲットデバイスをGPUにする(GPUで実行させる)ことを意味する
- `$ nvc++ -mp=gpu -gpu=cc80,mem:unified:nomanagedalloc`
  - Unified Memory 使用時のコンパイル方法
  - Managed Memory 使用時には`mem:managed` を指定する
  - リンク時にも `-mp=gpu -gpu=mem:unified:nomanagedalloc` を指定する
- Cの場合には, コンパイラをnvcに変更
- `-Minfo=accel,opt,mp` の追記で, コンパイル時にGPU化関連メッセージを出力
  - OpenMP関連のメッセージを出力させるためにmpの指定も必要

OpenMPコードのコンパイル方法(AMD)

- AMD GPU向け(ROCmを使用)
- `$ amdclang++ -fopenmp --offload-arch=gfx942`
  - `--offload-arch=gfx942` により, ターゲットデバイスを指定
    - gfx942 はAMD MI300A/MI300Xを意味する
- CPU･GPU間のメモリ管理をランタイム任せにする方法(XNACK使用)
  - 環境変数として `HSA_XNACK=1` を指定して実行
  - `$ amdclang++ -fopenmp --offload-arch=gfx942:xnack+`
    - コンパイル時に:xnack+を追記し, XNACK機能の使用を明示しても良い
    - 反対に, XNACK機能の不使用をコンパイル時に指示するには, :xnack-をつければ良い
    - :xnack+や:xnack-が指定されていない際には, 両方に対応したバイナリが生成される
- Cの場合には, コンパイラをamdclangに変更

GPU上で実行されていることの確認方法

- GPU上で実行されているか簡易に確認する際に有用な環境変数がある
- NVIDIA GPU向けの環境変数（NVIDIA HPC SDKを使用）
  - `NVCOMPILER_ACC_NOTIFY=1`
    - GPU上でカーネルが実行されるたびに情報を出力
  - `NVCOMPILER_ACC_NOTIFY=3`
    - CPU-GPU間のデータ転送に関する情報も出力
  - `NVCOMPILER_ACC_TIME=1`
    - CPU-GPU間のデータ転送およびGPU上での実行時間を出力
- AMD GPU向けの環境変数（ROCmを使用）
  - `LIBOMPTARGET_INFO=-1`
    - GPU上で動作しているかなどの各種情報を出力

Solomon を用いた際の，バックエンド切替方法

- コンパイル時にフラグを渡すだけでOK

コンパイル時フラグ	バックエンド
-DOFFLOAD_BY_OPENACC	OpenACC を使用，kernels
-DOFFLOAD_BY_OPENACC -DOFFLOAD_BY_OPENACC_PARALLEL	OpenACC を使用，parallel
-DOFFLOAD_BY_OPENMP_TARGET	OpenMP target を使用，loop
-DOFFLOAD_BY_OPENMP_TARGET -DOFFLOAD_BY_OPENMP_TARGET_DISTRIBUTE	OpenMP target を使用，distribute

- 各コンパイラに対する OpenACC / OpenMP target を有効化するためのフラグは別途必要
  - OpenACC 無効化時には，-DOFFLOAD_BY_OPENACCも自動的に無効化される
- -DOFFLOAD_BY_OPENACCと-DOFFLOAD_BY_OPENMP_TARGETを両方指定した場合
  - OpenACC を用いてGPU化
- -DOFFLOAD_BY_OPENACCと-DOFFLOAD_BY_OPENMP_TARGETをどちらも指定しなかった場合
  - マルチコアCPU向けにOpenMPでスレッド並列化

計算機へのログインとサンプルの取得

- （事前にVPN接続しておく）
- `$ ssh USERNAME@g00.cfca.nao.ac.jp`
- `$ cd /cfca-work/$USER`
- `$ mkdir 260820gpu-nbody` # 何か適当なフォルダを作る
- `$ cd    260820gpu-nbody` # 先ほど作ったフォルダに入る
- `$ cp /cfca-work/gpuws00/samples/cfca_sample.tar.bz2 .`
  - N体シミュレーション学校の途中成果コード（CPUのみで動作する単純なN体シミュレーションコード）とMakefileだけが入っています
- `$ tar -xvf cfca_sample.tar.bz2`
- `$ cp /cfca-work/gpuws00/samples/run_nvhpc.sh .`

3日目(8/20)の予定

- 9:00--9:45　前日の復習
- 10:00--11:00　指示文を使用したGPU化1　[講義・演習]
- 11:15--12:15　指示文を使用したGPU化2　[演習]
- 13:30--14:30　指示文＋CUDA混合コードの実装　[講義・演習]
- 14:45--15:45　MPIを使用したマルチGPUコードの実装　[講義]
- 16:00--17:00　最近＋近未来のGPUコンピューティング　[講義]

指示文＋CUDA混合コードの実装

- 移植工数削減や保守性向上のため，指示文でのGPU化を選択したとする
- 特定のカーネルが実行時間の大半を占めており，指示文では十分な性能最適化ができないということはよくある
- 特定のカーネルだけをCUDAで実装して性能最適化できれば，両方のメリットを取りに行ける

- （全体の指示文実装＋個別関数のCUDA化は完了済みとして）
  指示文実装とCUDA実装を結合するために必要な処理は何か？
  - CUDA実装を呼び出す関数に対して，「GPU上に確保されているメモリ」であるということを教えてあげる必要がある（またはUnified/Managed Memoryを利用）
    - 指示文実装からGPU向けライブラリを呼び出すためにはどうすれば良いか？ と実質的に同じ
  - コンパイル・リンクする際に，適切なオプションをつけるなどの工夫をする
    - 正直，このあたりはAIにうまくやって，と言ってMakefileをいじってもらうのが正解かもしれない

CUDA実装を指示文実装コードから呼び出す方法

- CUDA実装を呼び出す関数に対して,「GPU上に確保されているメモリ」であるということを教えてあげる必要がある
  - 指示文実装でMPIを用いてマルチGPU化する際やGPU向けライブラリを呼ぶ際にも同様の処理が発生する
- OpenACCでの例:

```
#pragma acc host_data use_device(pos) {
    w_init = calc_potential_energy(ni, pos, ni, pos, eps2);
  }

#pragma acc host_data use_device(pos, acc)
  calc_force(ni, pos, acc, ni, pos, eps2);
```

- OpenMPでの例:

```
#pragma omp target data use_device_addr(pos) {
    w_init = calc_potential_energy(ni, pos, ni, pos, eps2);
  }

#pragma omp target data use_device_addr(pos, acc)
  calc_force(ni, pos, acc, ni, pos, eps2);
```

- Solomonでの例:

```
  USE_DEVICE_DATA_FROM_HOST(pos) {
    w_init = calc_potential_energy(ni, pos, ni, pos, eps2);
  }

  USE_DEVICE_DATA_FROM_HOST(pos, acc)
  calc_force(ni, pos, acc, ni, pos, eps2);
```

Makefile にも工夫が必要

- 指示文実装のコードは nvc/nvc++/nvfortran でコンパイルする
  - 今回はCコードのため, nvc を使っているはず
- CUDA実装のコードは nvcc でコンパイルする
- それぞれのコンパイラでコンパイルし, 適切なオプションをつけてリンク
- うまくいった手順(g00:/cfca-work/gpuws00/cfca-gpu2026f/nbody/directive_cuda にうまくいったサンプルがあります):
  - Cコードから呼び出される関数を記載した共通ヘッダにはデマングル対策を追加
    - C++は勝手に関数名を変えてしまう
    - CUDAコード内でも, C側から呼び出したい関数には extern “C” をつけておく

```
#ifdef __cplusplus
extern “C” {
#endif
  void function0(int num, float *pos);
  void function1(int num, float *vel);
#ifdef __cplusplus
}
#endif
```

  - 指示文実装はnvcで, CUDAコードはnvccでコンパイルして, 最後にnvcでリンクする際に -cuda -c++libs を付与する

3日目(8/20)の予定

- 9:00--9:45　前日の復習
- 10:00--11:00　指示文を使用したGPU化1　[講義・演習]
- 11:15--12:15　指示文を使用したGPU化2　[演習]
- 13:30--14:30　指示文＋CUDA混合コードの実装　[講義・演習]
- **14:45--15:45　MPIを使用したマルチGPUコードの実装　[講義]**
- 16:00--17:00　最近＋近未来のGPUコンピューティング　[講義]

MPIを用いた複数GPU実装

- GPU-aware MPI が使える環境であれば, 実装は超簡単
  - MPI関数にGPU上のアドレスをそのまま指定すれば実装完了
  - CPU-GPU 間のデータ転送を自分で書く必要なし
    - GPUDirect のことを考えると, むしろ書かない方が良い
- システムのMPIがGPUDirect RDMA(GDR)などをサポートしていれば, GPU間の(CPUを介しない)直接通信もできる
  - Open MPI (w/ UCX) や, MVAPICH2 GDR
  - NVIDIA HPC SDKに含まれる HPC-X は Open MPIベース
  - Unified/Managed Memory を使っていると, GDR などは使ってくれない
    - 一度ホスト側にデータをコピーした上で通信されてしまう
    - 回避策もあるが, (現時点での) Unified/Managed Memory の弱点の一つ
    - Miyabi-G に搭載されているNVIDIA GH200ではCPUとGPUが密結合されているため, この問題が顕在化しにくい(CPU-GPU間のデータ転送コストが見えづらい)

今回の講習会プログラムの中での実装手順

- 指示文・CUDAなどで実装したコードをMPI並列してもらうのは作業量が多いので(講習会のプログラムの中では)厳しい
  - そもそもMPIについてきちんと解説をしていないというのも..
- 今回はMPI並列版のサンプルコードを好きな方法でGPU化してくださいという手順を採用
  - MPI(+ OpenMP ハイブリッド)並列されたCPU向けコードをGPU化する, というのはよくあるケースなので, 今回はこちらのルートを想定
  - 単体GPU化したコードは既に持っているはずなので, 適宜つなぎあわせればOK
  - GPU化したコードであっても, MPI関数はホスト(CPU)から呼び出す点に留意
    - GPU上に確保されているメモリであると教えてあげる必要がある(指示文+CUDA実装と同様)(CUDA実装の場合には`cudaMalloc()`で配列確保しているため, この部分は自明に伝わる)

MPI関数を指示文実装コードから呼び出すには？

- MPI関数に対して,「GPU上に確保されているメモリ」であるということを教えてあげる必要がある
  - 指示文実装＋CUDA実装をした際にも同様の処理が発生した
  - 実はCUDA実装から呼び出す際には, 何もする必要がない
- GPU上のアドレスを指定したということをコンパイラに教える必要がある
  - OpenACCでの実装:

```
#pragma acc host_data use_device(recv_buf)
  MPI_Irecv(recv_buf, recv_count, MPI_DOUBLE, recv_from, tag, MPI_COMM_WORLD, &recv_req);
#pragma acc host_data use_device(send_buf)
  MPI_Isend(send_buf, send_count, MPI_DOUBLE, send_to, tag, MPI_COMM_WORLD, &send_req);
```

  - OpenMP targetでの実装:

```
#pragma omp target data use_device_addr(recv_buf)
  MPI_Irecv(recv_buf, recv_count, MPI_DOUBLE, recv_from, tag, MPI_COMM_WORLD, &recv_req);
#pragma omp target data use_device_addr(send_buf)
  MPI_Isend(send_buf, send_count, MPI_DOUBLE, send_to, tag, MPI_COMM_WORLD, &send_req);
```

  - Solomonでの実装:

```
USE_DEVICE_DATA_FROM_HOST(recv_buf)
  MPI_Irecv(recv_buf, recv_count, MPI_DOUBLE, recv_from, tag, MPI_COMM_WORLD, &recv_req);
USE_DEVICE_DATA_FROM_HOST(send_buf)
  MPI_Isend(send_buf, send_count, MPI_DOUBLE, send_to, tag, MPI_COMM_WORLD, &send_req);
```

用いるコンパイラ環境

- `$ module purge`
- `$ module load nvhpc-hpcx`
- `$ make`
- 指示文でのGPU化の場合にはコンパイラを `mpicc` とするだけでOK
  - `mpicc`がnvcをラップしている
- CUDAでのGPU化の場合にはコンパイラを `nvcc` にしておき, MPI関連の情報を付与してあげる
  - Open MPI で使える `--showme` の出力を適切にパースして渡せば良い
  - MPIコンパイラにGPU関連情報を付与する方針よりもこちらの方が楽なはず

```
CU  := nvcc

CUFLAGS  += $(shell mpicxx --showme:compile 2>/dev/null | sed 's/-pthread//g')
LDFLAGS := $(shell mpicxx --showme:link 2>/dev/null | sed 's/-pthread//g' | sed -e 's/-pthread//g' -e 's/-Wl,¥([^ ]*¥)/-Xlinker ¥1/g')

$(EXEC):    $(OBJ)
    $(CU) $(CUFLAGS) $(ARGS) -o $@ $(OBJ) $(LDFLAGS)
%.o:    %.cu
    $(CU) $(CUFLAGS) $(ARGS) -o $@ -c $<
```

ジョブスクリプト例

- 指示文実装, CUDA実装, どちらの場合でもこれでOK
- GPU数 = 全MPIプロセス数にしておく
  - MPIプロセスあたりのGPU数を1にしておくのがおすすめ

```
#!/usr/bin/env bash
#SBATCH --partition=gpuws
#SBATCH --time=00:05:00
#SBATCH --ntasks=8
#SBATCH --gres=gpu:8

if [ -z "${PARAM}" ]; then
    PARAM="params.ini"
fi

module purge
module load nvhpc-hpcx

cd ${SLURM_SUBMIT_DIR}

mpiexec -n ${SLURM_NTASKS} sh/common/wrapper.sh ./collapse_mpi ${PARAM}
```

環境変数を用いたGPUマッピング

- 主にノードあたりに複数GPUが搭載されているシステム向けの情報
  - Miyabi-G のようなノードあたり1 GPUのシステムでは, この内容は不要
- `omp_set_default_device()`, `hipSetDevice()` などを使わずに, 各MPIプロセスがどのGPUを用いるかをプログラムの外から制御できる
  - NVIDIA製GPUの場合には, `CUDA_VISIBLE_DEVICES`
  - AMD製GPUの場合には, `ROCR_VISIBLE_DEVICES`
- 例えばMPIプロセスあたりのGPU数を1とする(これがおすすめ)場合には,
  - `$ mpiexec -n 4 ./wrapper.sh ./a.out`
    - wrapper.sh の中身(chmod +x をお忘れなく):

```
#!/usr/bin/env bash
local_rank=${OMPI_COMM_WORLD_LOCAL_RANK:=${MV2_COMM_WORLD_LOCAL_RANK}}
GPU_ID=${local_rank}
export CUDA_VISIBLE_DEVICES=${GPU_ID}
export ROCR_VISIBLE_DEVICES=${GPU_ID}
$*
```

      - Open MPI/MVAPICH2, NVIDIA/AMD GPUに対応したスクリプト例
      - `*_LOCAL_RANK` は, ノード内のMPIランクを示す変数(0, 1, …)

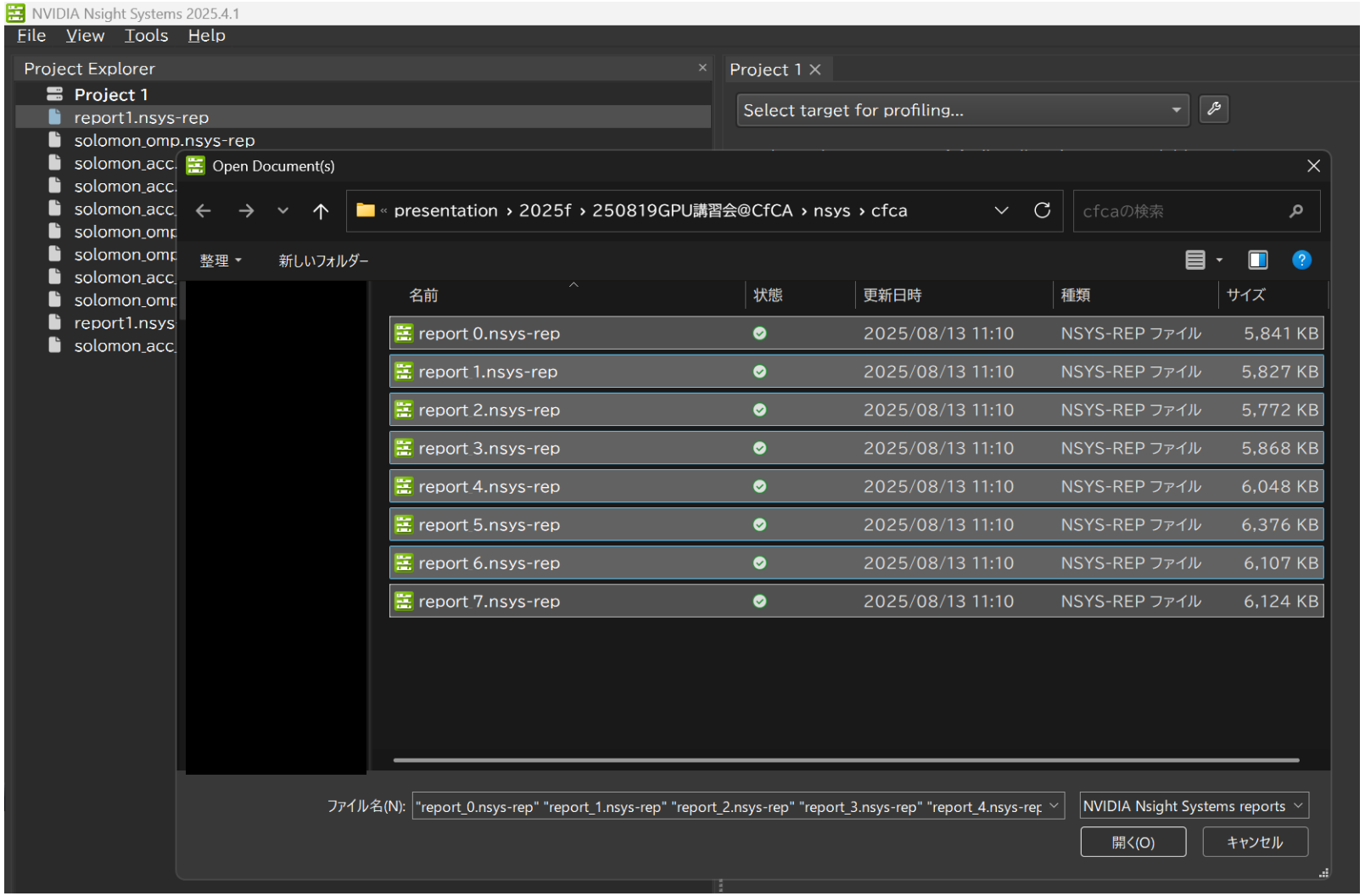
MPI版のコードのプロファイルを取得するには？

- ラッパースクリプトの最後の行を少し書き換える
  - 各MPIプロセスが別々のファイル名でプロファイルデータを出力する

```
#!/usr/bin/env bash

mpi_rank=${OMPI_COMM_WORLD_RANK:=${MV2_COMM_WORLD_RANK:=${PMI_RANK:=${PMIX_RANK:=0}}}}

# obtain process rank within a node
if [ -n "$OMPI_COMM_WORLD_LOCAL_RANK" ] || [ -n "$MV2_COMM_WORLD_LOCAL_RANK" ]; then
    local_rank=${OMPI_COMM_WORLD_LOCAL_RANK:=${MV2_COMM_WORLD_LOCAL_RANK}}
else
    cores_per_node=`LANG=C command lscpu | command sed -n 's/^CPU(s): *//p'`
    mpi_size=${OMPI_COMM_WORLD_SIZE:=${MV2_COMM_WORLD_SIZE:=${PMI_SIZE:=${OMPI_UNIVERSE_SIZE:=1}}}}
    procs_per_node=`expr $mpi_size / $cores_per_node`
    if [ $procs_per_node -lt 1 ]; then
        procs_per_node=$mpi_size
    fi

    local_rank=`expr $mpi_rank % $procs_per_node`
fi

GPU_ID=${local_rank}
export CUDA_VISIBLE_DEVICES=${GPU_ID}
export UCX_NET_DEVICES=mlx5_${GPU_ID}:1

nsys profile --stats=true --trace=osrt,cuda,openacc,openmp,nvtx,mpi,oshmem,ucx -f true -o report_${mpi_rank} $*
```

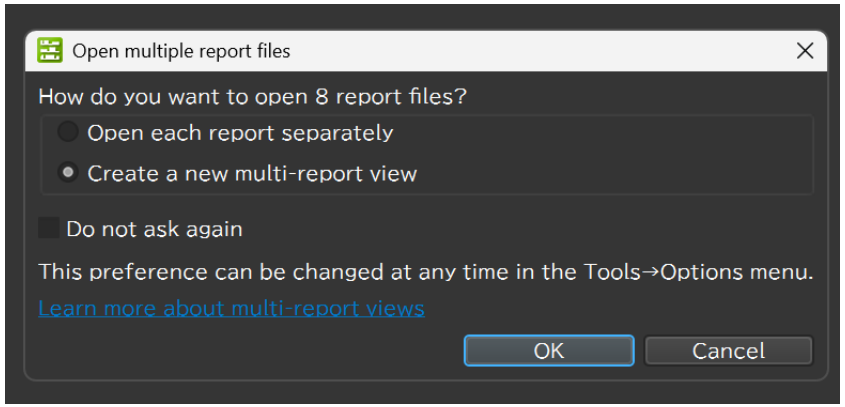
取得したプロファイル情報の可視化方法(1/3)

- Nsight Systems の File > Open から関連するファイルを全て選択して開く

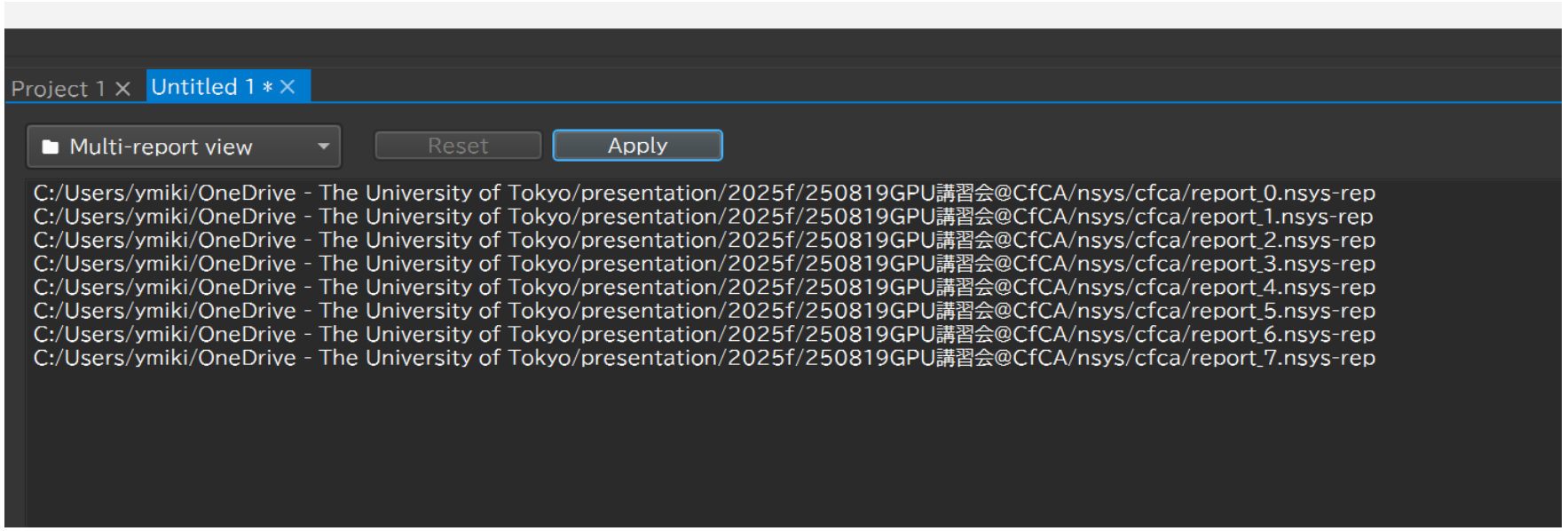


取得したプロファイル情報の可視化方法(2/3)

- “Create a multi-report view” を選択してOKをクリック

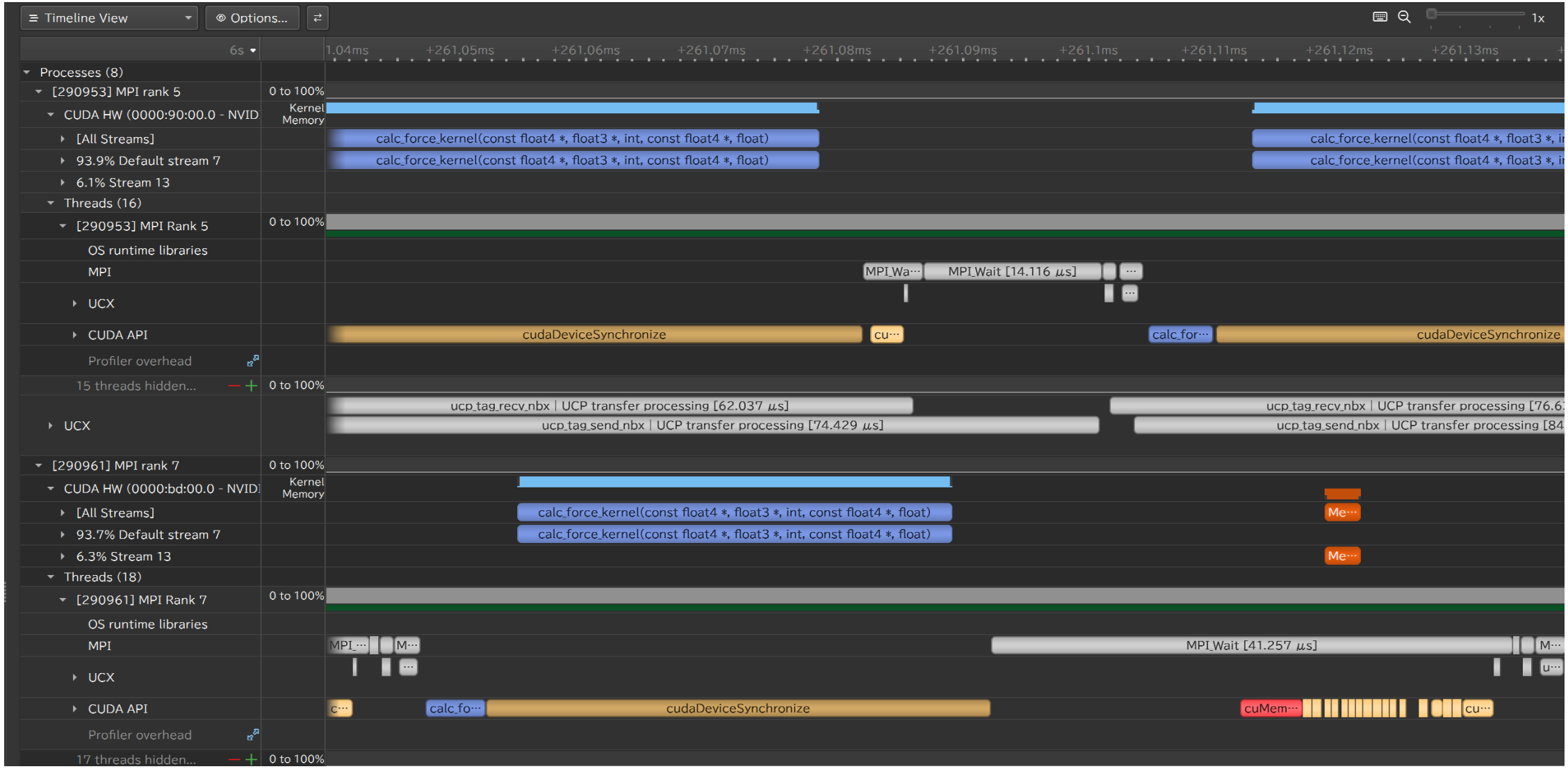


- 下記の画面になるので, Apply をクリック



取得したプロファイル情報の可視化方法(3/3)

- (問題サイズが小さすぎるものの)複数GPUが使えていることが確認できる



実習: コードをマルチGPU化してみる

- 新たに渡すサンプルコードを指示文あるいはCUDAを用いてGPU化

- `$ cd /cfca-work/$USER`
- `$ mkdir 260820gpu_mpi # 何か適当なフォルダを作る`
- `$ cd    260820gpu_mpi # 先ほど作ったフォルダに入る`
- `$ cp /cfca-work/gpuws00/samples/mpi_sample.tar.bz2 .`
  - CPUのみで動作するMPI並列化済みのN体シミュレーションコード, Makefile, ジョブスクリプト, 入力ファイルなどが入っています
- `$ tar -xvf mpi_sample.tar.bz2`

- `$ git clone https://github.com/ymiki-repo/solomon.git`
  - これは Solomon を使ってGPU化する人のみ実行
    (fortran ブランチの方が(C/C++部分も)新しいので, そちらをおすすめします)

参考用のコードなど

- `g00:/cfca-work/gpuws00/cfca-gpu2026f/nbody/` の中
  - `base`: 講師がGPU化前にリファクタリングしたコード（各種トラップを事前回避版）
  - `directive_managed`: 指示文＋managed memory実装
  - `directive_data`: 指示文＋自分でデータ転送
  - `directive_cuda`: 指示文＋CUDA実装
  - `cuda`: CUDA実装
  - `mpi_samples`: MPI実装（CPUコード）
  - `directive_mpi`: 指示文＋MPI実装
  - `cuda_mpi`: CUDA＋MPI実装
- https://github.com/ymiki-repo/nbody
  - CUDA, OpenACC, OpenMP target, C++17 stdpar でのN体実装例
  - 大抵のトラップは事前的に回避されている（実はCUDAを先に書いてCPU化した）

3日目(8/20)の予定

- 9:00--9:45　前日の復習
- 10:00--11:00　指示文を使用したGPU化1　[講義・演習]
- 11:15--12:15　指示文を使用したGPU化2　[演習]
- 13:30--14:30　指示文＋CUDA混合コードの実装　[講義・演習]
- 14:45--15:45　MPIを使用したマルチGPUコードの実装　[講義]
- 16:00--17:00　最近＋近未来のGPUコンピューティング　[講義]

JCAHPC第二世代システム Miyabi

- JCAHPC: 最先端共同HPC基盤施設(2013年～)
  - 筑波大学計算科学研究センターと東京大学情報基盤センターが共同で調達・運用
- 導入の経過
  - 2022年11月より調達プロセスを開始
    - 2022年6月には事前性能評価によりGPUアーキテクチャを決定
  - 2023年11月に開札, 富士通が落札
    - 準備期間を1年以上確保
- システムの特徴
  - システム全体性能の飛躍的な向上のため, 演算加速装置としてGPUを主体とするシステムへ
    - 消費電力も削減➔電力あたり性能の劇的な向上
    - CPU-GPU間が高速リンクで密結合され, 既存アプリのGPU化が容易に
  - GPU化が困難なアプリケーションのために, 汎用CPUのみの計算ノードも導入
  - ストレージの高性能化に向けてAll Flashを導入

Miyabiの外観

Miyabi

JCAHPC

筑波大学 University of Tsukuba

東京大学 THE UNIVERSITY OF TOKYO

Miyabi (OFP-II) (1/2)

JCAHPC

筑波大学 University of Tsukuba　東京大学 THE UNIVERSITY OF TOKYO

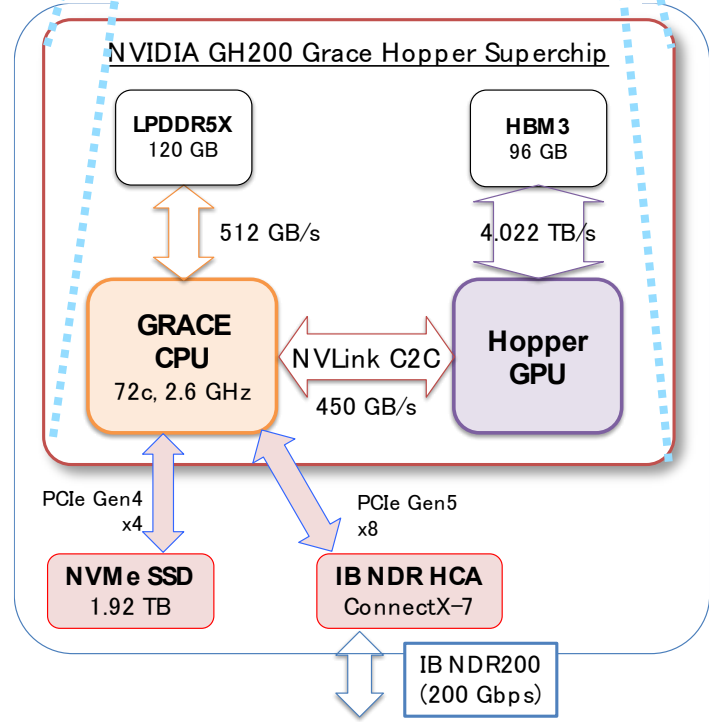
- **Miyabi-G: CPU+GPU: NVIDIA GH200**
  - Node: NVIDIA GH200 Grace-Hopper Superchip
    - Grace: 72c, 3.456 TF, 120 GB, 512 GB/sec (LPDDR5X)
    - H100: 66.9 TF DP-Tensor Core, 96 GB, 4,022 GB/sec (HBM3)
      - Cache Coherent between CPU-GPU
    - NVMe SSD for each GPU: 1.9TB, 8.0GB/sec, GPUDirect Storage
  - **Total (Aggregated Performance: CPU+GPU)**
    - **1,120 nodes, 78.8 PF, 5.07 PB/sec, IB-NDR 200**
- **Miyabi-C: CPU Only: Intel Xeon Max 9480 (SPR)**
  - Node: Intel Xeon Max 9480 (1.9 GHz, 56c) x 2
    - 6.8 TF, 128 GiB, 3,200 GB/sec (HBM2e only)
  - **Total**
    - **190 nodes, 1.3 PF, IB-NDR 200**
    - **372 TB/sec for STREAM Triad (Peak: 608 TB/sec)**

FUJITSU

SUPERMICRO

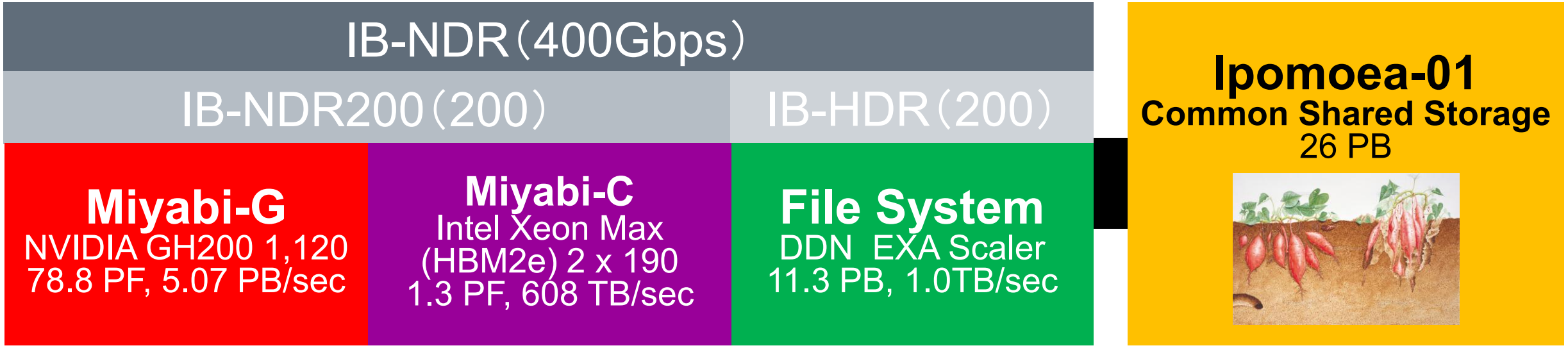
NVIDIA

intel



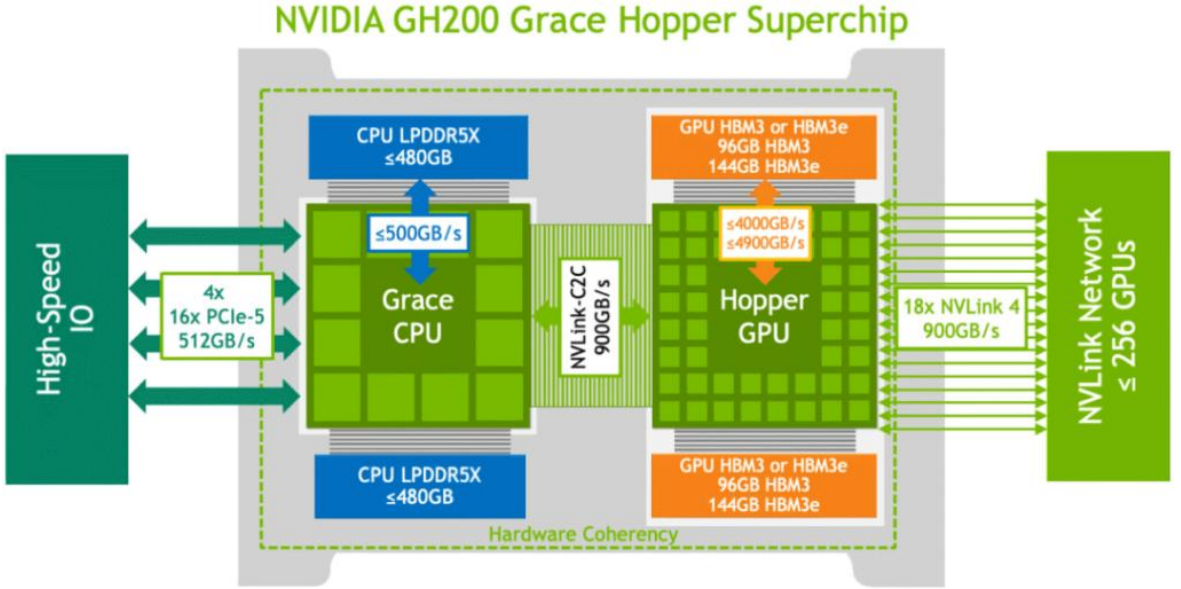
Miyabi (OFP-II) (2/2)

- ファイルシステム: **DDN EXA Scalar, Lustre FS**
  - 11.3 PB (NVMe SSD) 1.0TB/sec, “Ipomoea-01” (26 PB) も利用可能
- **Miyabi-G/C の全ノードはフルバイセクションバンド幅で接続**
  - **(400Gbps/8)×(32×20＋16×1) = 32.8 TB/sec**
- **2025年1月運用開始**
- Miyabi-G/C間の通信はh3-Open-SYS/WaitIO により実現



NVIDIA GH200の特性

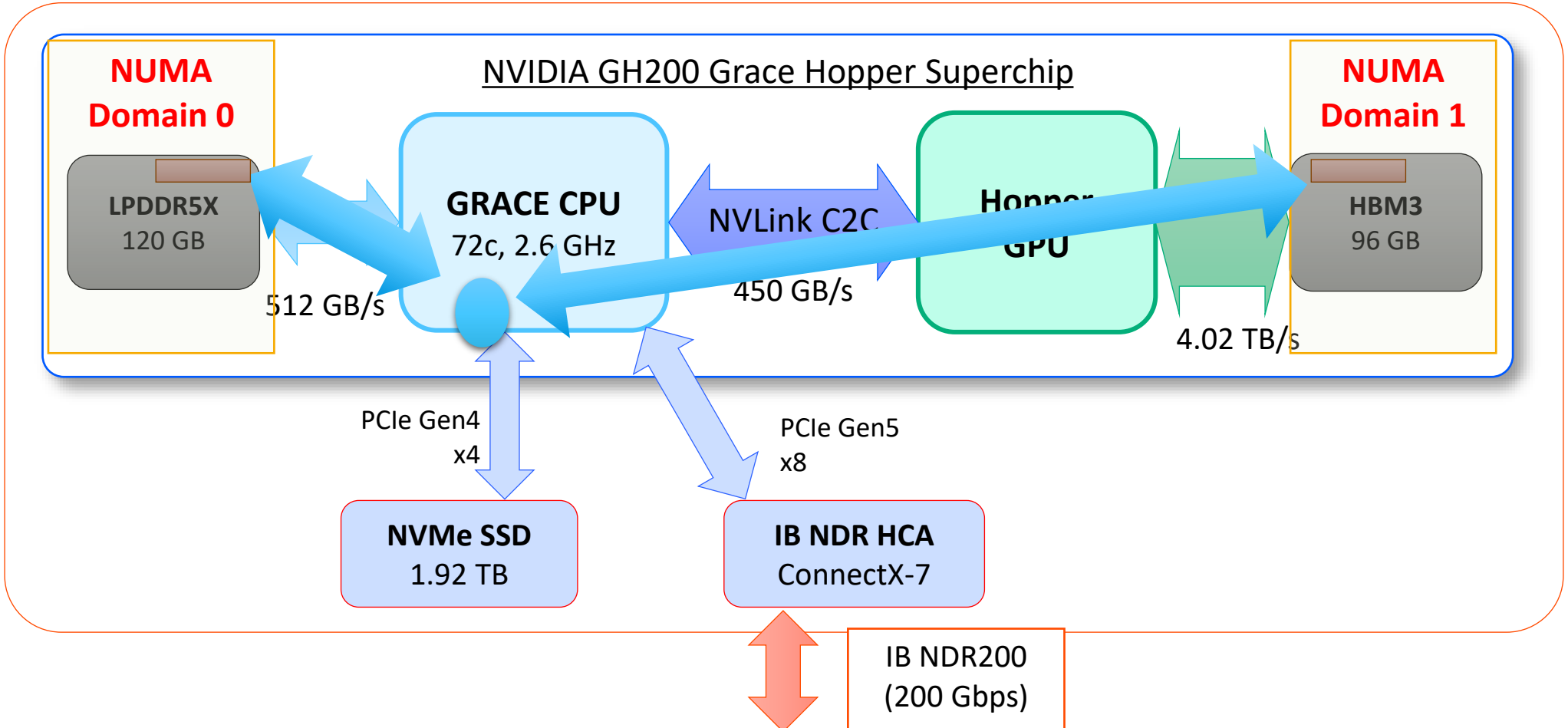
- Grace-Hopper相互のメモリ空間を直接参照可能, NUMA的な扱い
- CPU-GPU間: コヒーレントインタフェース(NVLink-C2C)
  - PCIe Gen 5の7倍以上の帯域(450GB/sec/dir)
  - CPU･GPUの効率的使い分けも可能
    - 従来はデータ転送がボトルネック
    - プログラミングも容易
    - AMD MI300Aも同じ方向性



- 小規模問題, GPUが不得意な計算をCPUが柔軟に処理することも可能

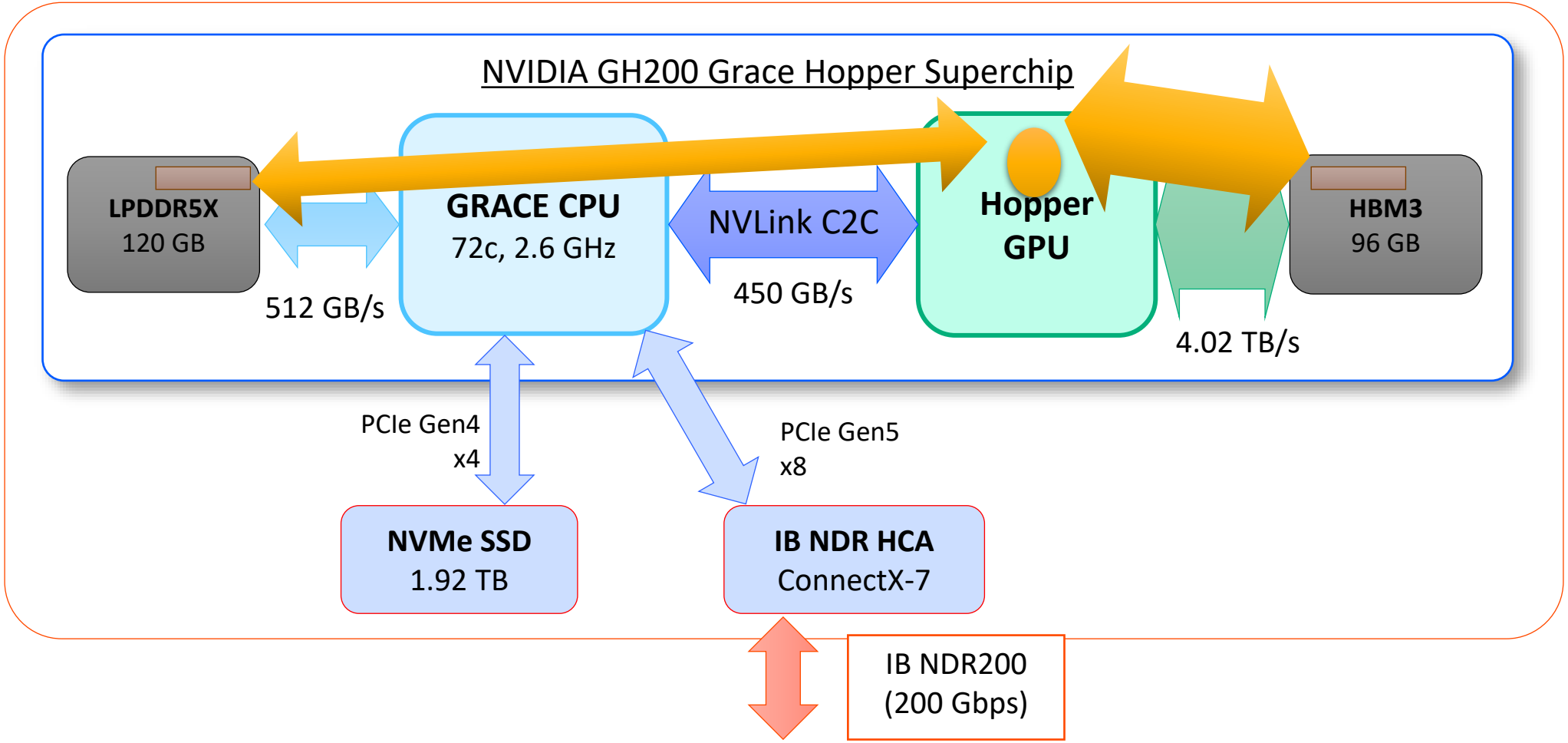
Graceからのメモリビュー

- NUMAとして見える
  - `malloc()`ではFirst Touchが大事
- 従来のCUDAのメモリモデルも使えて普通に動く(コード改変不要)＋転送が速い
  - `cudaMalloc() + cudaMemcpy()`
  - `cudaMalloc()`した領域はGraceからアクセス不可



Hopperからのメモリビュー

- Graceのアドレスを使って直接アクセス可能
- 従来のCUDAのコードはH100とほぼ挙動が変わらない（メモリバンド幅比相当の性能向上）



NVIDIA GH200のメモリアクセス

- CUDA

メモリ	メモリモード	確保される場所	Access-based Migration	CPUからアクセス	GPUからアクセス
System-allocated (malloc, new)	Unified相当	First-touch (GPU または CPU)	○	○	○
CUDA managed (cudaMallocManaged)	Managed相当	First-touch (GPU または CPU)	○	○	○
CUDA device memory (cudaMalloc)	Separate相当 (Device)	GPU			○
CUDA host memory (cudaMallocHost)		CPU		○	○

- OpenACC, OpenMP target, stdpar

メモリモード	コンパイルフラグ	デフォルトとなる環境	
Separate	`-gpu=mem:separate`	OpenACC OpenMP target	GPU上のデータはGPUからのみアクセス可能 GPU-CPU間の明示的なデータ移動が必要
Managed	`-gpu=mem:managed`	stdpar (Managed Memory のみの環境)	動的メモリ確保されたデータはGPU, CPU どちらからもアクセス可能
Unified	`-gpu=mem:unified`	stdpar (Unified Memory 対応の環境)	全てのデータはGPU, CPU どちらからもアクセス可能

GPU移行プラットフォームとしてのMiyabi

- 日本国内でも, GPU搭載スパコンが続々と増えている
  - TSUBAME4.0@科学大, 玄界@九大, Miyabi@JCAHPC, ABCI 3.0@AIST, …
  - 科学技術計算用のGPUスパコンとしてはMiyabi-Gが国内最大のシステム
- GPU初心者にとって移行が一番簡単なシステムはMiyabi-G
  - GH200では, GPU初心者がはまりやすい罠が大幅に軽減されている
    - CPUからもGPUからも相互にメモリ空間が参照できる
    - CPU-GPU間のデータ転送が(x86-Hopperに比べて)性能ボトルネックになりづらい
  - コードを部分的にGPU移植しながら動作テストするのも比較的容易
    - CPU-GPU間のデータ転送コストが(通常のPCIe接続に比べて)大幅に軽減されている
      (コードを移植途中に「遅くなった」と思って熱が冷めるリスクが減る)
  - GPU移植しづらい部分をCPUに置き去りにしても, 性能面で問題になりづらい
- 東大情報基盤センターとしてGPUを主体とするシステムはMiyabiが初
  - したがって, ユーザコードの移植支援にも力を入れている状態
  - GPU関係の講習会, GPU移行相談会, ポータルサイトでの情報提供など
    - ユーザアカウントを持っていなくてもOK, すべて無料

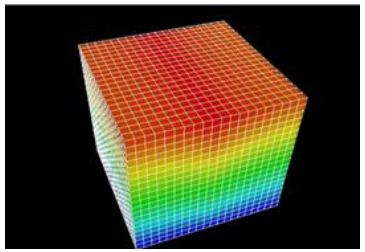
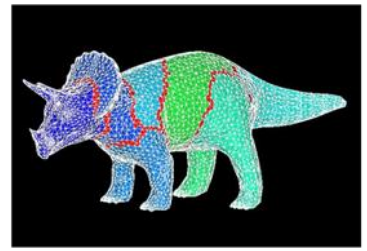
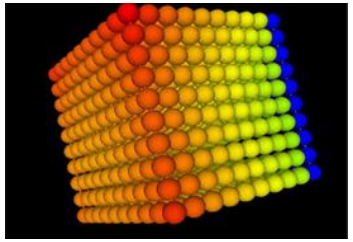
GPUコンピューティングの推進(1/2)
HAIRDESC: 次世代HPC・AI研究開発支援センター

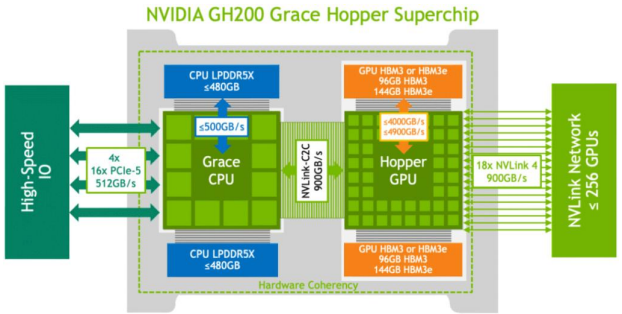
- 2025年11月開始, 文部科学省の支援による4.5年間のプロジェクト
- HAIRDESCは, 富岳NEXT時代を見据えて, HPCおよびAI分野におけるGPUを活用した次世代アプリケーション向けに高度な技術支援を提供する
- 実施体制
  - 代表機関: 高度情報科学技術研究機構(RIST)
  - 中核機関:筑波大学, 東京大学, 東京科学大学
    - 大学:北海道大学, 東北大学, 名古屋大学, 京都大学, 大阪大学, 九州大学(JHPCN構成拠点の大学)
    - 理研 R-CCS
    - NVIDIA, AMD

GPUコンピューティングの推進(2/2)
HAIRDESC: 次世代HPC・AI研究開発支援センター

- 東大情報基盤センターの役割は以下の通り:

- 計算科学:アプリケーションGPU化
  - 構造格子系(e.g. FDM)
  - 非構造格子系(e.g. FEM, FVM)
  - 多体系
  - CPUとGPUの両方の効率的な活用
- AI for Science
  - h3-Open-BDECによるシミュレーション(計算)/データ/学習の融合
- 性能可搬性プログラミング環境
  - Kokkos
  - Solomon [Miki & Hanawa 2024]
    - 指示文ベースのGPUオフロードのための統一インターフェース
  - Fortranからの脱出



Hierarchical, Hybrid, Heterogeneous
h3-Open-BDEC
Big Data & Extreme Computing

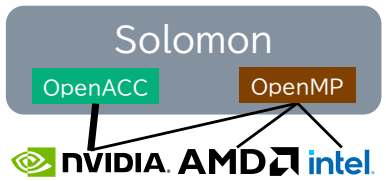
kokkos

Standard Implementation for OpenACC & OpenMP

```
#if defined(OFFLOAD_BY_OPENACC)
#if defined(OFFLOAD_BY_OPENACC_KERNELS)
#pragma acc kernels
#pragma acc loop
#else
#pragma acc parallel
#pragma acc loop
#endif
#endif
#if defined(OFFLOAD_BY_OPENMP_TARGET)
#if defined(
    OFFLOAD_BY_OPENMP_TARGET_LOOP)
#pragma omp target teams loop
#else
#pragma omp target teams distribute
    parallel for
#endif
#endif
  for (int i = 0; i < N_i; i++) {
    // loop body A
  }
```

Implementation using Solomon

```
OFFLOAD()
for (int i = 0; i < N_i; i++) {
  // loop body A
}
```



GPUプログラミング教材の整備と教育コース

- GPUプログラミング教材の作成方針
  - 「標準的GPU化」コースの向けた教材作成
  - 主に, 既存のOpenMP化されたCPUコードのGPU化を扱う
    - 最初からGPUコード開発を行う方を対象とした教材も検討中
  - 特定のGPUベンダーによらないマテリアル（NVIDIA, AMD対応）
- 初級から上級までのレベル別教材を準備整備（2025年度末に初版公開）
  - 初級編
    - GPUプログラミング概要
    - 指示文を用いたGPU化（OpenACC編, OpenMP target編）
  - 中級編
    - 複数GPUによる計算の概要
    - MPIによる複数GPU計算（OpenACC編, OpenMP target編）
  - 性能チューニング編（プロファイリング）とライブラリ編
  - 再配布NGなので, 皆さんご自身でユーザー登録して取得してください
- ハッカソン, チュートリアルの実施

GPUプログラミング初心者・初級者向けのサンプル集

- https://github.com/SCD-ITC-UTokyo/gpu-apps-tutorial
  - 整備の一部は文部科学省「次世代HPC・AI開発支援拠点形成事業」の支援を受けた
- 拡散方程式, 有限要素法, N体問題のサンプルコード
  - メモリ律速な問題, 不規則アクセスがある問題, 演算律速な問題
- 複数のプログラミングモデルの実装例を紹介
  - OpenMP for CPU
  - OpenMP target (for GPU)
  - OpenACC
  - 標準言語(C++ stdpar / Fortran do concurrent)
  - Kokkos
- 解説, 使い方, 性能評価ダッシュボードも整理されている
  - データ整理に生成AIツールを使っており, まだ誤りが含まれている可能性が高いです
  - 何かお気づきの際には, お気軽にお問い合わせください

東大スパコンの利用制度(https://www.cc.u-tokyo.ac.jp/guide)

- 一般利用
  - 大学・公共機関に在籍の方(大学院生は代表者としては申し込めません)
  - 電気代相当料金の利用負担金支払いが必要
- 企業利用
  - 企業に在籍の方
  - 書面・ヒアリング審査あり
  - 成果公開型: 利用成果報告書を公開, 利用負担金は一般利用の約1.2倍
  - 成果非公開型: 報告書・テーマ・社名など非公開, 利用負担金は一般利用の約4倍
- 若手・女性利用(後期は8/28 17:00締切)
  - 大学・公共機関に在籍の方
  - 4月1日現在40歳以下の若手, または女性, または学生
  - 利用負担金なし
  - 書類審査あり, 成果報告義務あり
- 学際大規模情報基盤共同利用・共同研究拠点(JHPCN)への課題申請
- HPCI課題への課題申請

もう一つ新しい制度の紹介

- 萌芽共同研究公募課題「GPU移行によるアプリケーションの高速化」（試行）

- 既存のCPUアプリケーションをGPUへ移植・最適化し，アプリケーションの高速化を目指す提案を募集
- 本センター教員と共同研究の形で実施

- 公募は年2回受付
  - 次の締切は8/31
  - 採択されれば，10/1から共同研究開始

再掲：GPUスパコンの近況

- GPUは最近のスパコンでのメジャーな演算加速器（特に上位システムで顕著）
- 国内ほとんどのGPU搭載スパコンはNVIDIA製GPUを採用
  - 筑波大CCSのSiriusはAMD MI300Aを搭載
  - 海外のハイエンドスパコンではAMD, Intel製GPUも採用
    - El Capitan（2位, AMD MI300A), Frontier（3位, AMD MI250X), Aurora（4位, Intel GPU Max）など, AMDやIntelのGPUを搭載したスパコンがTOP500上位にランクイン
      ※括弧内はTOP500順位（June 2026）と搭載GPU
- GPUベンダー間の競争が活性化
  - 発散するプログラミング環境への対応も必要
- GPU搭載スパコンは今後も増えていく傾向, GPU移植は必須と言える
  - ポスト富岳（富岳NEXT）は演算加速器（NVIDIA製GPU）を搭載
  - GPUベンダ間競争の活性化に伴い、GPU搭載スパコンに多様性がでる可能性も
  - 求める性能, 払えるプログラミングコスト, アプリの特性・特徴, 対象とするGPUベンダーを勘案して採用するGPUプログラミング手法を選定する必要がある

国内でもGPU移植を支援するための枠組みが開始

- HPCIに資源提供されているスパコンではGPU率が上昇中
- 一方，今までの日本のフラッグシップシステム（京，富岳）はCPUベース
- 富岳の後継機である「富岳NEXT」はNVIDIA製GPUを搭載する
- したがって，国内HPCユーザーのGPU移植が必須
- HAIRDESC（次世代HPC・AI研究開発支援センター）もこの文脈
  - GPUプログラミング教材の整備
  - 講習会，ミニキャンプの開催
- 文科省「次世代計算科学グランドリーチプログラム」
  - https://www.mext.go.jp/b menu/boshu/detail/mext 00494.html
  - 「富岳」成果創出加速プログラムの後継課題
  - 区分B・C・Dの選定結果通知は「令和8年8月下旬～9月中旬」
- そうは言っても国内のGPU資源は不足しているので，環境整備も加速中
  - 文科省「AI for Scienceに不可欠な計算資源の戦略的増強」
  - 選定結果は https://www.mext.go.jp/content/20260604-mext jyohoka01-000046874 .pdf を参照

倍精度(ベクトル)演算の性能が上がらない可能性も

- 少なくともNVIDIAの次世代GPU(Rubin)では, FP64 vectorの性能はHopper, Blackwell から下がる
  - https://developer.nvidia.com/blog/inside-the-nvidia-rubin-platform-six-new-chips-one-ai-supercomputer/

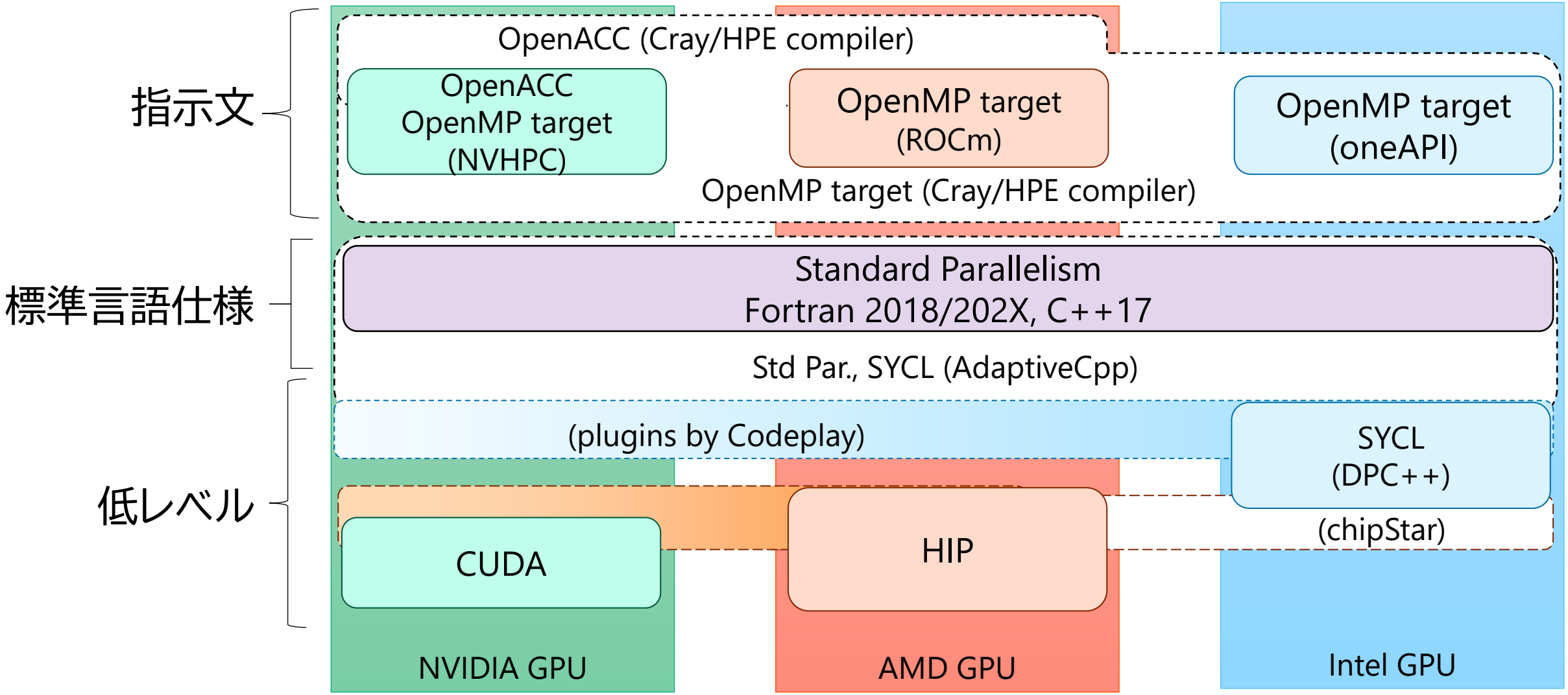
Feature	Hopper GPU	Blackwell GPU	Rubin GPU
FP32 vector (TFLOPS)	67	80	130
FP32 matrix (TFLOPS)	67	227*	400*
FP64 vector (TFLOPS)	34	40	33
FP64 matrix (TFLOPS)	67	150*	200*

Table 3. NVIDIA GPU FP32 and FP64 compute capability.
**Peak performance using Tensor Core-based emulation algorithms*

- 単精度(FP32)であれば, 世代ごとに性能増加はまだ続く
- 倍精度(FP64)では, 密行列の行列行列積限定でOzaki Schemeによって高速化
- AI向けに低精度行列行列積の高速化に全投資されている影響

再掲：GPU向けのプログラミング環境

2024年2月のPCCC AI/HPC OSS活用WSでの講演資料
(https://www.pccluster.org/ja/event/data/240205_pccc_wsAI-HPC-OSS_06_hanawa-miki.pdf)



再掲：GPUプログラミング手法の比較表

利用可能な機能や性能はプログラミング手法に依存し，調査検討が必要

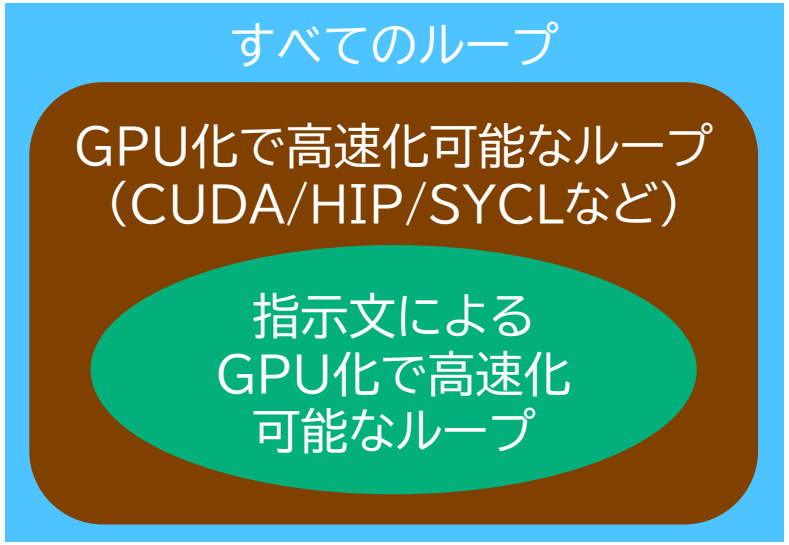
手法	ベース言語など	強み	弱み	Fortran対応	対象GPUベンダー
CUDA C++	C++	詳細な最適化可能 最新機能が使える	記述量が多い GPU専用コード	CUDA Fortran	NVIDIA限定
HIP	C++	詳細な最適化可能 ほぼCUDA C++	記述量が多い GPU専用コード	hipfort	AMD, NVIDIA （Intel: chipStar?）
SYCL	C++	詳細な最適化可能 ラムダ式	記述量が多い ラムダ式	N/A	Intel, NVIDIA, AMD
OpenACC	C/C++ Fortran	移植コスト低 CPUコードと共通化可能	CUDAより遅い （メモリ律速ならそれなり？）	標準の指示文で対応可能	ほぼNVIDIA限定 （HPE Cray コンパイラはAMDも対応）
OpenMP （target指示文）	C/C++ Fortran	移植コスト低 CPUコードと共通化可能	CUDAより遅い （メモリ律速ならそれなり？）	標準の指示文で対応可能	NVIDIA, AMD, Intel
言語の標準規格	C++17以降 Fortran 2008	CPUでも同じコードが動く	多くの制約あり CUDAより遅い	Fortran 2008	NVIDIA, AMD, Intel

2018年にあったやり取り

- 木構築なども含めて全てGPU上で動作する重力ツリーコードGOTHICをCUDA Cで開発(YM & Umemura 2017, New Astronomy, 52, 65-81)
- 論文を見た海外の研究者からコンタクト
  - 加速度に使っている定式化が特殊だが, GOTHICに組み込めるか？
  - (論文を見る限りは, おそらく)可能
  - 自分の環境でどのぐらいの実行時間になるか見積もってくれませんか？
  - アクセスできるGPUの情報を教えてください
  - AMD Vega 64です
  - CUDA実装なので, AMD製GPU上では動きません. NVIDIA製GPUへのアクセスはありませんか？
  - 以降返信なし
- 興味を持った人がアクセスできる・動かしたい環境と開発者が想定する環境が一致する保証はない(OSS開発者はこの視点を持っておくことも重要)
  - ベンダーロックインは自分が困ることも多々あるが, 共同研究の機会を逃すことにも

FortranベースのGPU化とベンダーロックイン

- AMD/IntelのFortranサポートは，基本的に指示文（OpenMP target）
  - 指示文だけでGPU化でき，かつ性能に満足できるならば
    ベンダーニュートラル化は可能
  - HIPやSYCLはFortranを直接サポートしていない
    - NVIDIAの場合にはCUDA Fortranがある
  - 指示文だけでGPU化できない and/or 十分な性能が出せない場合には
    1. 一部をC/C++に書き換えて，
    2. HIPやSYCLでCUDAレベルの実装を作り，
    3. Fortran側から呼び出す
  - ➔ Fortran だけで完結しない ＝ C/C++ のコードも読み書きできないといけない



- Fortran で完結，かつ性能も犠牲にしたくないと思うと何が起こる？
  - ベンダーニュートラルな実装は，（少なくとも現時点では）実質的に不可能
  - 市場原理が働かなくなるため，調達価格は高止まりする（競合相手がいれば安くなる）
  - 結果的に，導入されるシステム規模が小さくなる ＝ 実はFortranユーザも損している
    - C/C++ユーザからすれば，自分たちとは関係ないところで勝手にシステム規模を小さくされた話

Fortranユーザへのメッセージ（危機感の共有）

- 今どきのGPU向けプログラミング環境は基本的にC++ベース
  - CUDA C++, HIP C++, SYCL, Kokkos, etc.
  - AMD, IntelはCUDA Fortran相当のコンパイラを提供していないというのが現状
    - 指示文（OpenMP target）だけで済むならOKだが，全員がそれで済む保証は存在しない
  - 演算加速器向けにコードを書き換える「ついでに」言語も乗り換えるチャンス
    - データ構造を大幅にいじらないと性能が出ないという場合はあり，コードの大改修をする場合も
    - 大昔に設計したコードの設計見直し，アルゴリズムの改造などのタイミングでも同様
- 自分ではC++への移行はできないという方は？
  - LLMを使ってC++に変換という手法も模索されはじめている
    - ChatHPC, Code-Scribe, Fortran2CPP, CodeRosettaなど（Fortran to CUDA C++自動変換）
    - 変換後のC++コードを解読できるようにならないといけない（C++の学習コストはゼロにはならない）
  - 学生さんがプログラミングを始める際に（Fortranではなく）C++を習得してもらい，協力してコード移行するというのが現実的？
  - 短期間で一気に言語移行するのは困難なので，長期計画で（当然，早く着手すべきではある）
  - 新しいFortranユーザを増やすのはもうやめにしませんか？（できればコードも）
- 東大情報基盤センターとしても重要な課題と認識しているので，各種情報を収集しつつFortranからの移植をどうサポートすれば良いかを検討中
  - 現在GPU移植支援に力を入れているのと同様に，将来的にはFortranからの離脱支援も

ベンダーニュートラルGPUコンピューティング

- どの環境でも動くコードでないと，研究のsustainabilityが担保できない
  - ＊社のGPUが高くなりすぎて導入できません，となった時に道連れになりたくはない
  - 自分が開発しているOSSであれば，ユーザーが使いたい環境は自分と違う場合も
- 指示文実装の場合
  - OpenACC（実質的にNVIDIA向け）とOpenMP target（全社動く）の二択
  - 性能や，どう書けば性能が出やすいかといった違いはあったりする
  - どちらの指示文を使うか選択しなくて済むように，Solomonを開発した
    - 元々C/C++向けだったが，現在はFortran向けインタフェースも追加したものを公開済み https://github.com/ymiki-repo/solomon/tree/fortran
- より低レベルな実装の場合
  - CUDAはNVIDIA GPU専用言語だが，実は他社製GPUに移植しやすいという話も（GPUの構造が各社で大きく違うということはない，各社は移植ツールを開発・提供）
  - Kokkos, SYCL, RAJA, Alpaka などC++ラムダベースの性能可搬フレームワーク

SYCL環境の構築

- Intel oneAPI（コンパイラは icpx）
- Intel SYCL(コンパイラはclang++)
  - https://github.com/intel/llvm で公開されている(Arm でも使える)
- AdaptiveCpp（コンパイラは acpp）
  (https://github.com/AdaptiveCpp/AdaptiveCpp)
  - ドキュメントにしたがってインストールすれば特に苦労なく使えた
    - Intel GPU 向けのドキュメントが見当たらなかったので, NVIDIA/AMD GPUs 限定の話
    - NVIDIA GH200 向けにもインストールできた(Arm CPU なので, oneAPI が使えない)
      - ただし, nvclangをバックエンドに取ることはできなかった(`llvm-tblgen`が不足)
  - いくつかのインストール方法が提示されているが, 試したのは以下の手順
    1. LLVM を手でインストール(NVPTX or AMDGPU を有効にしておく)
       - ドキュメントでは LLVM は dnf install することを推奨されていたが, 管理者権限なしでインストールできる方法を取ることにした(スパコンに一般ユーザとしてインストールして使うことを想定した予行演習)
    2. AdaptiveCpp のソースを取ってきて, cmake してコンパイル
  - レジスタを大量に消費する場合に計算がこける場合がある
    - スレッド数が512以上かつILP数8以上というかなり極端な条件なので, 普通は出くわさないはず

コードの実装手順・勉強手順

- CUDA C++版
  - 簡単なのでスクラッチから実装
- HIP C++版
  - CUDA版のうちいくつか簡単なものを hipify-clang を用いて変換
    (シェアードメモリの使い方など, ドキュメントを眺めるよりも簡単に使い方が分かる)
  - 感触をつかんだ後は, 普通にスクラッチから実装できるようになる
- SYCL版
  - CUDA版のうちいくつか簡単なものを dpct (今はSYCLomatic)を用いて変換
  - デフォルトでは各queueが Out-of-Order 実行されるので適宜 `wait()` をかける
  - `sycl::property::queue::in_order` を指定して queue を作成すると, CUDAのdefault streamを使ったときと同じ振る舞い
    - CUDAに慣れている人にとってはこちらの使い方のほうが馴染みやすいと思われる
  - 感触をつかんだ後は, 普通にスクラッチから実装できるようになる
- 注:「簡単」,「普通に」などはあくまでも個人の感想なので保証はしません

CUDAコードをHIP化するにはどうすれば良いか？

- HIPIFYをかけると, CUDAコードをHIP化してくれる
  - `$ hipify-clang *.cu *.cuh --cuda-path=/cuda/path -I/include/path`
  - これだけで大体のことは完了してしまう
  - 他に hipify-perl というツールもある
    - hipify-clang の方がよりコンパイラ寄りのツールかつ推奨ツールとのこと
- 以前(数年前に)使った際に失敗した例
  - 注:今は挙動が変わっている可能性があるので, まずは試してみてください
  - #ifdef … #else … #endif のようなマクロスイッチがある時
    - #ifdef などを無視して変換するようで, const を付けた同名変数を複数回定義している, などというエラーが出たりする
    - 変換後のコードで有効にするフラグを切り替えることを想定すると仕方ない気もする
  - CUDA版ライブラリとROCm版ライブラリで不整合がある場合
    - cuRAND → rocRAND で, ヘッダの名前やライブラリのマクロ名などが違っていた
      (主に引っかかったのは Mersenne Twister まわり)
    - (AMD GPUで動かすには)変換後のコードを自分で手直しする必要があった

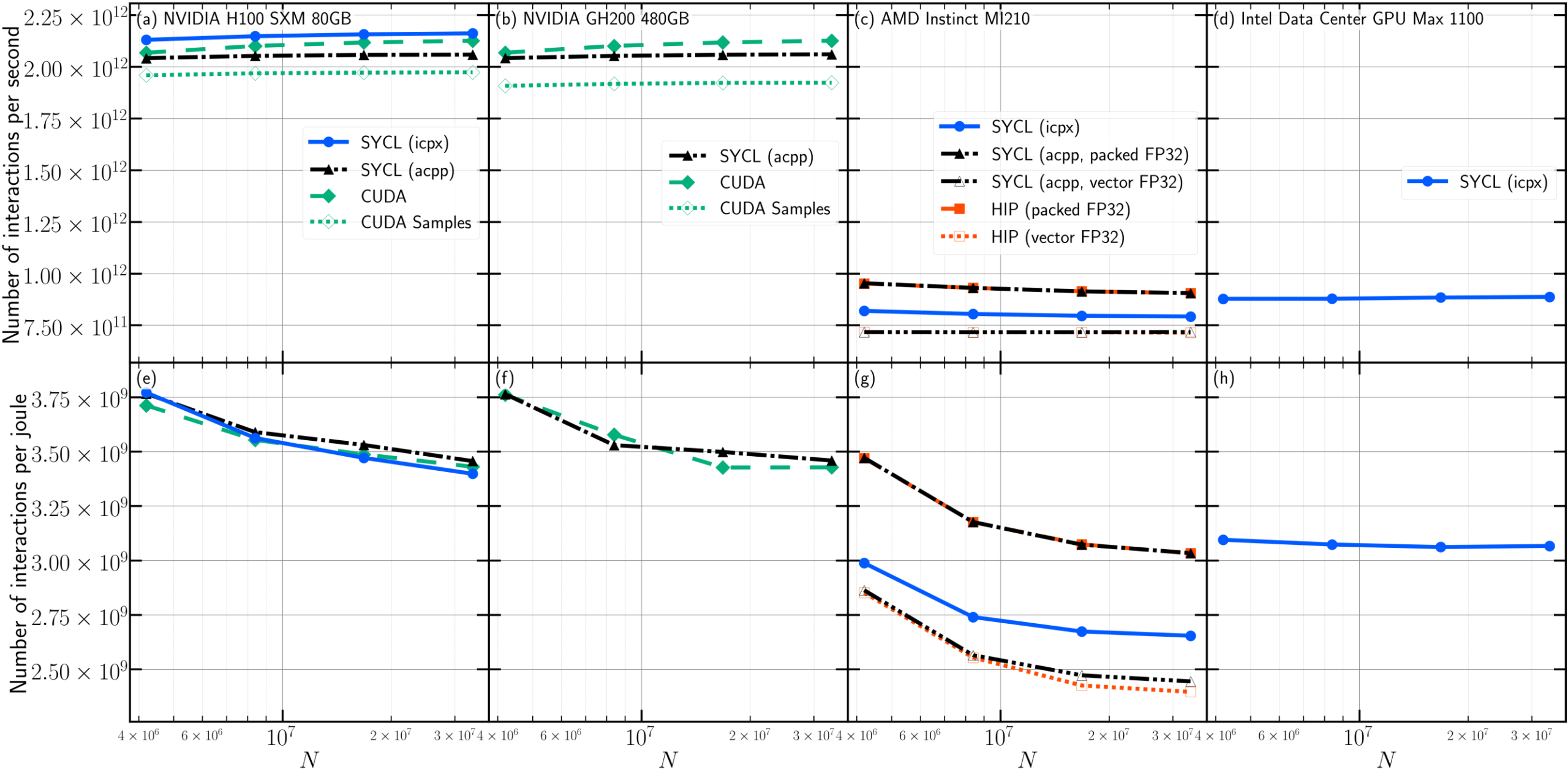
HIPとCUDAでの実装の違いは何か？

- 大雑把な違い：
  - ホストで呼ぶ `cuda*` という関数・変数には，`hip*` が対応
    - 新しいCUDA関数については，対応するHIP版がないこともある
    - cudaMallocHost ➔ hipHostMalloc のような例外もある（HIPIFYに任せればOK）
  - GPU上で動かすカーネル関数の起動方法
    - `func<<<blck, thrd, shmem, stream>>>(var0, var1, …);`
      - shmem, streamは省略することも多い（デフォルトで0が入る）
    - `hipLaunchKernelGGL(func, blck, thrd, shmem, stream, var0, var1, …);`
      - shmem, stream は省略できない（HIPIFY すると自動的に入れてくれていた）
      - （ドキュメントによると，CUDAと同じ記法でも良いらしい）
- 特にデバイス関数の中身については，HIPとCUDAはほぼ同じ（前ページ）
- ハードウェア側の違いを吸収しておく方が重要
  - NVIDIA GPUでは，`warpSize = 32` を基本単位として考える
  - （CDNA系の）AMD GPUでは，`waveSize = 64` を基本単位として考える
    - AMD GPUでも，RDNA系であれば `waveSize = 32, 64` で切り替え可能

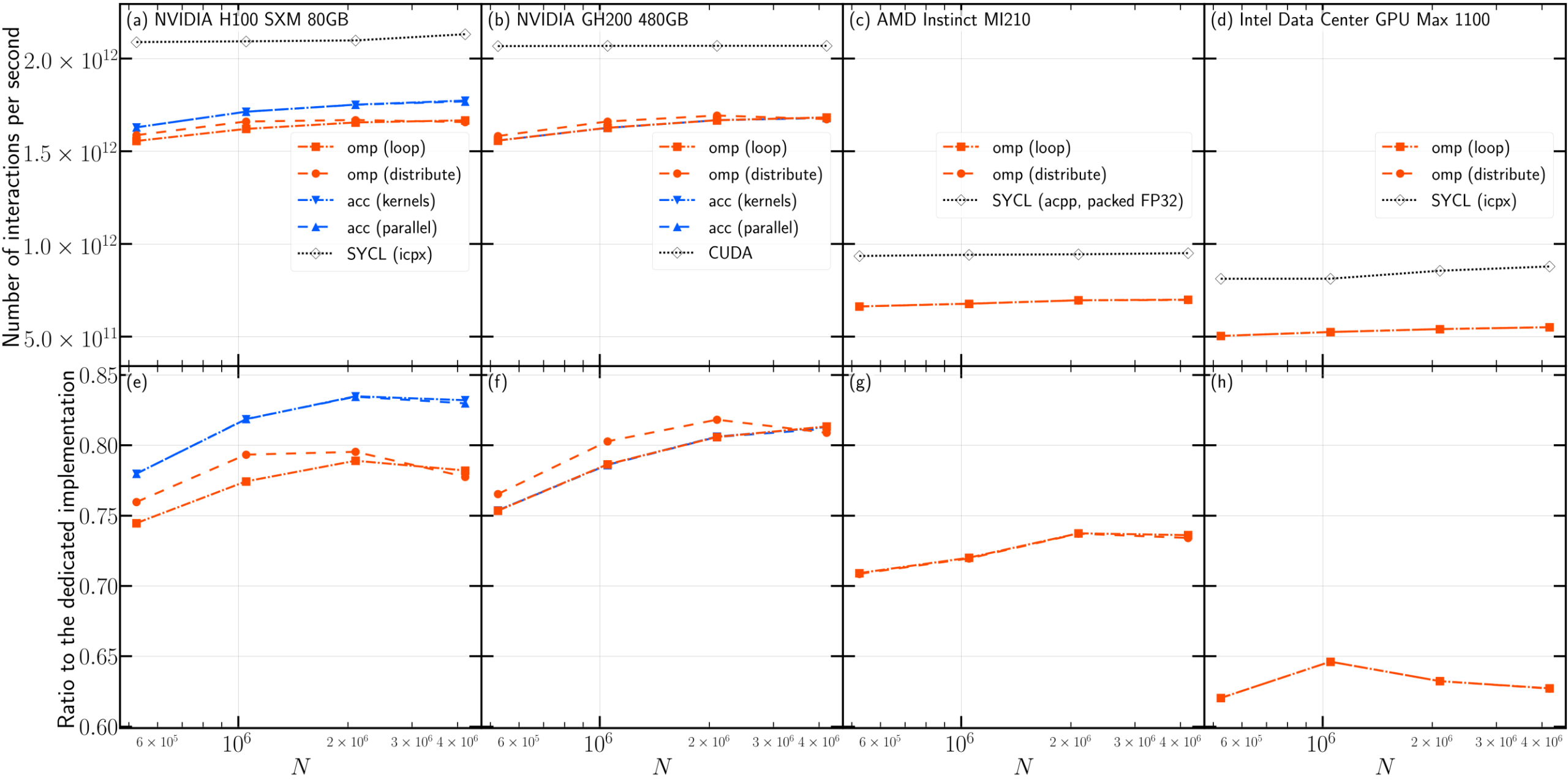
CUDAコードをSYCL化するにはどうすれば良いか？

- SYCLomatic（旧dpct）
  - コマンドは c2s
  - （試したのがかなり昔なので，`$ c2s --help` で使い方を表示して試してください）
- 変換前のコードがコメントアウトされた状態で残されており，
  また（必要に応じて）追加メッセージが書き込まれていることも
  - CUDAとSYCLを見比べやすいという意味で親切な設計と言える
- CUDAとHIPはよく似ていたが，SYCLはそれなりに雰囲気が異なる
  - C++のラムダ式を活用した実装になる
  - SYCLはむしろKokkosに近い（差分もかなりあるので，似ているとは言い難いが）
- （HIPIFYも同様だが）自分が中身を理解しているコードを別言語で実装し直したコードが出力される
  ➔自分向けにカスタマイズされたサンプルコードを作ってくれるツール
  - ドキュメントだけを眺めているよりも短時間でHIPやSYCLを学習できる

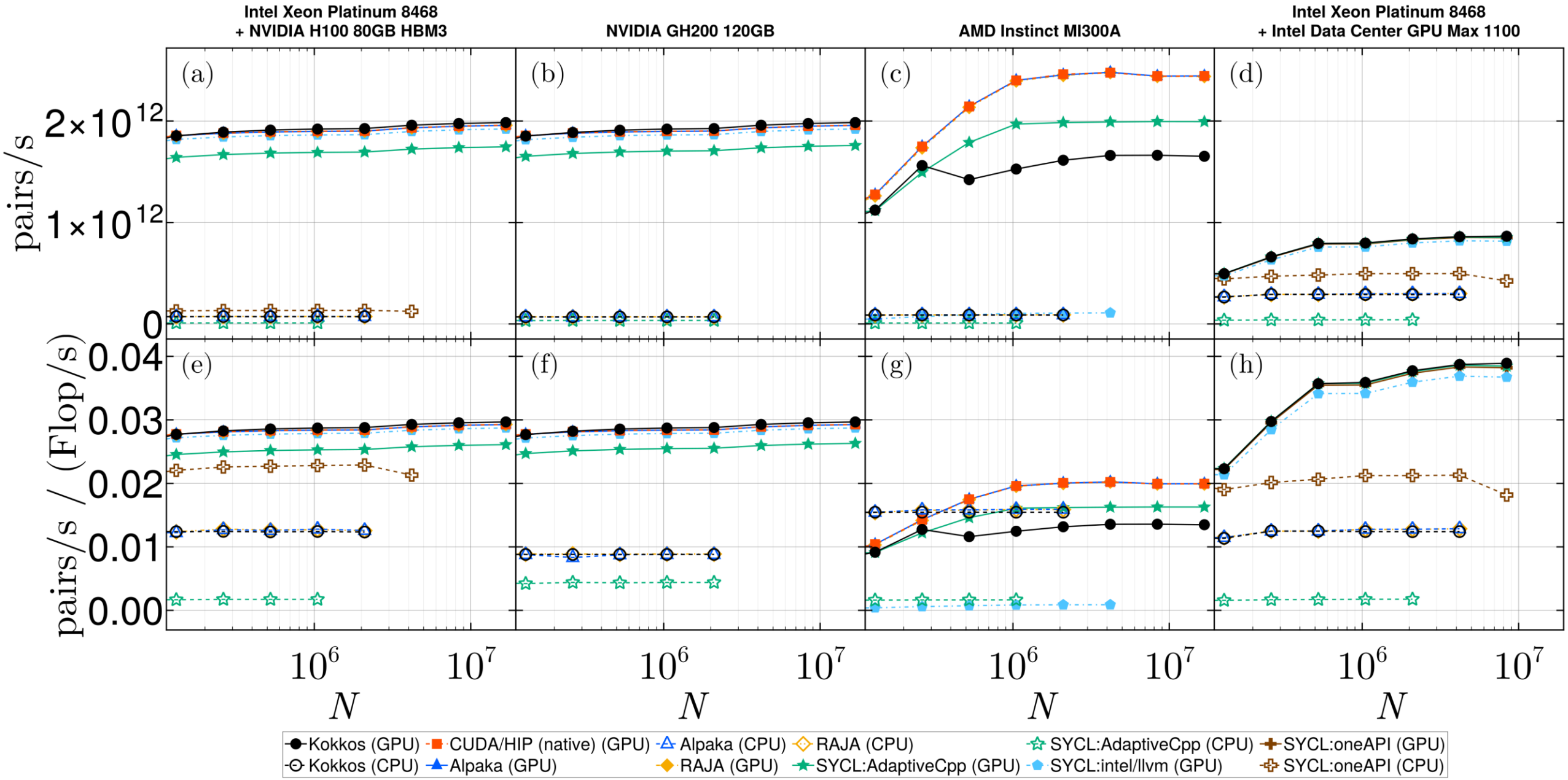
NVIDIA/AMD/Intel製GPUの性能比較



指示文実装との性能比較



Bandiera(仮)を仮実装・テスト中



最近のトレンド：AIコーディング

- エージェンティックAIの登場によって，AIコーディングはかなり流行っている
  - Claude Code, Codex など
  - 開発スピードは大幅に高速化される，簡単だけど面倒な実装作業からは解放される
- 簡単な問題であれば，AIお任せで実装からテスト・デバッグまでやってくれたりする
  - 簡単というのは，AIがよく学習している問題，という意味合いも強い
- ただし，以下の点には気をつけておいた方が良さそう
  - 場当たり的，その場しのぎの対応に終始することが多い
  - 割とひどいやらかしもしてくるので，きちんと監視していないと危ない（監視用エージェントを立てておいたとしても，数手先を見通せているわけでもない）
    - 要は，AI丸投げでは危ないので，きちんとした知識を元に適切な指示を与える必要がある
  - テスト問題として渡している問題に過剰に集中する傾向
  - 十分な材料を揃える前に方針を決めて突っ走ろうとする（そして失敗・手戻り）
    - 各種トークンを無駄消費することになるので，地味に痛い
  - 共有計算機上で動かすと，周囲に迷惑を与えてしまう

最後に

- 今回の演習の参照実装(注:全力最適化版ではない)を下記に置いておきます:
  - g00:/cfca-work/gpuws00/cfca-gpu2026f/nbody
- 今後もGPUを搭載した計算機は増えていきそう
  - 通常のCPUに比べて消費電力あたりの演算性能が高いため
  - ポスト富岳(富岳NEXT)も演算加速器を搭載したシステム
- GPU向けプログラミング手法は多数ある
  - 性能, 移植コスト, 可搬性などを考慮して自分に適したものを選択
  - 今回きちんと紹介していないものではKokkos, RAJA などのフレームワークも
  - ベンダーニュートラルなGPUプログラミング手法(Kokkos, SYCLなど)はいずれもC++ベースであるため, C/C++ユーザにとっての敷居はそこまで高くない
    - ラムダ式に慣れておくと良い
    - 身の回りに Fortranユーザがいる際には, C++への移行を勧めてあげてください
- 実は最適化も(世間一般で言われているほどは)難しくない
  - もちろんGPU向けにアルゴリズムを設計するのは大変(ただしCPUでも同様)
- GPU移行に関するポータルサイト, HAIRDESCページに情報を記載中
  - https://jcahpc.github.io/gpu_porting/
  - https://hairdesc.jp