2025年度GPU講習会

三木 洋平
（東京大学 情報基盤センター）

国立天文台三鷹キャンパス 講義室

講習会プログラム

- 8/19(火)
  - 9:00—10:00　[講義] GPUプログラミングの基礎(指示文編)
  - 10:10—12:00　[実習]
  - 12:00—13:30　昼休み
  - 13:30—14:30　[実習]
  - 14:40—15:40　[講義] GPUプログラミングの基礎(CUDA編)
  - 15:50—17:00　[実習]
- 8/20(水)
  - 9:00—10:00　[講義] GPU化したN体コードの高速化手法
  - 10:10—12:00　[実習]
  - 12:00—13:30　昼休み
  - 13:30—14:30　[講義] MPIを用いたマルチGPU化
  - 14:40—17:00　[実習]

Contents

- GPU(Graphics Processing Units)の紹介
- 指示文を用いたGPU化
  - OpenACC, OpenMP, Solomon
- CUDAを用いたGPU化
  - 指示文を用いてGPU化したコードの一部をCUDA化
  - コード全体をCUDAで実装
- ベンダーニュートラルGPUコンピューティングの紹介(HIP, SYCL)
- GPUコードの性能最適化に向けて
  - 気をつけるべき項目や機能の紹介
  - プロファイラの使い方の紹介
- MPIを用いたマルチGPU化

Contents

- GPU(Graphics Processing Units)の紹介
- 指示文を用いたGPU化
  - OpenACC, OpenMP, Solomon
- CUDAを用いたGPU化
  - 指示文を用いてGPU化したコードの一部をCUDA化
  - コード全体をCUDAで実装
- ベンダーニュートラルGPUコンピューティングの紹介(HIP, SYCL)
- GPUコードの性能最適化に向けて
  - 気をつけるべき項目や機能の紹介
  - プロファイラの使い方の紹介
- MPIを用いたマルチGPU化

GPU(Graphics Processing Units)

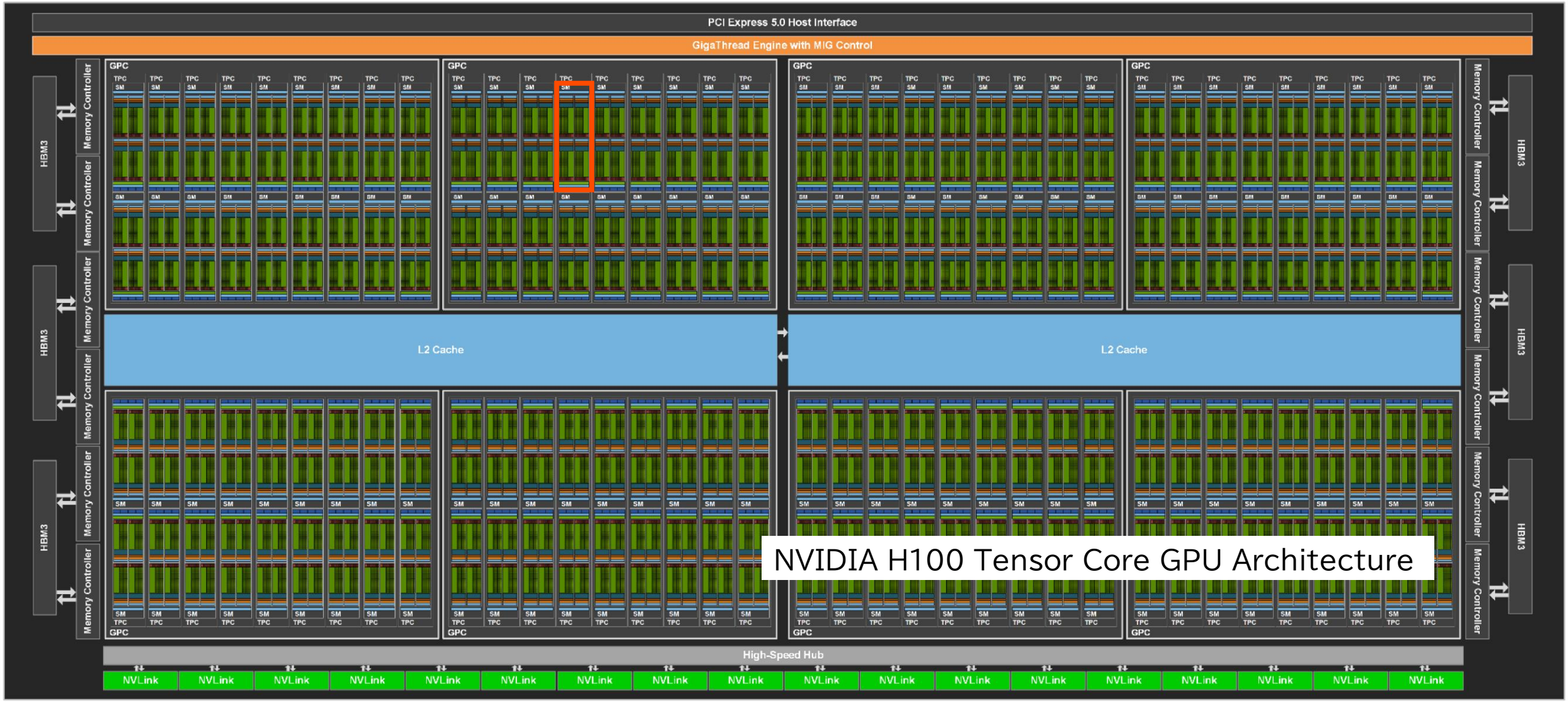
- 元々は画像処理を行うために特化していた専用プロセッサを汎用計算にも使えるように拡張
  - 最近のGPUには(主に深層学習用に)行列積向けユニットも
- 高い演算性能, 太いメモリバンド幅, 消費電力あたり性能も高い
  - 太古の昔には安かったが, 最近はとても高い
- 多数のコア(最新GPUは1万越え)を搭載した並列計算機
  - NVIDIA A100 (SXM, PCIe): $64 \times 108\ \mathrm{SMs} = 6912\ \mathrm{cores}$
  - NVIDIA H100 (SXM): $128 \times 132\ \mathrm{SMs} = 16896\ \mathrm{cores}$
  - NVIDIA H100 (PCIe): $128 \times 114\ \mathrm{SMs} = 14592\ \mathrm{cores}$
  - AMD MI250X: $64 \times 110\ \mathrm{CUs} \times 2\ \mathrm{GCDs} = 14080\ \mathrm{cores}$
  - AMD MI250: $64 \times 104\ \mathrm{CUs} \times 2\ \mathrm{GCDs} = 13312\ \mathrm{cores}$
- ざっくり言うと, コア数の10倍以上の並列度があると性能を発揮しやすい
- スレッド並列(e.g., OpenMP)的考えが分かっていると良い

Green500 Ranking (June 2025)

https://www.top500.org/lists/green500

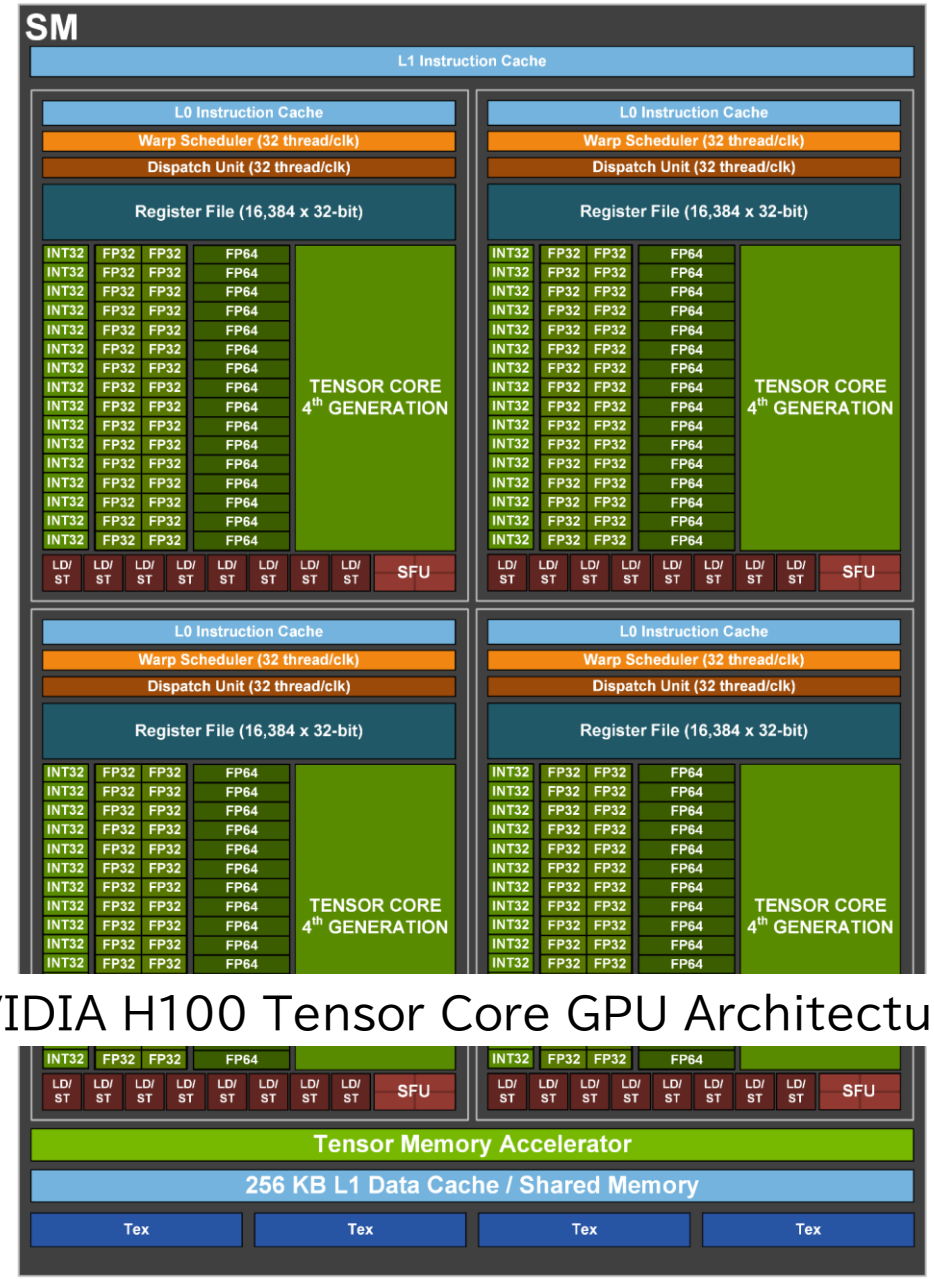
	TOP 500	System	Accelerator	Cores	HPL Rmax [PFlop/s]	Power [kW]	GFLOPS/W
1	259	JEDI, EuroHPC/FZJ, Germany	NVIDIA GH200 Superchip	19,584	4.50	67	72.733
2	148	ROMEO-2025, ROMEO HPC Center - Champagne-Ardenne, France	NVIDIA GH200 Superchip	47,328	9.86	160	70.912
3	484	Adastra 2, GENCI-CINES, France	AMD Instinct MI300A	16,128	2.53	37	69.098
4	183	Isambard-AI phase 1, University of Bristol, UK	NVIDIA GH200 Superchip	34,272	7.42	117	68.835
5	255	Otus (GPU only), Universitaet Paderborn - PC2, Germany	NVIDIA H100 SXM5 80GB	19,440	4.66		68.177
6	66	Capella, TU Dresden, ZIH, Germany	NVIDIA H100 SXM5 94GB	85,248	24.06	445	68.053
7	304	SSC-24 Energy Module, Samsung Electronics, South Korea	NVIDIA H100 SXM5 80GB	11,200	3.82	69	67.251
8	85	Helios GPU, Cyfronet, Poland	NVIDIA GH200 Superchip	89,760	19.14	317	66.948
9	399	AMD Ouranos, Atos, France	AMD Instinct MI300A	16,632	2.99	48	66.464
10	412	Henri, Flatiron Institute, USA	NVIDIA H100 80GB PCIe	8,288	2.88	44	65.396
99	131	PRIMEHPC FX1000, Central Weather Administration, Taiwan	(CPU-only: Fujitsu A64FX)	184,320	11.16	674	16.575
119	126	Carpenter, ERDC DSRC, USA	(x86 CPU-only: AMD EPYC 9654)	276,480	11.62	1,100	10.561

(NVIDIAの)GPUの構成(1/2)



(NVIDIAの)GPUの構成(2/2)

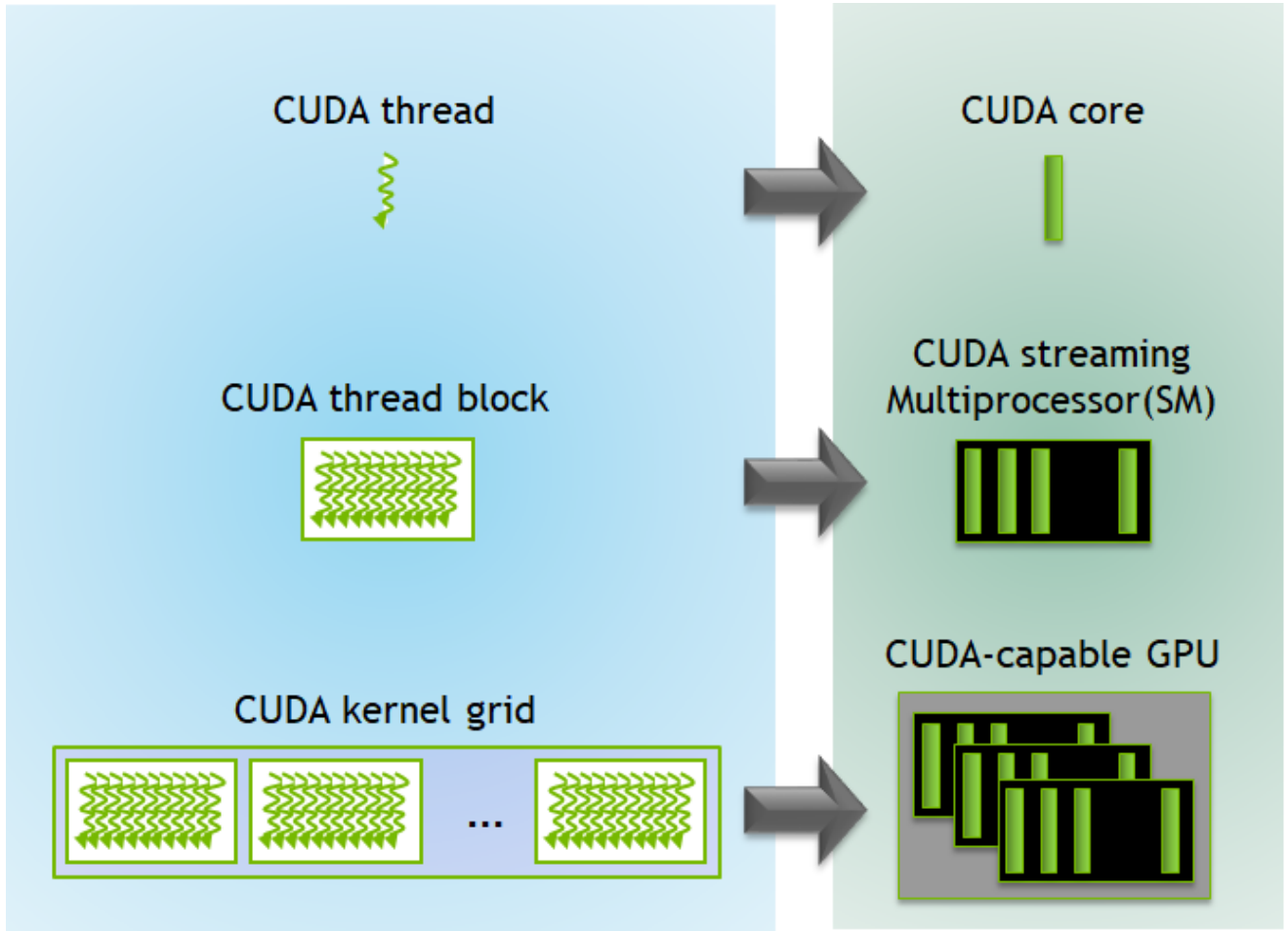
- SM:Streaming Multiprocessor
  - AMDの場合には Compute Unit (CU)
  - Intelの場合には Execution Unit (EU)
- SMの中での構造は気にしなくても性能が出せる
  - 注:32スレッドのグループ(ワープ)内で同じ動作をするように意識しておく
- SM内ではL1キャッシュ/シェアードメモリを共有(H100では256KB)
  - 配分は(ある程度)調節可能
  - シェアードメモリはCUDAでは使えるがOpenACCでは(明示的には)使えない機能
  - グローバルメモリよりも圧倒的に速い



NVIDIA H100 Tensor Core GPU Architecture

GPUプログラミングでの基本思想

- このスライドではCUDA用語で説明
- 演算器の数よりも多数のスレッドを立てて計算
  - 各種レイテンシを隠蔽するため
- スレッドブロックが基本単位
  - ブロックあたりのスレッド数は32の倍数, できれば128以上が推奨
  - SMに複数ブロックを割り当てるのが普通
  - ブロック内では32スレッド単位で動作（この組をワープと呼ぶ）
- スレッドブロックの集合をグリッドと呼ぶが, 意識しなくてOK



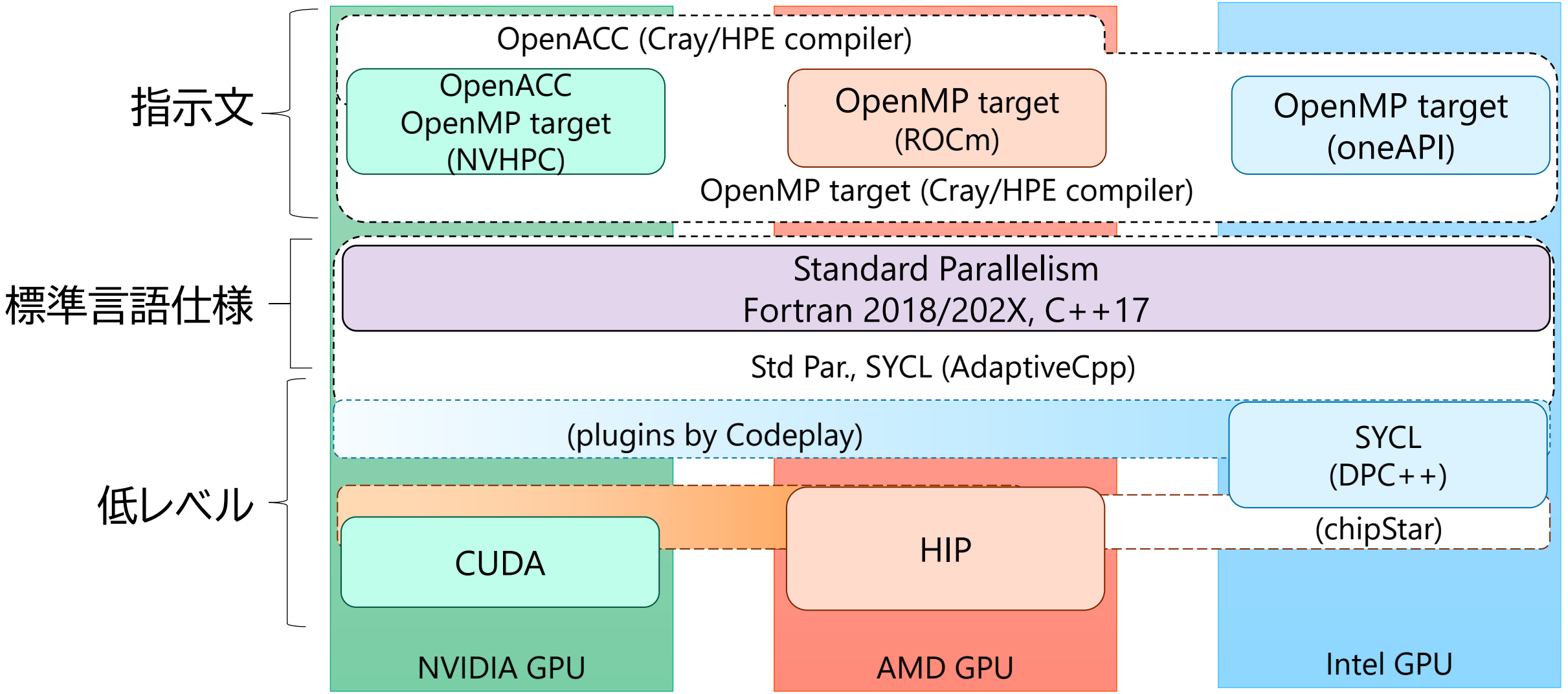
https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

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)
- 「GPU = NVIDIAのGPU」と言って良いぐらいの独占状態
  - 国内では, 現在HPCIに資源提供されている全てのGPUスパコンはNVIDIA製
    - 2024年度稼働開始のTSUBAME4.0, 玄界, Miyabi もNVIDIA製GPUを搭載
    - 2025年度はQST/NIFS, 筑波大CCSがそれぞれAMD MI300Aを搭載したシステムを運用開始
  - 海外のハイエンドスパコンではAMD, Intel製GPUも採用
    - Frontier などAMD製GPUを搭載したシステム, AuroraなどIntel製GPUを搭載したシステム
    - TOP500の上位にAMD, Intel, NVIDIA製GPUがランクイン
- GPUベンダー間の競争が活性化
  - NVIDIA製GPUの性能向上: P100, V100, A100の間は各世代$\sqrt{2}$倍➔H100では約3倍
  - 発散するプログラミング環境への対応も必要
- ポスト富岳(富岳NEXT)は演算加速器を搭載する
  - どのベンダー製の演算加速器なのか(= どうプログラミングすれば良いか)が不明
  - HPCI構成機関が今後どのようなシステムを入れてくるかも不明, 拠点ごとの多様性も?
  - ➔ ベンダーニュートラルな実装手法を採用したいが, 実現可能か? 十分な性能は出せるか?

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

- 独断と偏見に基づく不完全な比較表であることに注意
  - 特に Fortran はケアできていない（Fortranを読み書きできない人が作った資料）
    - Fortran対応で「OK」というのは動作するという意味. C/C++と同程度の性能が出るかは別問題

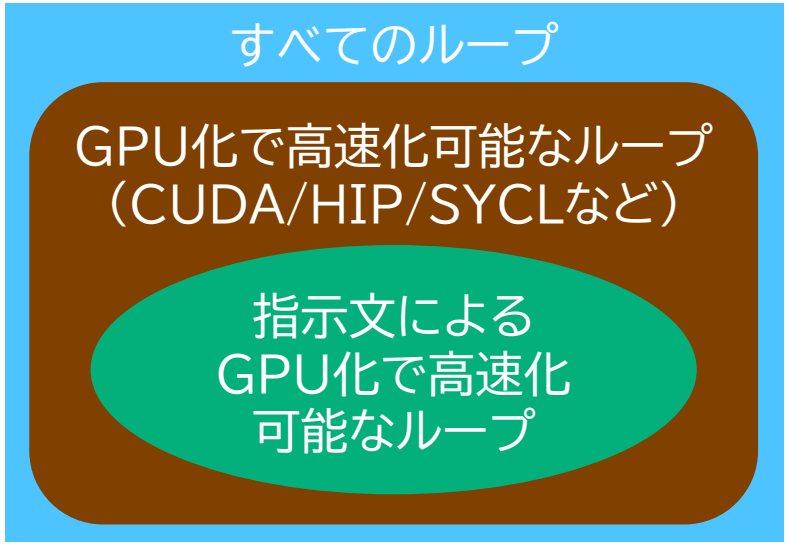
手法	ベース言語など	強み	弱み	Fortran対応	対象GPU
CUDA C++	C++	詳細な最適化可能 最新機能が使える	記述量が多い GPU専用コード	CUDA Fortran	NVIDIA限定
HIP	C++	詳細な最適化可能 ほぼCUDA C++	記述量が多い GPU専用コード	GPUFORT？ （開発停滞中？）	AMD, NVIDIA （Intel: chipStar?）
SYCL	C++	詳細な最適化可能 ラムダ式	記述量が多い ラムダ式	N/A	Intel, NVIDIA, AMD
OpenACC	指示文	移植コスト低 CPUコードと共通化可能	CUDAより遅い （メモリ律速ならそれなり？）	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, Intel （AMD: roc-stdpar?）

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

GPUプログラミングに関する資料

- 「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/242/
  - OpenACC
- 「OpenACC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/228/
  - OpenACC+MPI
- GPU移行に関するポータルサイト
  - https://jcahpc.github.io/gpu_porting/

Contents

- GPU(Graphics Processing Units)の紹介
- 指示文を用いたGPU化
  - OpenACC, OpenMP, Solomon
- CUDAを用いたGPU化
  - 指示文を用いてGPU化したコードの一部をCUDA化
  - コード全体をCUDAで実装
- ベンダーニュートラルGPUコンピューティングの紹介(HIP, SYCL)
- GPUコードの性能最適化に向けて
  - 気をつけるべき項目や機能の紹介
  - プロファイラの使い方の紹介
- MPIを用いたマルチGPU化

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

指示文(OpenACC/OpenMP)でのGPU化方針

1. Unified/Managed Memory を使って実装

- NVIDIA GH200の場合: CPU/GPU側のメモリは相互に読み書き可能
  - `-gpu=mem:unified` を指定(`-gpu=mem:unified:nomanagedalloc` の方がおすすめ)
    注:この書き方はNVIDIA HPC SDK用の記法
- 通常環境の場合: GPU上のメモリ確保, CPU-GPU 間のデータ転送は全てお任せ
  - `-gpu=mem:managed` を指定
- まずは演算部分のGPU実装に注力
- (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

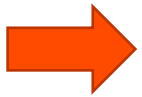
2. (Unified/Managed Memory 実装での性能に満足できない場合) データ指示文を使ってコードをアップデート

- 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
- Unified/Managed Memory ではCPU上のアドレスとGPU上のアドレスを同一視してしまうので, GPUDirect系の機能を活用する際に不利

OpenACCでの実装(演算部分)

- `#pragma omp parallel for` を置き換えていく
  - `#pragma acc kernels` はGPU化するかコンパイラが判断(GPU化しないことも)
  - `#pragma acc parallel` はGPU化できるとユーザが保証(GPU化される)
- 性能的に特に重要なものについては, スレッド数の調整も
  - `vector_length(スレッド数)` をkernels/parallel指示文に付与して示唆
  - `vector(スレッド数)` をloop指示文に付与しても同じことができる

```
#pragma omp parallel for
  for (int i = 0; i < Ni; i++) {
    // ループ内の実装は省略
  }


#pragma omp parallel for
  for (int i = 0; i < Ni; i++) {
    // ループ内の実装は省略
  }
```

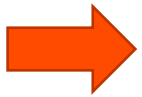


```
#pragma acc kernels
#pragma acc loop independent
  for (int i = 0; i < Ni; i++) {
    // ループ内の実装は省略
  }

#pragma acc kernels vector_length(NTHREADS)
#pragma acc loop independent
  for (int i = 0; i < Ni; i++) {
    // ループ内の実装は省略
  }
```

OpenMPでの実装(演算部分:distribute)

- `#pragma omp parallel for` を置き換えていく
- 性能的に特に重要なものについては, スレッド数の調整も
  - `thread_limit(スレッド数)` としてコンパイラに示唆(強制はできない)
  - `num_teams(チーム数)` という方法もある(が, あまり使いたくはない)
    - 全体の問題サイズが決まっている際にはスレッド数の指定と等価になるが, 実行時にパラメータファイルを読み込んで問題サイズを設定するような実装には不向き

```
#pragma omp parallel for
  for (int i = 0; i < Ni; i++) {
    // ループ内の実装は省略
  }


#pragma omp parallel for
  for (int i = 0; i < Ni; i++) {
    // ループ内の実装は省略
  }
```

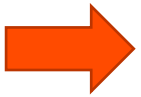


```
#pragma omp target teams distribute parallel for simd
  for (int i = 0; i < Ni; i++) {
    //ループ内の実装は省略
  }


#pragma omp target teams distribute parallel for simd
thread_limit(NTHREADS)
  for (int i = 0; i < Ni; i++) {
    //ループ内の実装は省略
  }
```

OpenMPでの実装(演算部分:loop)

- `#pragma omp parallel for` を置き換えていく
- (先述の)distributeよりも, コンパイラに多くを委ねる実装法
  - OpenMP 5.0 で導入された記法
- 性能的に特に重要なものについては, スレッド数の調整も
  - `thread_limit(スレッド数)` としてコンパイラに示唆(強制はできない)
  - `num_teams(チーム数)` という方法もある(が, あまり使いたくはない)

```
#pragma omp parallel for
  for (int i = 0; i < Ni; i++) {
    // ループ内の実装は省略
  }

#pragma omp parallel for
  for (int i = 0; i < Ni; i++) {
    // ループ内の実装は省略
  }
```



```
#pragma omp target teams loop
  for (int i = 0; i < Ni; i++) {
    //ループ内の実装は省略
  }

#pragma omp target teams loop thread_limit(NTHREADS)
  for (int i = 0; i < Ni; i++) {
    //ループ内の実装は省略
  }
```

多重ループではどのループをGPU化すべきか？

- まずは最外側ループに指示文を付与してGPU化する
  - OpenMP を使ってマルチコアCPU向けに並列化する際と同様（スレッドの生成・消滅コストを削減）
  - 十分な並列度（目安：コア数の10倍以上）が確保できない場合には，内側ループも
- N体の場合には外側ループだけで十分な並列度が確保できることが多い
  - 内側ループも並列化すると，reduction/atomic系の処理が入ってむしろ性能低下
- メッシュ流体の場合には大外ループだけでは十分な並列度には達しないはず
  - GPUあたりのメッシュ数が256^3であれば，最外側ループの並列度は（たった）256
  - 多重ループをcollapseするのが常套手段（上記の例では256^3 =16M）

```
#pragma acc kernels
#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++) {
      // loop body
    }}}
```

```
#pragma omp target teams loop collapse(3)
for (int i = 0; i < nx; i++) {
  for (int j = 0; j < ny; j++) {
    for (int k = 0; k < nz; k++) {
      // loop body
    }}}
```

OpenACCでの実装(データ転送部分)

- Unified/Managed Memoryを使わない場合のみ必要
  - GPU上に置くべきデータ, 必要なデータ転送を指定

```
// メモリ確保
#pragma acc enter data create(pos_ptr [0:num], vel_ptr [0:num], acc_ptr [0:num])

// CPU → GPU のデータ転送
#pragma acc update device(pos_ptr [0:num], vel_ptr [0:num])

// GPU → CPU のデータ転送
#pragma acc update host(acc_ptr [0:num])

// メモリ解放
#pragma acc exit data delete (pos_ptr [0:num], vel_ptr [0:num], acc_ptr [0:num])
```

OpenMPでの実装(データ転送部分)

- Unified/Managed Memoryを使わない場合のみ必要
  - GPU上に置くべきデータ, 必要なデータ転送を指定

```
// メモリ確保
#pragma omp target enter data map(alloc : pos_ptr [0:num], vel_ptr [0:num], acc_ptr [0:num])

// CPU → GPU のデータ転送
#pragma omp target update to(pos_ptr [0:num], vel_ptr [0:num])

// GPU → CPU のデータ転送
#pragma omp target update from(acc_ptr [0:num])

// メモリ解放
#pragma omp target exit data map(delete : pos_ptr [0:num], vel_ptr [0:num], acc_ptr [0:num])
```

指示文によるGPU化が意味するものは何か？

- 具体的にどうGPU化するかはコンパイラ任せ
  - 細部まで指定したい人は CUDA/HIP/SYCL を使う
  - つまり, 実装の詳細はブラックボックスであってもGPU化されていればOK
- CUDA/HIP/SYCL で全力最適化した場合よりも性能が低いことは受け入れているユーザ層
  - 最大性能よりも, CPUコードとの互換性や移植工数削減の方を優先
  - OpenACC と OpenMP targetの性能差は CUDA/HIP/SYCL からの性能差に比べれば小さいはず
- 本質的には OpenACC or OpenMP target？ はどちらでも良い(はず)
  - 何か指示文的なものを書いたらGPU化してくればOK
  - 新しい指示文を作るようなことは検討しない
    - 新しい移行コストが発生するだけ
  - 新しいコンパイラを開発するようなことは検討しない
    - コンパイラの開発・メンテナンスが止まった時にユーザーを道連れにしてしまう
    - 大変なので決してやりたくない, プロの方々にお任せする

マクロを用いた指示文のブラックボックス化

- Solomon(Simple Off-LOading Macros Orchestrating multiple Notations)を実装
  - Miki & Hanawa (2024, IEEE Access)
- バックエンドで OpenACC or OpenMP target に展開
  - Fallback mode (マルチコアCPU向けのOpenMPに展開)も実装済み
- ユーザ的にはオーバーオールのフラグ制御だけで OpenACC or OpenMP target の切り替えが可能
  - NVIDIA GPU 上では OpenACC で, AMD/Intel GPU 上では OpenMP target で動かす, ということが可能になる
  - 最適化レベルを揃えた上で OpenACC と OpenMP target の性能比較
- コンパイラではないのでベンダー製のコンパイラ性能をそのまま利用できる
  - (GPU向けプログラミングに詳しい人は)HIP の指示文版とイメージすると良い
  - 自作コンパイラの場合には, 最新機能への追随のためのコストが継続的に生じる
- 開発者が更新をさぼっても, 自分でマクロを付け足すことも簡単
  - 新コンパイラ/新指示文の実装であれば, 一般ユーザはほぼ手出しできない

Solomonを用いた実装例

- 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 (int32_t i = 0; i < N; i++) {
```

```
#ifdef OFFLOAD_BY_OPENACC
#pragma acc kernels vector_length(NTHREADS)
#pragma acc loop independent
#endif  // OFFLOAD_BY_OPENACC
#ifdef OFFLOAD_BY_OPENMP_TARGET
#ifdef OFFLOAD_BY_OPENMP_TARGET_LOOP
#pragma omp target teams loop
thread_limit(NTHREADS)
#else  // OFFLOAD_BY_OPENMP_TARGET_LOOP
#pragma omp target teams distribute parallel
for simd thread_limit(NTHREADS)
#endif  // OFFLOAD_BY_OPENMP_TARGET_LOOP
#endif  // OFFLOAD_BY_OPENMP_TARGET
  for (int32_t i = 0; i < N; i++) {
```

Solomonでの実装例(データ転送に関連する部分)

- ホスト(CPU)からデバイス(GPU)にデータ転送する場合

```
MEMCPY_H2D(pos [0:num], vel [0:num])                                    # 簡易記法
PRAGMA_ACC_UPDATE_DEVICE(pos [0:num], vel [0:num])                      # OpenACC的記法
PRAGMA_OMP_TARGET_UPDATE_TO(pos [0:num], vel [0:num])                   # OpenMP的記法
```

- デバイス(GPU)からホスト(CPU)にデータ転送する場合

```
MEMCPY_D2H(pos [0:num], vel [0:num])                                    # 簡易記法
PRAGMA_ACC_UPDATE_HOST(pos [0:num], vel [0:num])                        # OpenACC的記法
PRAGMA_OMP_TARGET_UPDATE_FROM(pos [0:num], vel [0:num])                 # OpenMP的記法
```

- デバイス(GPU)上のメモリを確保・解放する場合

```
MALLOC_ON_DEVICE(pos [0:num], vel [0:num])                              # 簡易記法
PRAGMA_ACC_ENTER_DATA_CREATE(pos [0:num], vel [0:num])                  # OpenACC的記法
PRAGMA_OMP_TARGET_ENTER_DATA_MAP_ALLOC(pos [0:num], vel [0:num])        # OpenMP的記法
```

```
FREE_FROM_DEVICE(pos [0:num], vel [0:num])                              # 簡易記法
PRAGMA_ACC_EXIT_DATA_DELETE(pos [0:num], vel [0:num])                   # OpenACC的記法
PRAGMA_OMP_TARGET_EXIT_DATA_MAP_DELETE(pos [0:num], vel [0:num])        # OpenMP的記法
```

Solomon のインターフェース(指示文)

- 簡易記法, OpenACC的記法, OpenMP的記法の3種を提供

入力		出力	
簡易記法	OpenACC/OpenMP 的記法	展開先	バックエンド
OFFLOAD(...)	PRAGMA_ACC_KERNELS_LOOP(...)	_Pragma("acc kernels __VA_ARGS__") _Pragma("acc loop __VA_ARGS__")	OpenACC (kernels)
	PRAGMA_ACC_PARALLEL_LOOP(...)	_Pragma("acc parallel __VA_ARGS__") _Pragma("acc loop __VA_ARGS__")	OpenACC (parallel)
	PRAGMA_OMP_TARGET_TEAMS_LOOP(...)	_Pragma("omp target teams loop __VA_ARGS__")	OpenMP (loop)
	PRAGMA_OMP_TARGET_TEAMS_DISTRIBUTE_PARALLEL_FOR(...)	_Pragma("omp target teams distribute parallel for __VA_ARGS__")	OpenMP (distribute)
MALLOC_ON_DEVICE(...)	PRAGMA_ACC_ENTER_DATA_CREATE(...)	_Pragma("acc enter data create(__VA_ARGS__)")	OpenACC
	PRAGMA_OMP_TARGET_ENTER_DATA_MAP_ALLOC(...)	_Pragma("omp target enter data map(alloc: __VA_ARGS__)")	OpenMP
FREE_FROM_DEVICE(...)	PRAGMA_ACC_EXIT_DATA_DELETE(...)	_Pragma("acc exit data delete(__VA_ARGS__)")	OpenACC
	PRAGMA_OMP_TARGET_EXIT_DATA_MAP_DELETE(...)	_Pragma("omp target exit data map(delete: __VA_ARGS__)")	OpenMP
MEMCPY_D2H(...)	PRAGMA_ACC_UPDATE_HOST(...)	_Pragma("acc update host(__VA_ARGS__)")	OpenACC
	PRAGMA_OMP_TARGET_UPDATE_FROM(...)	_Pragma("omp target update from(__VA_ARGS__)")	OpenMP
MEMCPY_H2D(...)	PRAGMA_ACC_UPDATE_DEVICE(...)	_Pragma("acc update device(__VA_ARGS__)")	OpenACC
	PRAGMA_OMP_TARGET_UPDATE_TO(...)	_Pragma("omp target update to(__VA_ARGS__)")	OpenMP
DATA_ACCESS_BY_DEVICE(...)	PRAGMA_ACC_DATA(...)	_Pragma("acc data __VA_ARGS__")	OpenACC
	PRAGMA_OMP_TARGET_DATA(...)	_Pragma("omp target data __VA_ARGS__")	OpenMP
DATA_ACCESS_BY_HOST(...)	PRAGMA_ACC_HOST_DATA(...)	_Pragma("acc host_data __VA_ARGS__")	OpenACC
	PRAGMA_OMP_TARGET_DATA(...)	_Pragma("omp target data __VA_ARGS__")	OpenMP
SYNCHRONIZE(...)	PRAGMA_ACC_WAIT(...)	_Pragma("acc wait __VA_ARGS__")	OpenACC
	PRAGMA_OMP_TARGET_TASKWAIT(...)	_Pragma("omp taskwait __VA_ARGS__")	OpenMP
ATOMIC(...)	PRAGMA_ACC_ATOMIC(...)	_Pragma("acc atomic __VA_ARGS__")	OpenACC
	PRAGMA_OMP_TARGET_ATOMIC(...)	_Pragma("omp atomic __VA_ARGS__")	OpenMP
DECLARE_OFFLOADED(...)	PRAGMA_ACC_ROUTINE(...)	_Pragma("acc routine __VA_ARGS__")	OpenACC
	PRAGMA_OMP_DECLARE_TARGET(...)	_Pragma("omp declare target __VA_ARGS__")	OpenMP
DECLARE_OFFLOADED_END	PRAGMA_OMP_END_DECLARE_TARGET	_Pragma("omp end declare target")	OpenMP (only)

Solomon のインターフェース(指示節)

- 簡易記法, OpenACC的記法, OpenMP的記法の3種を提供
  - 1つの指示文に対して記述の中で複数の記法を混ぜてもOK

入力		出力	
簡易記法	OpenACC/OpenMP 的記法	展開先	バックエンド
AS_INDEPENDENT	ACC_CLAUSE_INDEPENDENT	independent	OpenACC
	OMP_TARGET_CLAUSE_SIMD	simd	OpenMP
NUM_THREADS(n)	ACC_CLAUSE_VECTOR_LENGTH(n)	vector_length(n)	OpenACC
	OMP_TARGET_CLAUSE_THREAD_LIMIT(n)	thread_limit(n)	OpenMP
COLLAPSE(n)	ACC_CLAUES_COLLAPSE(n)	collapse(n)	OpenACC
	OMP_TARGET_CLAUSE_COLLAPSE(n)		OpenMP
REDUCTION(...)	ACC_CLAUSE_REDUCTION(...)	reduction(__VA_ARGS__)	OpenACC
	OMP_TARGET_CLAUSE_REDUCTION(...)		OpenMP
AS_ASYNC(...)	ACC_CLAUSE_ASYNC(...)	async(__VA_ARGS__)	OpenACC
	OMP_TARGET_CLAUSE_NOWAIT	nowait	OpenMP

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)
- AS_INDEPENDENT は(複数の)指示節の先頭に置いておく(必須)
  - バックエンドを OpenMP にした時には simd に変換されるが, simd は構文の一部分であるため, 中間に他の指示節が紛れ込むとエラーとなってしまう
  - 指示節候補をソートした後で判定・有効化するよう実装すれば回避できるはず(未対応)
- 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](…)

OpenACCコードのコンパイル方法など

- NVIDIA HPC SDK向けの情報(NVIDIA A100 = cc80向け)
- `$ nvc -acc=gpu -gpu=cc80 -Minfo=accel,opt`
  - リンク時にも `-acc=gpu` を指定する
- `$ nvc -acc=gpu -gpu=cc80,mem:managed -Minfo=accel,opt`
  - こちらはManaged Memory使用時のコンパイル方法
    - GH200の場合(Unified Memory 使用)は, `-gpu=mem:unified:nomanagedalloc`を推奨
  - リンク時にも `-acc=gpu -gpu=mem:managed` を指定する
- デバッグ時などに便利な環境変数
  - `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

OpenMPコードのコンパイル方法など

- NVIDIA HPC SDK向けの情報(NVIDIA A100 = cc80向け)
- `$ nvc -mp=gpu -gpu=cc80 -Minfo=accel,opt,mp`
  - リンク時にも `-mp=gpu` を指定する
- `$ nvc -mp=gpu -gpu=cc80,mem:managed -Minfo=accel,opt,mp`
  - Managed Memory使用時のコンパイル方法
  - リンク時にも `-mp=gpu -gpu=mem:managed` を指定する
- 参考:AMD製GPU(MI210)向けにコンパイルする場合
  - `$ amdclang++ -fopenmp -offload-arch=gfx90a`
- 参考:Intel製GPU(DC GPU Max 1100)向けにコンパイルする場合
  - `$ icpx -fiopenmp -fopenmp-targets=spir64_gen -Xs “-device pvc”`

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 250819gpu_directive` # 何か適当なフォルダを作る
- `$ cd    250819gpu_directive` # 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 .`

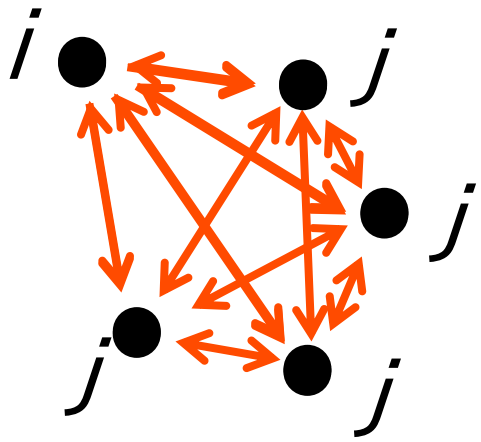
- `$ git clone https://github.com/ymiki-repo/solomon.git`
  - これは Solomon を使ってGPU化する人のみ実行

N体計算(重力多体計算)

- 粒子どうしに働く自己重力による系の時間進化を，運動方程式に基づいて計算
  - データ量: $\mathcal{O}(N)$
  - 重力計算: $\mathcal{O}(N^2)$
  - 時間積分: $\mathcal{O}(N)$

$$\boldsymbol{a}_i = \sum_{\substack{j=0 \\ j\neq i}}^{N-1} \frac{G m_j \left(\boldsymbol{x}_j - \boldsymbol{x}_i\right)}{\left(\left|\boldsymbol{x}_j - \boldsymbol{x}_i\right|^2 + \epsilon^2\right)^{3/2}}$$

- N体業界的によく使う用語
  - i-粒子: 重力を受ける粒子
  - j-粒子: 重力を及ぼす粒子

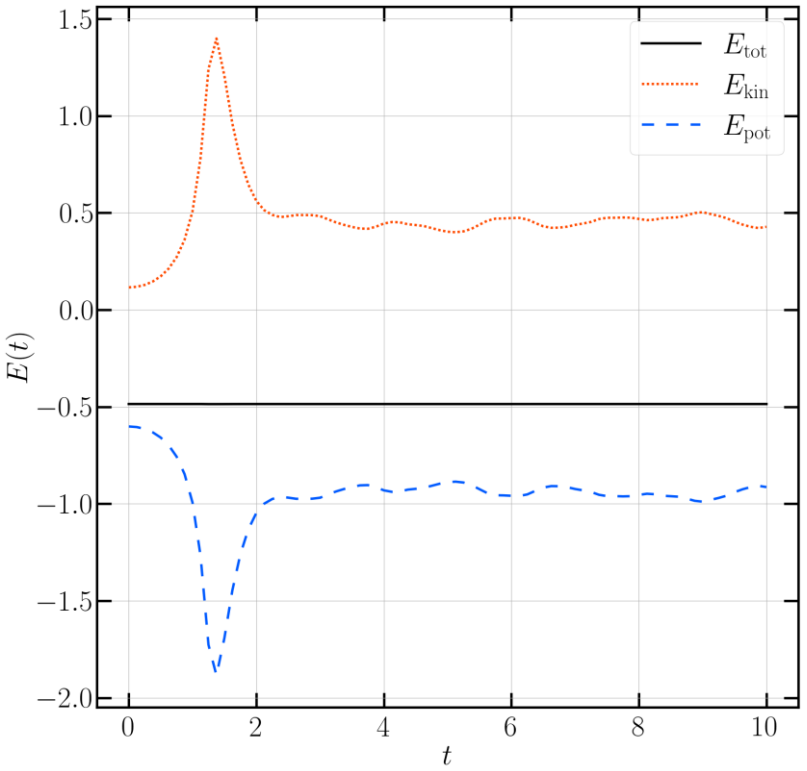


- 直接法のソルバーはツリーコードのバックエンドとして使えるので，高速化しておく価値が高い(また，衝突系N体計算であれば直接法を用いる)

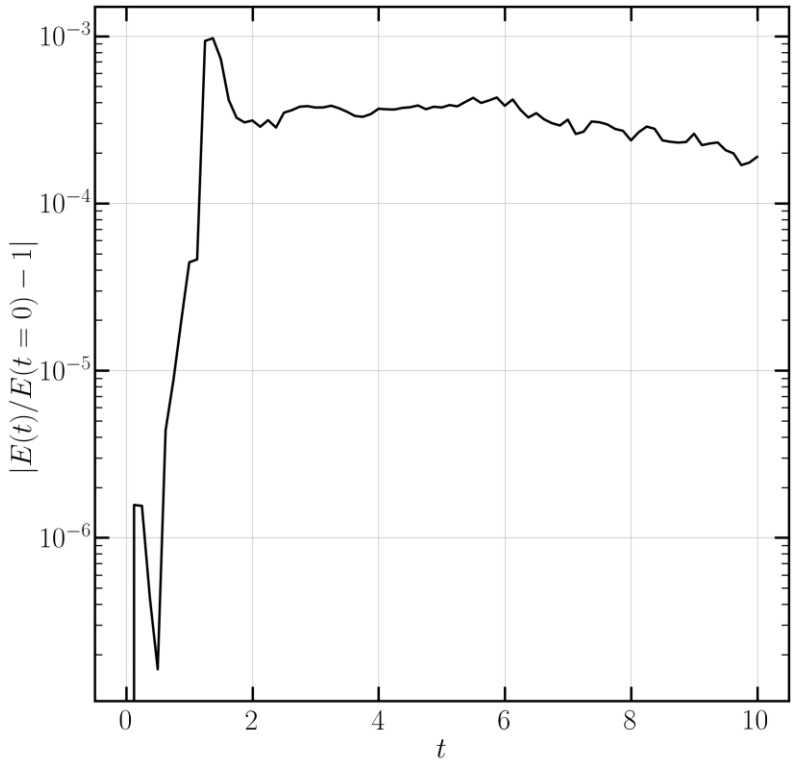
計算結果の可視化例(1/2)

- https://github.com/ymiki-repo/nbody/tree/main/gallery/validation/fig

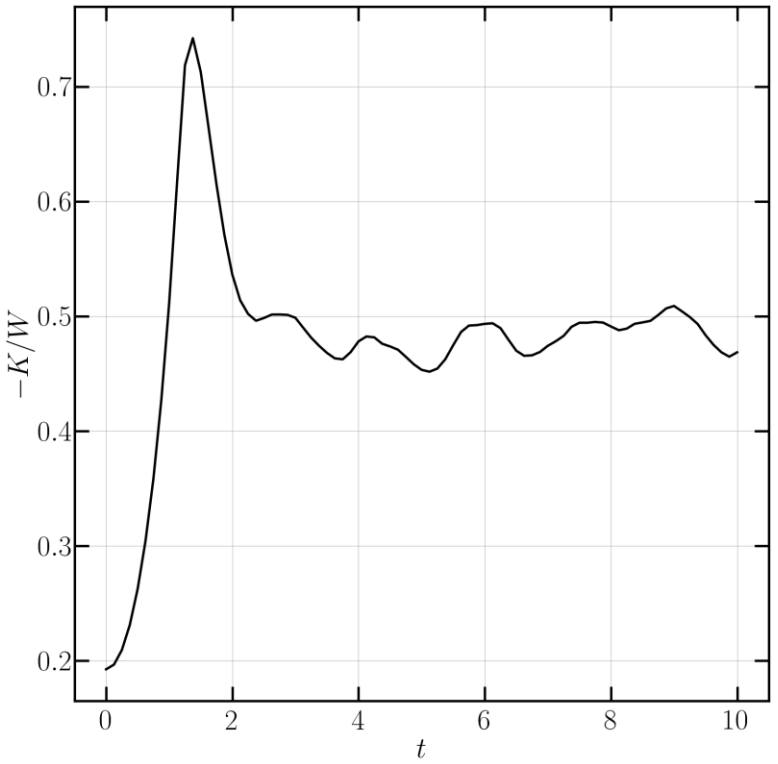
エネルギーの時間進化



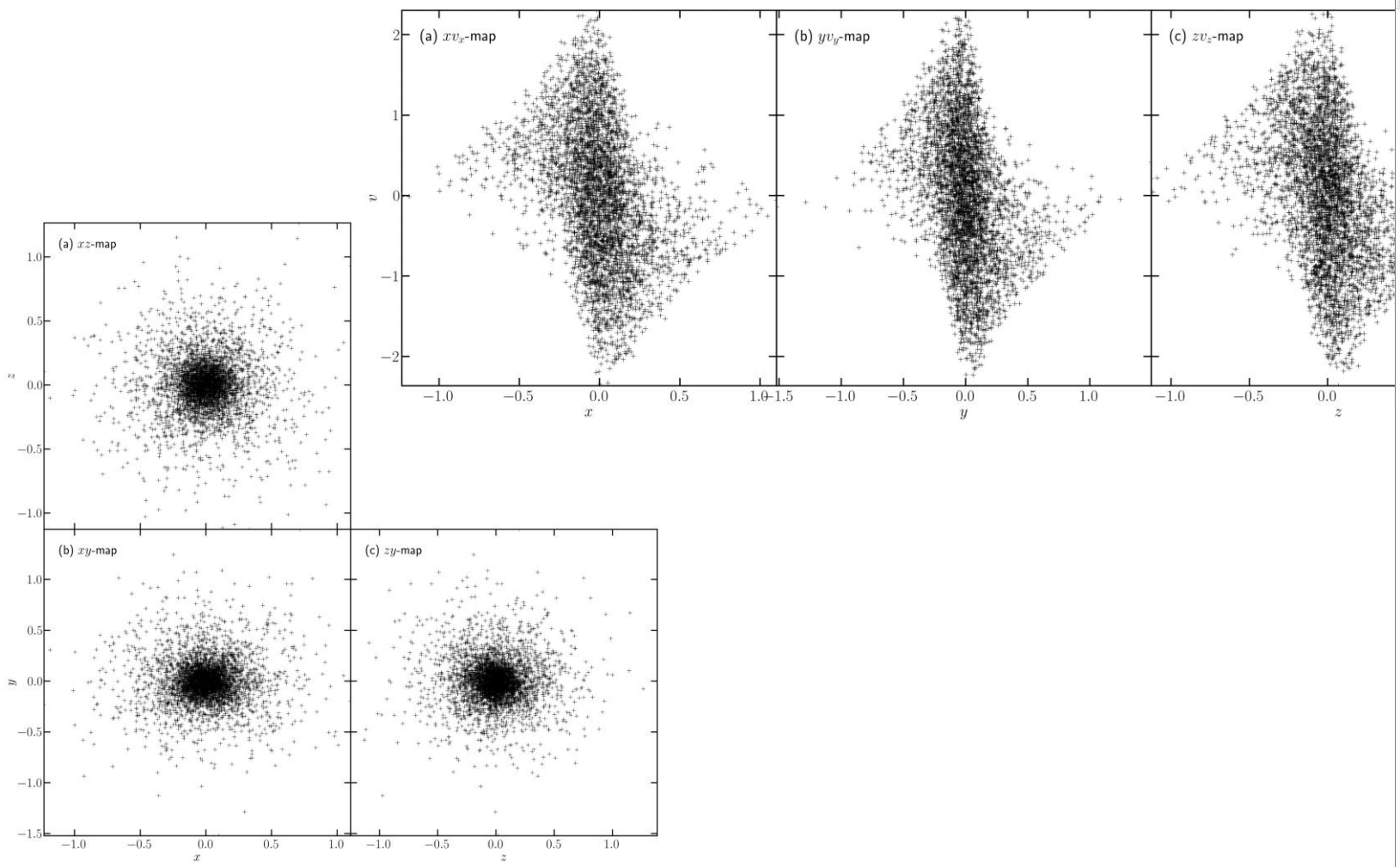
エネルギー保存

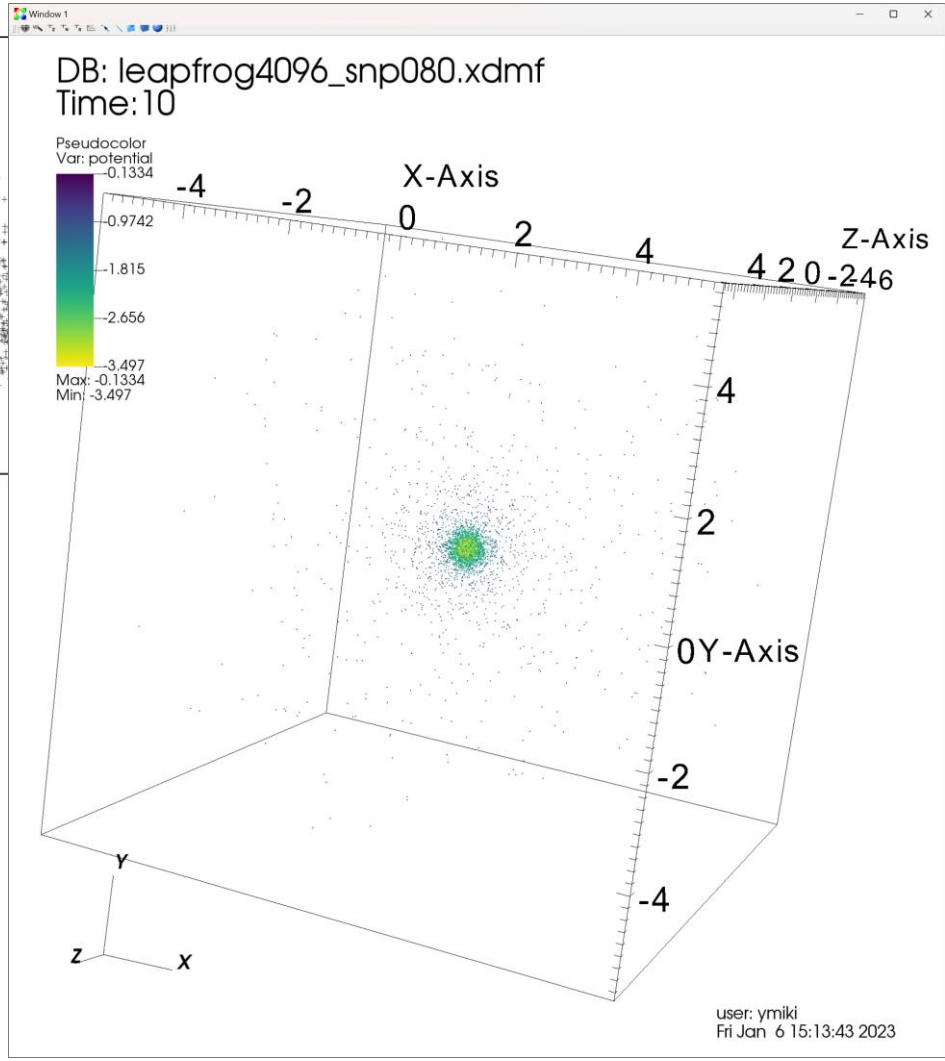


ビリアル比の時間進化



計算結果の可視化例(2/2)



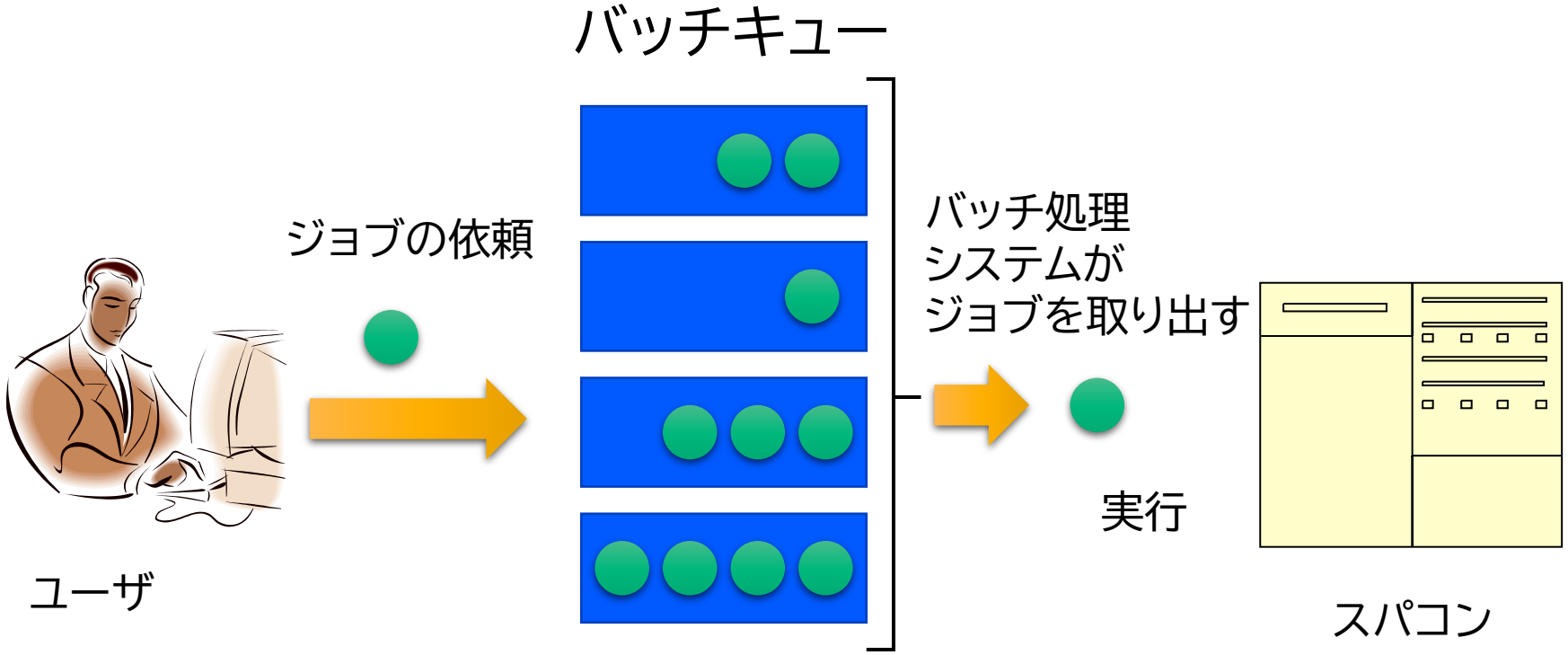


コンパイル方法

- `$ module purge`
  - デフォルトでロードされているモジュールと衝突する場合(今回は不要)
- `$ module load nvhpc/25.7`
  - 前ページなどで紹介した `-gpu=mem:managed` という記法はNVIDIA HPC SDK 24.5から導入されたもので, デフォルトのnvhpc/22.2では使えません
  - 単に `$ module load nvhpc` とせずに, バージョン指定するよう注意してください
- `$ make`
  - はじめのうちは直接nvcを叩いても構いませんが, すぐ後で複数ファイルの結合(指示文実装+CUDA実装)なども扱うので, はじめから Makefile を作るほうが楽です

バッチ処理とは

- スパコン環境では，通常は，インタラクティブ実行(コマンドラインで実行すること)はできません
- ジョブはバッチ処理で実行します



演習環境におけるバッチ処理

- 今回の演習環境でのバッチ処理は, Slurmで管理
- 主要コマンド:
  - ジョブの投入: `sbatch <ジョブスクリプト名>`
  - 自分が投入したジョブの状況確認: `squeue`
  - 投入ジョブの削除: `scancel <ジョブID>`

- ジョブスクリプト例:
  - 今回の講習会では `gpuws` を使用
  - 使用GPU数を `--gres=gpu:N` として指定
  - 実行時間の上限は10分
  - モジュール設定は基本的にコンパイル時と揃える

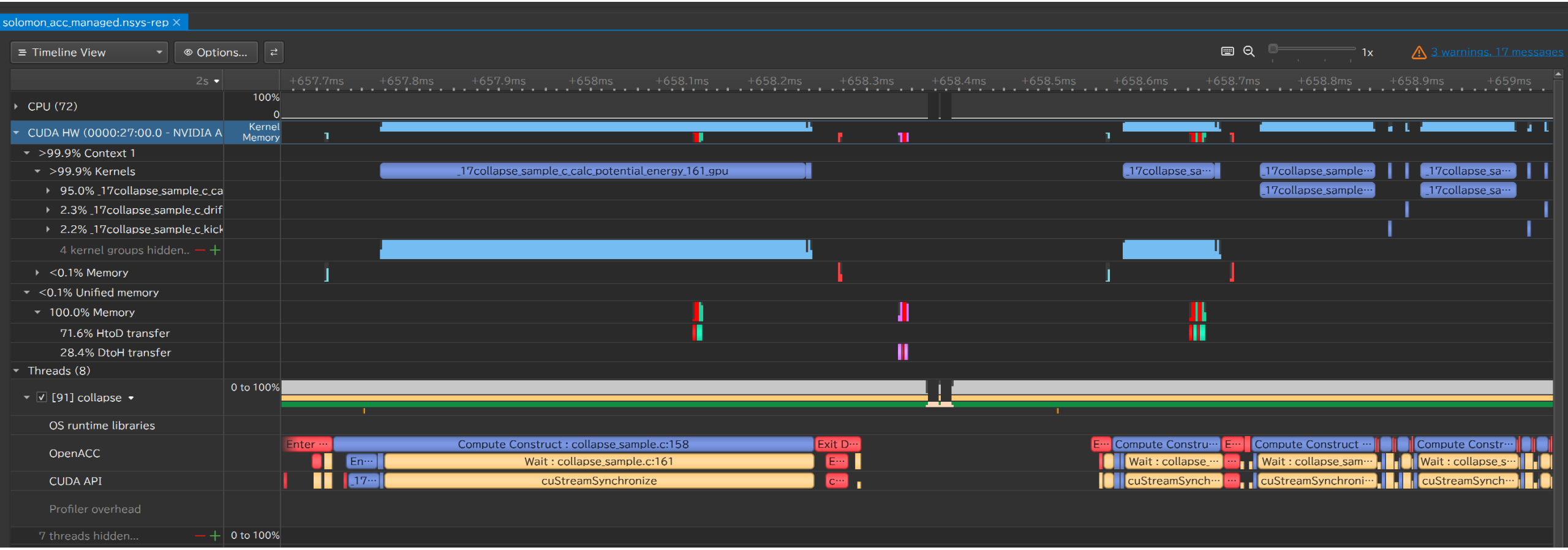
```
#!/usr/bin/env bash
#SBATCH --partition=gpuws
#SBATCH --gres=gpu:1
#SBATCH --time=00:05:00

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

module purge
module load nvhpc/25.7

cd ${SLURM_SUBMIT_DIR}

./collapse ${PARAM}
```

プロファイラの使い方(簡易版)

- GPU化の大まかな方針
  - 実行時間を測定し, 処理が重い部分を見つける
  - 処理が重い部分から順番にGPU化していく
  - CPU･GPU間のデータ転送は後回し(managed/unified memoryを利用)
- まずはプロファイラを使って重い部分を見つけていく
  - (今回のサンプルコードに関しては重い部分は自明(重力計算をしているところ))
- `$ nsys profile --stats=true ./a.out`
  - `--stats=true` をつけておくと, 結果の概要も出力されるので便利
  - report?.nsys-rep というファイルが生成されるので, (手元で or Xを飛ばして) nsys-ui で開く
    - どちらかというと, 手元の環境にNVIDIA Nsight Systemsをインストールする方が楽
      - Windows, macOS, Linux のどれでもOK
      - https://developer.nvidia.com/nsight-systems

プロファイラの使い方紹介

- 下記の例はOpenACC + managed memory の場合(Wisteria-A)
  - CUDA HW の中にはGPUで実行されている関数が表示される
  - Managed memory 任せにしたデータ転送は, Unified memory 内に表示



N体コード動作時に入力するパラメータなど

- 元コードではいくつかのパラメータを実行時に入力する仕様になっている
  - バッチジョブ実行する関係上, このままでは少し使いづらいので書き換え推奨（GPU化とは直接関係ない部分）
  - 書き換え方針A: コード内で直接パラメータを指定してしまう
  - 書き換え方針B: 入力パラメータを記載したファイルを作り, ファイルから読み取る
- 動作テスト時のパラメータ指定（以下の値を厳密に守る必要はありません）
  - 重力ソフトニング: 1.5625e-2（1/100程度の値. 初期球の半径が1の系）
  - 時間刻み: -7（これは2^{-7}という意味. 重力ソフトニングより小さい値にしておくと, $v_{\max} \lesssim 1$であれば十分な時間分解能になる）
    - 最終的なエネルギー保存の誤差が0.5%よりも小さくなるように設定（あくまでも一つの目安）
  - 計算終了時刻: 10.0（自由落下時間が1程度の系）
  - 粒子数: 1024（小さい値にしておいた方が実行時間が短く, バグを見つけやすい）
  - 初期ビリアル比: 0.2（0.1でも良い. 小さくするほど数値計算的に難しい設定）
- 元コードでは定期的にGNUPLOTで描画する仕様になっているが, バッチジョブでは使いづらいためにコメントアウトしてしまう方が楽
  - 途中結果を把握したければ, スナップショットファイルを出力する方式を推奨

実習:指示文を使ってのGPU化

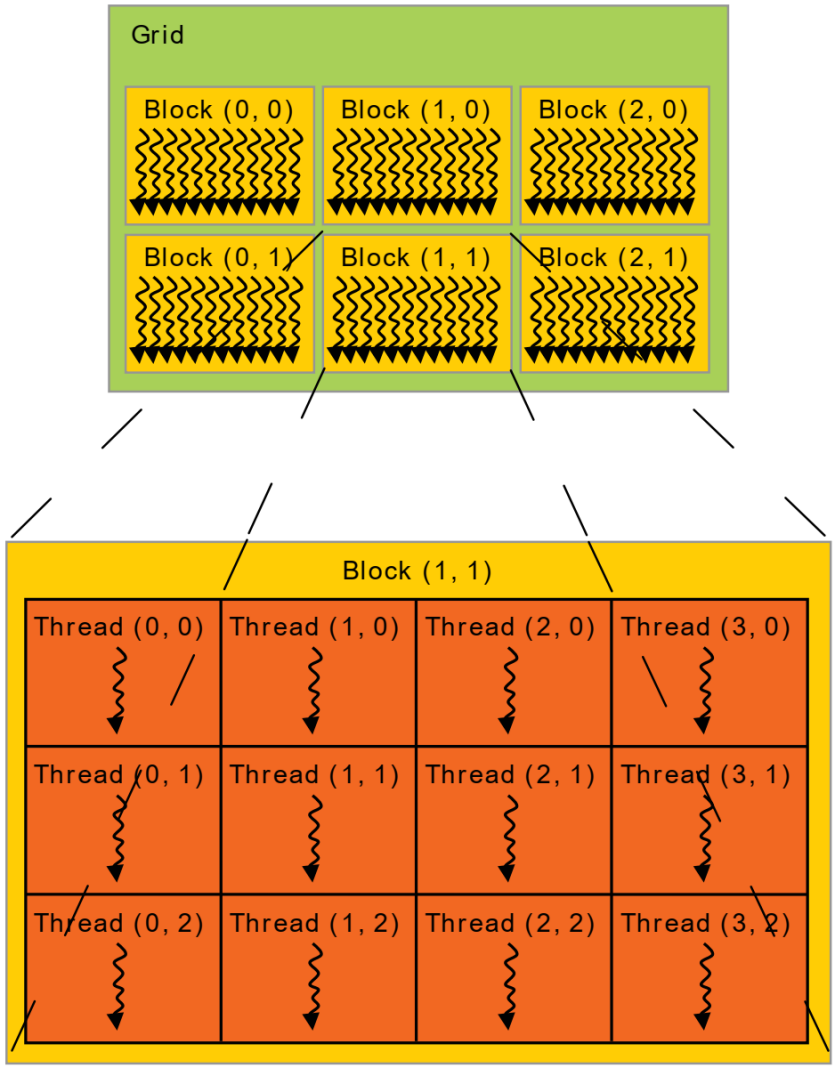
1. 演算部分を指示文を使ってGPU化してみてください
   - データ転送などは, managed memoryを使ってシステムに丸投げ
   - OpenACC, OpenMP target, Solomon のどれを使ってもOK
   - サンプルコードをそのままGPU化しようと思うと面倒な部分もあるため, 適宜リファクタリングもした方が良いです
2. メモリ確保, データ転送部分も指示文でGPU化してみてください
   - 1. でGPU化したコードを拡張する方針
   - Makefile を編集するのをお忘れなく
     (`-gpu=mem:managed` が入っていると, データ指示文などは全て無視されます)

Contents

- GPU(Graphics Processing Units)の紹介
- 指示文を用いたGPU化
  - OpenACC, OpenMP, Solomon
- CUDAを用いたGPU化
  - 指示文を用いてGPU化したコードの一部をCUDA化
  - コード全体をCUDAで実装
- ベンダーニュートラルGPUコンピューティングの紹介(HIP, SYCL)
- GPUコードの性能最適化に向けて
  - 気をつけるべき項目や機能の紹介
  - プロファイラの使い方の紹介
- MPIを用いたマルチGPU化

Compute Unified Device Architecture

- NVIDIA社のGPUを対象としたプログラミング環境
  - C++の拡張(昔はC/C++の拡張だったが…)
  - GPUの内部構造を(あまり)意識しないで良い
- 多くのスレッドを生成
  - 多数のコアを有効に使うため
  - グローバルメモリへのアクセスレイテンシを隠蔽
- ブロック:スレッドの集合
  - 典型的には128-512スレッド程度
  - シェアードメモリ, L1キャッシュを共有
  - 同期はいくつかの粒度で取れる
  - 最も遅いスレッドが全体を律速



CUDA C++ Programming Guide (Version 11.2)

簡易GPU化との比較

- CUDAでは, GPU上で実行される関数は非同期実行される
  - GPU上での実行完了を待たずにCPU上で次の命令が実行される
  - OpenACCなどでは, 同期的に実行されていた
  - `cudaDeviceSynchronize()` によって完了が保証できる
  - 特に Unified/Managed Memory を使う場合には気をつけないと危険
- GPU上で実行される関数などはCUDAストリームに紐付く
  - デフォルトストリームのみを使っている場合には, 意識する必要なし
  - 同一ストリーム内では実行順序などの一貫性が保証される
    - `cudaDeviceSynchronize()` を入れて同期する必要はない
  - 複数のCUDAストリームを使って処理をオーバーラップすることも可能
    - `cudaStreamSynchronize()` などを使って適切に管理する
- OpenACCでCUDAの8-9割を目指す or 達成とは言うものの,
  - CUDAで全力の最適化を施したコードと比較されていないことが多い
  - CUDAで実装しなおしました, という程度のコードとの比較が多い
  - たまにCUDAより速くなったという結果を見せられることもありますが, CUDA側の最適化が足りていないのでは？ と思って聞いていることが多い

CUDA C++で実装する際の記述項目

- GPUの起動(cudaSetDevice())
  - これが必要なのは複数GPUを触りに行く場合
  - 1 GPU / MPI プロセスであれば, CUDA_VISIBLE_DEVICESが便利
- デバイスメモリの確保(cudaMalloc())
- データ転送用メモリの確保(cudaMallocHost())
- CPU⇔GPU間のデータ転送(cudaMemcpy())
- __global__関数の実装(CPUから起動するGPU関数)
- __device__関数の実装(GPU関数から呼び出すGPU関数)
- カーネル立ち上げ命令の追加(kernel<<<blck, thrd>>>())
- 確保したメモリの開放(cudaFree(), cudaFreeHost())

CUDA C++でのコンパイル方法

- `$ nvcc -gencode arch=compute_80,code=sm_80 -Xptxas -v,-warn-spills,-warn-lmem-usage –lineinfo`
  - `-gencode arch=compute_80,code=sm_80`
    - 用いるGPUの世代を指定するオプションで, 2つの数字は揃えなくても良い (`arch=compute_60,code=sm_80` は Ampere(80)世代のGPUをPascal(60)世代の動作モードで使用するという意味. ただしCUDA 13からはPascalサポートがなくなるとのこと)
  - `-Xptxas -v,-warn-spills,-warn-lmem-usage`
    - 性能最適化用に出力することが多いオプション. レジスタスピルやローカルメモリ落ちすると性能低下するので, コンパイル時にそのあたりのことが起こっていないかを確認
  - `-lineinfo`
    - デバイスコードにコードの行数を埋め込める

CUDA C++での実装(お手軽版:1/5)

- まずはGPU上で動作させたい関数をGPU化
  - 関数定義の先頭に__global__をつける
    - GPU上の関数から呼ぶ関数については, __device__をつける
  - 一番外側のfor文を削除し, 代わりに自動設定されるスレッドIDと紐づけ
    - if(i < Ni){…} をつけないですむように, スレッド数の定数倍のメモリを確保
    - 余分に確保したメモリ領域には, 質量0の粒子を置くなどの工夫を施す
    - (ツリー法などもう一工夫必要な場合もあるが, 細かい話なのでここでは省略)

```
void calc_acc(const type::int_idx Ni, const type::position *const ipos, …) {
#pragma omp parallel for
  for (type::int_idx i = 0U; i < Ni; i++) {
    // 関数の中身は省略
  }}
```

```
__global__ void calc_acc_device(const type::position *const ipos, …) {
  const type::int_idx i = blockIdx.x * blockDim.x + threadIdx.x;
  // 関数の中身は省略
}
```

CUDA C++での実装(お手軽版:2/5)

- GPU化した関数をCPUから起動する
  - スレッド数, 問題サイズから必要なブロック数を設定(マクロ関数が便利)
    - スレッド数: NTHREADS
    - ブロック数: マクロ関数 BLOCKSIZE を使用
  - <<<ブロック数, スレッド数, 動的確保するシェアードメモリ容量, ストリーム>>>
    - 後ろ2つは省略されることが多い(デフォルト設定をそのまま使用)

```
constexpr auto BLOCKSIZE(const type::int_idx num, const type::int_idx thread) {
 return (1U + ((num - 1U) / thread));
 }

static inline void calc_acc(const type::int_idx Ni, const type::position *const ipos,
type::acceleration *__restrict iacc, const type::int_idx Nj, const type::position *const
jpos, const type::flt_pos eps2) {
  calc_acc_device<<<BLOCKSIZE(Ni, NTHREADS), NTHREADS>>>(ipos, iacc, Nj, jpos, eps2);
}
```

CUDA C++での実装(お手軽版:3/5)

- Managed Memory を使用する場合
  - メモリの確保・解放だけ記述すればOK
    - GH200でUnified Memoryを使用する際には, malloc/new などでメモリを確保するだけ
  - (CPU-GPU間のデータ転送は自分では何もしない)

```
// 配列サイズを NTHREADS の整数倍にするための細工
auto size = static_cast<size_t>(num);
if ((num % NTHREADS) != 0U) {
  size += static_cast<size_t>(NTHREADS - (num % NTHREADS));
}

// マネージドメモリの確保
cudaMallocManaged((void **)pos, size * sizeof(type::position));

// マネージドメモリの解放
cudaFree(pos);
```

CUDA C++での実装(お手軽版:4/5)

- Unified/Managed Memoryを使わない場合は, データ転送も記述

```
// GPU上のメモリ確保
cudaMalloc((void **)pos_dev, size * sizeof(type::position));
// CPU上の（pinned）メモリ確保（CPU・GPU間のデータ転送高速化のため）
cudaMallocHost((void **)pos_hst, size * sizeof(type::position));

// 上記で確保したメモリの解放
cudaFree(pos_dev);
cudaFreeHost(pos_hst);

// CPU → GPUのデータ転送
cudaMemcpy(pos_dev, pos_hst, num * sizeof(type::position), cudaMemcpyHostToDevice);

// GPU → CPUのデータ転送
cudaMemcpy(acc_hst, acc_dev, num * sizeof(type::acceleration), cudaMemcpyDeviceToHost);
```

CUDA C++での実装(お手軽版:5/5)

- もし必要があれば, `cudaDeviceSynchronize()`を追加
  - GPU上で関数を起動すると, 関数の終了を待たずにCPUに処理が帰る
  - 複数のCUDAストリームを使った場合には, どこかで同期が必要
  - 性能測定時など, GPU上の関数の状態を把握すべき場合(下の例)
  - Unified Memoryを使用した際に, CPUから読み出したデータが不正だった場合

```
auto timer = util::timer();
cudaDeviceSynchronize();
timer.start();

calc_acc(num, pos_dev, acc_dev, num, pos_dev, eps2);

cudaDeviceSynchronize();
timer.stop();
```

CUDA実装を指示文実装コードから呼び出すには？

- （前述した部分の情報で，完全CUDA実装は可能になっている）
- 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_ptr(pos) {
    w_init = calc_potential_energy(ni, pos, ni, pos, eps2);
  }

#pragma omp target data use_device_ptr(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
- 普通に考えるとそれぞれのコンパイラでコンパイルしておき, 適切なオプションをつけてリンクすれば良い
  - 「適切」な手順を見つけ出すのは地味に面倒な手順だったりする
    - ChatGPT に聞きながら色々試したが, あまりうまくいかず..
  - 最近のCUDAは CUDA C++ であり, C++ベースになっているのも面倒さを助長
    - 昔は CUDA C/C++ と言っていた
- うまくいった手順:
  - 全てのコードを nvc++ でコンパイル・リンク
  - CUDAコードをコンパイルする際には -cuda オプションを付与
  - (今回の講師は指示文ベースのGPU化を一切していない人なのでこの辺は素人です. もっとエレガントなコンパイル方法を知っている/見つけた人はぜひ教えてください)

Contents

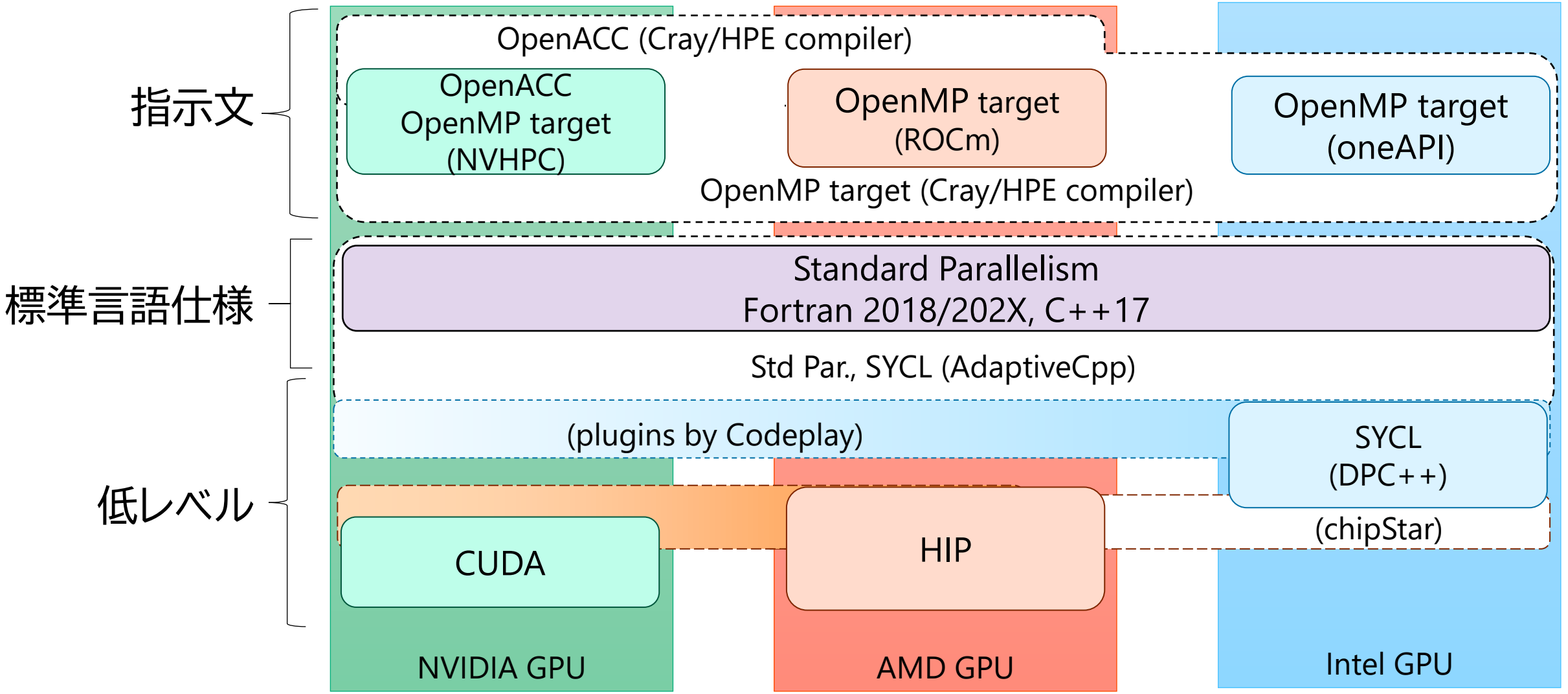
- GPU(Graphics Processing Units)の紹介
- 指示文を用いたGPU化
  - OpenACC, OpenMP, Solomon
- CUDAを用いたGPU化
  - 指示文を用いてGPU化したコードの一部をCUDA化
  - コード全体をCUDAで実装
- ベンダーニュートラルGPUコンピューティングの紹介(HIP, SYCL)
- GPUコードの性能最適化に向けて
  - 気をつけるべき項目や機能の紹介
  - プロファイラの使い方の紹介
- MPIを用いたマルチGPU化

ベンダーニュートラルなGPU実装に向けて

- （今回の講習会では実習しないため，あくまでも話題提供という位置づけ）
- CfCAが提供するGPU資源は全てNVIDIA GPU
  - 国内で使えるGPUスパコンもほとんどはNVIDIA GPUを搭載
  - OpenACC, CUDA が使えれば何も問題はない
- 今後，NVIDIA以外のGPUを搭載する環境が増える可能性はある
  - QST/NIFS のプラズマシミュレータ，筑波大CCSのPost CygnusはAMD MI300Aを搭載するため，OpenACC, CUDA は使えない
- CUDAを用いたGPU化ができていれば，HIPやSYCLへの変換ツールを用いてベンダーニュートラル化もできるので，かけた労力は無駄にならない
- OpenACCについても，OpenMP targetへの変換ツールが存在
  - そもそも Solomon を開発しようと思った動機もこのあたりにある
- その他の解としては，性能可搬ライブラリKokkosを使った実装など
  - C++のラムダ式を使った実装になる

GPU向けのプログラミング環境

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)



SYCL環境の構築

- Intel oneAPI（コンパイラは icpx）
  - Codeplay 社提供のプラグインをインストール（for NVIDIA/AMD GPUs）（https://codeplay.com/solutions/oneapi/plugins/）
- 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

とても簡単なコードであれば，区別がつかないことも

- N体計算の重力計算部分を実装した例（あまり最適化はしていない）

```
__global__ void calc_acc(const int Ni, float4 *ipos, float4 *iacc, const int Nj, float4
 *jpos, const float eps){
  const int i = blockIdx.x * blockDim.x + threadIdx.x;
  float4 pi = ipos[i];  pi.w = eps * eps;
  float4 ai = {0.0F, 0.0F, 0.0F, 0.0F};
  for(int j = 0; j < Nj; j++){
    const float4 pj = jpos[j];
    float4 rji;
    rji.x = pj.x - pi.x;    rji.y = pj.y - pi.y;    rji.z = pj.z - pi.z;
    const float r2 = fmaf(rji.z, rji.z, fmaf(rji.y, rji.y, fmaf(rji.x, rji.x, pi.w)));
    rji.w = 1.0F / sqrtf(r2);
    rji.w *= rji.w * rji.w;
    rji.w *= pj.w;
    ai.x = fmaf(rji.x, rji.w, ai.x);
    ai.y = fmaf(rji.y, rji.w, ai.y);
    ai.z = fmaf(rji.z, rji.w, ai.z);
  }
  iacc[i] = ai;
}
```

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

先程のN体コードに似せて実装したSYCLコード


```
#include <sycl/sycl.hpp>

static constexpr auto BLOCKSIZE = [](const auto num, const auto num_threads) {
  return (1 + ((num - 1) / num_threads));
};
static constexpr auto KERNEL_PARAM_1D = [](const size_t num, const size_t num_threads) {
  const auto grid = sycl::range<1>{BLOCKSIZE(num, num_threads)};
  const auto block = sycl::range<1>{num_threads};
  return (sycl::nd_range<1>{grid * block, block});
};

void calc_acc_kernel(sycl::nd_item<1> nd, const int Ni, float4 *ipos, float4 *iacc, const int Nj, float4 *jpos, const
float eps) {
  const int ii = nd.get_global_id(0);
  float4 pi = ipos[ii];  pi.w = eps * eps;
  float4 ai = {0.0F, 0.0F, 0.0F, 0.0F};
  for (int jj = 0; jj < Nj; jj++) {
    const float4 pj = jpos[jj];
    float4 rji;    rji.x = pj.x - pi.x;    rji.y = pj.y - pi.y;    rji.z = pj.z - pi.z;
    const float r2 = sycl::fma(rji.z, rji.z, sycl::fma(rji.y, rji.y, sycl::fma(rji.x, rji.x, pi.w)));
    rji.w = 1.0F / sycl::sqrt(r2);
    rji.w *= rji.w * rji.w;    rji.w *= pj.w;
    ai.x = sycl::fma(rji.x, rji.w, ai.x);
    ai.y = sycl::fma(rji.y, rji.w, ai.y);
    ai.z = sycl::fma(rji.z, rji.w, ai.z);
  }
  iacc[ii] = ai;
}

void calc_acc(sycl::queue &queue, const int Ni, float4 *ipos, float4 *iacc, const int Nj, float4 *jpos, const float eps) {
  queue.parallel_for(KERNEL_PARAM_1D(Ni, NTHREADS), [=](sycl::nd_item<1> item) {
    calc_acc_kernel(item, Ni, ipos, iacc, Nj, jpos, eps);
  });
}
```

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 (OFP-II) (1/2)

JCAHPC

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

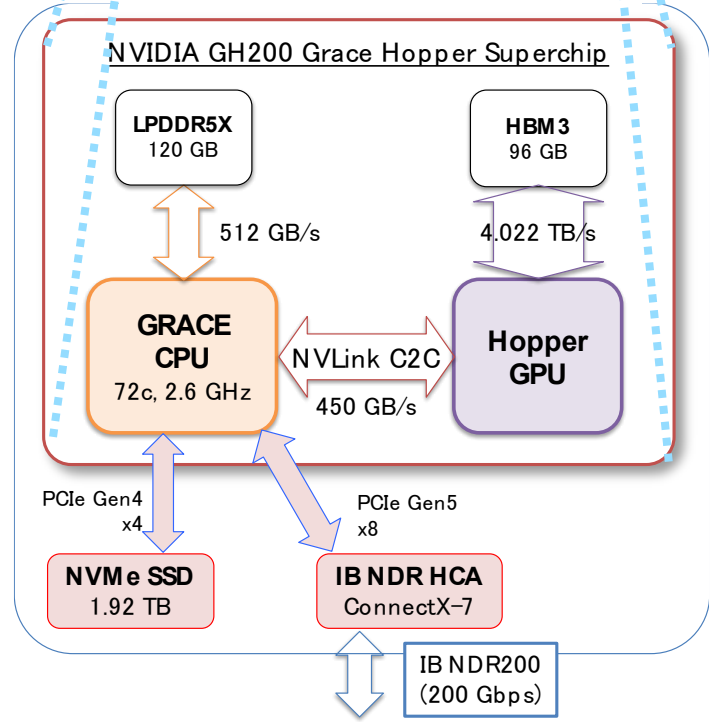
- **Miyabi-G: CPU+GPU: NVIDIA GH200**
  - Node: NVIDIA GH200 Grace-Hopper Superchip
    - Grace: 72c, 3.456 TF, 120 GB, 512 GB/sec (LPDDR5X)
    - H100: 66.9 TF DP-Tensor Core, 96 GB, 4,022 GB/sec (HBM3)
      - Cache Coherent between CPU-GPU
    - NVMe SSD for each GPU: 1.9TB, 8.0GB/sec, GPUDirect Storage
  - **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

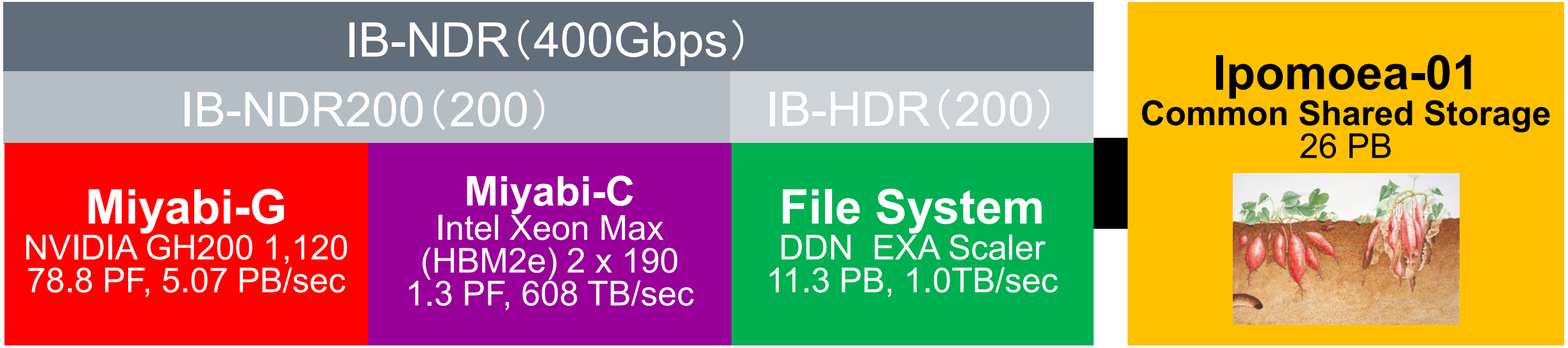
NVIDIA

intel



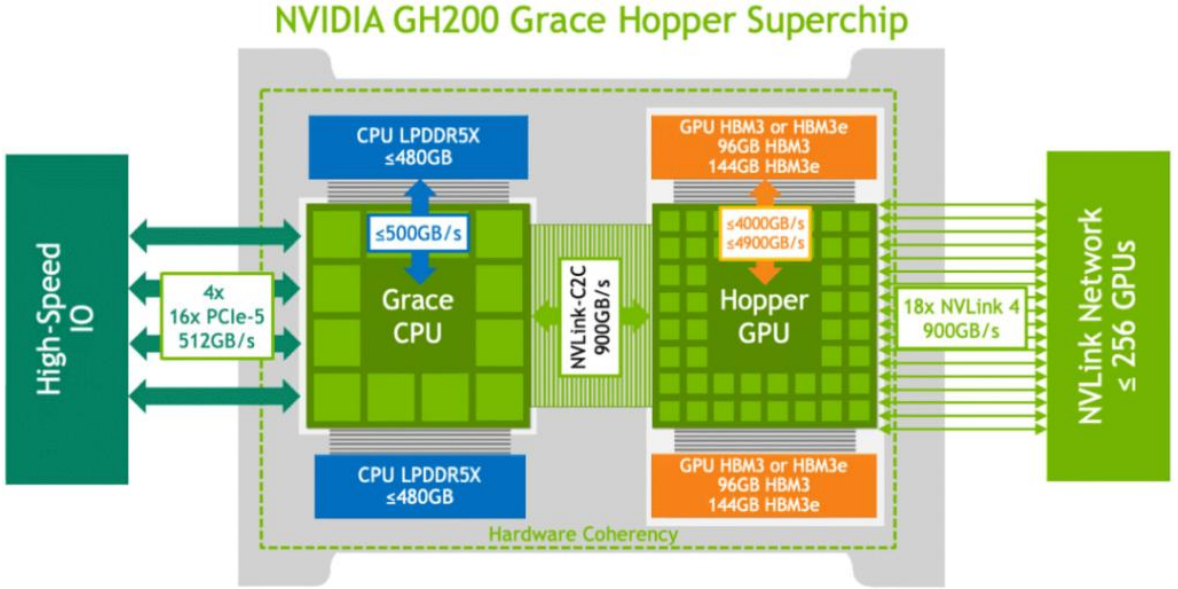
Miyabi (OFP-II) (2/2)

- 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

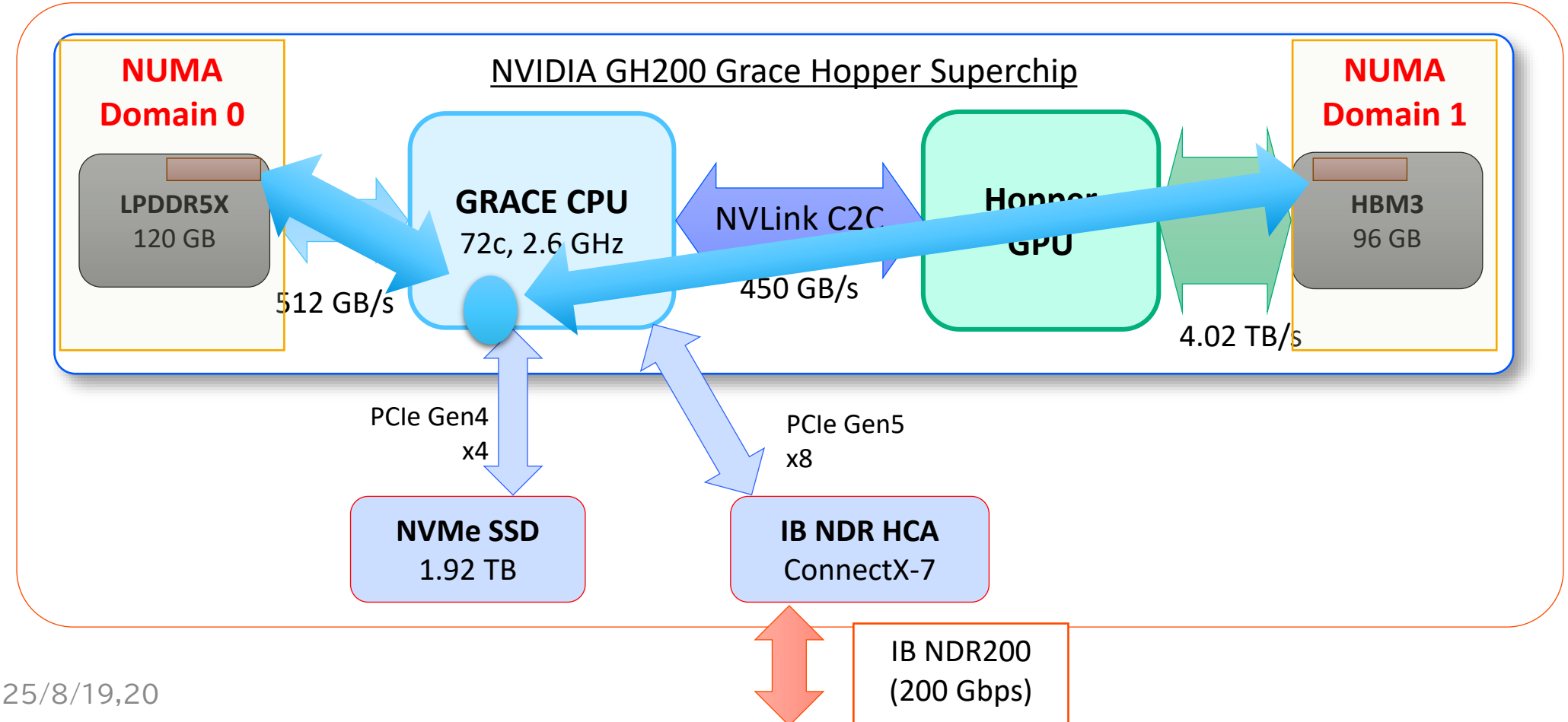
- Grace-Hopper相互のメモリ空間を直接参照可能, NUMA的な扱い
- CPU-GPU間: コヒーレントインタフェース(NVLink-C2C)
  - PCIe Gen 5の7倍以上の帯域(450GB/sec/dir)
  - CPU･GPUの効率的使い分けも可能
    - 従来はデータ転送がボトルネック
    - プログラミングも用意
    - AMD MI300Aも同じ方向性



- 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

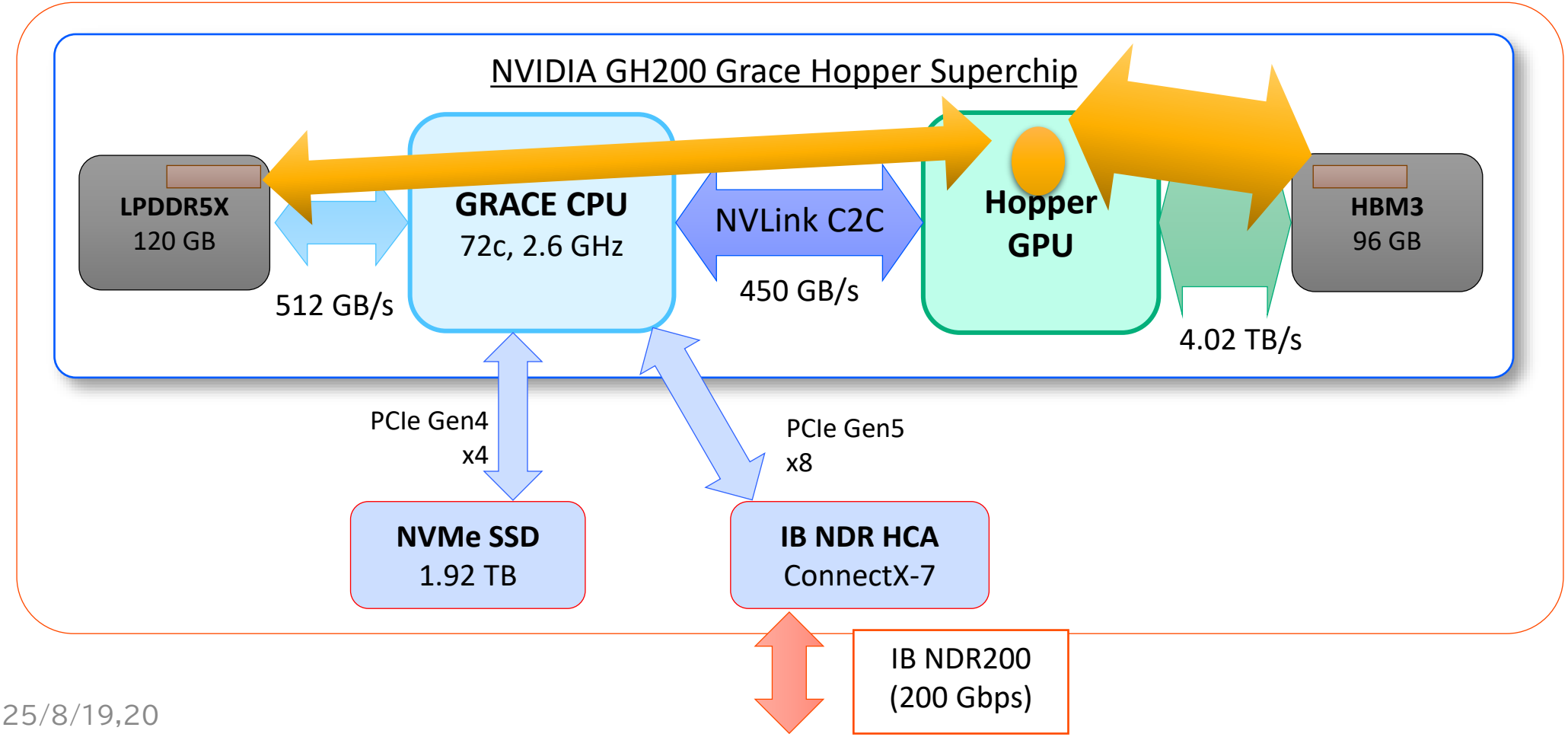
Graceからのメモリビュー

- 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

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8/29 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

実習: 指示文+CUDAでのGPU化をやってみる

- 指示文を用いてGPU化したコードをさらにいじる
- 完全CUDA化に手を出してもOK

講習会プログラム

- 8/19(火)
  - 9:00—10:00　[講義] GPUプログラミングの基礎(指示文編)
  - 10:10—12:00　[実習]
  - 12:00—13:30　昼休み
  - 13:30—14:30　[実習]
  - 14:40—15:40　[講義] GPUプログラミングの基礎(CUDA編)
  - 15:50—17:00　[実習]
- 8/20(水)
  - 9:00—10:00　[講義] GPU化したN体コードの高速化手法
  - 10:10—12:00　[実習]
  - 12:00—13:30　昼休み
  - 13:30—14:30　[講義] MPIを用いたマルチGPU化
  - 14:40—16:00　[実習]
  - 16:00—17:00　[解説] 参照実装の紹介

Contents

- GPU(Graphics Processing Units)の紹介
- 指示文を用いたGPU化
  - OpenACC, OpenMP, Solomon
- CUDAを用いたGPU化
  - 指示文を用いてGPU化したコードの一部をCUDA化
  - コード全体をCUDAで実装
- ベンダーニュートラルGPUコンピューティングの紹介(HIP, SYCL)
- GPUコードの性能最適化に向けて
  - 気をつけるべき項目や機能の紹介
  - プロファイラの使い方の紹介
- MPIを用いたマルチGPU化

GPU版N体コードの最適化

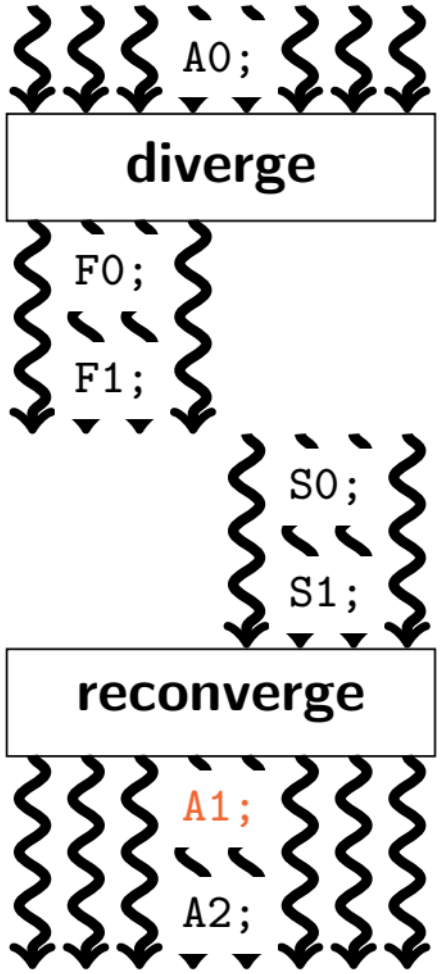
- 最適なスレッド数の選択
  - スレッド数を変えるとどの程度性能が変わるか実験してみてください
- できるだけ条件分岐を削除(CPUコードの場合も同じ)
- FMA(fused multiply-add)命令の発行率を増やす
- 高速な近似逆数平方根命令の使用
  - N体計算の場合は, `rsqrtf()`による高速化が最重要(NVIDIA GPU)
  - 指示文の場合には, コンパイルオプションに `-Mfprelaxed=rsqrt` などを追加)
  - AMD GPUの場合には, `__frsqrtf_rn()`がベスト
- シェアードメモリの活用
  - A100でL2キャッシュの容量が増えた(6 MB➔40 MB)こともあり効果減(=最適化なしの状態でもかなり速くなった)
  - 追加でループアンローリングも適用するとさらに高速化
- 命令レベルの並列性の導入
- H100以降では, `memcpy_async()`の活用も効果あり

ワープ分裂(warp divergence)

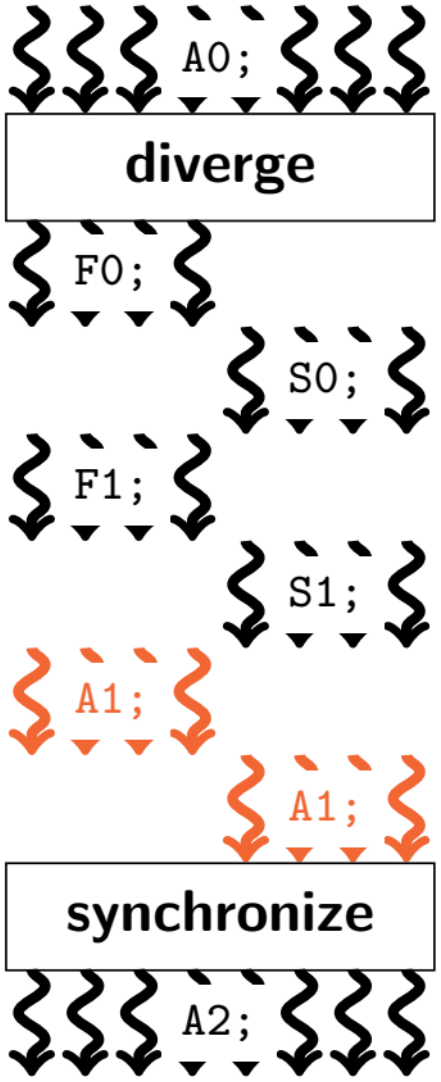
Pseudocode

```
A0;
if( threadIdx.x < 4 ){
  F0;
  F1;
} else {
  S0;
  S1;
}
A1;
#if __CUDA_ARCH__ >= 700
__syncwarp();
#endif
A2;
```

Pascal or earlier



Volta or later



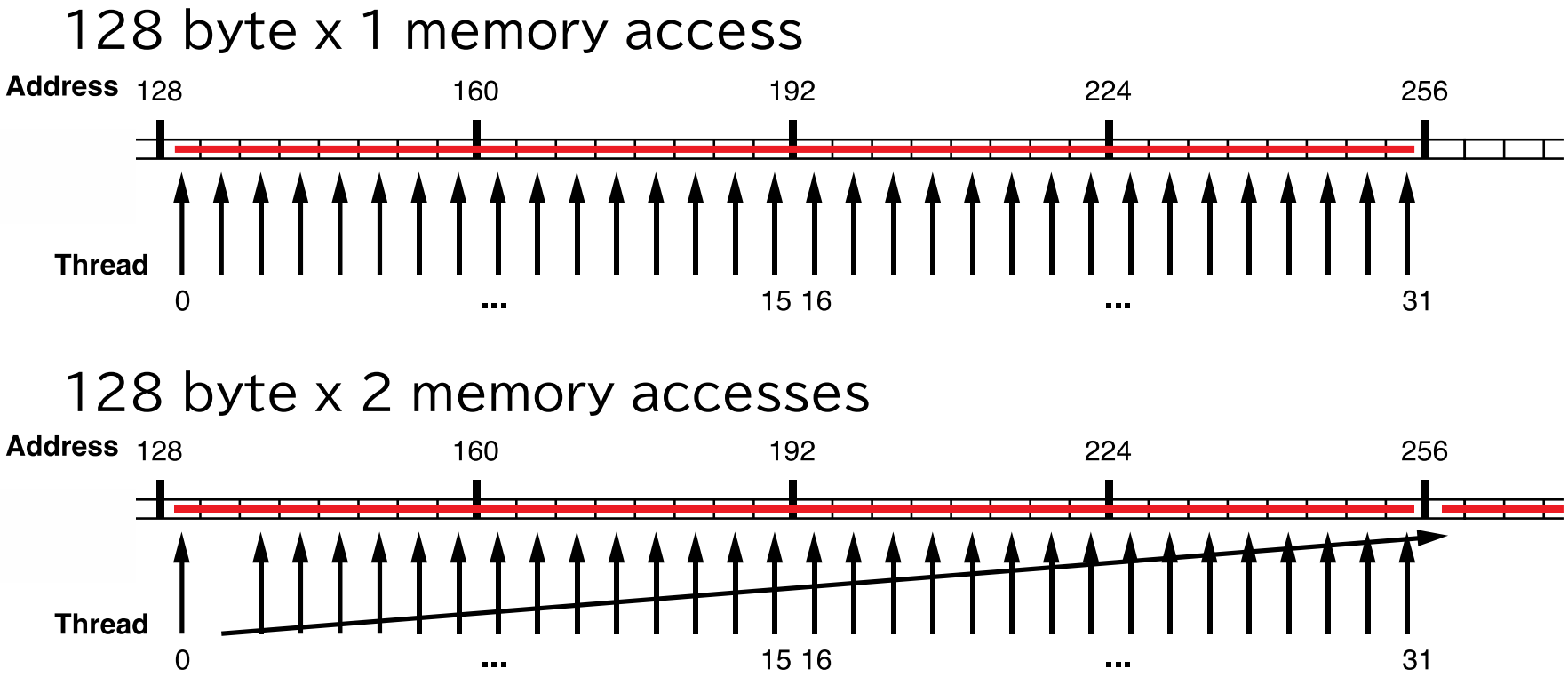
シェアードメモリを使った実装の例

- 静的に確保するには, GPU関数内で __shared__をつけて宣言しておけばOK
- ブロック内の全スレッドから読み書き可能なので, シェアードメモリ上のデータを変更する際には block.sync() などをつけて制御する(Cooperative Groupsを使用)
  - 以前は __syncthreads() を使っていた(今も使える)
- 実行性能はループアンローリングの段数にも依存

```
#include <cooperative_groups.h>
__global__ void calc_acc_device(…) {
  // 初期化部分省略
  auto block =
cooperative_groups::this_thread_block();
  __shared__ type::position jpos_shmem[NTHREADS];

  for (int jh = 0; jh < Nj; jh += NTHREADS) {
    // load j-particle
    const auto pj_tmp = jpos[jh + threadIdx.x];
    block.sync();
    jpos_shmem[threadIdx.x] = pj_tmp;
    block.sync();

    PRAGMA_UNROLL
    for (int j = 0; j < NTHREADS; j++) {
      const auto pj = jpos_shmem[j];
      // 以下省略
    }
  }
  iacc[i] = ai;
}
```

コアレスドアクセス(Coalesced access)

- ワープ(連続する32スレッドの集合)内のスレッドが, 連続したメモリアドレスにアクセスする際には, 実際の処理はまとめて実行される
  - メモリアクセスは128バイト単位で実行される
  - スレッドの順番とメモリアドレスの順番は一致していなくてもOK
  - メモリアライメントには気をつけておく



メモリアクセス命令の発行数削減

- 64ビットアクセス命令, 128ビットアクセス命令があり, こうした命令を発行できれば総メモリアクセス命令数が削減できて高速化につながる場合も
- 例えば単精度(float: 32ビット=4バイト)でN体コードを実装した場合:
  - 普通にj-粒子を取ってくると, x, y, z, m の4要素を4回のロード命令で取得
  - float4というデータ型(x, y, z, wをメンバに持つ構造体)を使うと, 1回のロード命令(128ビットアクセス命令が発行される)で取得できる
    - 粒子質量を w に入れておくと良い
- 同様のデータ型は float2, double2, double4 など
  - float3 などは(構造体の規約的に)メモリ容量を無駄遣いするのでおすすめしない(実装を簡単に済ませる目的ならばOK, 性能向上を意図した実装には適さない)
  - CUDA 13 からは double4, long4, ulong4, longlong4, ulonglong4 などが非推奨になった
    - 32バイトアライメントのdouble4_32a, 16バイトアライメントのdouble4_16a などが新設

より高度な内容

- 以降はGPUを使い始める(た)段階の人向けにはハードルが高めの情報なので, 演習などには含めず情報提供のみとしておきます

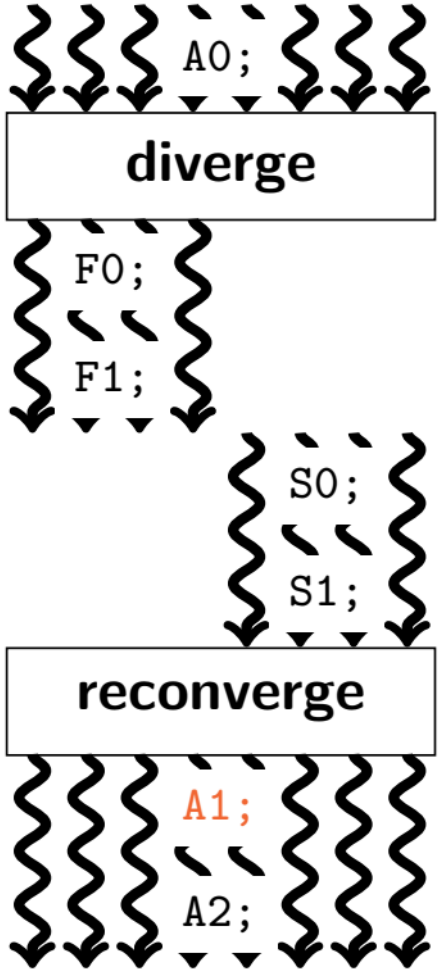
- Volta世代以降のGPUでのワープの挙動
- 最近のGPU上で(CUDAから)使える命令・機能

Independent thread scheduling

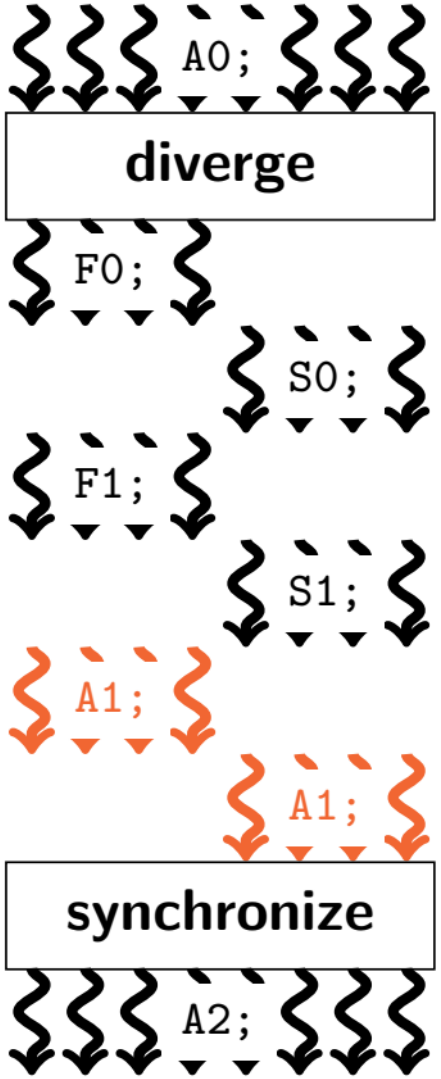
Pseudocode

```
A0;
if( threadIdx.x < 4 ){
  F0;
  F1;
} else {
  S0;
  S1;
}
A1;
#if __CUDA_ARCH__ >= 700
__syncwarp();
#endif
A2;
```

Pascal or earlier



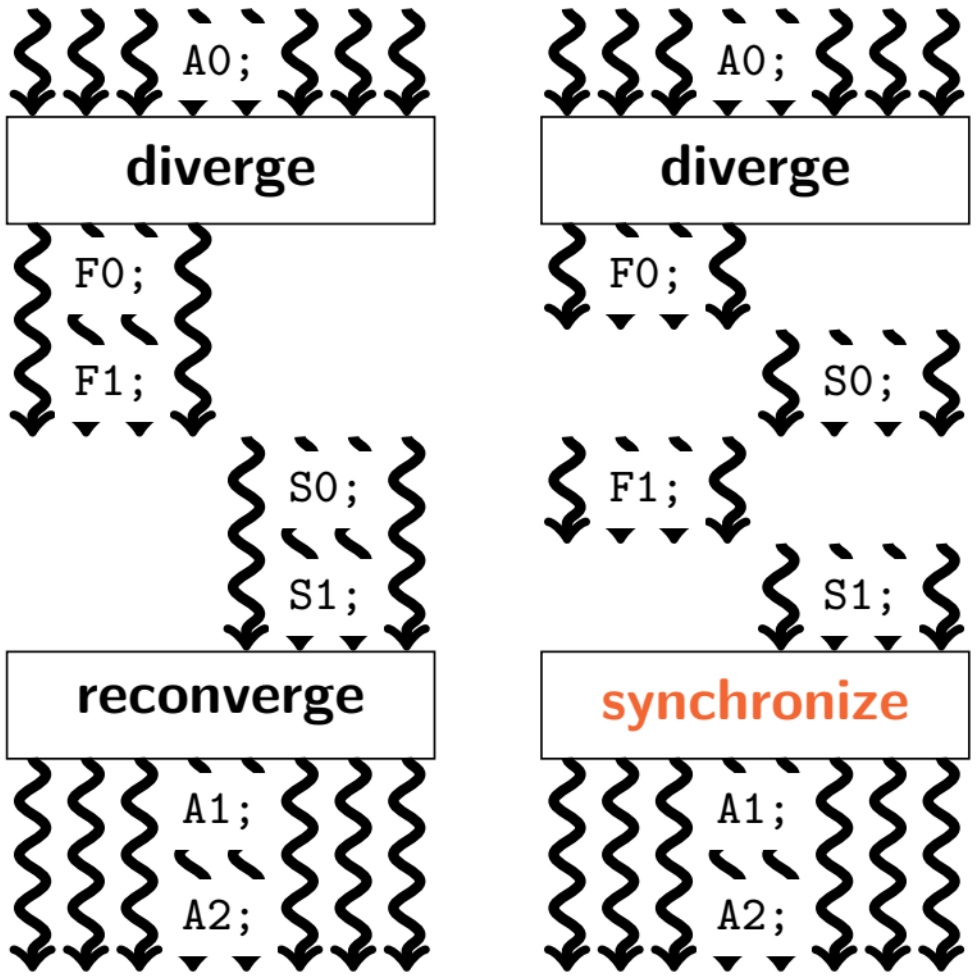
Volta or later



Independent thread scheduling

Pseudocode

```
A0;
if( threadIdx.x < 4 ){
  F0;
  F1;
} else {
  S0;
  S1;
}
#if __CUDA_ARCH__ >= 700
__syncwarp();
#endif
A1;
A2;
```



暗黙の同期の挙動変更に対する処方箋

- (暗黙の同期を使わないように)コードを書き換える
  - NVIDIA的にはこちらが推奨方針
  1. ワープシャッフル命令などに _sync を追加(マスクは慎重に！)
  2. 暗黙の同期点, `if()`文直後などに `__syncwarp()` を追加
  3. ブロック同期用の `__syncthreads()` の挿入位置を再検討
- 暗黙の同期を無理矢理使う(勝手にPascalモードと呼ぶ)
  - 実はこちらの方が速かったりする(今後どうなるかは不明)
  - Ampere: コンパイル時に `arch=compute_60,code=sm_80` を指定
  - Hopper: `arch=compute_60,code=sm_90` を指定
  - Ampereから導入された warp-wide reduction は, Pascalモードでは使えない模様(コンパイルが通らなかった)
  - CUDA 13.0でPascalサポートが消えたので, 今後この裏技は封じられることに

Warp shuffle命令

- 同一ワープ(32スレッドの塊)内にある他スレッドのレジスタの値を(シェアードメモリなどを介さずに)取得できる

```
T __shfl_sync(unsigned mask, T var, int srcLane, int width=warpSize);
T __shfl_up_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
T __shfl_down_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
T __shfl_xor_sync(unsigned mask, T var, int laneMask, int width=warpSize);
```

- maskの値は基本的には 0xffffffff(=32スレッド全員)でOK
  - 複雑な実装をしている場合は, これではNGの場合もある
  - 同時に入ってくるスレッド数にデータ依存性があって予測できない場合には, __activemask()命令を使って動的に制御をかける

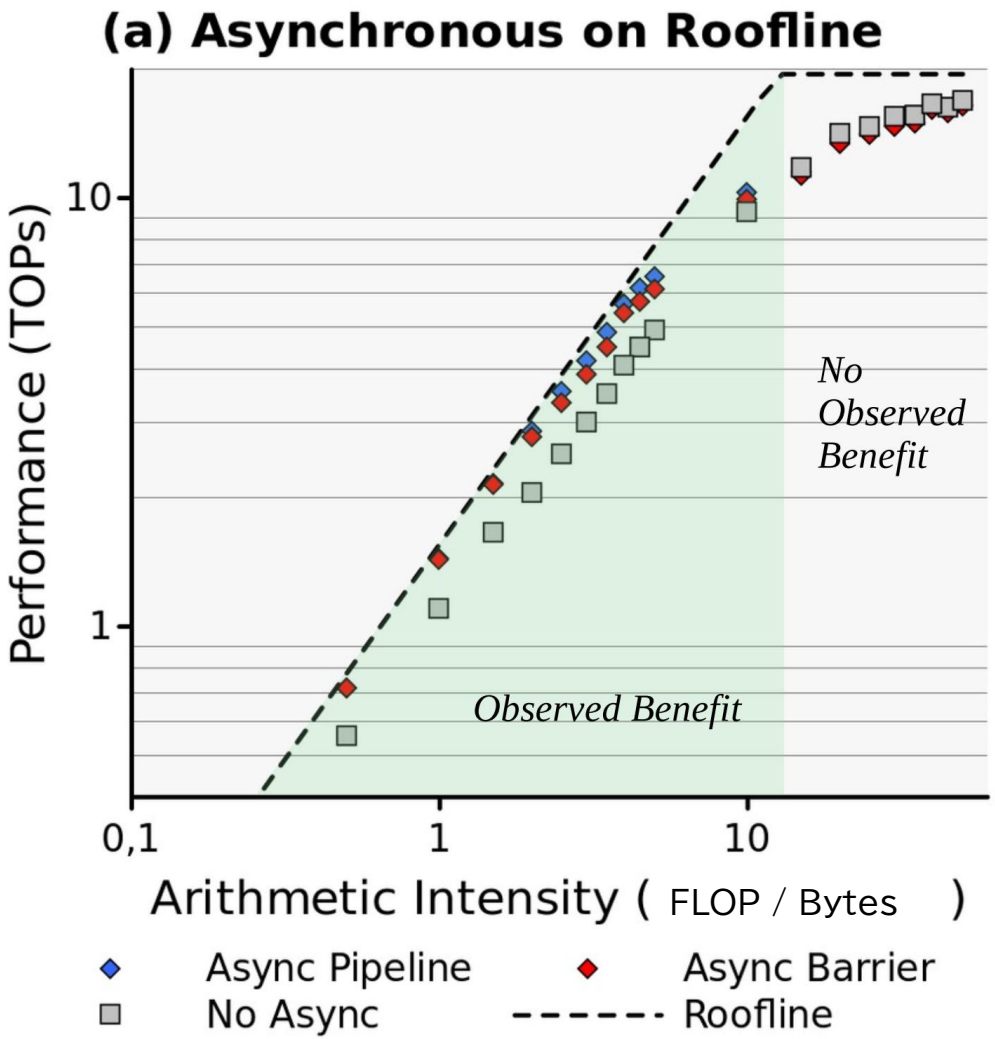
A100/CUDA 11から導入された機能

- テンソルコア関連の機能
  - Sparsity: 小行列の中に0が入っていたらその分をスキップして高速化
  - TF32: 仮数部10ビット, 指数部8ビットのデータ型(実態はFP19)
- Asynchronous copy/barrier
  - グローバルメモリからシェアードメモリに直接(レジスタを介さずに)データを置ける
- Warp-wide reduction
  - ハードウェア的な高速化が効く
- L2 cache residency control
  - (1つの連続な)データ領域をL2キャッシュに固定しておける

Asynchronous copy (memcpy_async)

- グローバルメモリからシェアードメモリへの direct memcpy
- A100では, hardware accelerated
- Svedin et al. (2021):
  - Pipeline API は Barrier API よりも高速
  - メモリ律速な問題では性能向上
  - 演算律速な問題では性能低下
    - N体問題は残念ながらこちらの場合
- H100ではハード的な改良(TMA)が入るので, 演算律速な場合でも性能低下しない
  - N体問題でも memcpy_async を使った方が速くなることを確認済み(H100, GH200)

Svedin et al. 2021



Warp-wide reduction

- Throughputは16(INT32演算が64なので, 4演算相当)
  - Warp shuffleが32(2演算相当)なので, 5段(32スレッド)必要な旧実装に比べて(演算コストを無視しても)圧倒的に速い
- `__reduce_*_sync(unsigned mask, T value)`
  - T: unsigned/intに対してはadd, min, max
    - 小細工すればfloatに対してmin/maxを返させることは可能
  - T: unsignedに対してはand, or, xor も可能
  - Supported by devices of compute capability 8.x or higher
    - Pascalモード(arch=compute_60,code=sm_80)の場合には, コンパイルが通らなかった(compute_80の指定が必要)
    - Pascalモード使用による高速化か, Ampereモード+warp-wide reductionによる高速化か, の競争(tree構築はPascalモードの勝ち)

L2 cache residency control

- Best Practice Guideからの抜粋
  ”A portion of the L2 cache can be set aside for persistent accesses to a data region in global memory”
- 1/16(= 2.5 MB)刻みで調整可能
  - White paper中の記述
- CUDAストリームごとに1つの配列中の連続領域を指定
  - 1/16刻みで調整可能であれば, 最大16個の領域を指定できても良さそうだが, 残念ながらそういうAPIにはなっていない

Contents

- GPU(Graphics Processing Units)の紹介
- 指示文を用いたGPU化
  - OpneACC, OpenMP, Solomon
- CUDAを用いたGPU化
  - 指示文を用いてGPU化したコードの一部をCUDA化
  - コード全体をCUDAで実装
- ベンダーニュートラルGPUコンピューティングの紹介(HIP, SYCL)
- GPUコードの性能最適化に向けて
  - 気をつけるべき項目や機能の紹介
  - プロファイラの使い方の紹介
- MPIを用いたマルチGPU化

プロファイラなどの使い方

- `$ nsys profile --stats=true ./a.out`
  - `--stats=true` をつけておくと, 結果の概要も出力されるので便利
  - report?.nsys-rep というファイルが生成されるので, (手元で or Xを飛ばして) nsys-ui で開く
    - どちらかというと, 手元の環境にNVIDIA Nsight Systemsをインストールする方が楽
      - Windows, macOS, Linux のどれでもOK
      - https://developer.nvidia.com/nsight-systems
  - より詳細な使い方については, https://www.cc.u-tokyo.ac.jp/events/lectures/239/20250131_openhackathon_day0.pdf を参照してください(前半はGH200の内容, 後半がプロファイラ関連)
- `$ cuobjdump -sass ./a.out`
  - ディスアセンブルして, 発行されている命令を確認したくなった人向け
  - 時折 PTX の確認方法も紹介されるが, おすすめできません
    (PTX から実行ファイル生成までにもう一段コンパイルが入るため,
    実際に発行されている命令とは違っている場合があります)

nsys を使う際のトレース対象

- $ nsys profile --help すると下記のトレース対象がリストされる

```
-t, --trace=
      Possible values are 'cuda', 'nvtx', 'cublas', 'cublas-
verbose', 'cusolver’, 'cusolver-verbose', 'cusparse',
'cusparse-verbose', 'mpi', 'oshmem', 'ucx’, 'osrt', 'cudnn',
'opengl', 'opengl-annotations', 'openacc', 'openmp’,
'nvvideo', 'vulkan', 'vulkan-annotations', 'python-gil',
'syscall' or 'none’.
```

- トレース対象としたいものを nsys profile 実行時のオプションに渡す
  - 今回の講習会の範囲では, --trace=osrt,cuda,openacc,openmp,nvtx,mpi,oshmem,ucx とすれば十分（NVTXには触れていないので, 若干過剰気味）

性能プロファイルを取得する際の注意点

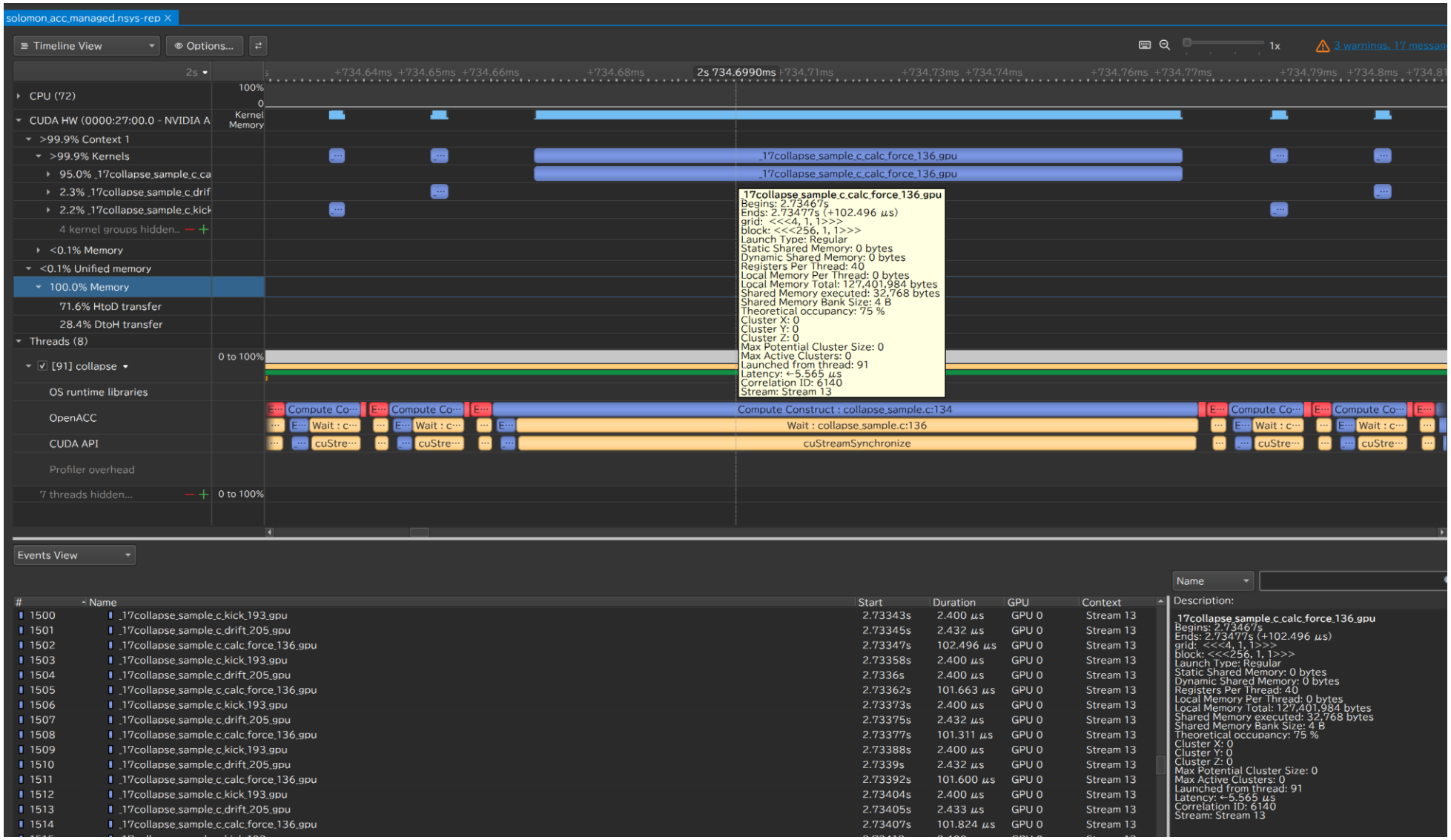
- 問題サイズは実際の計算に揃えてあげる
  - 実使用時とかけ離れた状況で測定しても意味がない
  - 今回の場合, GPUを埋められる程度には粒子数を増やしておく
  - 下記のスクリプト, コマンドであれば実行パラメータファイルだけ入れ替え可能
    `$ sbatch --export=PARAM=bench.ini sh/cfca/nsys_nvhpc.sh`

```
#!/usr/bin/env bash
#SBATCH --partition=gpuws
#SBATCH --gres=gpu:1
#SBATCH --time=00:05:00

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

module purge
module load nvhpc/25.7

cd ${SLURM_SUBMIT_DIR}

nsys profile --stats=true --trace=osrt,cuda,openacc,openmp,nvtx,mpi,oshmem,ucx ¥
 ./collapse ${PARAM}
```

- 実行時間が伸びるので, 総ステップ数に上限を設定するなど配慮する
  - あるいは, `--duration=30` などを付与して測定時間を設定(この場合は30秒)

Nsight Systems の出力例

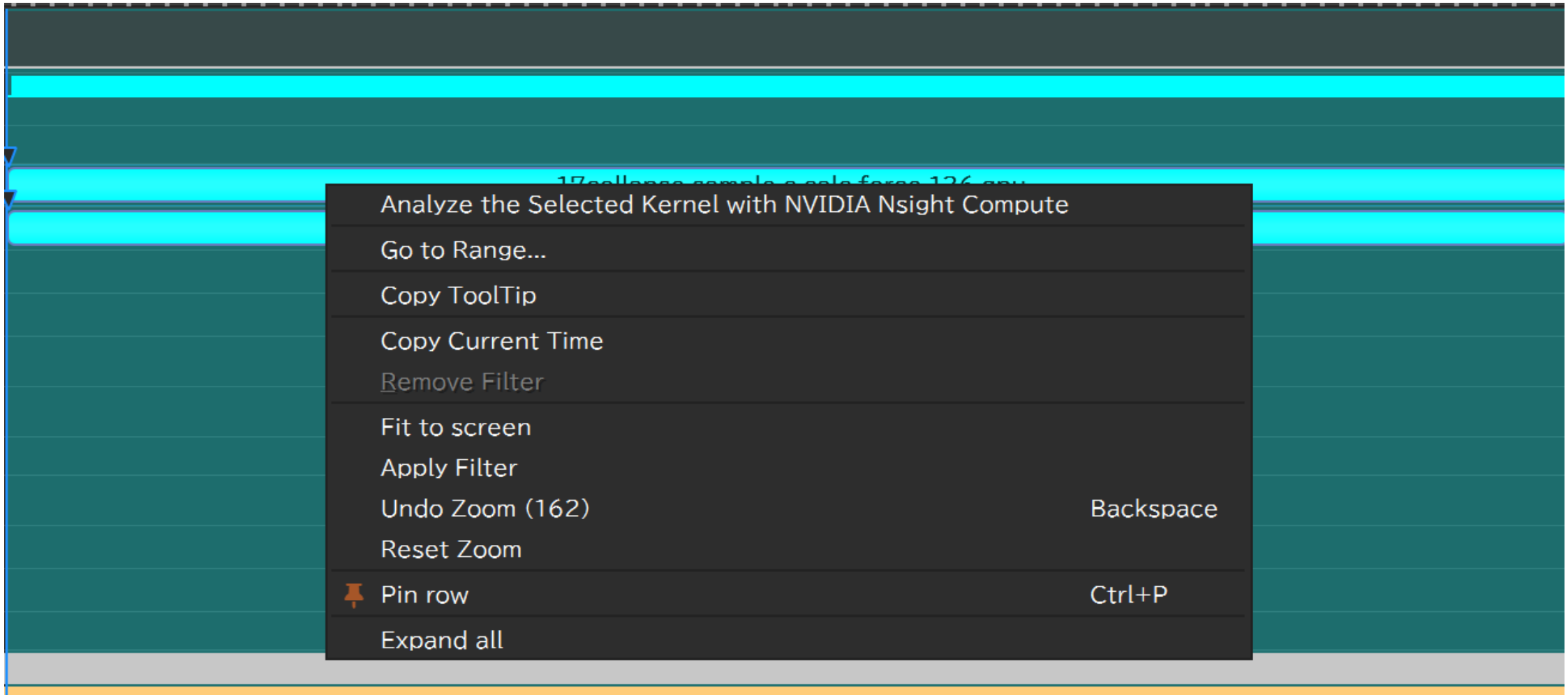
- 注目する関数をダブルクリック, マウスオーバーなどすると主な情報が見える



#	Name	Start	Duration	GPU	Context
1500	_17collapse_sample_c_kick_193_gpu	2.73343s	2.400 µs	GPU 0	Stream 13
1501	_17collapse_sample_c_drift_205_gpu	2.73345s	2.432 µs	GPU 0	Stream 13
1502	_17collapse_sample_c_calc_force_136_gpu	2.73347s	102.496 µs	GPU 0	Stream 13
1503	_17collapse_sample_c_kick_193_gpu	2.73358s	2.400 µs	GPU 0	Stream 13
1504	_17collapse_sample_c_drift_205_gpu	2.7336s	2.400 µs	GPU 0	Stream 13
1505	_17collapse_sample_c_calc_force_136_gpu	2.73362s	101.663 µs	GPU 0	Stream 13
1506	_17collapse_sample_c_kick_193_gpu	2.73373s	2.400 µs	GPU 0	Stream 13
1507	_17collapse_sample_c_drift_205_gpu	2.73375s	2.432 µs	GPU 0	Stream 13
1508	_17collapse_sample_c_calc_force_136_gpu	2.73377s	101.311 µs	GPU 0	Stream 13
1509	_17collapse_sample_c_kick_193_gpu	2.73388s	2.400 µs	GPU 0	Stream 13
1510	_17collapse_sample_c_drift_205_gpu	2.7339s	2.432 µs	GPU 0	Stream 13
1511	_17collapse_sample_c_calc_force_136_gpu	2.73392s	101.600 µs	GPU 0	Stream 13
1512	_17collapse_sample_c_kick_193_gpu	2.73404s	2.400 µs	GPU 0	Stream 13
1513	_17collapse_sample_c_drift_205_gpu	2.73405s	2.433 µs	GPU 0	Stream 13
1514	_17collapse_sample_c_calc_force_136_gpu	2.73407s	101.824 µs	GPU 0	Stream 13

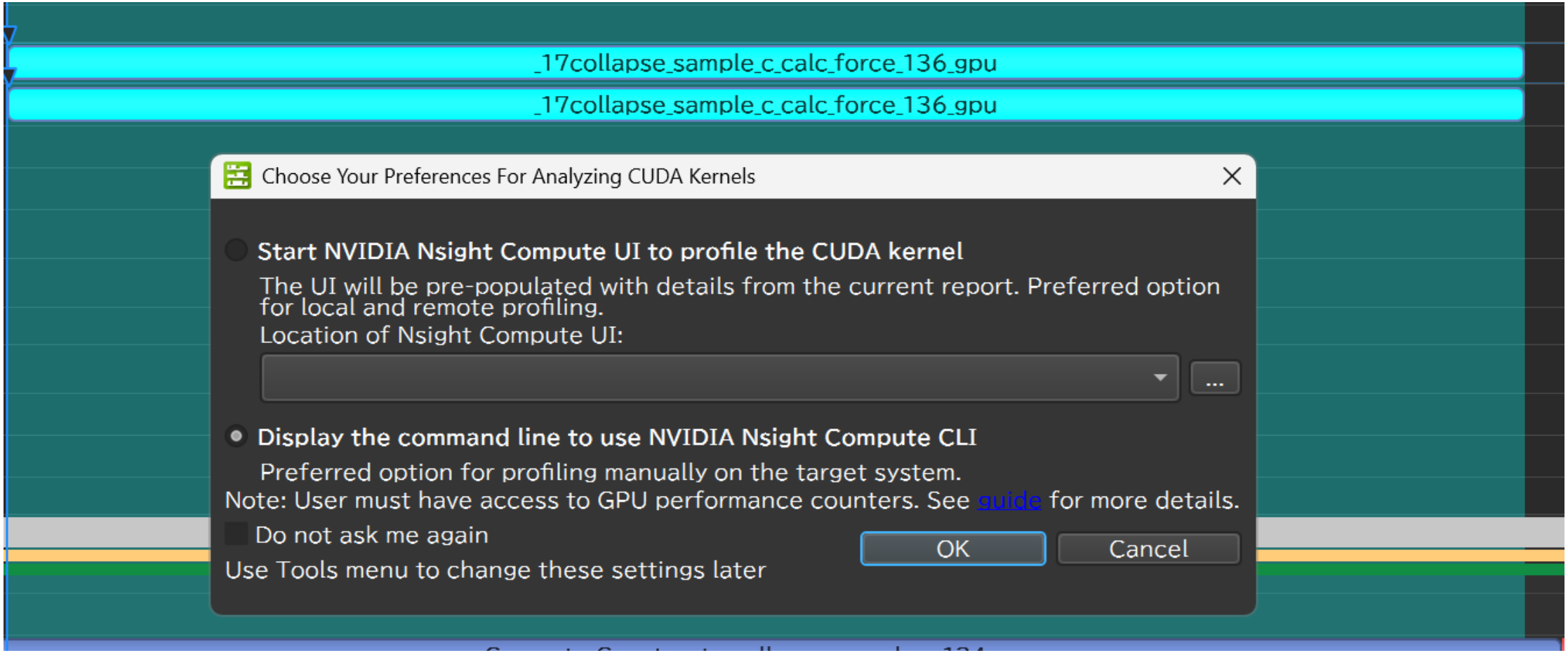
詳細な性能解析をする方法(1/4)

- 注目する関数を右クリックすると下記のメニューが表示されるので,”Analyze the … Compute”をクリック



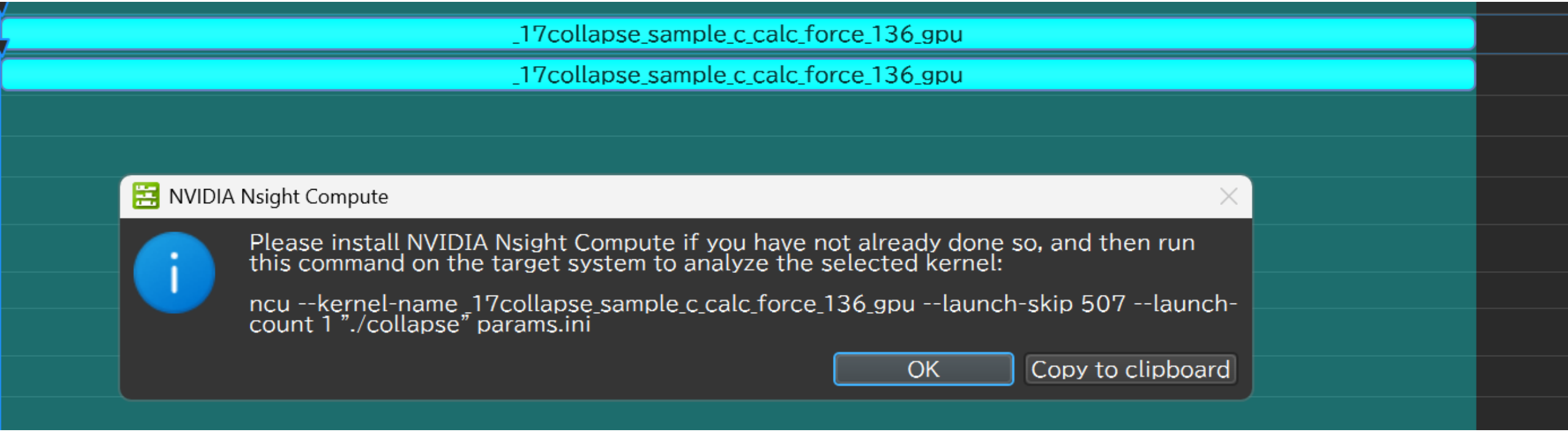
詳細な性能解析をする方法(2/4)

- ポップアップが出てくるので, ”Display the command line…”の方にチェックを入れて OK をクリック
  - 手元のGPU環境でGUI操作できる状態であれば, 上の”Start …”でもOK



詳細な性能解析をする方法(3/4)

- 実行すべきコマンドが表示されるので, ジョブスクリプトにこれを記入すれば Nsight Compute を使った詳細解析が実行できる



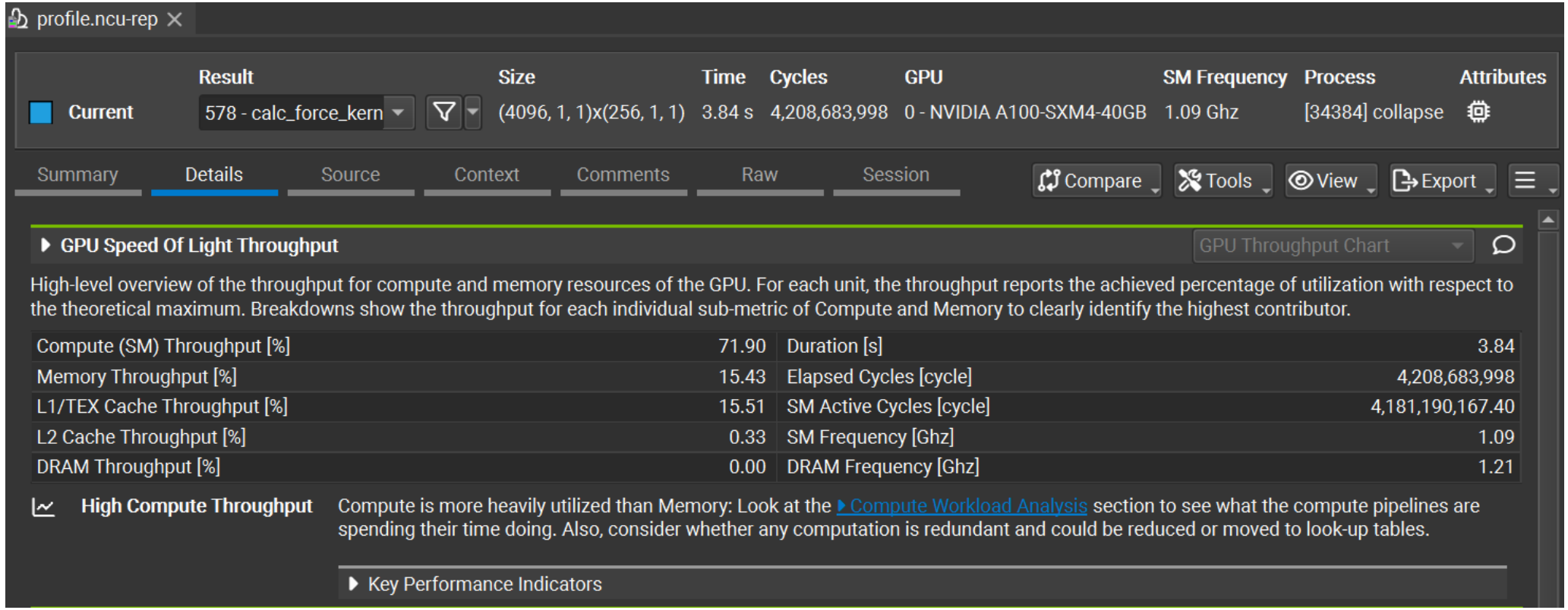
詳細な性能解析をする方法(4/4)

- 先程表示されたコマンドをそのまま実行するとテキストで結果が出力される
- Nsight Systems 同様にGUIで表示したい場合には, 下記も追加
  `--target-processes all --export profile --force-overwrite`
  - `--export profile` の部分が重要
  - `--force-overwrite` をつけているので出力は上書きされるという点に注意
- 後は Nsight Compute のGUIツールを用いて開けば良い

- Compute (SM) Throughput, Memory Throughput あたりが十分高い数字になっていればOK
  - ボトルネックになるメトリックに集中
  - (過去の)記憶では80%以上になっていれば別関数の最適化を推奨される
  - Direct N-body のコードを最適化すると, 90%台には到達できます

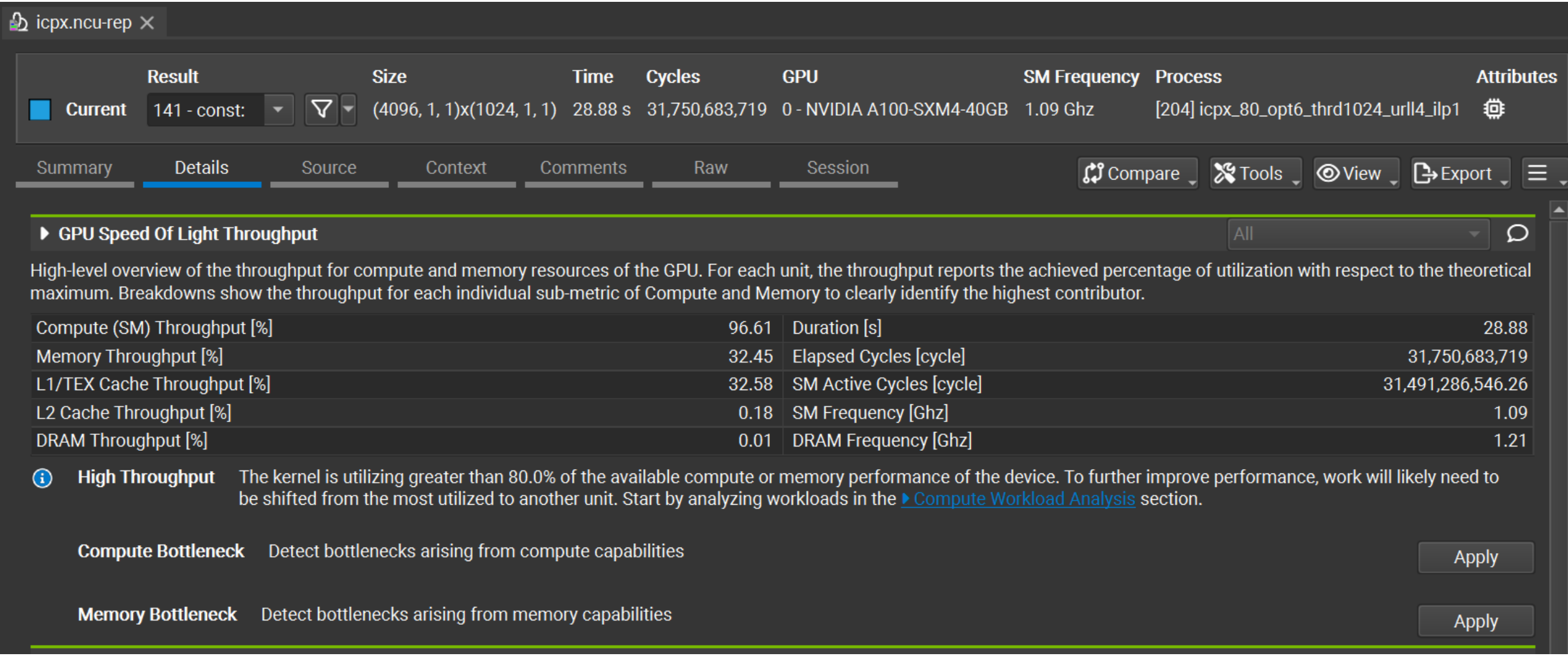
ncu で取得したデータをGUI表示する方法

- *.ncu-rep というファイルが生成されるので,（手元で or Xを飛ばして）ncu-ui で開く
  - 手元の環境にNVIDIA Nsight Computeをインストールする方が楽
    - Windows, macOS, Linux のどれでもOK
    - https://developer.nvidia.com/nsight-compute



(性能最適化済み)N体コードのプロファイル結果

- 下図は(なぜかCUDAより速かった)SYCL実装の測定結果



チューニング・性能評価環境

GPU	NVIDIA H100 SXM 80GB	NVIDIA GH200 480GB	AMD Instinct MI210	Intel Data Center GPU Max 1100
FP32 peak	66.9 TFlop/s	66.9 TFlop/s	22.6 TFlop/s	22.2 TFlop/s
# of units	132 SMs	132 SMs	104 CUs	448 EUs
FP32 parallelism	16896	16896	6656	7168
Clock frequency	1980 MHz	1980 MHz	1700 MHz	1550 MHz
TDP	700 W	(全体で)1000 W	300 W	300 W
Host CPU	Intel Xeon Platinum 8468	NVIDIA Grace	AMD EPYC 7713	Intel Xeon Platinum 8468
	48 cores × 2 sockets	72 cores	64 cores × 2 sockets	48 cores × 2 sockets
	2.1 GHz	3.0 GHz	2.0 GHz	2.1 GHz
	CUDA 12.3	CUDA 12.4	ROCm 6.0.2, 5.4.3	
Intel oneAPI	2024.1.0		2024.1.0	2024.1.0
LLVM	18.1.7	18.1.7	18.1.7	
AdaptiveCpp	24.02.0	24.02.0	24.02.0	

最適な逆数平方根演算命令の特定

- N=4Mにおいて，1秒あたりに計算できた相互作用ペア数を比較

GPU	Compiler	1.0F/std::sqrt()	rsqrtf()	__frsqrt_rn()	sycl::rsqrt()	sycl::native::rsqrt()
NVIDIA H100	nvcc	8.06×10^{11}	1.51×10^{12}	7.68×10^{11}		
NVIDIA H100	icpx	1.15×10^{12}			1.77×10^{12}	1.77×10^{12}
NVIDIA H100	acpp	7.90×10^{11}			8.08×10^{11}	8.08×10^{11}
NVIDIA GH200	nvcc	8.15×10^{11}	1.51×10^{12}	7.68×10^{11}		
NVIDIA GH200	acpp	8.01×10^{11}			8.08×10^{11}	8.08×10^{11}
AMD MI210	hipcc	2.36×10^{11}	5.28×10^{11}	7.05×10^{11}		
AMD MI210	icpx	7.13×10^{11}			7.13×10^{11}	7.13×10^{11}
AMD MI210	acpp	2.35×10^{11}			7.06×10^{11}	7.06×10^{11}
Intel 1100	icpx	5.52×10^{11}			5.52×10^{11}	5.52×10^{11}

NVIDIA製GPU向けの実装

- CUDA, SYCL 実装を測定
  - HIP版はCUDA版と同性能が得られると分かっているので, 今回は割愛
    （管理者権限なしで(= rpmパッケージを使わずに)HIP環境を構築するのは面倒）
- 逆数平方根, ブロックあたりのスレッド数, ループアンローリング段数, Instruction Level Parallelism(ILP)数については, パラメータ探査の結果最高性能だったものを採用
  - シェアードメモリの使い方(single or double buffer, ブロック単位 or ワープ単位)やmemcpy_async()の使用についても同様
    - ただしmemcpy_async()の使用はCUDA版のみで実装
  - 非正規化数についてFTZ(flush-to-zero)を有効化するかどうかも比較
- パラメータ探査時の粒子数はN=4M
  - 逆数平方根はrsqrtf()を使用
    （acppでは__hipsycl_if_target_cuda()経由，icpxではsycl::rsqrt()）
  - CUDA: memcpy_async()を使ったほうが速くなった(FTZ有効時限定)
  - 最適なパラメータはCUDA, SYCL (icpx), SYCL (acpp) それぞれで異なる

AMD製GPU向けの実装

- HIP, SYCL 実装を測定
- 逆数平方根, ブロックあたりのスレッド数, ループアンローリング段数, Instruction Level Parallelism(ILP)数については, パラメータ探査の結果最高性能だったものを採用
  - シェアードメモリの使い方(single or double buffer, ブロック単位 or ウェーブ単位)の使用についても同様
  - Packed FP32命令(FP32の加算・乗算・積和算の性能が2倍)の使用・不使用
    - CDNA 2世代のGPUで導入された
  - 非正規化数についてFTZ(flush-to-zero)を有効化するかどうかも比較

- パラメータ探査時の粒子数はN=4M
  - 逆数平方根は`__frsqrt_rn()`を使用 (acppでは`__hipsycl_if_target_hip()`経由, icpxでは`sycl::rsqrt()`)
  - スレッド数は256, シェアードメモリは不使用

Intel製GPU向けの実装

- SYCL 実装を測定, SYCL も Intel oneAPI のみ
- 逆数平方根, ブロックあたりのスレッド数, ループアンローリング段数, Instruction Level Parallelism(ILP)数については, パラメータ探査の結果最高性能だったものを採用
  - シェアードメモリの使い方(single or double buffer, ブロック単位 or ワープ単位)の使用についても同様
  - 非正規化数についてFTZ(flush-to-zero)を有効化するかどうかも比較

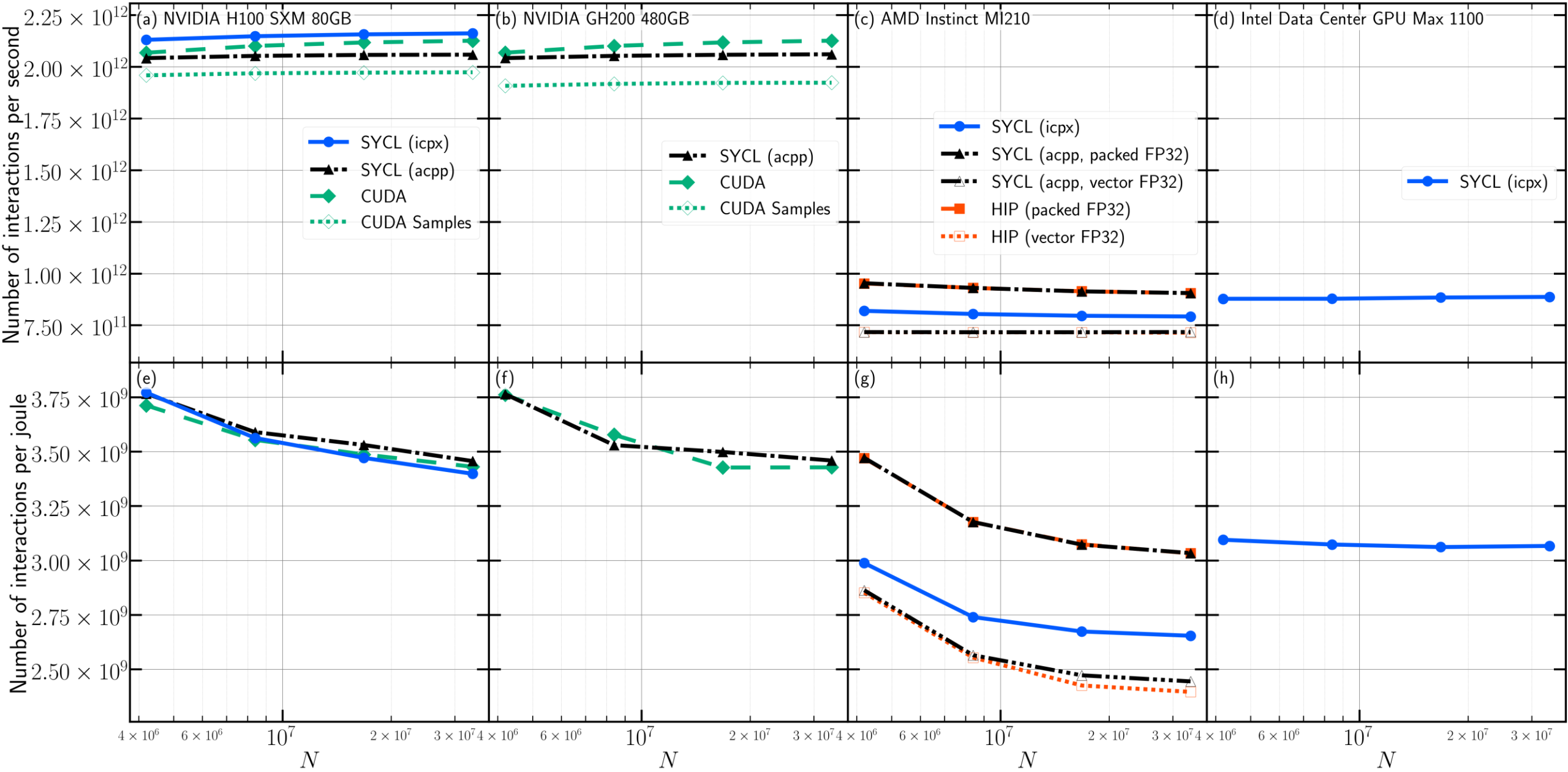
- パラメータ探査時の粒子数はN=4M
  - 逆数平方根は`sycl::rsqrt()`を使用
  - スレッド数は1024
  - シェアードメモリを使用(single buffer)
  - ループアンローリングは2段
  - ILPは使用せず(= 通常の実装)

GPU状態の監視

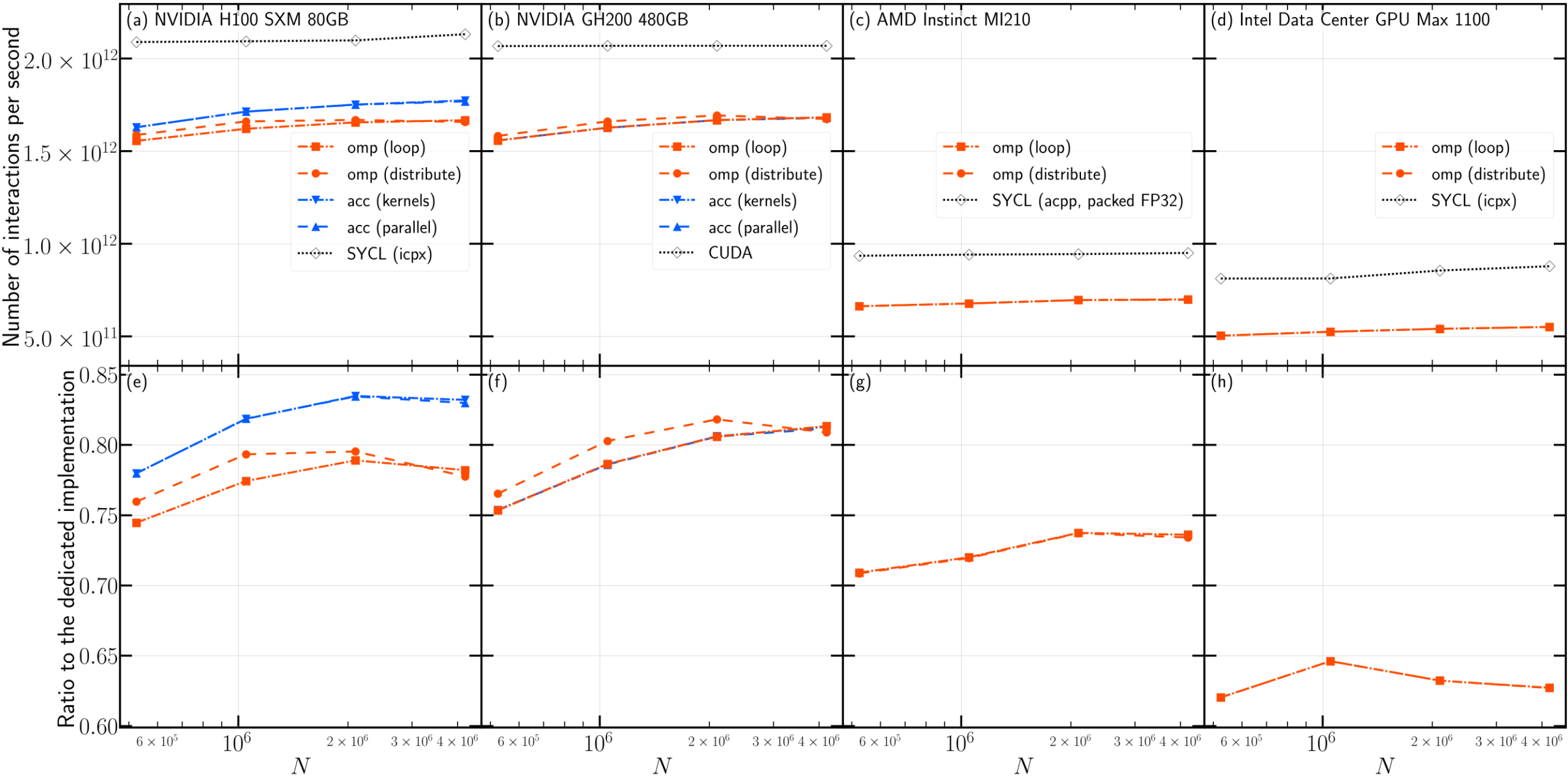
- GPUの温度, 動作周波数, 消費電力を計算中に取得
  - GPUの温度が下がった状態から測定するため, 毎回の測定前に10分間スリープ
- NVIDIA GPU: NVML
- AMD GPU: ROCm SMI
- Intel GPU: Level-Zero Sysman API
  - 管理者権限が要求されるメトリックあり
  - 電力測定方法はNVIDIA, AMDと少し違う
- 本計算に用いていない余剰コアを用い, 0.1秒間隔で状態を取得
  - OpenMPのタスク指示文を活用
  - 本計算側のコードについては一切変更なし

```
static bool repeat;
repeat = true;
#pragma omp parallel num_threads(2)
{
#pragma omp single
  {
#pragma omp task
    {
      while (repeat) {
        observe_GPU();
        sleep();
      }
    }
#pragma omp task
    {
      run_simulation();
      repeat = false;
    }
#pragma omp taskwait
  }
}
```

NVIDIA/AMD/Intel製GPUの性能比較

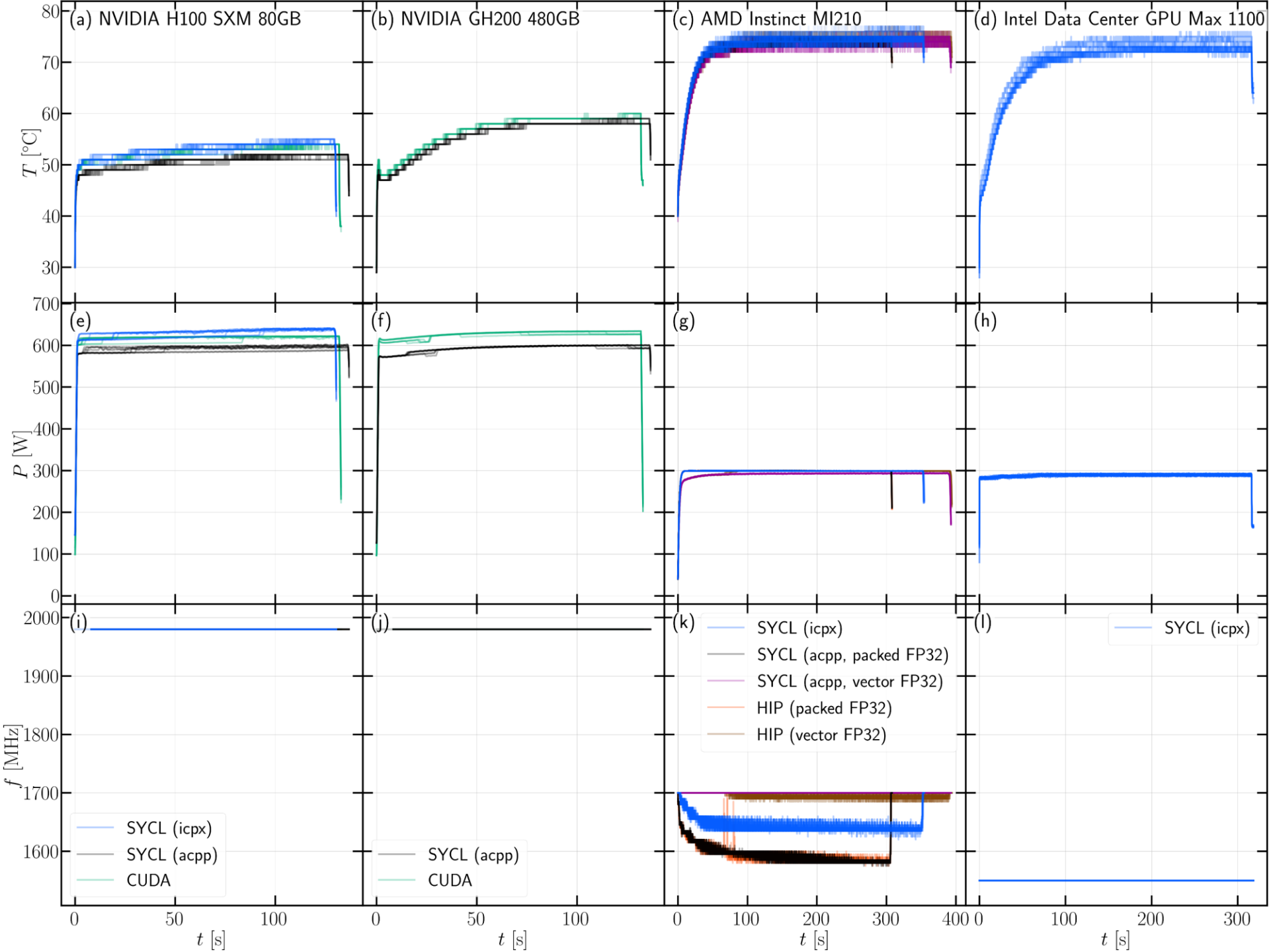


指示文実装との性能比較



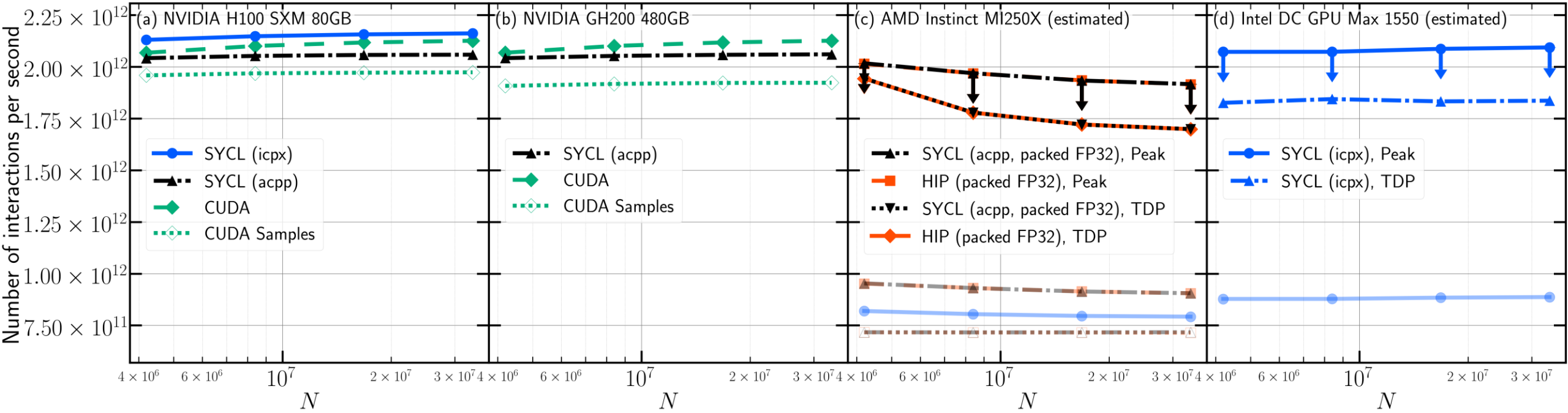
GPUの状態

- GPUの温度, 消費電力, 動作周波数を監視
- GPU温度上昇による動作周波数低下はなかった
- AMD MI210では, 供給電力不足による動作周波数低下（packed FP32命令の使用頻度に依存）



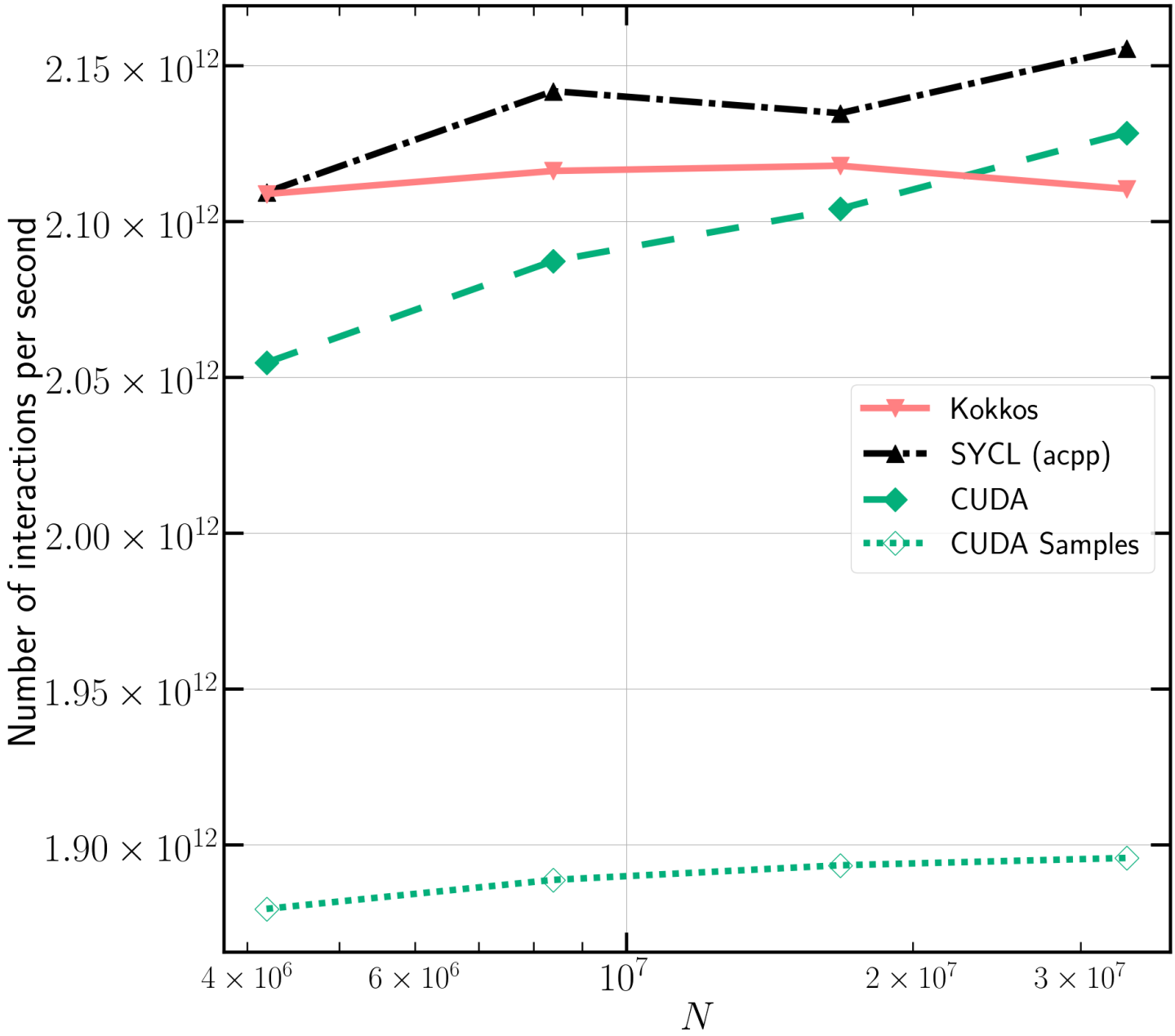
各世代の最上位製品どうしの比較

- AMD MI210, Intel Data Center GPU Max 1100 は最上位製品ではないので, 最上位製品を用いた際の性能を推定
  - 理論ピーク性能に基づく推定：実際には供給電力が不足するので上限値（下矢印つき）
    - AMD MI250X: MI210の2.12倍の理論ピーク性能, 1.87倍のTDP（注：MI210でも不足）
    - Intel Data Center GPU Max 1550: 1100の2.36倍の理論ピーク性能, 2倍のTDP
  - 消費電力に基づく推定：電力性能が変わらなければこちらの方が正解に近いはず
    - 電力性能は動作周波数に依存するため, （動作周波数が下がると）推定値より上がることもある



Kokkos との性能比較

- 欧米では性能可搬フレームワークKokkosがよく使われている
  - CMakeの使い方に一癖あり，(CMakeにはそれなりに慣れていたが)CMakeまわりの調整に一番時間がかかった
- Miyabi-G 上での測定
  - NVIDIA GH200 120GB
  - CUDA 12.6
  - AdaptiveCpp 24.10.0
    - LLVM 19.1.7
  - Kokkos 4.5.99
    - `nvcc_wrapper` は手直しした
- (なぜか)SYCL (acpp) が一番速い
  - Kokkos も良い勝負



昨日GPU化してもらった際のはまりポイント

- 複数の人向けに紹介した点を全体に共有
  - 元のサンプルコードの実装から素直にGPU化すると皆が陥りやすいトラップだった
- メモリの確保方法
  - 元の実装では, 粒子配列は`static`宣言付きで静的に確保されていた
  - 指示文実装などでGPUとのデータのやり取りをする際には, メモリを動的に確保する（`malloc`, `new` など）
- 並列リージョン(指示文の影響範囲)よりも前に宣言されている変数は, 全スレッドで共有される
  - （注:マルチコアCPU向けのOpenMPでも同様）
  - ループカウンタ`i`, `j`などは各スレッドが独立な値として持っていないといけない
  - C99以降の形式であれば`for`文の中で変数を定義することもできるため, 変数が必要になったタイミングで宣言する. または, 指示文に `private(i, j)` などと追記する
  - 配列はもっと扱いが面倒なので, `r[3]` などは `dx`, `dy`, `dz` などにする方が楽

C/C++ 向けの最適化アドバイス

- 配列を複数渡す関数(calc_forceなど)では, 各配列のメモリ空間が重なっていないことをコンパイラに教えてあげると良い
  - Intelコンパイラは賢いので, このあたり何も言わずとも対応してくれることも
  - restrict という修飾子があるので, これを使う

```
void calc_force(int n, double m[restrict], double x[restrict][3], double a[restrict][3],
double eps2){
```

```
void calc_force(const int ni, const float4 *pos_i, float4 * restrict acc_i, const int nj,
const float4 *pos_j, const float eps2) {
```

  - CUDAの場合には, __restrict__ がこれに対応

```
__global__ void calc_force_kernel(const float4 *pos_i, float4 *__restrict__ acc_i, const
int nj, const float4 *pos_j, const float eps2) {
```

  - 中身を書き換えないものについては, const をつけておくのも良い
- コンパイラに対して多くの情報を渡してあげるほど, 高速な実行ファイルを作ってくれる(基本的には安全側に倒して保守的なファイルを生成するので, 過剰な配慮が不要であると伝えてあげると良い)

実習: 昨日GPU化したコードを高速化してみる

- 先ほど紹介した高速化手法のうち, 簡単に試せそうなものを実装してみる

- プロファイラを使って遅くなっている原因を探し出し, 解決を図る

- 参考用のコードなど
  - `g00:/cfca-work/gpuws00/cfca-gpu2025f/` 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
  - コンパイルしたときのログも入っているので, 参考にしてください

TIPS: コンパイルしたときのログをファイルに保存

- 参考の .bashrc に設定を書き込んでいます
  - g00:/cfca-work/gpuws00/samples/.bashrc

```
# Save input command and results
function cmd_logger() {
    local tag=$1
    local cmd=$*
    local DATE=$(date +%y%m%d%H%M%S)
    local log="${DATE}${tag}.log"
    command echo "$ $cmd" > $log
    command hostname  2>&1 | command tee --append $log
    module list       2>&1 | command tee --append $log
    eval command $cmd 2>&1 | command tee --append $log
}

# Save logfile of ninja-build
function ninja() {
    cmd_logger ninja -v $*
}

# Save logfile of make
function make() {
    cmd_logger make $*
}

# Save logfile of cmake
function cmake() {
    cmd_logger cmake $*
}
```

Contents

- GPU(Graphics Processing Units)の紹介
- 指示文を用いたGPU化
  - OpenACC, OpenMP, Solomon
- CUDAを用いたGPU化
  - 指示文を用いてGPU化したコードの一部をCUDA化
  - コード全体をCUDAで実装
- ベンダーニュートラルGPUコンピューティングの紹介(HIP, SYCL)
- GPUコードの性能最適化に向けて
  - 気をつけるべき項目や機能の紹介
  - プロファイラの使い方の紹介
- MPIを用いたマルチGPU化

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)

環境変数 CUDA_VISIBLE_DEVICES

- 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 などのノードあたり1 GPUのシステムでは, この内容は不要
- この環境変数を用いると, cudaSetDevices() などを使わずにどのGPUを用いるかを外から制御できる
  - AMD製GPUでも, ROCR_VISIBLE_DEVICESで同様の処理ができる
- 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):

```
#!/bin/sh
export
LOCAL_ID=$OMPI_COMM_WORLD_LOCAL_RANK
export CUDA_VISIBLE_DEVICES=$LOCAL_ID
$*
```

    - これはOpen MPI の場合の例

今回の講習会の中での実装手順

- 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した演習
  - 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
- OpenACCでの例:

```
#pragma acc host_data use_device(recv_buffer)
MPI_Irecv(recv_buffer, next_nj * 4, MPI_FP_M, recv_from, 0, MPI_COMM_WORLD, &recv_req);
#pragma acc host_data use_device(compute_buffer)
MPI_Isend(compute_buffer, nj * 4, MPI_FP_M, send_to, 0, MPI_COMM_WORLD, &send_req);
```

- OpenMPでの例:

```
#pragma omp target data use_device_ptr(recv_buffer)
MPI_Irecv(recv_buffer, next_nj * 4, MPI_FP_M, recv_from, 0, MPI_COMM_WORLD, &recv_req);
#pragma omp target data use_device_ptr(compute_buffer)
MPI_Isend(compute_buffer, nj * 4, MPI_FP_M, send_to, 0, MPI_COMM_WORLD, &send_req);
```

- Solomonでの例:

```
USE_DEVICE_DATA_FROM_HOST(recv_buffer)
MPI_Irecv(recv_buffer, next_nj * 4, MPI_FP_M, recv_from, 0, MPI_COMM_WORLD, &recv_req);
USE_DEVICE_DATA_FROM_HOST(compute_buffer)
MPI_Isend(compute_buffer, nj * 4, MPI_FP_M, send_to, 0, 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
- 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}
```

ラッパースクリプト例

- ここまで凝った実装にする必要はまったくないが，大抵のMPIライブラリに対応している
- IB HCAのマッピングもかけている（環境によってはかなり効く）
  - `$ nvidia-smi topo -m` の出力を見て適切に割り付ける

```
#!/usr/bin/env bash
# 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
    mpi_rank=${OMPI_COMM_WORLD_RANK:=${MV2_COMM_WORLD_RANK:=${PMI_RANK:=${PMIX_RANK:=0}}}}

    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
$*
```

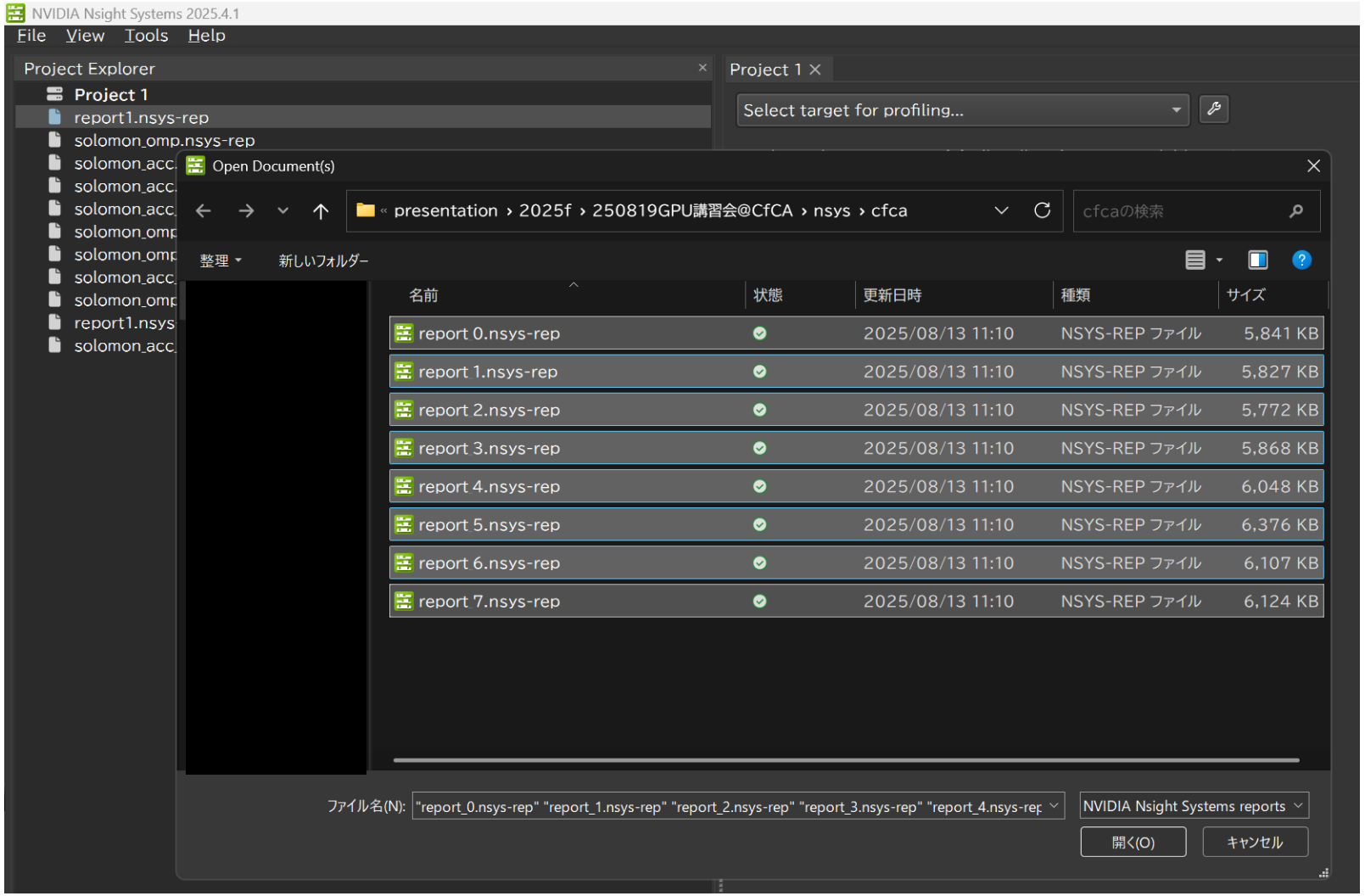
MPI版のコードのプロファイルを取得するには？

- 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} $*
```

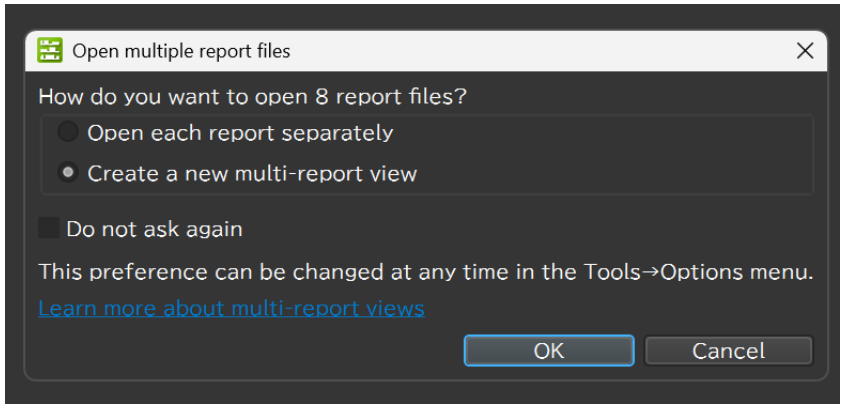
取得したプロファイル情報の可視化方法(1/3)

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

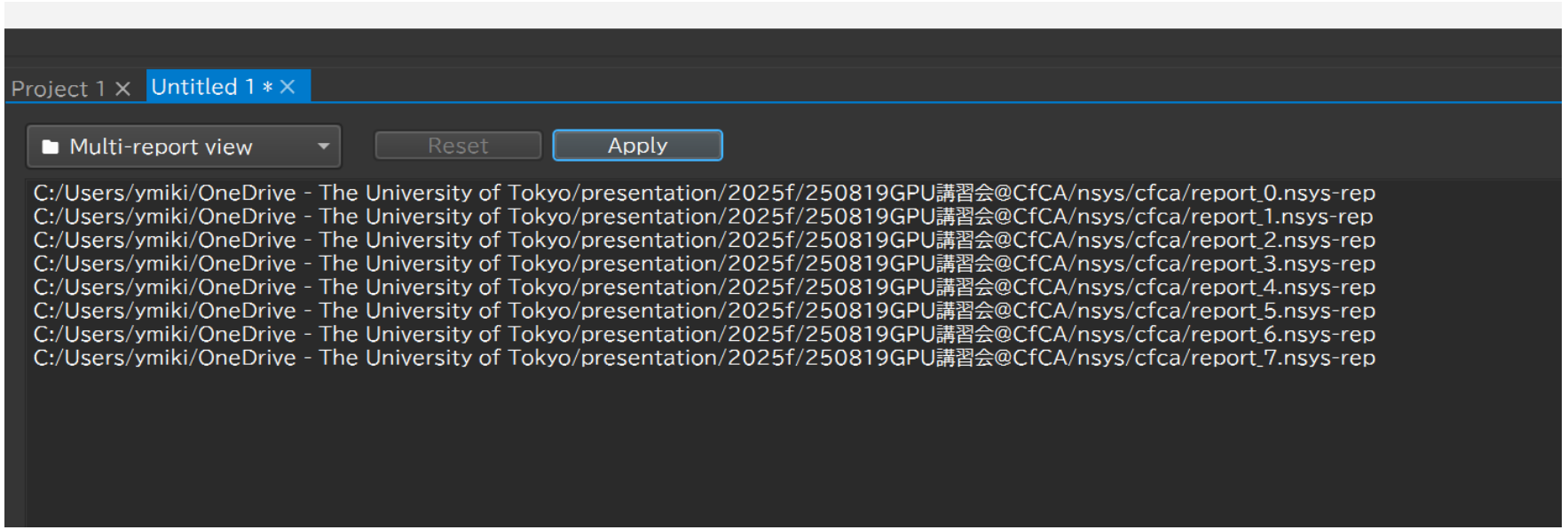


取得したプロファイル情報の可視化方法(2/3)

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

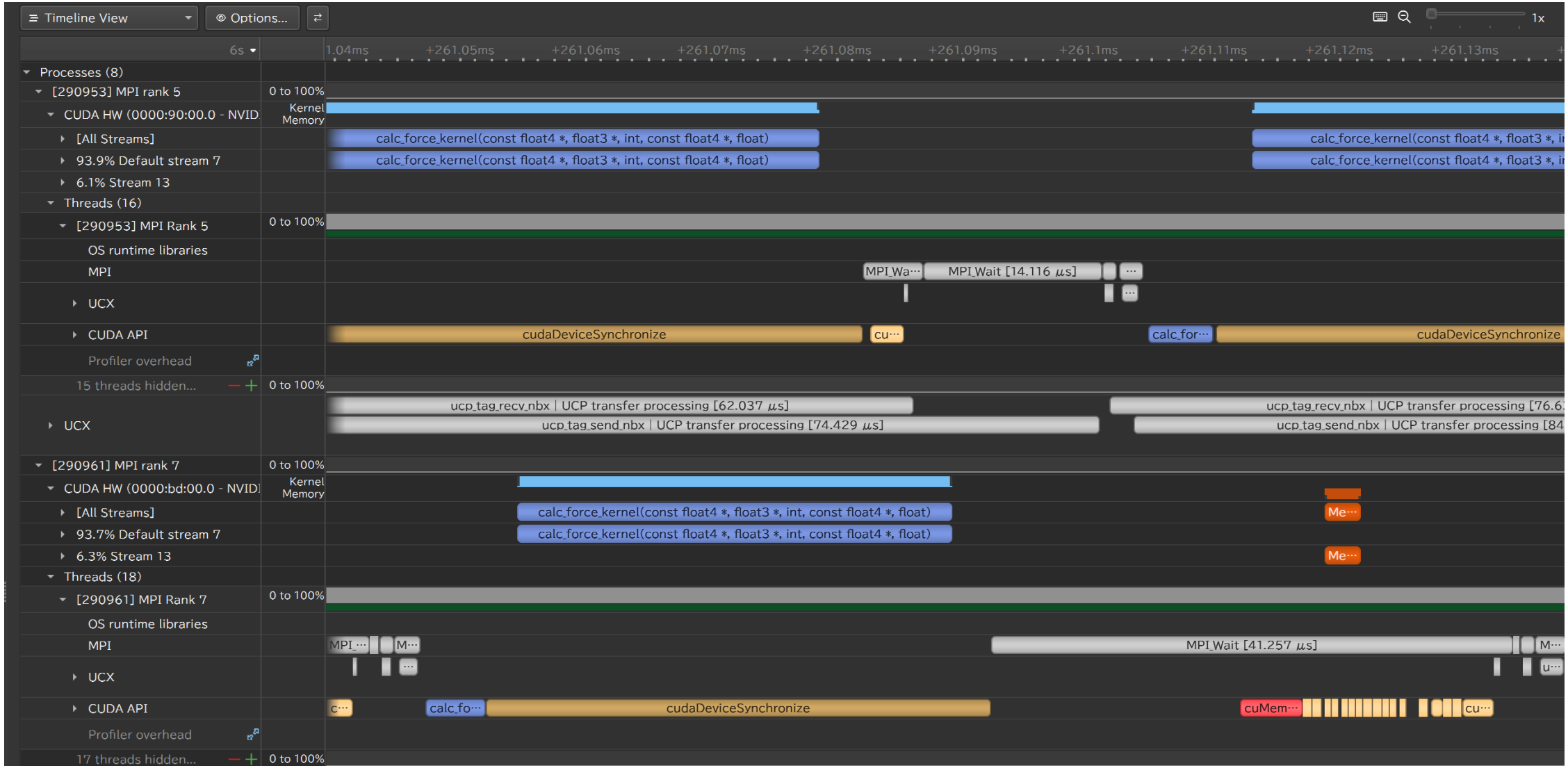


- 下記の画面になるので, Apply をクリック



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 250820gpu_mpi # 何か適当なフォルダを作る`
- `$ cd    250820gpu_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

参考用のコードなど(これから解説します)

- g00:/cfca-work/gpuws00/cfca-gpu2025f/ 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

- コンパイルしたときのログも入っているので, 参考にしてください

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-gpu2025f
- 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に情報を記載中
  - https://jcahpc.github.io/gpu_porting/