

第229回お試しアカウント付き並列プ
ログラミング講習会
「OpenMPで並列化されたC++プロ
グラムのGPU移植手法」

東京大学 情報基盤センター
三木 洋平

講習会概略

- 開催日: 2024年8月26日(月) 13:00--17:00
- 形態: ZoomおよびSlackを用いたオンライン講習会
- 使用システム: Wisteria/BDEC-01 (Aquarius)
- 講習会プログラム:
 - 13:00--13:30 GPU向けプログラミング手法の紹介(座学)
 - 13:40--14:20 OpenACCを用いたGPU化(座学+演習)
 - 14:30--15:10 OpenMPを用いたGPU化(座学+演習)
 - 15:20--15:50 C++17を用いたGPU化(座学+演習)
 - 16:00--17:00 CUDA C++を用いたGPU化(座学+演習)

Contents

- GPUに関する基礎知識
- (主にNVIDIA製の)GPU向け各種プログラミング手法の紹介
 - OpenACC
 - OpenMP の target 指示文(OpenMP GPU offloading)
 - 言語の標準規格を利用したのGPU化(C++17以降)
 - CUDA C++
- 発展的な内容(情報提供のみ)
 - Independent Thread Scheduling
 - A100/CUDA 11から導入された機能
 - CUDA-aware MPI

講習会概略

- 開催日: 2024年8月26日(月) 13:00--17:00
- 形態: ZoomおよびSlackを用いたオンライン講習会
- 使用システム: Wisteria/BDEC-01 (Aquarius)
- 講習会プログラム:
 - 13:00--13:30 GPU向けプログラミング手法の紹介(座学)
 - 13:40--14:20 OpenACCを用いたGPU化(座学+演習)
 - 14:30--15:10 OpenMPを用いたGPU化(座学+演習)
 - 15:20--15:50 C++17を用いたGPU化(座学+演習)
 - 16:00--17:00 CUDA C++を用いたGPU化(座学+演習)

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): 64×108 SMs = 6912 cores
 - NVIDIA H100 (SXM): 128×132 SMs = 16896 cores
 - NVIDIA H100 (PCIe): 128×114 SMs = 14592 cores
 - AMD MI250X: 64×110 CUs \times 2 GCDs = 14080 cores
 - AMD MI250: 64×104 CUs \times 2 GCDs = 13312 cores
- 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 2024)

	TOP 500	System	Accelerator	Cores	HPL Rmax [PFlop/s]	Power [kW]	GFLOPS/W
1	189	JEDI, EuroHPC/FZJ, Germany	NVIDIA GH200 Superchip	19,584	4.50	67	72.733
2	128	Isambard-AI phase 1, University of Bristol, UK	NVIDIA GH200 Superchip	34,272	7.42	117	68.835
3	55	Helios GPU, Cyfronet, Poland	NVIDIA GH200 Superchip	89,760	19.14	317	66.948
4	328	Henri, Flatiron Institute, USA	NVICIA H100 80GB PCIe	8,288	15.47	240	65.396
5	71	preAlps, CSCS, Switzerland	NVIDIA GH200 Superchip	81,600	15.47	240	64.381
6	299	HoreKa-Teal, KIT, Germany	NVIDIA H100 94GB SXM5	13,616	3.12	50	62.964
7	54	Frontier TDS, ORNL, USA	AMD Instinct MI250X	120,832	19.20	309	62.684
8	11	Venado, LANL, USA	NVIDIA GH200 Superchip	481,440	98.51	1,662	59.287
9	20	Adastra, GENCI-CINES, France	AMD Instinct MI250X	319,072	46.10	921	58.021
10	28	Setonix-GPU, Pawsey, Australia	AMD Instinct MI250X	181,248	27.16	477	56.983
61	86	PRIMEHPC FX1000, Central Weather Administration, Taiwan	(CPU-only: Fujitsu A64FX)	184,320	11.16	674	16.575
82	84	Carpenter, ERDC DSRC, USA	(x86 CPU-only: AMD EPYC 9654)	276,480	11.62	1,100	10.561

OFP-IIへの道(1/2)

- OFP (Oakforest-PACS) 後継機種 (OFP-II)
 - JCAHPC (筑波大学と共同)
 - 2025年1月運用開始予定
- スパコンへの性能要求, 省電力, 脱炭素化⇒演算加速器搭載は不可避(電気代も高騰)
 - 2021年秋には方針決定



筑波大学
University of Tsukuba



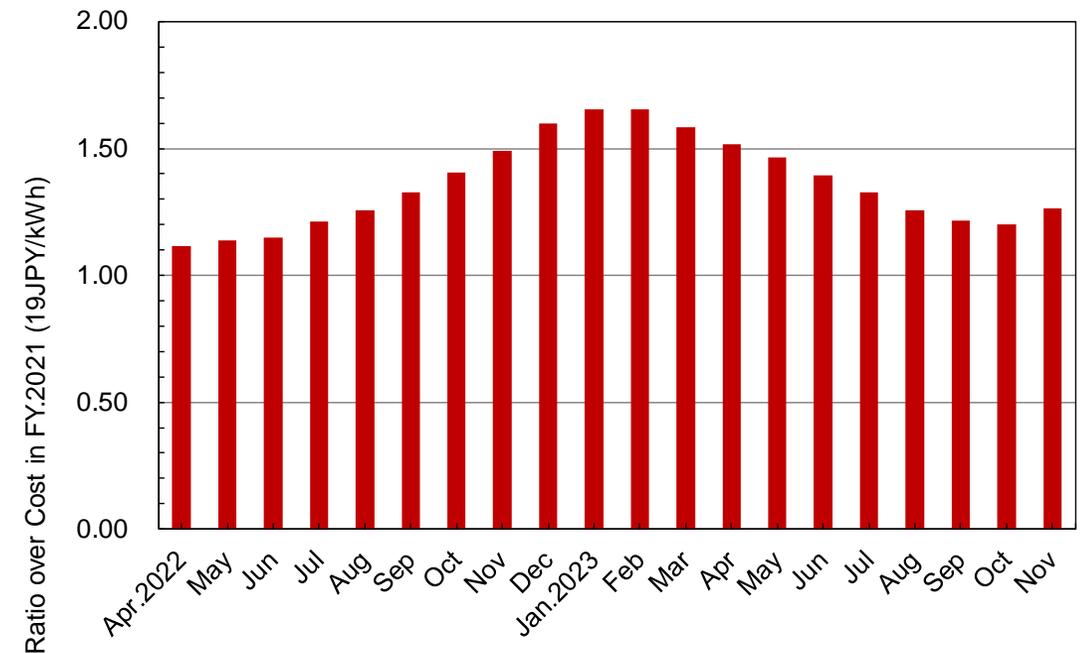
東京大学
THE UNIVERSITY OF TOKYO



System (Top/Green 500)	HW	GF/W
Henri (255,1)	NVIDIA H100	65.4
Frontier (1,6)	AMD MI250X	52.6
Leonardo (4,15)	NVIDIA A100	32.2
Fugaku (2, 49)	A64FX	15.4

2024/8/26

講習会:GPU/C++



電力単価推移(対2021年度比)

OFP-IIへの道(2/2)

- OFP-II (2025年1月運用開始)
 - 汎用CPUクラスタ(CPU-Group) + GPUクラスタ(Acc-Group)
 - 「計算・データ・学習」融合路線は継続
- GPUの選定は2022年6月に終了
 - OFPユーザー(3,000人以上)のGPUへの移行には18-30ヶ月必要
 - 7種類のベンチマーク
- NVIDIA社製GPU採用に決定(2022年6月)
 - H100もしくははその後継機
 - 決め手
 - 性能そのもの
 - Fortranで記述されたアプリケーションのポータビリティ
 - OpenACC/StdPar(Standard Parallelism)によるGPU化は比較的簡単, OpenMP/MPIハイブリッドによって並列化されたプログラムに適している

CPU-Group
CPU only

Acc-Group
CPU+GPU

Miyabi (OFP-II) (1/2)

• 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)**

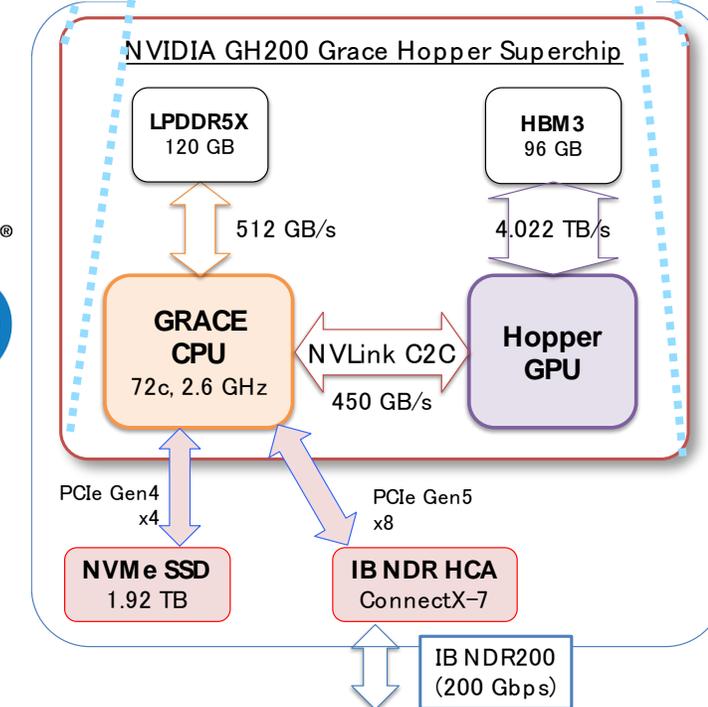
FUJITSU

SUPERMICR



nVIDIA

intel



Miyabi (OFP-II) (2/2)

FUJITSU



JCAHPC¹⁰

- ファイルシステム: DDN EXA Scaler, 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**

- $(400\text{Gbps}/8) \times (32 \times 20 + 16 \times 1) = 32.8 \text{ TB/sec}$

- **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



筑波大学
University of Tsukuba



東京大学
THE UNIVERSITY OF TOKYO



NVIDIA



IB-NDR (400Gbps)

IB-NDR200 (200)

IB-HDR (200)

Miyabi-G

NVIDIA GH200 1,120
78.2 PF, 5.07 PB/sec

Miyabi-C

Intel Xeon Max
(HBM2e) 2 x 190
1.3 PF, 608 TB/sec

File System

DDN EXA Scaler
11.3 PB, 1.0TB/sec

Ipomoea-01
Common Shared Storage
26 PB

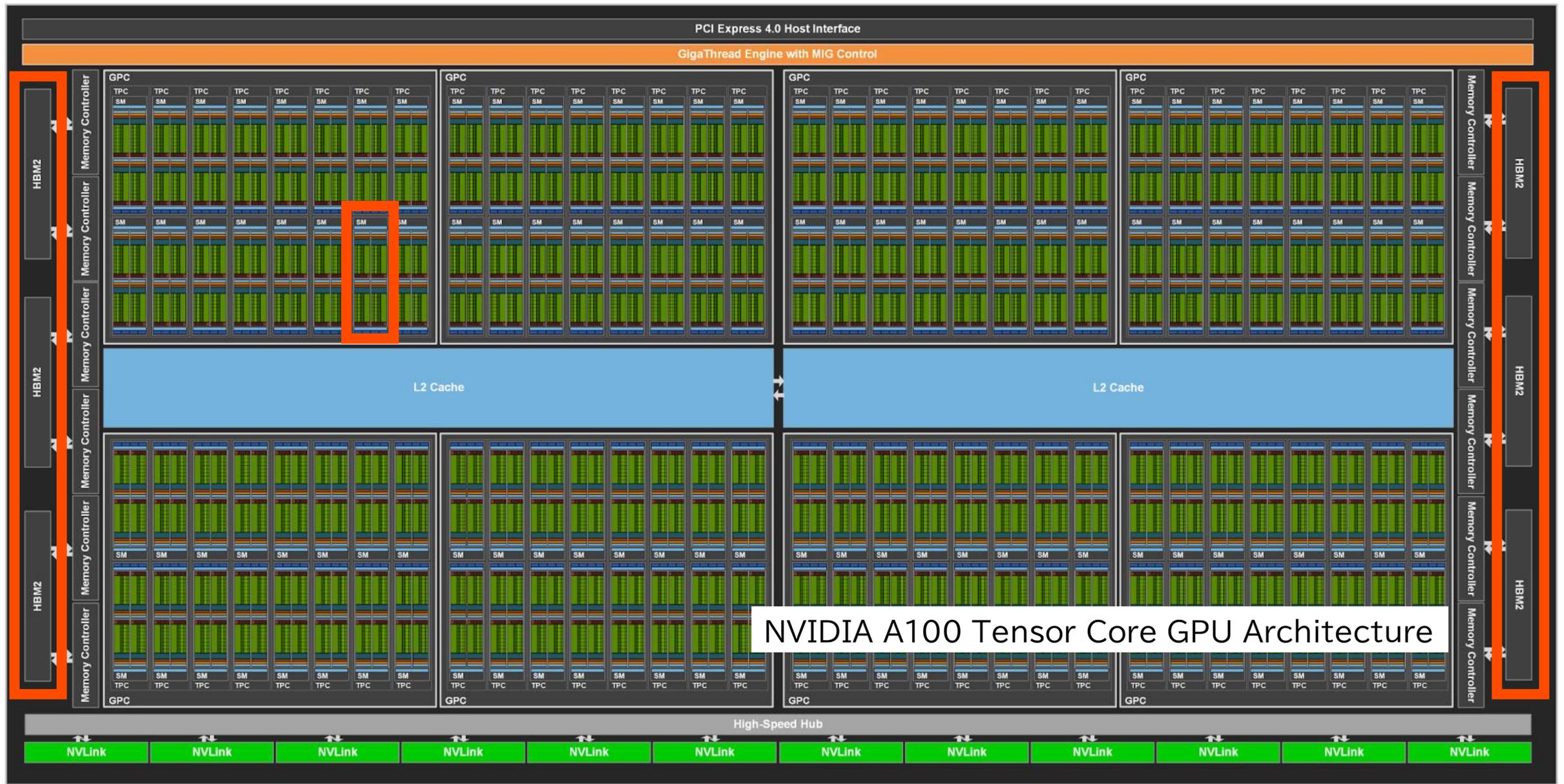


GPU移植・移行の計画

- NVIDIA Japanの協力
- 3,000人以上のOFP利用者:2つの形態
- 「自己移植(Self Porting)」:様々なオプション
 - 8日間のハッカソン(ミニキャンプ), 3ヶ月に1回, オンライン・ハイブリッド, Slack併用
 - 毎月開催される「相談会」(Zoom, 非ユーザーも自由に参加できる)
 - 移行ポータルサイト, 各種講習会
 - https://jcahpc.github.io/gpu_porting/
- 「サポート移植(Supported Porting)」, 2022年10月開始
 - 多くのユーザーを有するコミュニティコード(17種類, 次頁), OpenFOAM(NVIDIA)
 - 外注のための予算も確保(落札ベンダーが担当する予定)
 - 「サポート移植」グループメンバー(主に若手)はハッカソン・相談会にも積極的に参加
- 基本的にOpenACC/StdPar(Standard Parallelism)推奨



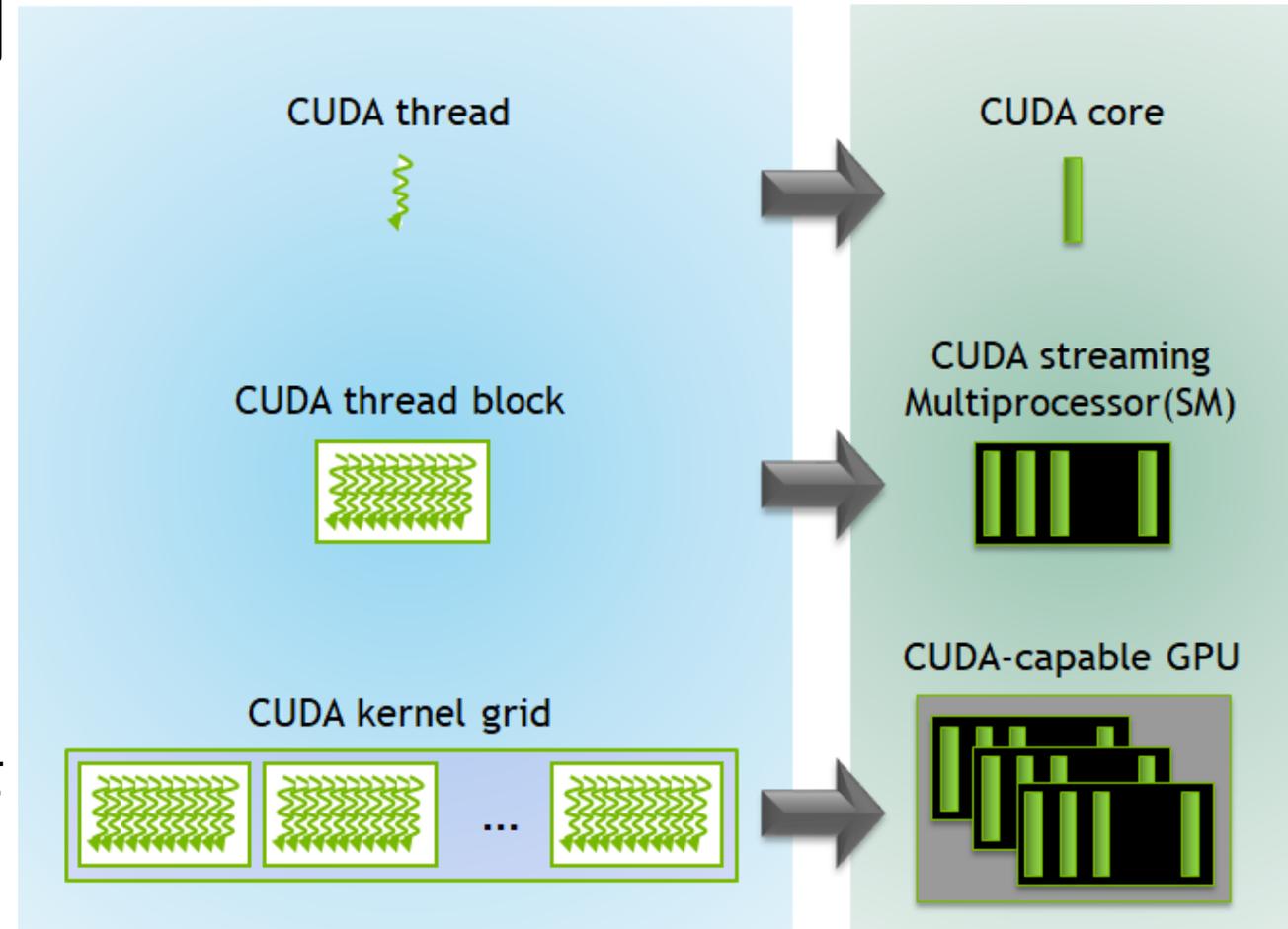
(NVIDIAの)GPUの構成(1/2)



NVIDIA A100 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/>

他手法での用語紹介(覚える必要なし)

- Vector: CUDA でのスレッドに対応
 - この並列度は性能へのインパクト大(32の倍数, できれば128以上)
 - OpenMPのGPUオフローディングでは SIMD に対応
- Worker: CUDAでのスレッドブロックに対応
 - Vector の集合
 - Worker 単位でSMに投入される(が, こんなことまで気にしなくてOK)
 - OpenMPのGPUオフローディングでは thread に対応
- Gang: CUDAでのグリッドに対応
 - Worker の集合
 - ジョブの全体ということになるが, 全く気にしなくてOK
 - OpenMPのGPUオフローディングでは teams に対応

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 まわりはまったくケア・更新していない(今日はC/C++が対象)

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の 発展に期待?	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?)

大まかな考え方の違い

- 簡易GPU化(OpenACC, OpenMP, 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

- 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, C++17ではGPU化が難しいループであっても, GPU化できる
 - 例: ツリー構築のGPU化は指示文ベースでできる気はしないが, CUDAなら可能

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/226/>
 - 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/](https://jcahpc.github.io/gpu%20porting/)

Zoom関連

- 「手をあげる」機能
 - 質問がある際, 全体の状況を確認するため使用
- ブレークアウトセッション
 - 画面を共有しながらエラー対応する際に使用
 - (なるべく口頭でのやりとりやSlackで対応する予定)
- [https://utelecon.adm.u-tokyo.ac.jp/zoom/how to use](https://utelecon.adm.u-tokyo.ac.jp/zoom/how%20to%20use)

「手を挙げる」方法

1. Zoomメニュー中の「リアクション」をクリック



2. ポップアップで表示された「挙手」をクリック



手が挙がっていることの確認方法

1. Zoomメニュー中の「参加者」をクリックして、参加者一覧を表示



2. 表示された参加者一覧の、自分のところを見ると手が挙がっている



「手をおろす」方法

1. ポップアップで表示されている「手をおろす」をクリック



- もし「手をおろす」が表示されていないならば、「リアクション」の中から探す

ブレイクアウトルーム(1/6)

- 演習時に使用するかもしれません
- 演習中に「ヘルプを求める」ことができます
 - ホストを招待した後に「画面を共有」することで、皆さんの記述したプログラムを一緒に見ながら問題解決にあたります
- Zoomメニュー中の「ブレイクアウトルーム」をクリック



ブレイクアウトルーム(2/6)

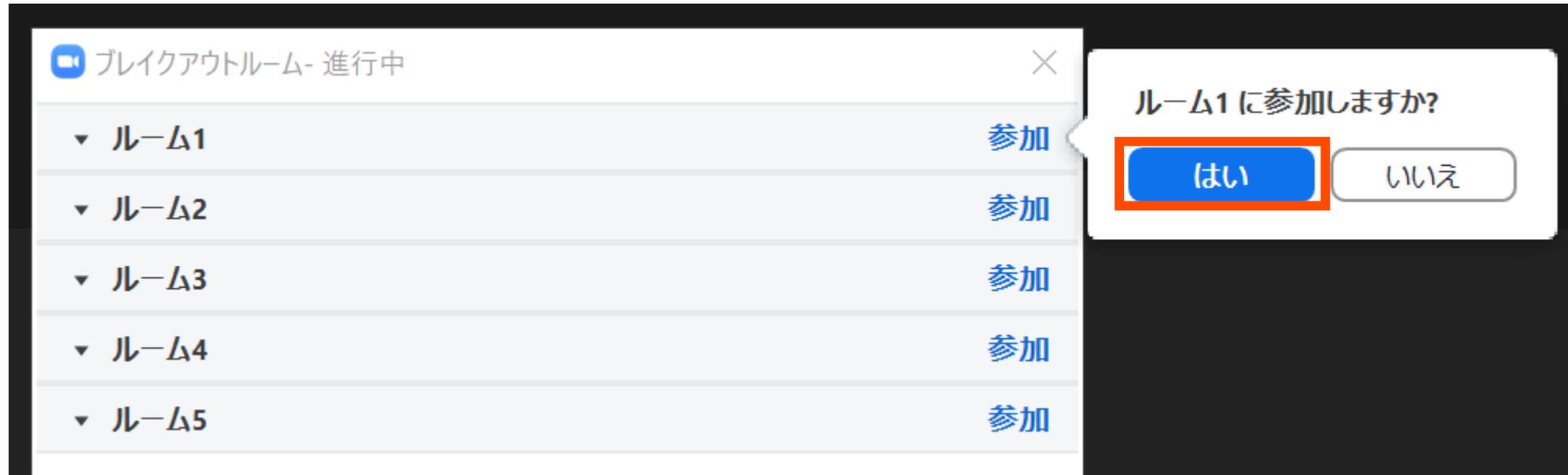
- 進行中のブレイクアウトルームのリストが表示されるので、空いている部屋に「参加」してください
 - 左の例では5部屋がすべて空室、右の例ではルーム1のみ参加者がいる

ブレイクアウトルーム- 進行中		×
▼ ルーム1	参加	
▼ ルーム2	参加	
▼ ルーム3	参加	
▼ ルーム4	参加	
▼ ルーム5	参加	

ブレイクアウトルーム- 進行中		×
▼ ルーム1	参加	
● tABCDE		
▼ ルーム2	参加	
▼ ルーム3	参加	
▼ ルーム4	参加	
▼ ルーム5	参加	

ブレイクアウトルーム(3/6)

- 「参加」をクリックすると確認画面が出てくるので、「はい」を選択すると入室できます



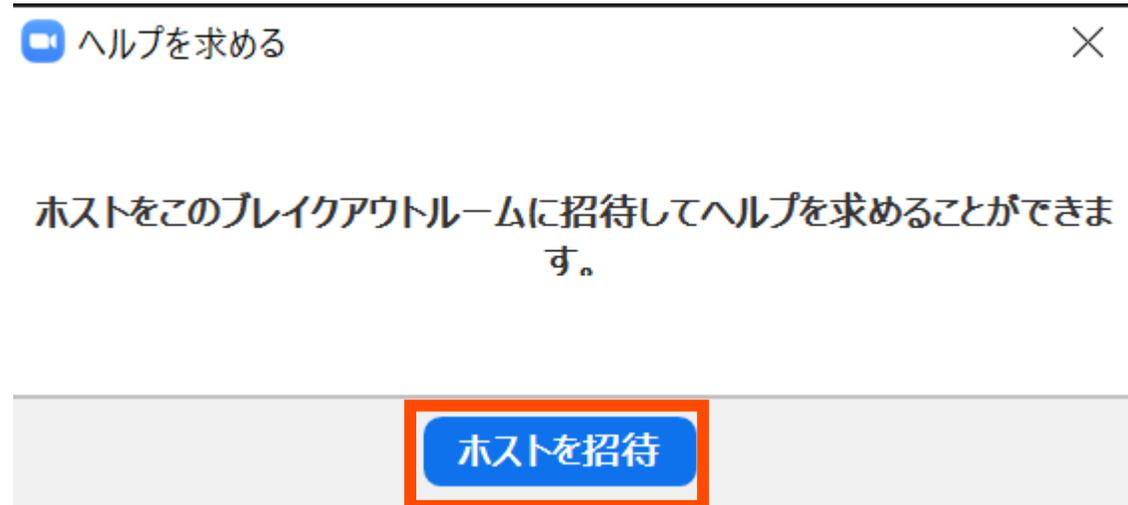
ブレイクアウトルーム(4/6)

- 再度メニュー中の「ブレイクアウトルーム」をクリックすると、「ヘルプを求める」が増えているのでクリックしてください



ブレイクアウトルーム(5/6)

- ポップアップで出てきた「ホストを招待」をクリック
- ホスト(講師)の参加待ちに移行します(画面はそのまま)
 - 他の受講者のヘルプ中など, 直ちに対応できない場合もあります

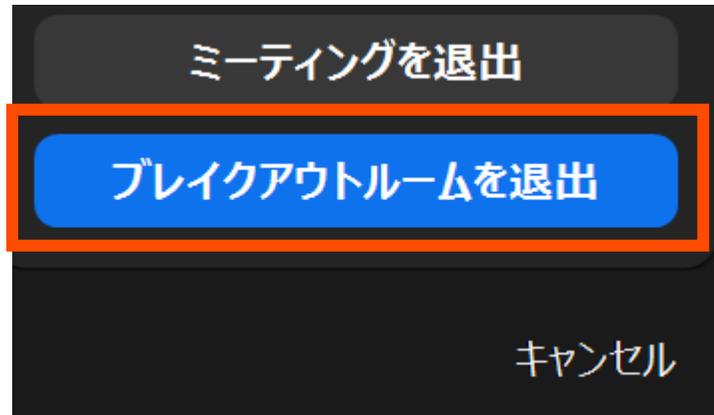


ブレイクアウトルーム(6/6)

- 問題解決後は, Zoomメニュー中の「ルームを退出する」



- 「ブレイクアウトルームを退出」が表示されるのでクリックして, 元の講習会会場にお戻りください
 - 間違えて「ミーティングを退出」すると講習会から退出します



Slack関連

- ブラウザ上で使う場合には:
 - <https://w1590055008-bgo338004.slack.com/>
 - 注:ログインには, 事前にお配りしたリンクからの登録が必要です
- 質問対応に使用
- コードの貼り付け方
- スレッドの確認方法

- 以下, ブラウザ版で説明しますがアプリ版でも操作は同じです

質疑応答チャンネルへの移動

- 左側のメニューバーのチャンネル一覧内に「第229回-cpp2gpu」があるので、クリック
- 見つからない場合
 1. 「チャンネルを追加する」をクリック
 2. 「チャンネル一覧を確認する」をクリック
 3. 「第229回-cpp2gpu」があるので「参加する」をクリック

[🔗 プランをアップグレード](#) 未読 スレッド 下書き & 送信済み

▼ チャンネル

general

random

第133回-gpuプログラミング入門

第141回-mpi基礎

第153回-mpi基礎

第156回-wisteria実践

第161回-wisteria実践

第165回-mpi基礎

第170回-wisteria実践

第176回-mpi基礎

第181回-wisteria実践

第185回-wisteria実践

第189回-mpi基礎

第199回-wisteria実践

第203回-mpi基礎

第207回-utokyo-n-ways-to-gpu...

第208回-cpp2gpu

第213回-mpi基礎

第220回-wisteria実践

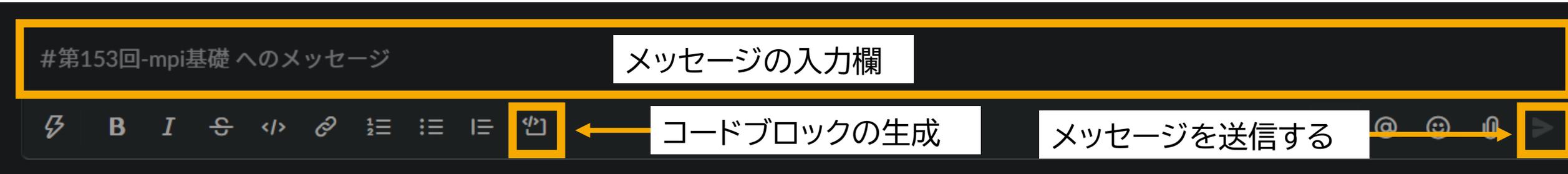
第224回-mpi基礎

第229回-cpp2gpu

+ チャンネルを追加する

メッセージの入力方法

- 最下部に入力欄があるので、質問内容を記載して Ctrl+Enter
 - 入力後に右下の「メッセージを送信する」をクリックしても同じ
(メッセージ入力前には、「メッセージを送信する」は押せない)



- コードを入力する際には、「コードブロック」がおすすめ
 - 枠が生成されるので、この中にコピペするのが簡単かつ見やすい
 - `` (JIS配列ならばShift+@を3連打)しても枠が生成される

自分が参加したスレッドのみの表示方法

- 左上の「スレッド」をクリックすると、自分が参加しているスレッドの一覧が表示されます
 - 質問内容には、「スレッドで返信する」形式で回答するので、自分が聞いた内容のみが表示できます



ユーザアカウント

- 使用システム: Wisteria/BDEC-01 (Aquarius)
 - `$ ssh USERNAME@wisteria.cc.u-tokyo.ac.jp`
- 本講習会でのユーザ名(上記コマンドでの USERNAME のこと)
 - 利用者番号: `tABCDE` (ABCDEは, 適宜書き換えてください)
 - 利用グループ: `gt00`
- 利用期限
 - `9/26 9:00`まで有効
- 注:本講習会関連の質問はymiki[at]cc.u-tokyo.ac.jpまで
 - Slack で質問していただいても結構です
 - (講習会アカウントでは)公式の相談対応システムは使わないでください

講習会概略

- 開催日: 2024年8月26日(月) 13:00--17:00
- 形態: ZoomおよびSlackを用いたオンライン講習会
- 使用システム: Wisteria/BDEC-01 (Aquarius)
- 講習会プログラム:
 - 13:00--13:30 GPU向けプログラミング手法の紹介(座学)
 - 13:40--14:20 OpenACCを用いたGPU化(座学+演習)
 - 14:30--15:10 OpenMPを用いたGPU化(座学+演習)
 - 15:20--15:50 C++17を用いたGPU化(座学+演習)
 - 16:00--17:00 CUDA C++を用いたGPU化(座学+演習)

NVIDIA GPU向けに各手法でGPU化

- 対象とする計算:N体計算(直接法)
 - 無衝突系向け(2次Leapfrog, shared time step)
 - 衝突系向け(4次Hermite, block time step)
- 今日試してみるプログラミング手法
 - OpenACC
 - OpenMP
 - C++17
 - CUDA C++
- 今日使うサンプルコード(上記のGPU実装)
 - <https://github.com/ymiki-repo/nbody>
 - CMake のサンプルとしても参照可(HDF5部分は実装が特殊)

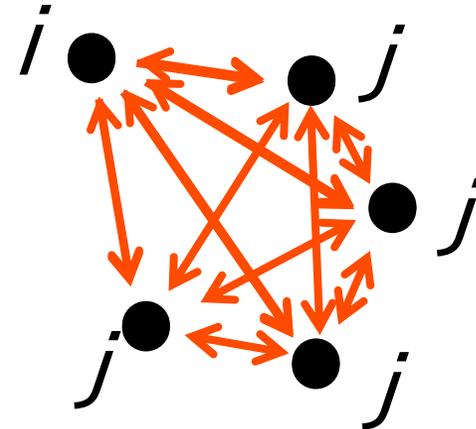
N体計算(重力多体計算)

- 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: $O(N)$
- 重力計算: $O(N^2)$
- 時間積分: $O(N)$

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

- 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



- (N体計算の詳細は国立天文台CfCAの[N体学校](#)を参照)

OpenACC/OpenMP での実装方針

1. Unified Memory を使って実装

- GPU上のメモリ確保, CPU-GPU 間のデータ転送は全てお任せ
- 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 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

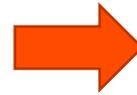
- Unified Memory では, 一旦メモリを読んで, ページフォルトがあればハードウェアレベルでページ単位で転送という仕組み
- 必要なデータ転送は自分で指示した方が必然的に速くなる
- Unified Memory ではCPU上のアドレスとGPU上のアドレスを同一視してしまうので, GPUDirect系の機能を活用する際に不利

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 (type::int_idx ii = 0U; ii < Ni; ii++) {
    // ループ内の実装は省略
}
```

```
#pragma omp parallel for
for (type::int_idx ii = 0U; ii < Ni; ii++) {
    // ループ内の実装は省略
}
```



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

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

OpenACCコードのコンパイル方法など

- NVIDIA HPC SDK向けの情報
- `$ 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,managed -Minfo=accel,opt`
 - Unified Memory使用時のコンパイル方法
 - リンク時にも `-acc=gpu -gpu=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

計算機へのログインとサンプルの取得

- `$ ssh USERNAME@wisteria.cc.u-tokyo.ac.jp`
- `$ cd /work/gt00/$USER`
- `$ git clone https://github.com/ymiki-repo/nbody.git`
- `$ cd nbody`

コンパイル方法

- `$ module load cmake nvidia/22.7 hdf5`
 - nvidia/23.3(default) 用のHDF5環境が未構築のため
- `$ cmake -S . -B build_nvhpc`
 - `$ mkdir build_nvhpc; cd build_nvhpc; cmake .. # 同じ`
- `$ cd build_nvhpc`
- `$ ccmake -S .. # デフォルトの設定を調整したい場合`
 - 倍精度で計算したければ, `FP_L`, `FP_M`を64に変更
 - Hermite法で試したい方は, `HERMITE_SCHEME`をONに変更
- `$ ninja acc_managed`
 - 単に `$ ninja` だけでもOKですが, 他にもたくさんのビルドが走ります

moduleの指定

- コンパイラ・ライブラリ等の環境をセットアップ
 - Odyssey向け富士通コンパイラ、MPIを使用
 - `$ module load fj` (fjmpiは自動的にロードされる) または
 - `$ module load odyssey`
 - Aquarius向けgcc、cuda, Open MPI(CUDA対応)を使用
 - `$ module load gcc cuda omp-cuda` または
 - `$ module load aquarius cuda omp-cuda`
- 現在指定済みのmoduleを確認するには
 - `$ module list`
- 困った時(module環境を整理したくなった時)は
 - `$ module purge`

moduleの一覧

- 現在利用中の環境で追加できるものを確認
`$ module avail`
- 使い方の確認(例はModuleNameというmoduleのヘルプを表示)
`$ module help ModuleName`
- 設定されるPATHなどの確認(例はModuleNameというmoduleの場合)
`$ module show ModuleName`
- 全ての環境を確認(Wisteria/BDEC-01向けに提供されるコマンド)
`$ show_module`

ApplicationName	ModuleName	Node	BaseCompiler/MPI
Archiconda	archiconda3/0.2.3	odyssey	-
Arm Forge	forge/21.0.2	aquarius	-

実習

- `$ pjsub -x EXEC=bin/acc_managed sh/wisteria/run_nvidia.sh`
 - `sh/wisteria/run_nvidia.sh` 内に直接 `EXEC=bin/acc_managed` と書き込んだ上で `$ pjsub sh/wisteria/run_nvidia.sh` としても同じ
- 正常に終了すると, 下記ファイルが出力されるはずですよ
 - `log/collapse_run.csv`
 - エネルギー保存していることを確認(右端2列の数字が 0.005 より十分に小さい)
 - `dat/collapse_snp[000 - 080].[h5 xdmf]`
- 本当はここで可視化して計算結果をチェックするところだが, (環境構築の都合上)今回は省略
 - Julia スクリプトを使ってエネルギー保存, ビリアル比の時間進化を表示
 - Julia スクリプトを使って粒子分布の時間進化を表示
 - VisIt や ParaView を使って粒子分布の時間進化を表示

ジョブスクリプトの説明

```
#!/bin/bash
#PJM -L rscgrp=lecture-a
#PJM -L gpu=1
#PJM -L elapse=0:15:00
#PJM -g gt00

if [ -z "${EXEC}" ]; then
    EXEC=bin/base
fi

module purge
module load nvidia/22.7
module load hdf5

cd $PJM_O_WORKDIR
numactl --localalloc ${EXEC} ${OPTION}
```

リソースグループ名: lecture-a

利用GPU数: 1 GPU使用

実行時間制限: 15分

利用グループ名: gt00

実行ファイルの指定

環境設定

GPUジョブを実行

Wisteria/BDEC-01でのジョブ実行

- 以下の2通りの実行形態があります

1. バッチジョブ実行

- バッチジョブシステムに処理を依頼して実行
- 実行したい処理をファイル(ジョブスクリプト)で指示
- スパコン環境で一般的
- 大規模実行用
 - Wisteria/BDEC-01 (Odyssey)では, 最大2304ノード(110592コア), 24時間まで
 - Wisteria/BDEC-01 (Aquarius)では, 最大8ノード(64 GPUs), 24時間まで

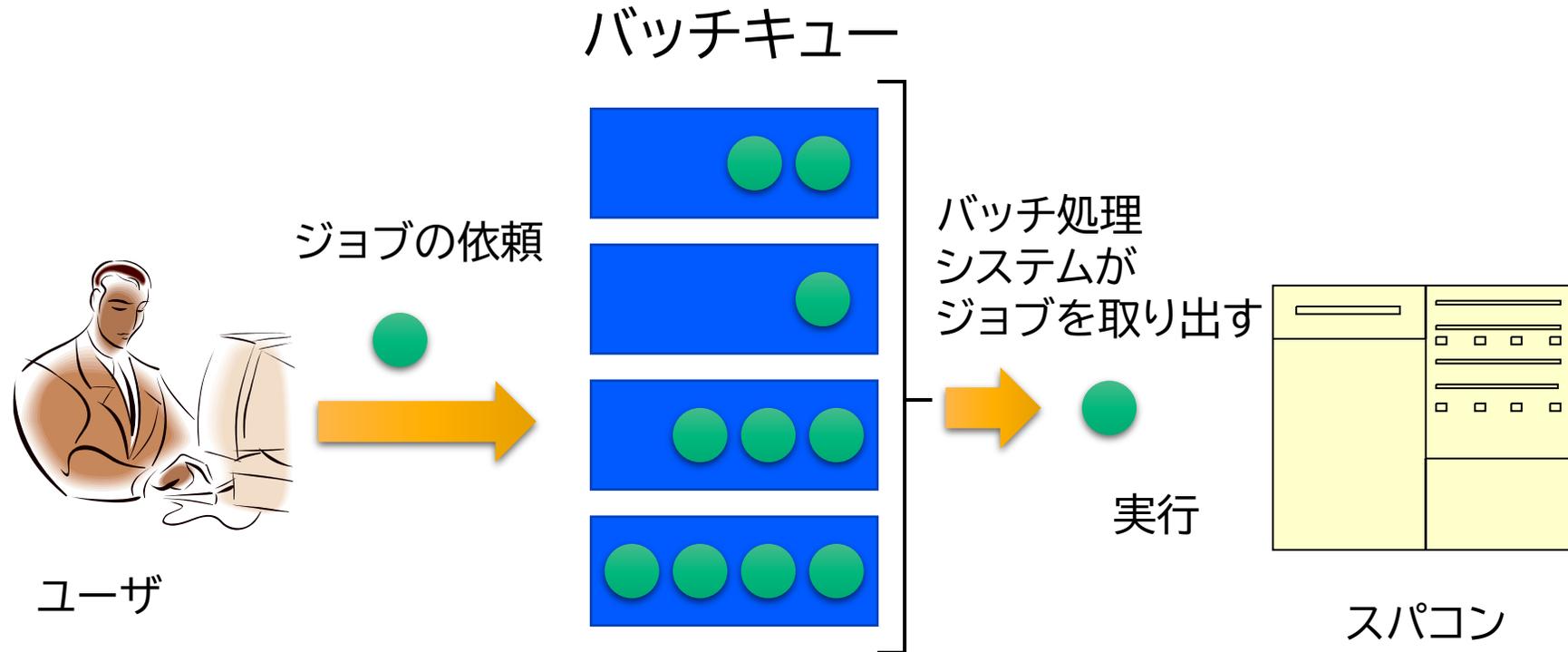
※講習会アカウントでは
バッチジョブ実行のみ,
最大4 GPUs 15分まで

2. インタラクティブジョブ実行

- PCでの実行のように, コマンドを入力して実行
- スパコン環境では一般的ではない
- デバッグ用, 大規模実行はできない
 - 1ノード(48コア): 30分まで
 - 12ノード(576コア): 10分まで

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



バッチキューの設定方法

- Wisteriaでのバッチ処理は, 富士通のバッチシステムで管理
- 主要コマンド:
 - ジョブの投入: `pjsub <ジョブスクリプト名>`
 - 自分が投入したジョブの状況確認: `pjstat`
 - 投入ジョブの削除: `pjdel <ジョブID>`
 - 計算ノードの込み具合を見る: `pjstat --rscuse`
 - バッチキューの状態を見る: `pjstat --rsc`
 - バッチキューの詳細構成を見る: `pjstat --rsc -x`
 - 投げられているジョブ数を見る: `pjstat --rsc -b`
 - 過去の投入履歴を見る: `pjstat -H`
 - 同時に投入できる数・実行できる数を見る: `pjstat --limit`

参考:インタラクティブ実行

- 1ノード実行の場合
\$ pjsub --interact -g グループ名 -L rg=interactive-a,elapsed=01:00
- 1 GPU実行の場合
\$ pjsub --interact -g グループ名 -L rg=share-interactive,elapsed=01:00
- インタラクティブ用のノードが全て使われている場合, 資源が空くまでログインできません
- 本講習会用のアカウントでは使えません

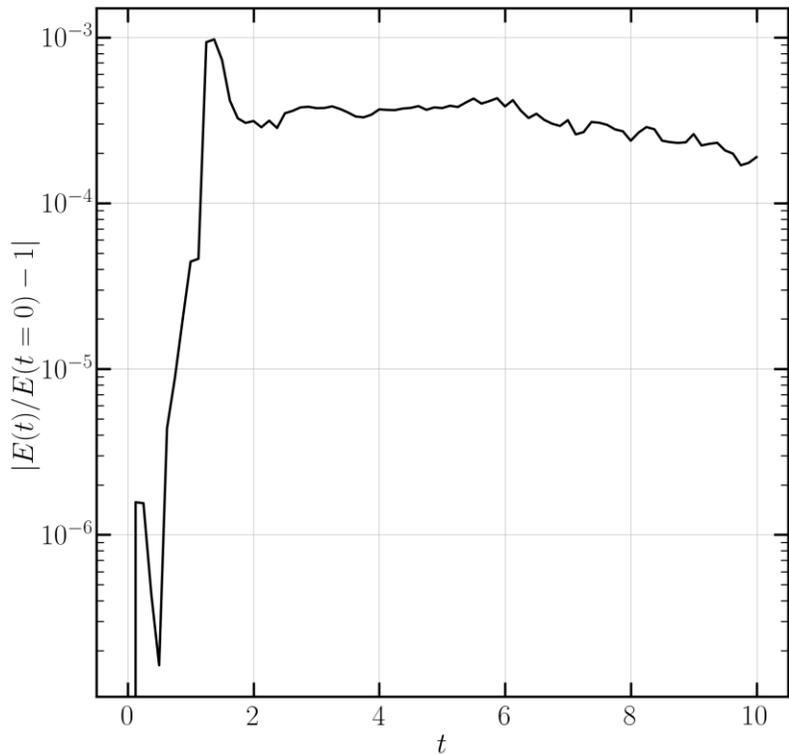
本お試し講習会でのキュー・グループ名

- 本講習会中のキュー名
 - tutorial-a
 - 最大15分まで
 - 最大GPU数は4 GPUs(1ノード内)まで
- 本講習会終了後のキュー名
 - lecture-a
 - 利用条件tutorial-aと同様
- グループ名: gt00

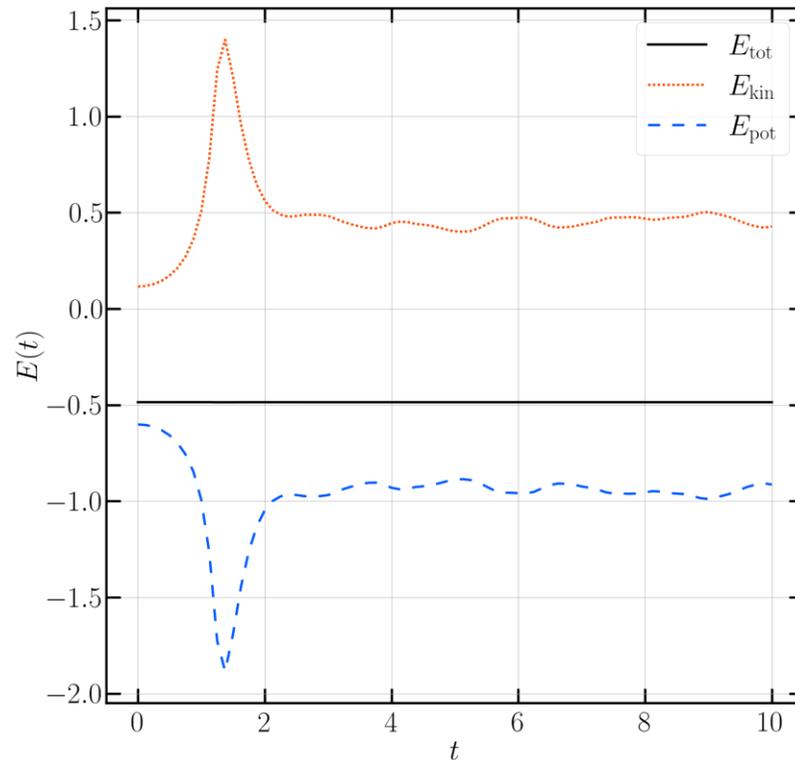
計算結果の可視化例(1/2)

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

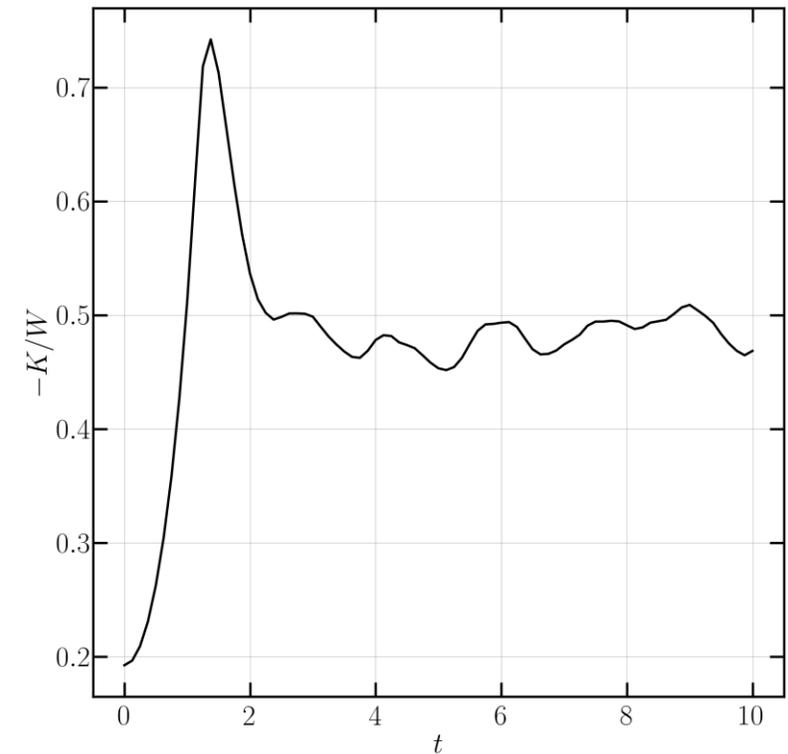
エネルギー保存



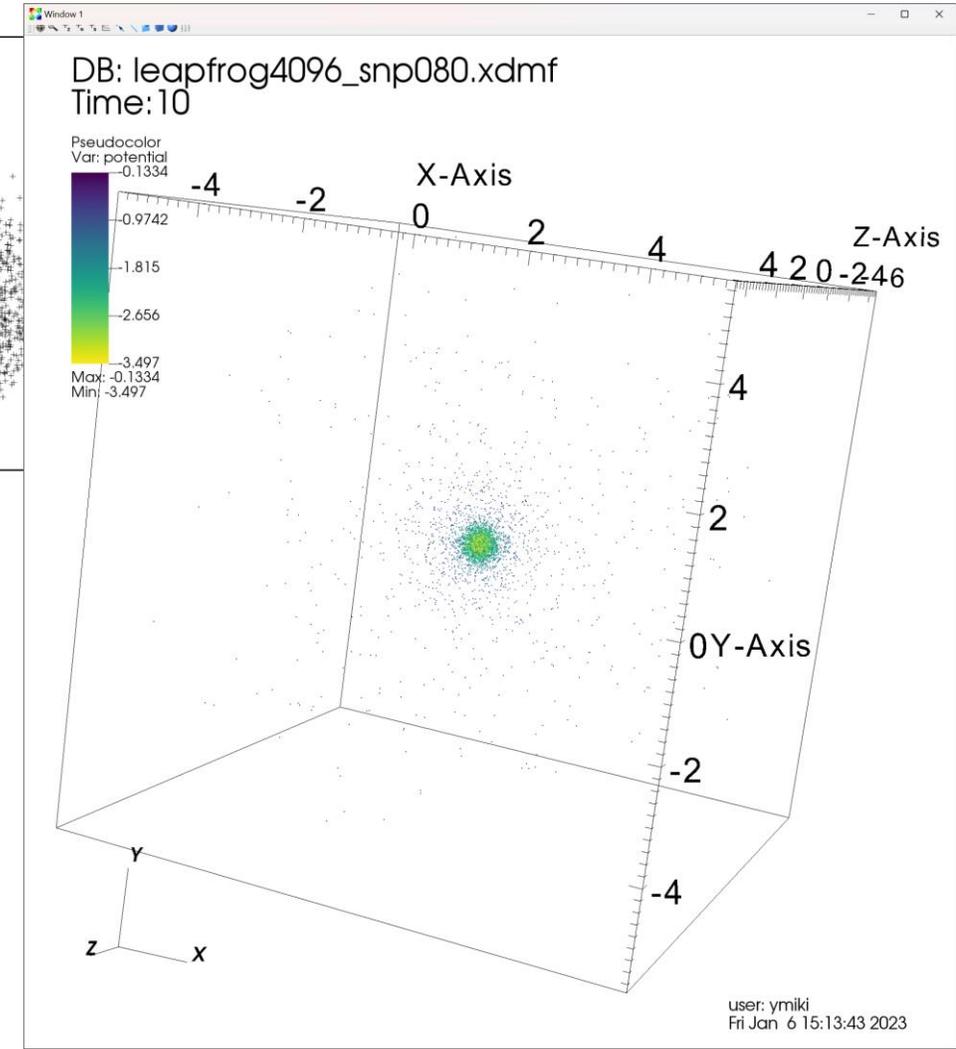
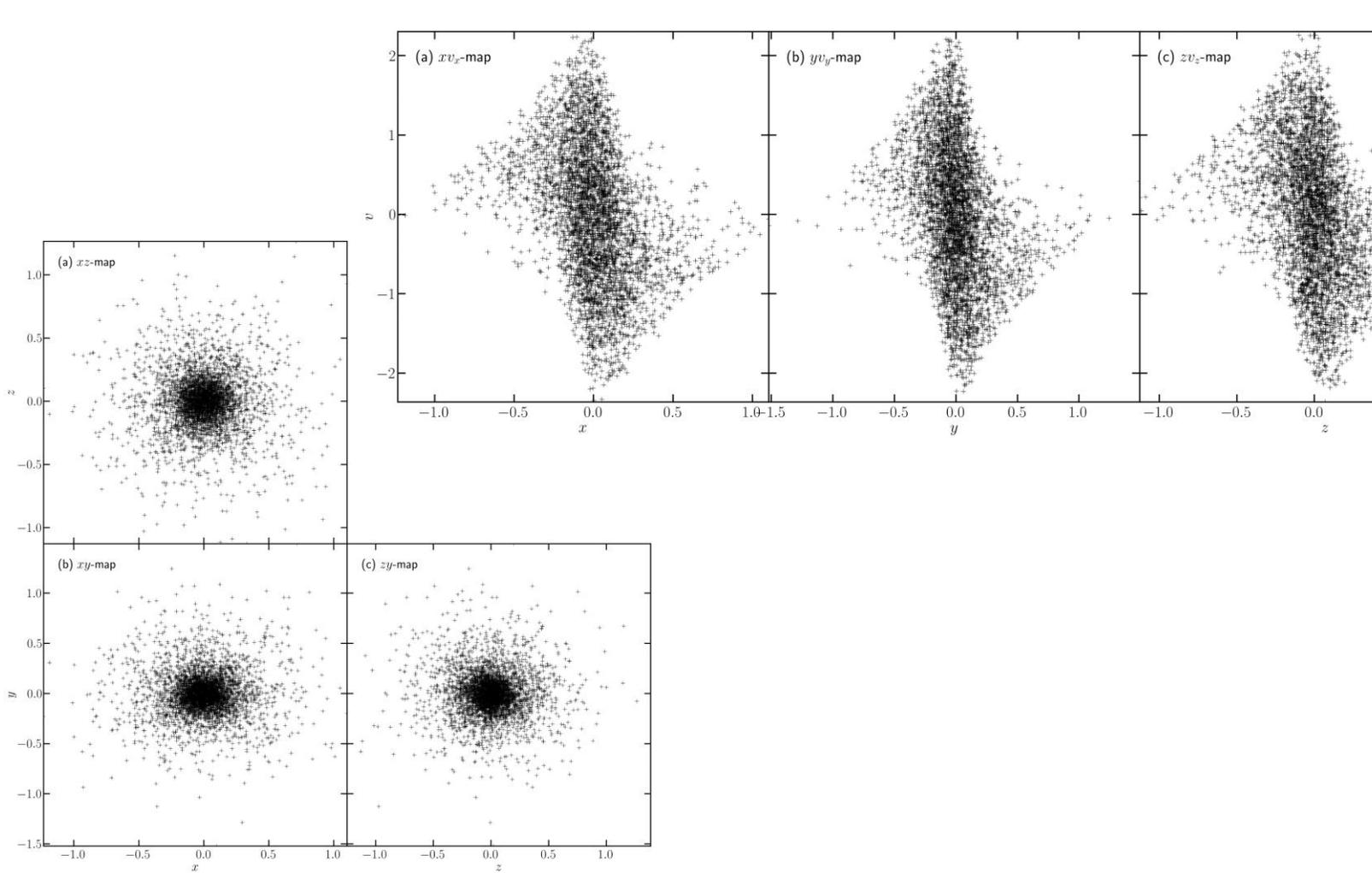
エネルギーの時間進化



ビリアル比の時間進化



計算結果の可視化例(2/2)



OpenACCでの実装(データ転送部分)

- Unified 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])
```

コンパイルと実行

- `$ ninja acc_data`
- `$ pjsub -x EXEC=bin/acc_data sh/wisteria/run_nvidia.sh`
- 出力されるファイル名を変更したければ,
 - `$ pjsub -x EXEC=bin/acc_data,OPTION="--file=NAME" sh/wisteria/run_nvidia.sh`
- 正常に終了すると, 下記ファイルが出力されるはずです
 - `log/collapse_run.csv`
 - エネルギー保存していることを確認(右端2列の数字が 0.005 より十分に小さい)
 - `dat/collapse_snp[000 - 080].[h5 xdmf]`

講習会概略

- 開催日: 2024年8月26日(月) 13:00--17:00
- 形態: ZoomおよびSlackを用いたオンライン講習会
- 使用システム: Wisteria/BDEC-01 (Aquarius)
- 講習会プログラム:
 - 13:00--13:30 GPU向けプログラミング手法の紹介(座学)
 - 13:40--14:20 OpenACCを用いたGPU化(座学+演習)
 - 14:30--15:10 OpenMPを用いたGPU化(座学+演習)
 - 15:20--15:50 C++17を用いたGPU化(座学+演習)
 - 16:00--17:00 CUDA C++を用いたGPU化(座学+演習)

OpenMPを用いてのGPU化

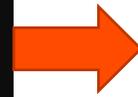
- OpenMP 4.0 以降ではアクセラレータへのオフロードをサポート
- OpenMP 5.0 では loop 指示節が追加
 - OpenACC 的な実装ができる
 - スレッドブロック構造などは全てコンパイラにお任せ
- 政治的にはOpenACCよりも優勢
 - NVIDIA, AMD, Intel が OpenMP でのGPU化をサポート
 - AMD, Intel は(きっと)OpenACCをサポートしない
 - HPE Cray コンパイラであればAMD GPU向けのOpenACCもサポート
- 自分にとってベンダーロックインはどこまで気にすべきか？

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)

```
#pragma omp parallel for
for (type::int_idx ii = 0U; ii < Ni; ii++) {
    // ループ内の実装は省略
}
```

```
#pragma omp parallel for
for (type::int_idx ii = 0U; ii < Ni; ii++) {
    // ループ内の実装は省略
}
```



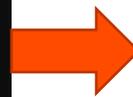
```
#pragma omp target teams distribute parallel for
simd thread_limit(NTHREADS)
for (type::int_idx ii = 0U; ii < Ni; ii++) {
    //ループ内の実装は省略
}
```

```
#pragma omp target teams distribute parallel for
simd
for (type::int_idx ii = 0U; ii < Ni; ii++) {
    //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
 - スレッド数をコンパイラに示唆する方法はない

```
#pragma omp parallel for
for (type::int_idx ii = 0U; ii < Ni; ii++) {
    // ループ内の実装は省略
}
```



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

OpenMPでの実装(データ転送部分)

- Unified 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])
```

OpenMPコードのコンパイル方法など

- NVIDIA HPC SDK向けの情報
- `$ 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,managed -Minfo=accel,opt,mp`
 - Unified Memory使用時のコンパイル方法
 - リンク時にも `-mp=gpu -gpu=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++ -target x86_64-pc-linux-gnu -fopenmp -fopenmp-targets=amdgcN-amd-amdhsa -Xopenmp-target=amdgcN-amd-amdhsa -march=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"`

コンパイルと実行

- `$ ninja omp_dist omp_dist_data omp_loop omp_loop_data`
- `$ pjsub -x EXEC=bin/omp_dist sh/wisteria/run_nvidia.sh` # 他
の実行ファイルも同様
- 出力されるファイル名を変更したければ,
 - `$ pjsub -x EXEC=bin/omp_dist,OPTION="--file=NAME" sh/wisteria/run_nvidia.sh`
- 正常に終了すると, 下記ファイルが出力されるはずですよ
 - `log/collapse_run.csv`
 - エネルギー保存していることを確認(右端2列の数字が 0.005 より十分に小さい)
 - `dat/collapse_snp[000 - 080].[h5 xdmf]`

おまけ: OpenACC/OpenMP統合マクロを整備中

- GPU向け指示文統合マクロSolomon (三木 & 埜, SWoPP 2024)
 - Simple Off-Loading Macros Orchestrating multiple Notations
 - プリプロセッサマクロ経由で指示文を記載するためのマクロ集
 - NVIDIA GPU上ではOpenACCを, AMD/Intel GPU上ではOpenMP targetを選択, ということができる
 - 簡易記法, OpenACC的記法, OpenMP的記法があるため, 学習コストを低減
 - GPU提供ベンダー製のコンパイラをそのまま使える
 - OpenACC と OpenMP target の性能比較も簡単にできる
- 現在は公開に向けてドキュメントを整備中
 - (公開より早くに)使いたいという要望があれば, 個別対応するのでご連絡ください

```
void calc_acc(...) {  
  OFFLOAD(AS_INDEPENDENT, NUM_THREADS(NTHREADS))  
  for (std::remove_const_t<decltype(Ni)> i = 0; i < Ni; i++) {  
    // 初期化 (省略)  
    PRAGMA_ACC_LOOP(ACC_CLAUSE_SEQ)  
    for (std::remove_const_t<decltype(Nj)> j = 0; j < Nj; j++) {  
      // ループ内は省略
```

```
set_uniform_sphere(num, pos, vel,  
Mtot, rad, virial, newton);  
MEMCPY_H2D(pos [0:num], vel [0:num])  
calc_acc(num, pos, acc, num, pos,  
eps);
```

講習会概略

- 開催日: 2024年8月26日(月) 13:00--17:00
- 形態: ZoomおよびSlackを用いたオンライン講習会
- 使用システム: Wisteria/BDEC-01 (Aquarius)
- 講習会プログラム:
 - 13:00--13:30 GPU向けプログラミング手法の紹介(座学)
 - 13:40--14:20 OpenACCを用いたGPU化(座学+演習)
 - 14:30--15:10 OpenMPを用いたGPU化(座学+演習)
 - 15:20--15:50 C++17を用いたGPU化(座学+演習)
 - 16:00--17:00 CUDA C++を用いたGPU化(座学+演習)

C++17 Parallel Algorithms版の実装

- `#pragma omp parallel for` を置き換えていく
- GPU向けに特化した実装方法ではない
 - まったく同じコードをマルチコアCPU向けにコンパイルできる
 - CUDA実装を混ぜることもできるが, 上記メリットが消えるので非推奨
 - 現時点では先進的(実験的)な部分もある組み合わせなので, 人柱精神の強い人でないならば様子見で良い(というのが個人的なコメント)

```
#pragma omp parallel for
for (type::int_idx ii = 0U; ii < Ni; ii++) {
    // ループ内の実装は省略
}
```



```
#include <algorithm>
#include <execution>
#include <boost/iterator/counting_iterator.hpp>

std::for_each_n(std::execution::par,
boost::iterators::counting_iterator<type::int_idx>(
0U), Ni, [=](const type::int_idx ii) {
    // for文の中身はそのままOK
});
```

標準言語での並列化に関する補足

- 前ページの例では, `counting_iterator` を Boost C++ Libraries から呼んだが, もちろんこれ以外の実装も可能
 - 自分で実装してもOK(面倒だが..)
 - NVIDIA HPC SDKの場合は, `thrust`から呼んでもOK
 - コンパイラがNVIDIA縛りになるので, 標準言語の機能を使う利点は減るが..
- 並列版実装が提供されているもの(`std::sort`, `std::reduce`など)については, 実行ポリシーを指定することで並列版アルゴリズムを実行
 - `std::execution::seq` (逐次処理)
 - `std::execution::par` (マルチスレッド化を許可)
 - `std::execution::par_unseq` (マルチスレッド化 and/or ベクトル化を許可)
- スレッド数を示唆する方法, データ指示文的な記法は存在しない
 - この手法は, GPU実装に特化したプログラミング手法ではない
 - Unified Memory を使うことが前提となっている

標準言語でのコンパイル方法

- NVIDIA HPC SDK向けの情報
- `$ nvc++ -stdpar=gpu -gpu=cc80 -Minfo=accel,opt,stdpar`
 - リンク時にも `-stdpar=gpu` を指定する
- `$ nvc++ -stdpar=multicore -Minfo=accel,opt,stdpar`
 - マルチコアCPU向けにコンパイルする場合

コンパイルと実行

- `$ ninja stdpar`
- `$ pjsub -x EXEC=bin/stdpar sh/wisteria/run_nvidia.sh`
- 出力されるファイル名を変更したければ,
 - `$ pjsub -x EXEC=bin/stdpar,OPTION="--file=NAME" sh/wisteria/run_nvidia.sh`
- 正常に終了すると, 下記ファイルが出力されるはずです
 - `log/collapse_run.csv`
 - エネルギー保存していることを確認(右端2列の数字が 0.005 より十分に小さい)
 - `dat/collapse_snp[000 - 080].[h5 xdmf]`

Fortranユーザ向けに

- Fortran 2008 から, 標準言語としての並列化がサポート
 - C++17 の場合と同様に, nvfortran ではGPU化の対象
- do文をdo concurrentに直して並列化可能であると明記
 - C++17 の場合よりも書き換えコストはかなり低い
- 各種制限などはC++とFortranで大きく違う
- Unified Memoryが前提なのはC++17の場合と同じ
- 参考資料
 - <https://www.cc.u-tokyo.ac.jp/events/lectures/211/>
 - OpenACC/OpenMP などについても説明されています
 - 次回は9/3開催, 申込締切は8/27 17:00

各実装手法間での性能比較を試みる

- `$ cmake -S ..`
 - `BENCHMARK_MODE=ON` に変更(当該行で Enter を入力)
 - `[c]onfigure` → `[g]enerate`
- `$ ninja`
- `$ pjsub -x EXEC=bin/acc_data,OPTION="--num_min=2097152 num_max=2097152 --num_bin=1 --file=benchmark" sh/wisteria/run_nvidia.sh`
 - 同様に `omp_dist_data`, `omp_loop_data`, `stdpar` も実行
- 性能を確認・比較してみよう
 - `log/benchmark_run.csv` の右端から3列目が演算性能
 - `$ cmake -S ..` から `RELAXED_RSQRT_ACCURACY=ON` すると？
 - `$ cmake -S ..` から `OVERWRITE_DEFAULT=ON` にしたうえで `NTHREADS` の値を切り替えるとどうなる？

実習環境の理論ピーク演算性能

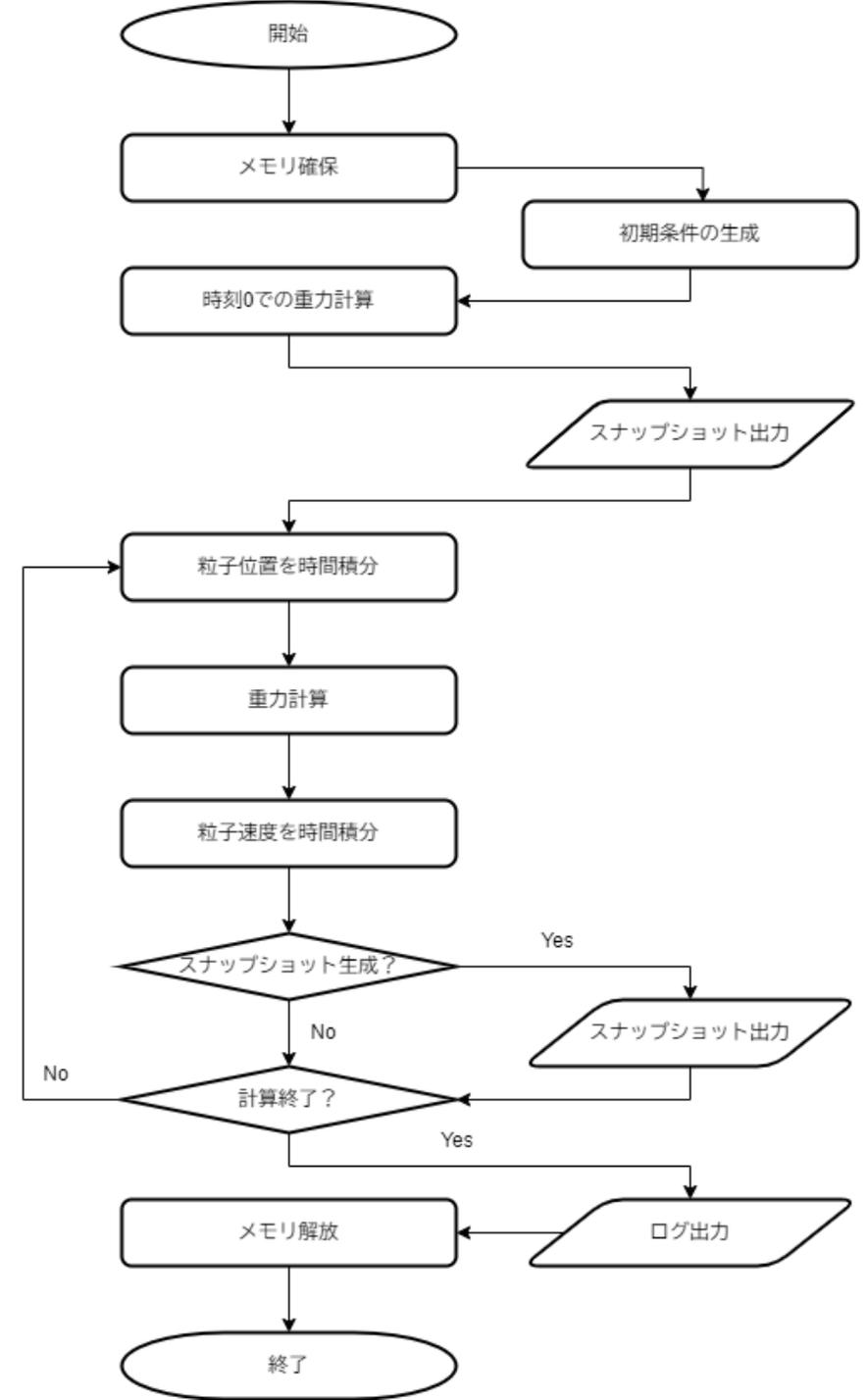
- NVIDIA A100 (SXM, 40GB)
 - 108 SMs (32 FP64 cores per SM, 64 FP32 cores per SM)
 - GPU Boost Clock: 1410 MHz
 - 注: テンソルコア (FP64には対応) はN体では使えないので考慮しない
 - FP64: 9.7 TFlop/s
 - FP32: 19.5 TFlop/s
- 理論ピーク性能はFMA (fused multiply-add) 命令が前提なので, N体計算で実際にこの性能が出ることはない
- 実習コードでの仮定 (逆数平方根が4 Flops相当と仮定)
 - Leapfrog: 24 Flops per interaction
 - Hermite: 46 Flops per interaction

自分でも実装してみる

- 対象のコード: `cpp/base/0_base` をコピーしただけのもの
 - OpenACC: `cpp/openacc/exercise/nbody_*.cpp`
 - OpenMP: `cpp/openmp/exercise/nbody_*.cpp`
- 実装方針:
 1. まずは Unified Memory を前提にして計算部分だけGPU化
 2. 次に, Unified Memory を外して実装
 - `cpp/[openacc openmp]/exercise/CMakeLists.txt` の中で, `managed` とある部分(2ヶ所)を編集
 - `-gpu=cc${NVIDIA_GPU_HARD},managed` → `-gpu=cc${NVIDIA_GPU_HARD}`
 - `-gpu=managed` とある行を削除
- `$ cmake -S ..` から `EXERCISE_MODE=ON` して `ninja`
 - OpenACC: `$ ninja acc_exercise`
 - OpenMP: `$ ninja omp_exercise`

フローチャート

- 右図はLeapfrog法の場合
 - Hermite法も大まかな流れは同様だが、時間積分が予測子・修正子法になっており、また粒子時刻によるソート処理も追加される
- 初期条件生成, ファイル出力部分はGPU化しなくてOK
 - 注:最新の粒子データはGPU上にあるため、スナップショット出力の直前にCPU上のデータを更新する作業は必要(データ指示文を使った実装の場合)
- 性能測定モードにおいては、重力計算だけを繰り返す実装になっている

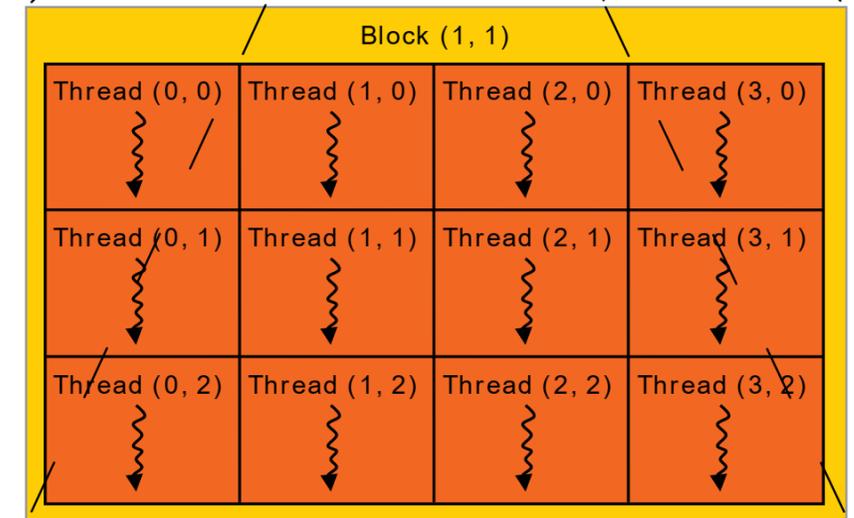
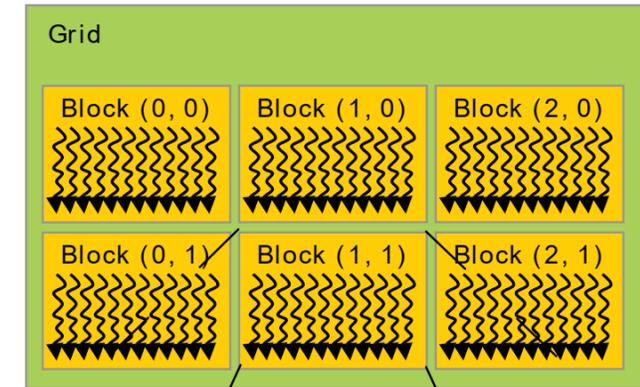


講習会概略

- 開催日: 2024年8月26日(月) 13:00--17:00
- 形態: ZoomおよびSlackを用いたオンライン講習会
- 使用システム: Wisteria/BDEC-01 (Aquarius)
- 講習会プログラム:
 - 13:00--13:30 GPU向けプログラミング手法の紹介(座学)
 - 13:40--14:20 OpenACCを用いたGPU化(座学+演習)
 - 14:30--15:10 OpenMPを用いたGPU化(座学+演習)
 - 15:20--15:50 C++17を用いたGPU化(座学+演習)
 - 16:00--17:00 CUDA C++を用いたGPU化(座学+演習)

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)意識しないが良い
- 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++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<<<block, thrd>>>()`)
- 確保したメモリの開放(`cudaFree()`, `cudaFreeHost()`)

簡易GPU化との性能比較

- OpenACCでCUDAの8-9割を目指す or 達成とは言いますが、
 - 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
- OpenACC/OpenMP/C++17 で実装したコードと, CUDA C++で実装したコードとの性能差を確認
 - 今回の講習会ではCUDA C++全力版までは紹介しませんが, その少し手前ぐらいのコードとの比較まで行います

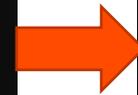
簡易GPU化との思想の違い

- 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 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
 - 同一流域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

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(ii < 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の粒子を置いておけば計算結果は正しい
 - (ツリー法の場合にはもう一工夫必要だが, 細かい話なので省略)

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



```
__global__ void calc_acc_device(const
type::position *const ipos, ...) {
  const type::int_idx ii = 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)

- Unified Memory を使用する場合
 - メモリの確保・解放だけ記述すればOK
 - (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 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 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`

コンパイル方法

- `$ cd ../` # 大元の CMakeLists.txt のあるフォルダに移動
- `$ module purge` # load済みのmoduleを一旦全て外す
- `$ module load cmake cuda gcc hdf5`
- `$ cmake -DUSE_CUDA=ON -S . -B build_cuda`
- `$ cd build_cuda`
- `$ cmake -S ..` # デフォルトの設定を調整したい場合
 - 倍精度で計算したければ, `FP_L`, `FP_M`を64に変更
 - Hermite法で試したい方は, `HERMITE_SCHEME`をONに変更
- `$ ninja cuda_unified_base cuda_memcpy_base`
 - 単に `$ ninja` だけでもOKですが, 他にもたくさんのビルドが走ります

実行

- `$ pjsub -x EXEC=bin/cuda_unified_base sh/wisteria/run_cuda.sh`
- `$ pjsub -x EXEC=bin/cuda_memcpy_base sh/wisteria/run_cuda.sh`
- 出力されるファイル名を変更したければ,
 - `$ pjsub -x EXEC=bin/cuda_unified_base,OPTION="--file=NAME" sh/wisteria/run_cuda.sh`
- 正常に終了すると, 下記ファイルが出力されるはずです
 - `log/collapse_run.csv`
 - エネルギー保存していることを確認(右端2列の数字が 0.005 より十分に小さい)
 - `dat/collapse_snp[000 - 080].[h5 xdmf]`

CUDA C++コードの最適化

- 最適なスレッド数の選択
 - 後で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
 - これはCPUコードの場合も同じ
- 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)
 - NVHPCで `RELAXED_RSQRT_ACCURACY=ON` すると速くなる理由
(`nvc++`のコンパイルオプションに `-Mfprelaxed=rsqrt` 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では, 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)
- 命令レベルの並列性の導入(今回はここまではやりません)

コンパイルと実行

- `$ ninja cuda_unified_rsqrt cuda_memcpy_rsqrt`
- `$ pjsub -x EXEC=bin/cuda_unified_rsqrt sh/wisteria/run_cuda.sh` # 他の実行ファイルも同様
- 出力されるファイル名を変更したければ,
 - `$ pjsub -x EXEC=bin/cuda_unified_rsqrt,OPTION="--file=NAME" sh/wisteria/run_cuda.sh`
- 正常に終了すると, 下記ファイルが出力されるはずです
 - `log/collapse_run.csv`
 - エネルギー保存していることを確認(右端2列の数字が 0.005 より十分に小さい)
 - `dat/collapse_snp[000 - 080].[h5 xdmf]`

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 `__syncthreads()` などをつけて制御する
- 実行性能はループアンローリングの段数にも依存

```
__global__ void calc_acc_device(...) {  
    // 初期化部分省略  
    __shared__ type::position jpos_shmem[NTHREADS];  
  
    for (type::int_idx jh = 0U; jh < Nj; jh += NTHREADS) {  
        // load j-particle  
        const auto pj_tmp = jpos[jh + threadIdx.x];  
        __syncthreads();  
        jpos_shmem[threadIdx.x] = pj_tmp;  
        __syncthreads();  
  
        PRAGMA_UNROLL  
        for (type::int_idx jj = 0U; jj < NTHREADS; jj++) {  
            const auto pj = jpos_shmem[jj];  
            // 以下省略  
        }  
    }  
    iacc[ii] = ai;  
}
```

コンパイルと実行

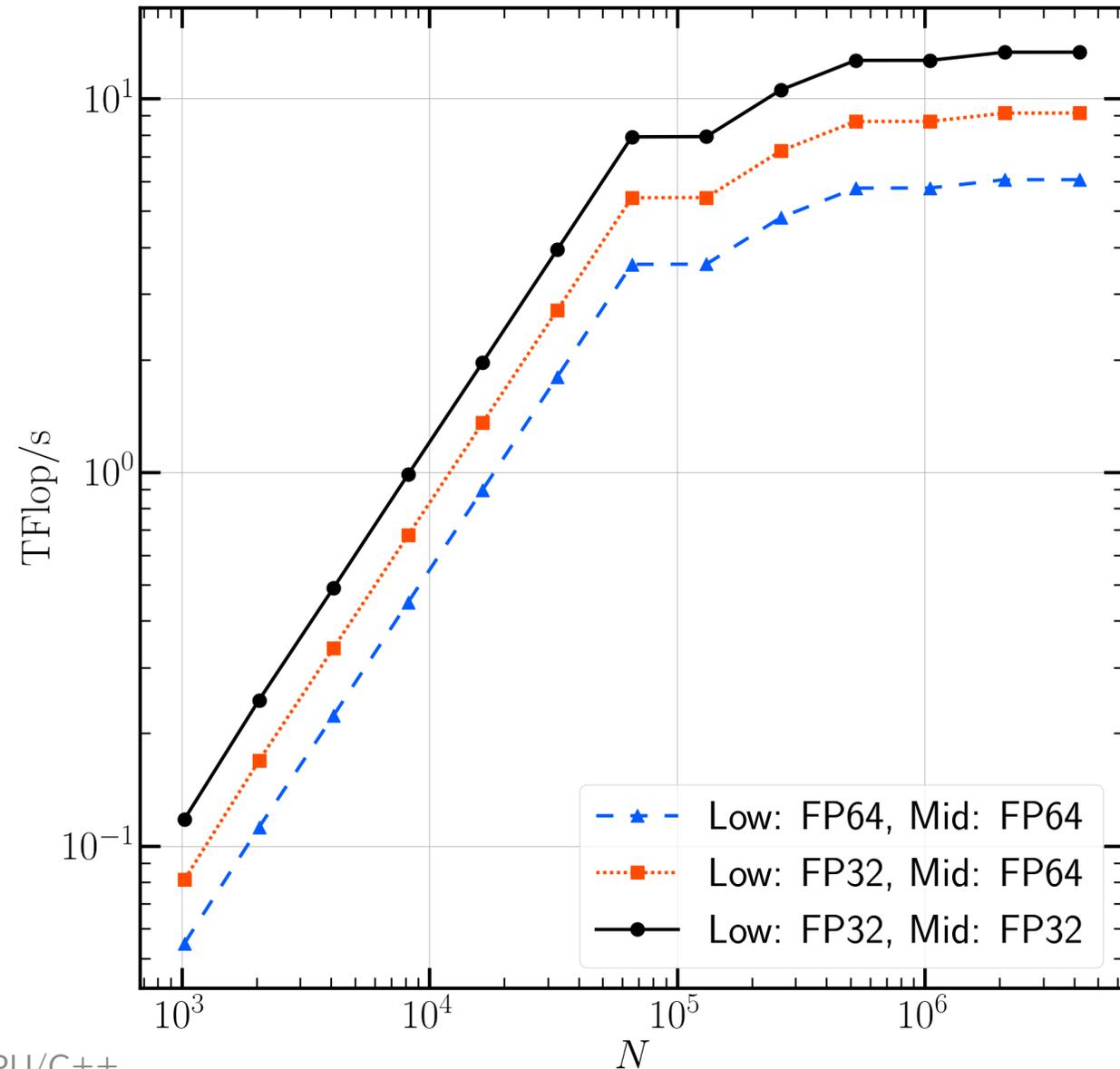
- `$ ninja cuda_unified_shmem cuda_memcpy_shmem`
- `$ pjsub -x EXEC=bin/cuda_unified_shmem sh/wisteria/run_cuda.sh` # 他の実行ファイルも同様
- 出力されるファイル名を変更したければ,
 - `$ pjsub -x EXEC=bin/cuda_unified_shmem,OPTION="--file=NAME" sh/wisteria/run_cuda.sh`
- 正常に終了すると, 下記ファイルが出力されるはずです
 - `log/collapse_run.csv`
 - エネルギー保存していることを確認(右端2列の数字が 0.005 より十分に小さい)
 - `dat/collapse_snp[000 - 080].[h5 xdmf]`

各実装手法間での性能比較を試みる

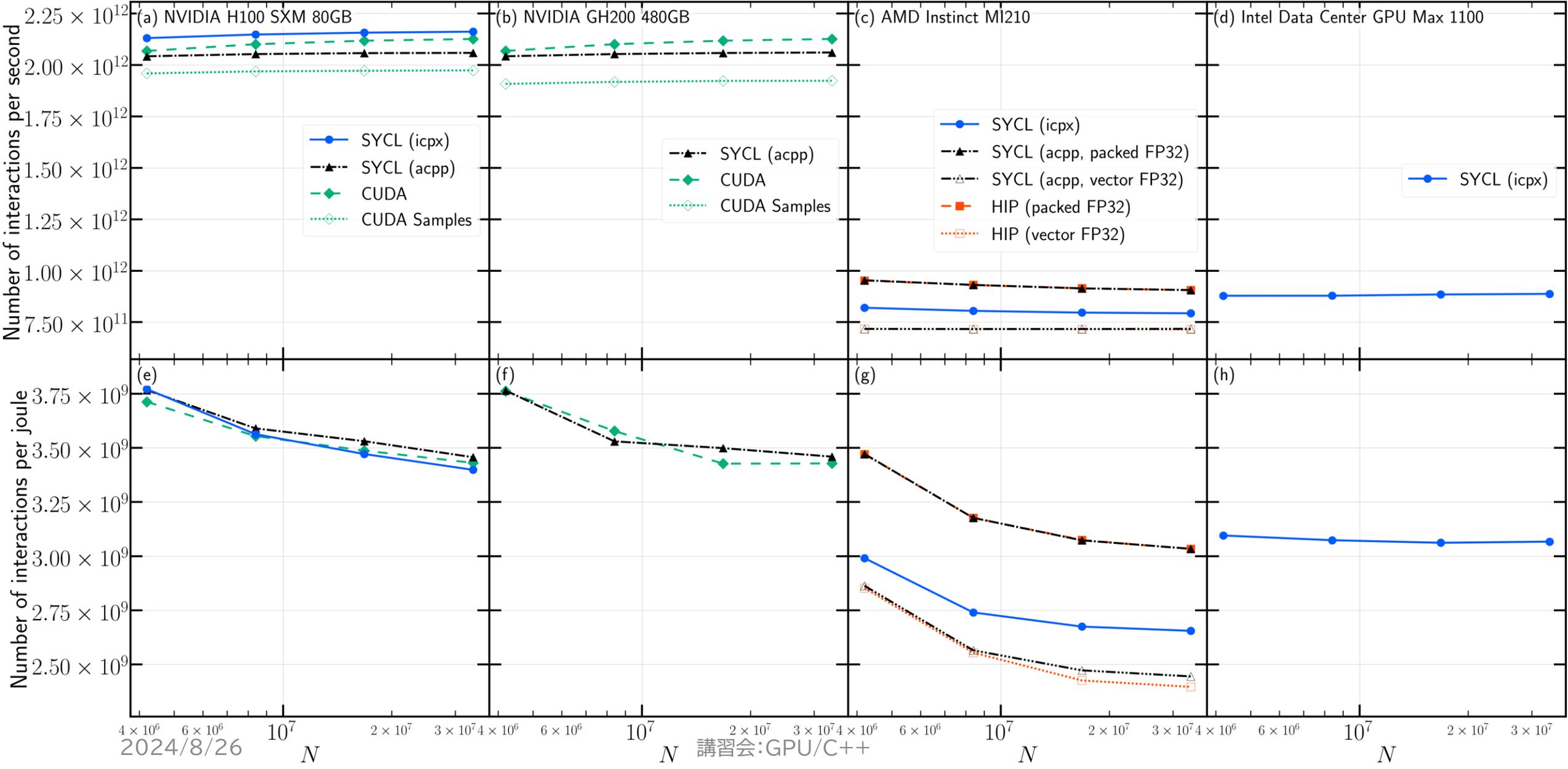
- `$ cmake -S ..`
 - `BENCHMARK_MODE=ON` に変更(当該行で Enter を入力)
 - `[c]onfigure` → `[g]enerate`
- `$ ninja`
- `$ pjsub -x EXEC=bin/cuda_memcpy_base,OPTION="--num_min=2097152 num_max=2097152 --num_bin=1 --file=benchmark" sh/wisteria/run_cuda.sh`
 - 同様に `cuda_memcpy_rsqr`, `cuda_memcpy_shmem` も実行
- 性能を確認・比較してみよう
 - `log/benchmark_run.csv` の右端から3列目が演算性能
 - 簡易GPU化したコードとの性能比は？
 - `$ cmake -S ..` から `OVERWRITE_DEFAULT=ON` にしたうえで `NTHREADS` の値を切り替えるとどうなる？

CUDA版コードの性能の粒子数依存性

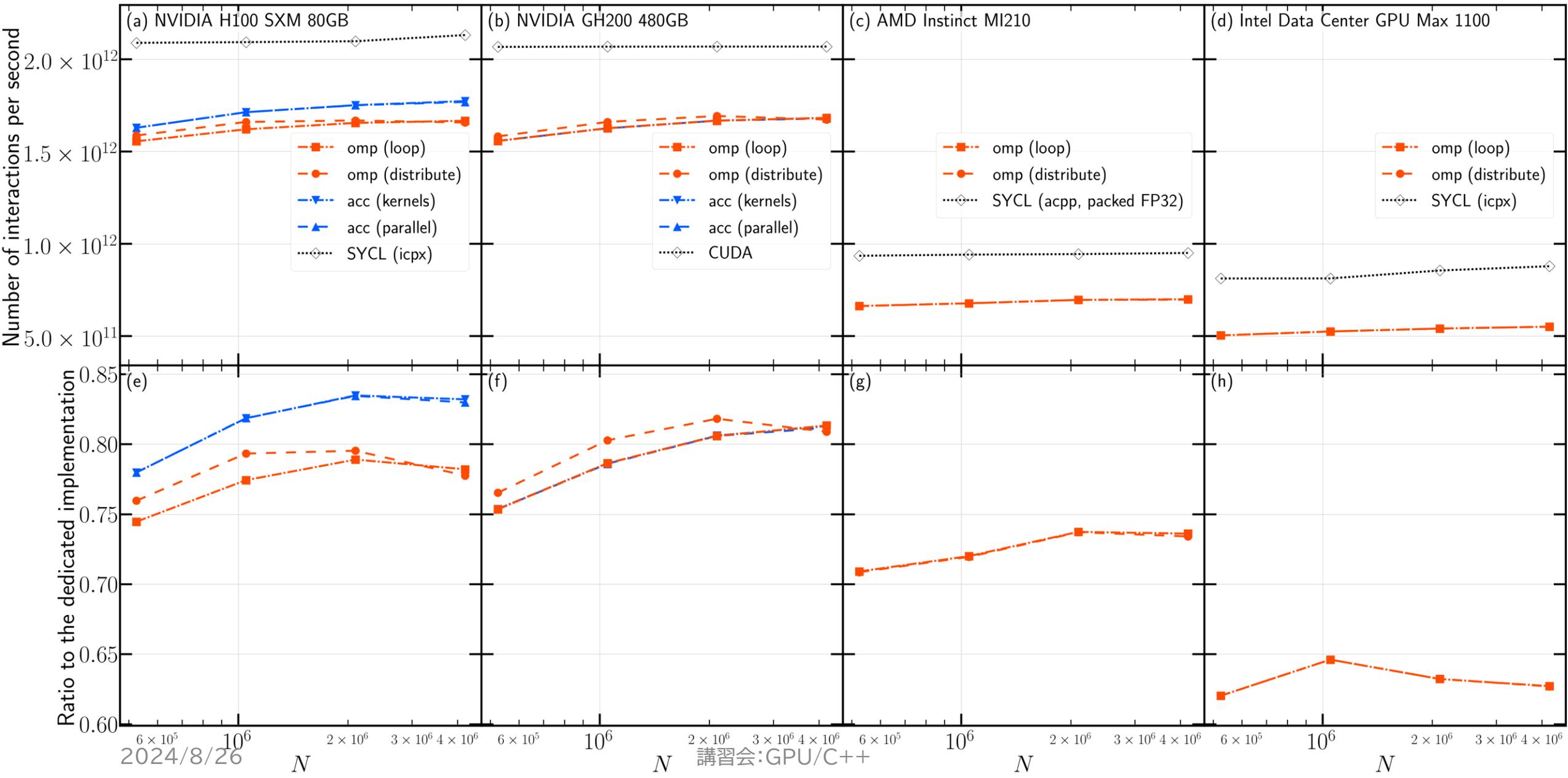
- Wisteria/BDEC-01 (Aquarius) 上の測定結果
 - NVIDIA A100 (SXM, 40GB)
 - CUDA 11.4
 - Leapfrog法向け実装
- 全てのSMが埋まるまでは、性能は粒子数 N に比例
 - コア数の10倍程度の粒子数が必要
- 13.6 TFlop/s (FP32) が最高
 - 全力版は 14.7 TFlop/s まで行く



参考：NVIDIA/AMD/Intel製GPUの性能比較



参考：指示文実装との性能比較



2024/8/26

講習会: GPU/C++

自分でも実装してみる

- 対象のコード: `cpp/base/0_base` をコピーしたただけのもの
 - `cpp/cuda/exercise/nbody_*.cu`
- 実装方針:
 1. まずは Unified Memory を前提にして計算部分だけGPU化
 2. 次に, Unified Memory を外して実装
- `$ cmake -S ..` から `EXERCISE_MODE=ON` して `make`
 - `$ ninja cuda_exercise`

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
- `$ 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)

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でのワークの挙動
- 最近のGPU上で(CUDAから)使える命令・機能
- 複数GPU実装に向けての情報

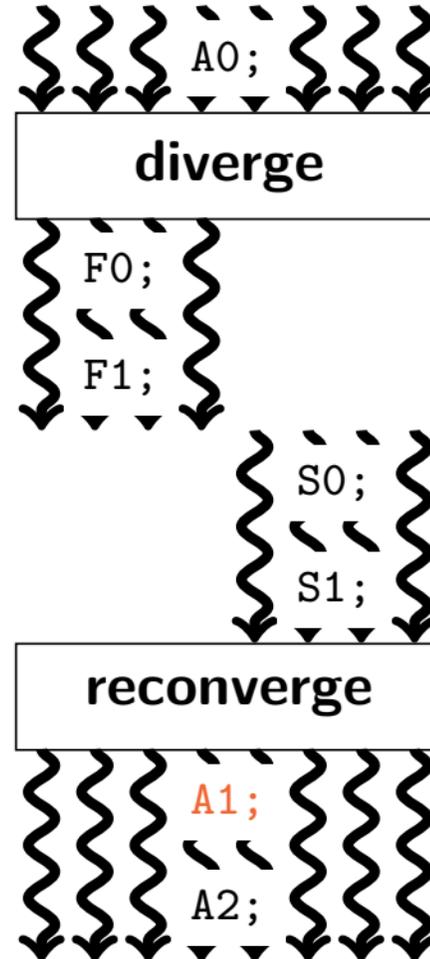
Independent thread scheduling

Pseudocode

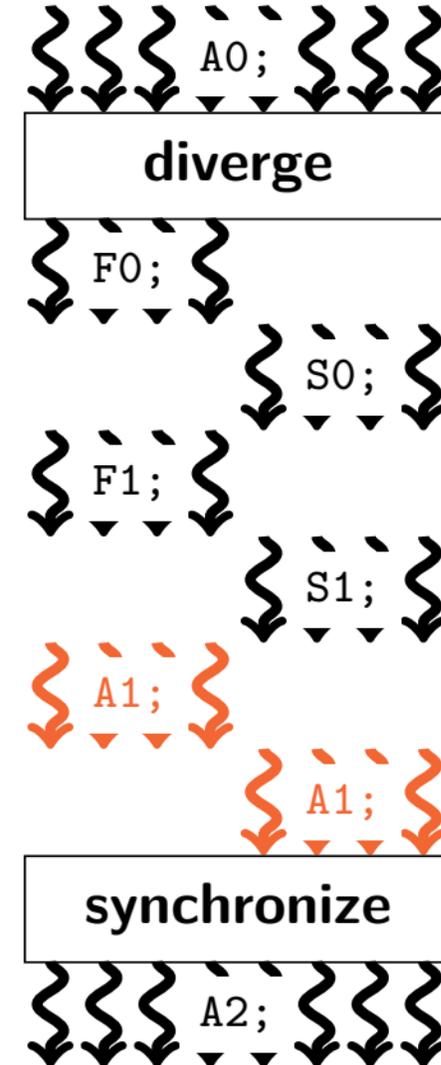
```

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

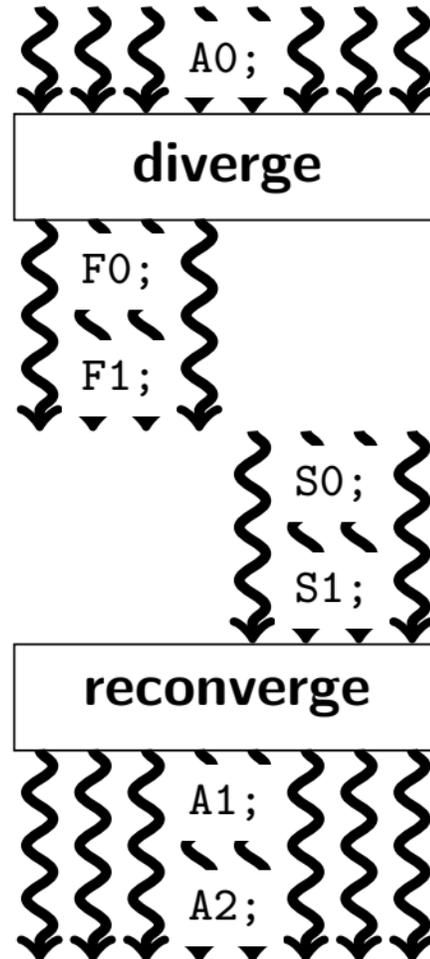
Pseudocode

```

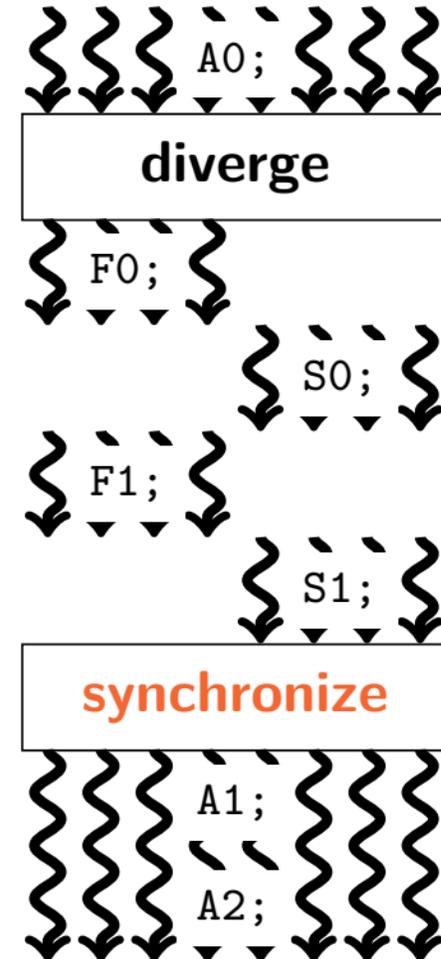
A0;
if( threadIdx.x < 4 ){
    F0;
    F1;
} else {
    S0;
    S1;
}
#if __CUDA_ARCH__ >= 700
__syncwarp();
#endif
A1;
A2;

```

Pascal or earlier



Volta or later



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)
 - Volta: コンパイル時に `arch=compute_60,code=sm_70` を指定
 - Ampere: `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)

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

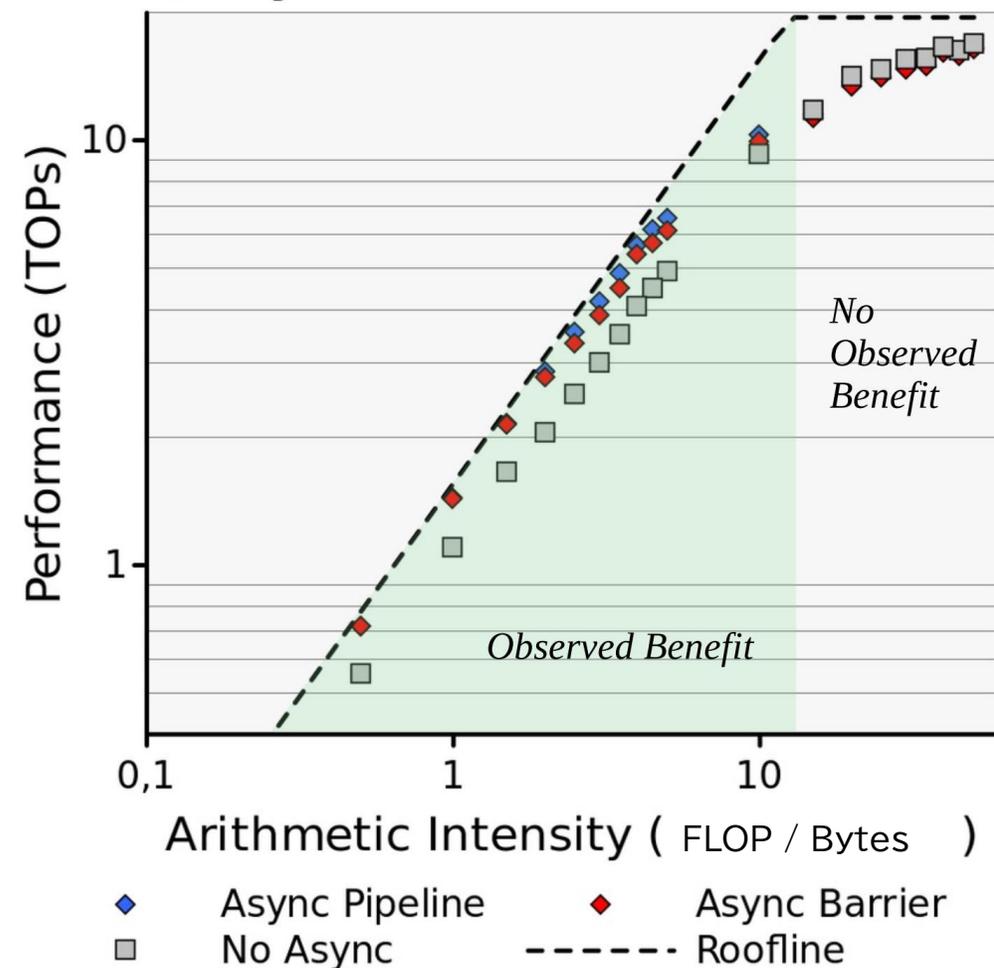
- テンソルコア関連の機能
 - Sparsity: 小行列の中に0が入っていたらその分をスキップして高速化
 - TF32: 仮数部10ビット, 指数部8ビットのデータ型(実態はFP19)
- **Asynchronous copy/barrier**
 - グローバルメモリからシェアドメモリに直接(レジスタを介さずに)データを置ける
- **Warp-wide reduction**
 - ハードウェア的な高速化が効く
- **L2 cache residency control**
 - (1つの連続な)データ領域をL2キャッシュに固定しておく

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

(a) Asynchronous on Roofline



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

複数GPU実装に関する情報提供

- CUDA-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よい
- 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を介さない)直接通信もできる
 - Open MPI (w/ UCX) や, MVAPICH2 GDR
 - Unified Memory を使っていると, GDR など使ってくれない
 - 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 Memory の弱点の一つ

環境変数 CUDA_VISIBLE_DEVICES

- 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

Miyabi (OFP-II) (1/2)

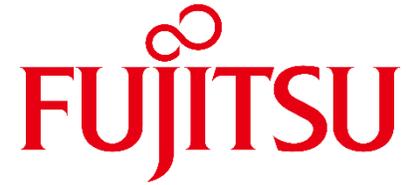
• Miyabi-G: CPU+GPU: NVIDIA GH200

- 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

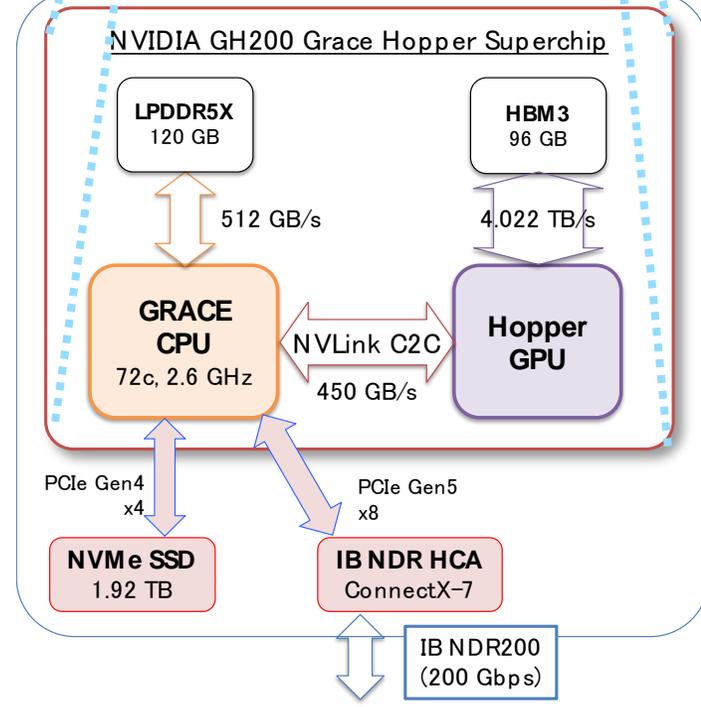
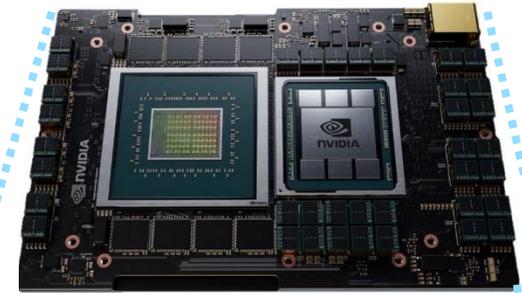
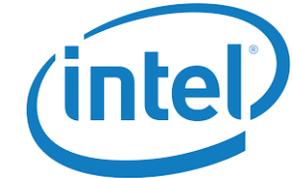
- **Total (Aggregated Performance: CPU+GPU)**
 - 1,120 nodes, 78.8 PF, 5.07 PB/sec, IB-NDR 200

• 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)

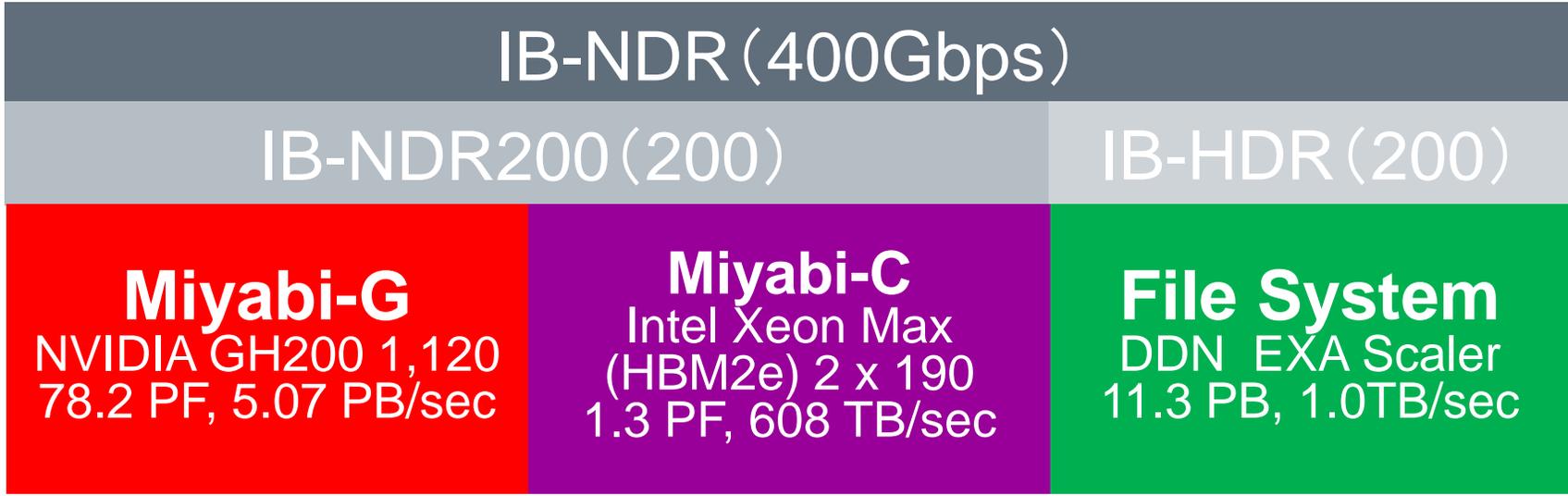
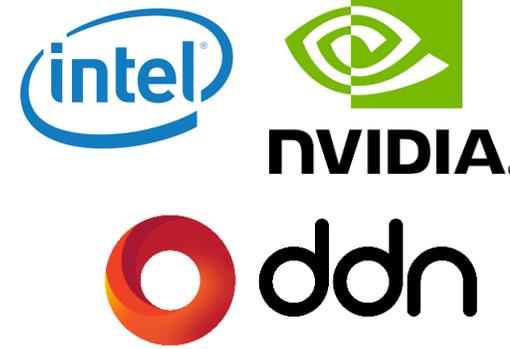


nVIDIA®



Miyabi (OFP-II) (2/2)

- ファイルシステム: DDN EXA Scaler, 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**
 - $(400\text{Gbps}/8) \times (32 \times 20 + 16 \times 1) = 32.8 \text{ TB/sec}$
- **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



Ipomoea-01
Common Shared Storage
26 PB



最後に(1/2)

- 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
 - 2025年1月稼働予定のMiyabi (OFP-II) by JCAHPCもGPUが主体
- 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
 - 今日紹介していないものではKokkos, RAJA などのフレームワークも
 - OpenACC/OpenMP targetを統合するライブラリも作成中
- 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
 - GPU向けにアルゴリズムを設計などと言い出すと話は別だが, その場合はCPU向けであっても大変なはず
- 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/

最後に(2/2)

- アンケートの回答をお願いします
 - 講習会の改善のために有用な情報なので、ご協力お願いします
- 本講習会アカウントは、9/26(木) 9:00まで使えます
 - キュー名はlecture-aです
(tutorial-aは8/26の17:00以降使えなくなります)
 - 最大15分まで
 - 最大GPU数は4 GPUまで
- 本講習会関連の質問はymiki[at]cc.u-tokyo.ac.jpまで
 - Slack で質問していただいても結構です
 - (講習会アカウントでは)公式の相談対応システムは使わないでください