2026年度GPU講習会（1日目）

三木 洋平
（東京大学 情報基盤センター）

2026年度GPU講習会@国立天文台三鷹キャンパス

初日(8/18)の予定

- 10:00--11:00　並列計算の初歩　[講義]
- 11:15--12:15　GPUコンピューティングの基礎概念　[講義]
- 13:30--14:30　簡単なCUDAコードの実装　[講義・演習]
- 14:45--15:45　簡易なプロファイラと性能評価　[講義・演習]
- 16:00--17:00　GPU移植時の罠　[講義・演習]

- GPUとはどんなものか, どういうことを考えれば良いのかつかむことが目的
  - AIコーディングする際に適切な指示出しをするために分かっているべき情報

2日目(8/19)の予定

- 9:00--9:45　　前日の復習
- 10:00--11:00　　無衝突系N体コードのCUDA移植1　　[講義・演習]
- 11:15--12:15　　無衝突系N体コードのCUDA移植2　　[演習]
- 13:30--14:30　　ライブラリの利用　　[講義・演習]
- 14:45--15:45　　衝突系N体コードのCUDA移植　　[講義・演習]
- 16:00--17:00　　CUDAコードの性能最適化　　[講義・演習]

- 演習中心として, CUDA実装に慣れることを目的にしたいと考えています

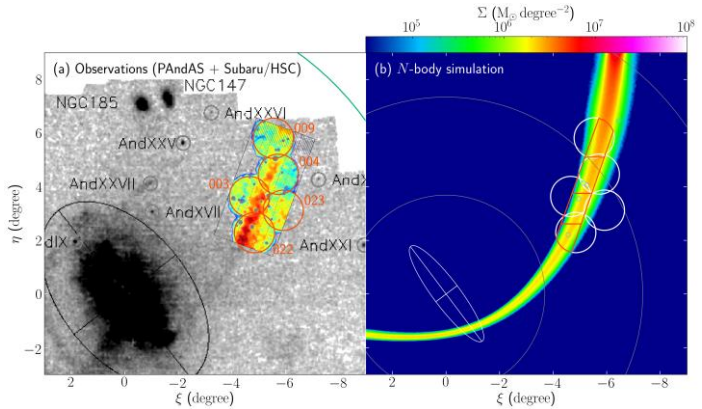
3日目(8/20)の予定

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

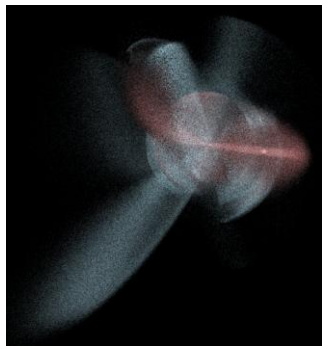
- 最終日は皆さんが自身のコードをGPU化する際に知っておくと便利な情報を提供することを目的にします

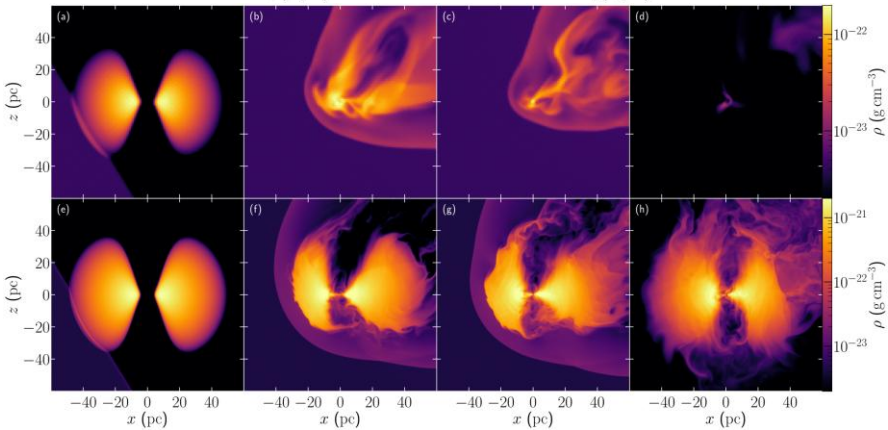
自己紹介

2005-2009	筑波大学 第一学群 自然学類
2009-2011	筑波大学大学院 数理物質科学研究科 物理学専攻(博士前期課程)
2011-2014	筑波大学大学院 数理物質科学研究科 物理学専攻(博士後期課程)
2011-2013	筑波大学大学院 システム情報工学研究科 コンピュータサイエンス専攻(博士前期課程)
2014-2017	筑波大学 計算科学研究センター 研究員
2017-2024	東京大学 情報基盤センター 助教
2024-	東京大学 情報基盤センター 准教授
2025-	工学系研究科 電気系工学専攻 准教授(兼担)



- 専門分野:宇宙物理学, 高性能計算
  - 銀河の形成・進化(銀河考古学, 銀河衝突)
  - 銀河と中心ブラックホールの共進化
  - GPUを用いた演算加速
  - 重力多体計算, 数値流体力学
  - (準)力学平衡状態生成コードの作成





初日(8/18)の予定

- 10:00--11:00　並列計算の初歩　[講義]
- 11:15--12:15　GPUコンピューティングの基礎概念　[講義]
- 13:30--14:30　簡単なCUDAコードの実装　[講義・演習]
- 14:45--15:45　簡易なプロファイラと性能評価　[講義・演習]
- 16:00--17:00　GPU移植時の罠　[講義・演習]

先出し: 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
  - GPGPU (General-Purpose computing on GPU) とも呼ばれていた
  - 最近のGPUは (主に深層学習用に) 行列積向けユニットも備える
- 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
  - 2010年頃のGPUコンピューティング黎明期には安かったが, 最近はとても高価
- 多数のコア(最新GPUは1万超)を搭載した並列計算機
  - NVIDIA H100 (SXM) : 8448 FP64 (16896 FP32) cores
  - NVIDIA B200 : 9472 FP64 (18944 FP32) cores
  - AMD MI300A : 14592 Stream Processors
  - AMD MI300X : 19456 Stream Processors
- GPUの多数のコアを使いこなすには
  - 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

(GPUに行く前に)CPU単体の理論ピーク性能の導出

- 単位時間あたりにどれだけ浮動小数点演算をできるか？
- 例:CfCAのXD2000(システムM)の場合
  Intel Xeon CPU Max 9480 (56 cores, 1.9 GHz)
  - $56\ [\mathrm{cores}] \times 1.9\ [\mathrm{GHz}] \times \frac{512\ [\mathrm{bit}]}{64\ [\mathrm{bit}]} \times 2\ (\#\ \mathrm{of\ AVX\ units}) \times 2\ (\mathrm{FMA}) = 3.4\ \mathrm{TFlop/s}$
    - CPUの動作周波数にも色々あるが, ここではベースクロック
    - 倍精度(64 bit)変数の理論ピーク性能を計算(単精度であればこの2倍)
    - SPR(Sapphire Rapids)世代ではAVX512命令に対応している
    - FMA: Fused Multiply-Add 命令は積和算($a \times b + c$, 実質2演算)を1命令1サイクルで実行
  - さらに, このCPUが1ノードの中に2ソケット搭載され, 全体で208ノードというのがシステムM
- やたらと色々な項が出てきたなと思うはず
  - (残念なことに)これを全部使わないと, 理論ピーク性能を引き出せない
  - 全コアを活用するという部分は並列計算
  - AVX という部分も並列計算(SIMD命令セットの実装例)

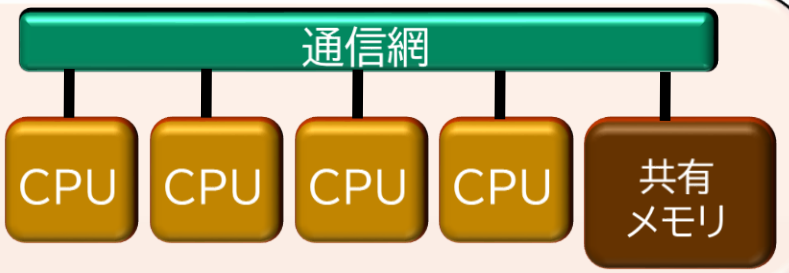
まずは言葉だけ：並列計算機の分類

- Michael J. Flynn氏の分類(1966)
- 単一命令・単一データ流
  (SISD: Single Instruction Single Data Stream)
- 単一命令・複数データ流
  (SIMD: Single Instruction Multiple Data Stream)
- 複数命令・単一データ流
  (MISD: Multiple Instruction Single Data Stream)
- 複数命令・複数データ流
  (MIMD: Multiple Instruction Multiple Data Stream)

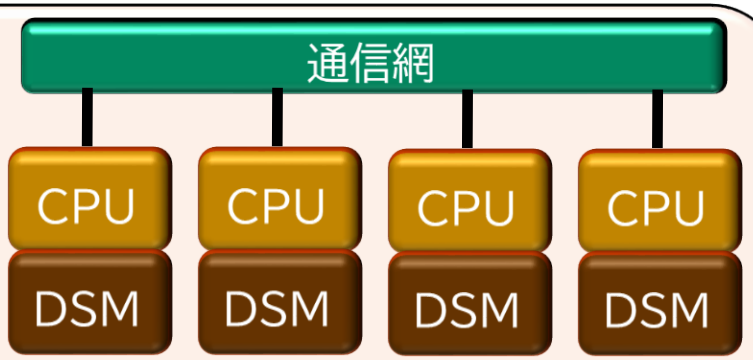
並列計算機のメモリ型による分類(1/2)

- A) メモリアドレスを共有している:互いのメモリがアクセス可能

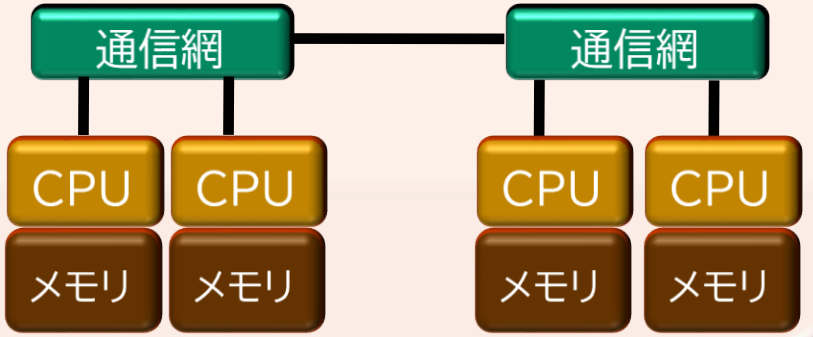
1. 共有メモリ型
   (SMP: Symmetric Multiprocessor,
   UMA: Uniform Memory Access)



2. 分散共有メモリ型
   (DSM: Distributed Shared Memory)



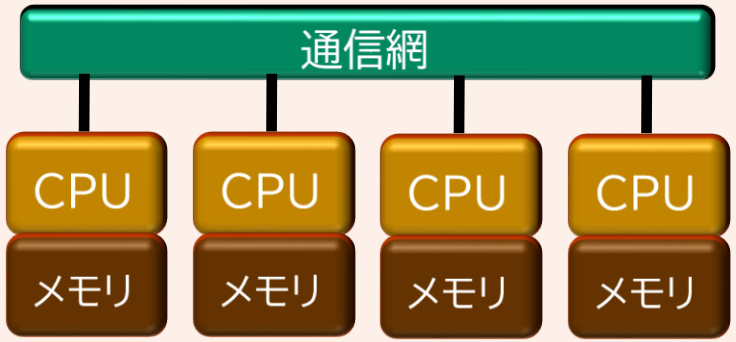
   共有・非一様メモリ型
   (ccNUMA: Cache Coherent
   Non-Uniform Memory Access)



並列計算機のメモリ型による分類(2/2)

- B）メモリアドレスは独立：互いのメモリはアクセス不可

3. 分散メモリ型
（メッセージパッシング）

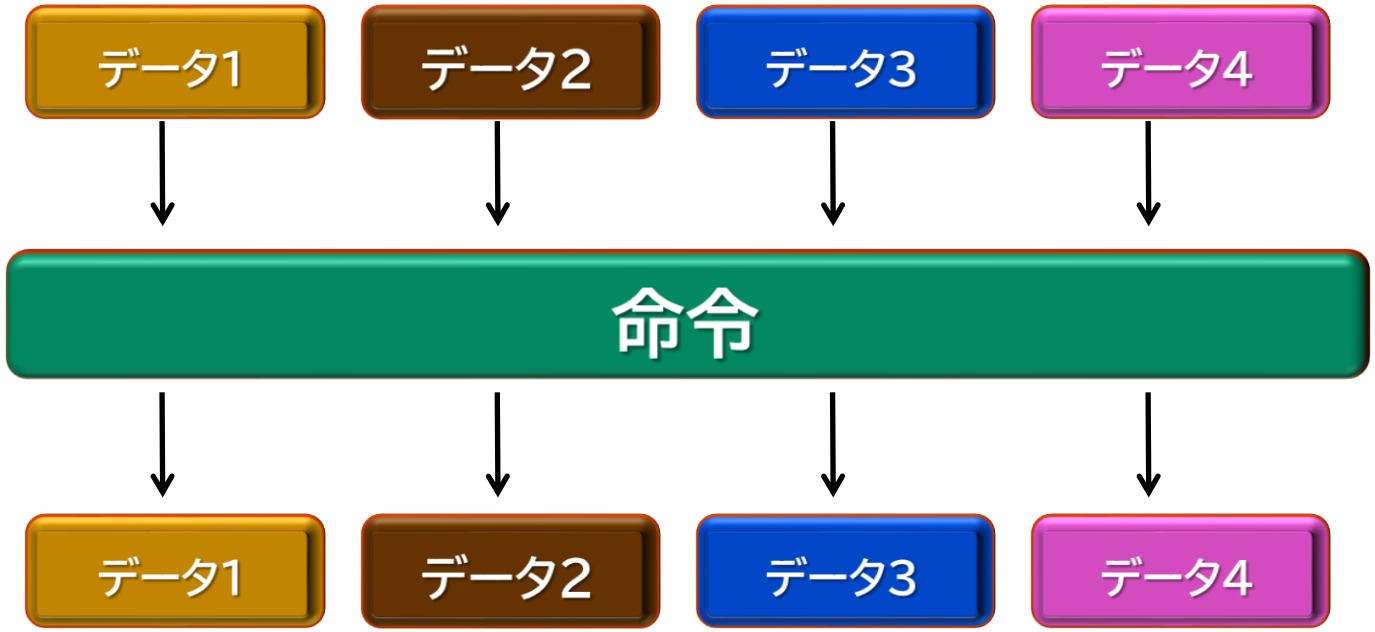


プログラミング手法から見た分類

1. マルチスレッド
   - Pthreads, …
2. データ並列
   - OpenMP
   - (最近の)Fortran
   - PGAS (Partitioned Global Address Space)言語: XcalableMP, UPC, Chapel, X10, Coarray Fortran, …
3. タスク並列
   - Cilk Plus, Threading Building Blocks (現 oneTBB), StackThreads, MassiveThreads, …
4. メッセージ並列
   - MPI (Message Passing Interface)

並列プログラミングのモデル(1/2)

- 実際の並列プログラムの挙動はMIMD
- アルゴリズムを考えるときにはSIMDが基本
  - いきなり複雑な挙動を考えるのは難しいので, なるべくシンプルに



並列プログラミングのモデル(2/2)

- 多くのMIMD上での並列プログラミングのモデル

1. SPMD (Single Program Multiple Data)

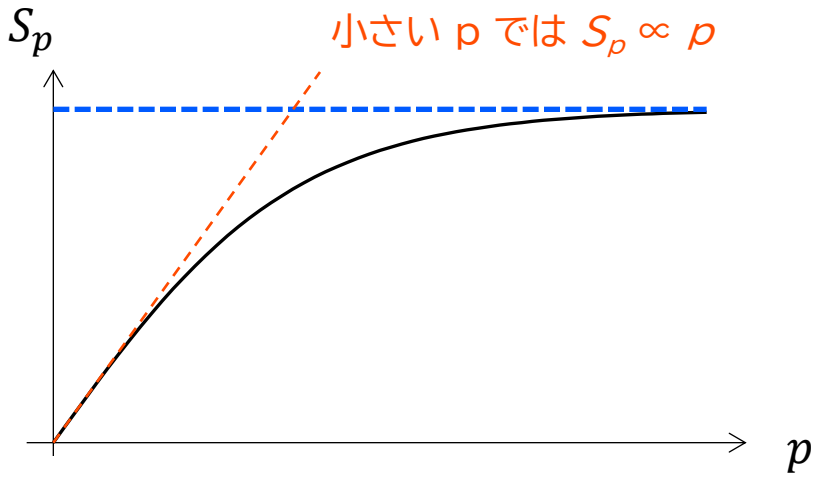
- 1つの共通のプログラムが, 並列処理開始時に, 全プロセッサ上で起動
- MPI(バージョン1)のモデル

2. Master/Worker

- 1つのプロセス(Master)が, 複数のプロセス(Worker)を管理(生成, 消去)
- 今回, こちらは対象にしません

性能評価指標 --台数効果--

- 台数効果
  - 式: $S_p = \frac{T_s}{T_p}$, ただしT_sは逐次の, T_pはp台での実行時間
  - p台用いて$S_p = p$のとき, 理想的な(ideal)速度向上
  - p台用いて$S_p > p$のとき, スーパーリニア・スピードアップ
    - 主な原因は, 並列化によってデータアクセスが局所化され, キャッシュヒット率が向上したことによる高速化
- 並列化効率
  - 式: $E_p = \frac{S_p}{p} \times 100\ [\%]$
- 飽和性能
  - 速度向上の限界



Weak ScalingとStrong Scaling

- 並列処理においてシステム規模を大きくする方法
- Weak Scaling: プロセッサあたりの問題サイズを変えず並列度をあげる
  - 全体の問題サイズが(並列数に比例して)大きくなる
  - 通信のオーバーヘッドはあまり変わらないか, やや増加
  - 今まで解けなかった規模の問題が解けるようになる
- Strong Scaling: 全体の問題サイズを変えずに並列度をあげる
  - プロセッサあたりの問題サイズは(並列数に反比例して)小さくなる
  - 通信のオーバーヘッドは相対的に大きくなる
  - 同じ問題規模でも, より短時間に結果が得られる(Weakよりも難しい)

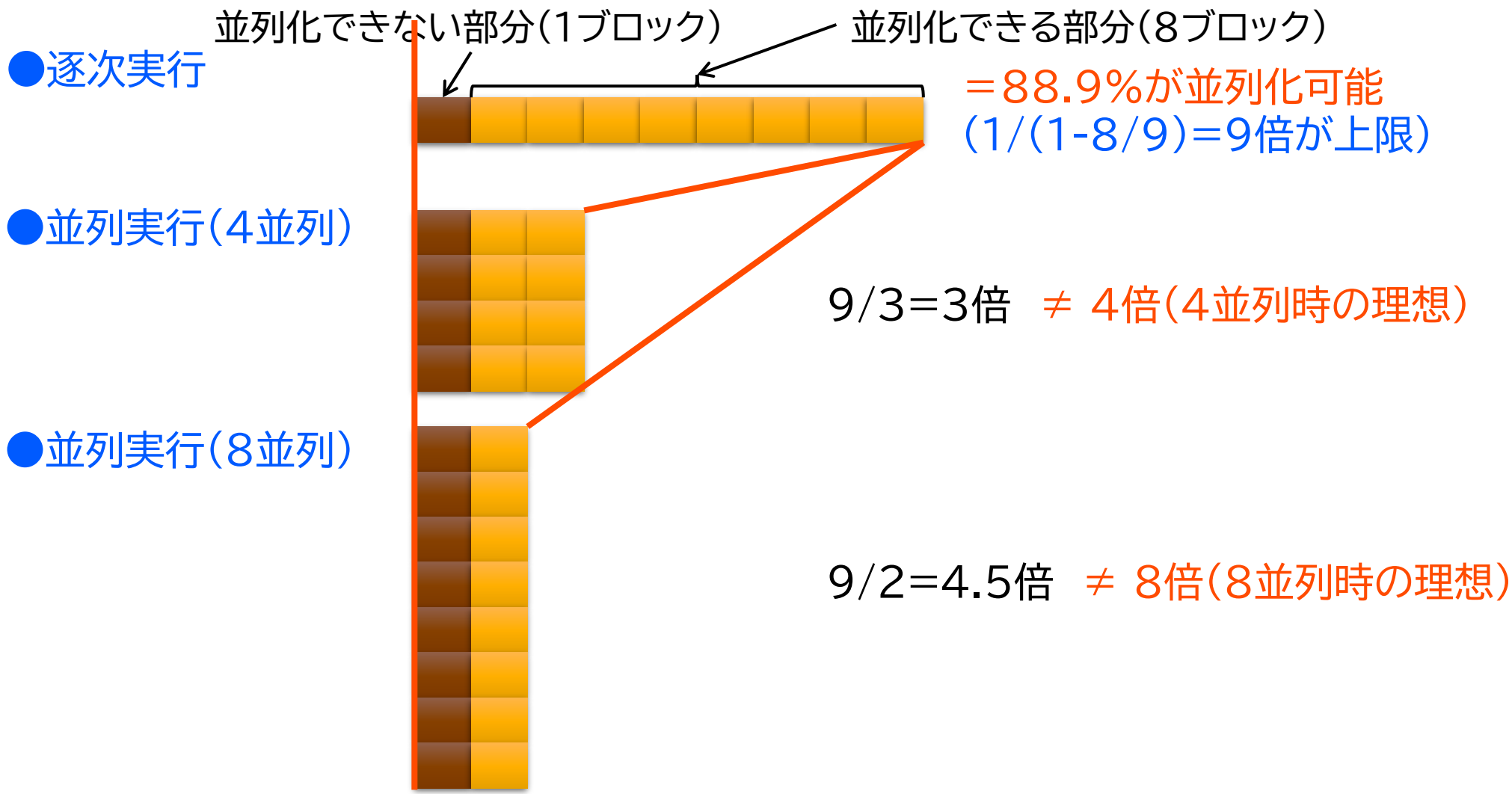
アムダールの法則

- 逐次実行時間をT_sとし，このうち並列化できる割合をαとする
- 台数効果は以下のように計算できる：

$$S_p = \frac{T_s}{\frac{\alpha T_s}{p} + (1-\alpha)T_s} = \frac{1}{\frac{\alpha}{p} + 1 - \alpha} = \frac{1}{1 + \alpha\left(\frac{1}{p} - 1\right)}.$$

- 無限大の数のプロセッサを使っても，台数効果は$1/(1-\alpha)$が上限（アムダールの法則）
  - 全体の90%が並列化できたとしても，台数効果は最大で$\frac{1}{1-0.9} = 10$倍
  - →高性能の達成には，少しでも並列可能な割合を上げる実装をすることが重要

アムダールの法則の直感例



以降では, メモリを共有している場合の並列化に集中

- 複数の演算器(コア)が, 共通のメモリにアクセスできる場合を想定
  - (単体)GPUはこのパターン
  - CPUスパコンの1ノードの中もこのパターン
  - 皆さんが今使っているノートPCもこのパターン

- プログラムを並列化する際に
  - どういう方法が使えるのか？
  - やってはいけないことは何か？
  - 気をつけるべきことは何か？
  - どうすれば性能を出しやすくなるか？
  - ↑ このあたりを把握していると, プログラムのGPU化にも役立つ

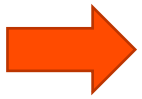
CPU性能を引き出すには，二段階の並列化が必要

- 再掲：CfCAのXD2000（システムM）に搭載されている Intel Xeon CPU Max 9480 (56 cores, 1.9 GHz)
  - $56\ \text{cores} \times 1.9\ \text{GHz} \times \frac{512\ \text{bit}}{64\ \text{bit}} \times 2\ (\text{AVX units}) \times 2\ (\text{FMA}) = 3.4\ \text{TFlop/s}$
- 1ソケットの中にも（ハードウェアとして）複数の並列度がある
  0. （大前提として）並列可能な処理がないといけない
     - 各処理に依存関係があると，独立に処理できない（＝（簡単には）並列可能ではない）
     - 各処理の実行順序が単体実行の時と同じであるとは保証されない
     - （メモリを共有しているので）メモリへの書き込みが競合しないよう注意
  1. 全56コアを活用する（こちらの方が簡単なので，先に説明）
     - OpenMP の指示文を入れてあげる（スレッド並列）のが一番簡単
  2. SIMD命令を活用する（実際に使うのは一筋縄で行かないこともある）
     - `#pragma omp simd` でできれば一番簡単
       （駄目ならintrinsicやインラインアセンブラといったより低レベルな実装に）
       - 低レベルというのはより機械に近いレベルという意味で，実装難度は数段上がっている

スレッド並列の基礎

- 「スレッド」が処理単位, CPUの場合にはスレッド数＝コア数と思って良い
  - ハイパースレッディング無効の場合(スパコン環境では無効にしていることが多い)
- スレッド並列における注意事項
  - スレッド間の同期は保証されない
    - 同期が必要な際には, 明示的に同期を取る(暗黙の同期ポイントもあるが, 今回は触れない)
  - スレッドの実行順序が単体実行時と同じである保証はない
  - 全体の実行時間は, 最後に処理を終えるスレッドが決める
    - 処理時間が長いスレッドがいると, 全体の足を引っ張ることになる
- スレッド並列が可能なループであることをユーザーが保証し, 指示文を追加
  - 「実はスレッド並列してはいけなかった」としてもコンパイラは並列化する
    - 計算結果が正しいか, はきちんと自分で確認すること
  - コンパイルオプションに-fopenmpなどを付与することもお忘れなく
    - コンパイラごとにオプションが違う点にも注意

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



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

例えば direct N-body の場合

- Direct N-body の場合には,
  重力計算は二重ループになる

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

- 右に示したコード例では,
  大外のi-ループをスレッド並列化
  - 各i-粒子が受ける重力は独立に計算可能
  - i-粒子のループをどの順番で実行してもOK
- 内側のj-ループを並列化するのは面倒
  - 「共通の」i-粒子への重力を積算する必要
    - `reduction` や `atomic` で実装できるが, 性能低下
  - スレッドの生成・消滅コストも乗ってくる
  - なるべく外側で並列化した方が良い,
    と思っておけば大抵うまくいく

```
static inline void calc_acc(const int32_t Ni, const float4
*const ipos, float4 *__restrict iacc, const int32_t Nj,
const float4 *const jpos, const float eps2) {
#pragma omp parallel for
  for (int32_t i = 0; i < Ni; i++) {
    const auto pi = ipos[i];
    float4 ai = {0.0F, 0.0F, 0.0F, 0.0F};

    for (int32_t j = 0; j < Nj; j++) {
      const auto pj = jpos[j];

      // particle-particle interaction
      const auto dx = pj.x - pi.x;
      const auto dy = pj.y - pi.y;
      const auto dz = pj.z - pi.z;
      const auto r2 = eps2 + dx * dx + dy * dy + dz * dz;
      const auto r_inv = 1.0F / std::sqrt(r2);
      const auto r2_inv = r_inv * r_inv;
      const auto alp = pj.w * r_inv * r2_inv;

      // accumulation
      ai.x += alp * dx;
      ai.y += alp * dy;
      ai.z += alp * dz;
      ai.w += alp * r2;
    }
    iacc[i] = ai;
  }
}
```

スレッド並列(CPU向けのOpenMP)まとめ

- 前提:
  - ループの各反復が独立な処理であり, 複数スレッドで分担して(並列に)実行できる
    - 各処理に依存関係はないか？
  - 全スレッドは同じメモリを読み書きできる(共有メモリモデル)
    - データ競合を起こすような実装になっていないか？

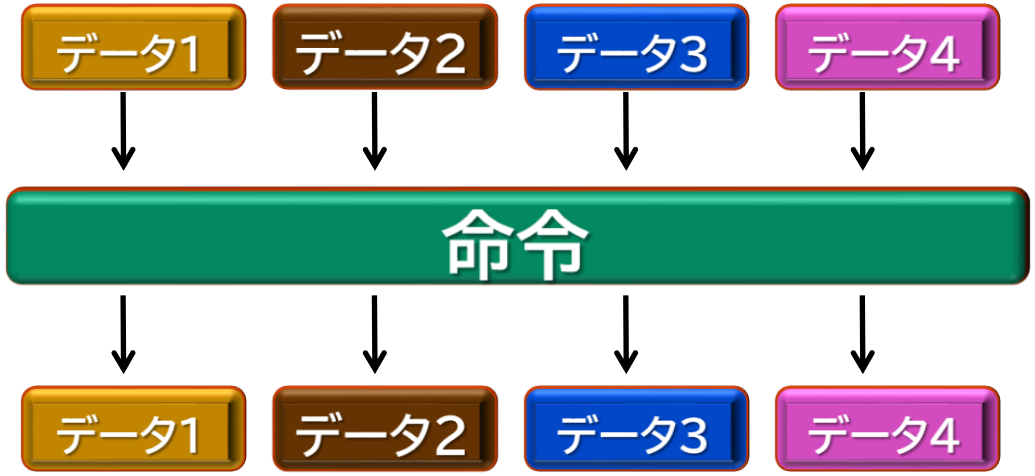
```
static inline void calc_acc(const int32_t Ni, const float4
*const ipos, float4 *__restrict iacc, const int32_t Nj,
const float4 *const jpos, const float eps2) {

  for (int32_t i = 0; i < Ni; i++) {
    const auto pi = ipos[i];
    float4 ai = {0.0F, 0.0F, 0.0F, 0.0F};
    for (int32_t j = 0; j < Nj; j++) {
      const auto pj = jpos[j];
      // particle-particle interaction
      const auto dx = pj.x - pi.x;
      ……
      const auto alp = pj.w * r_inv * r2_inv;
      // accumulation
      ai.x += alp * dx;
      ……
    }
    iacc[i] = ai;
  }
}
```

```
static inline void calc_acc(const int32_t Ni, const float4
*const ipos, float4 *__restrict iacc, const int32_t Nj,
const float4 *const jpos, const float eps2) {
#pragma omp parallel for
  for (int32_t i = 0; i < Ni; i++) {
    const auto pi = ipos[i];
    float4 ai = {0.0F, 0.0F, 0.0F, 0.0F};
    for (int32_t j = 0; j < Nj; j++) {
      const auto pj = jpos[j];
      // particle-particle interaction
      const auto dx = pj.x - pi.x;
      ……
      const auto alp = pj.w * r_inv * r2_inv;
      // accumulation
      ai.x += alp * dx;
      ……
    }
    iacc[i] = ai;
  }
}
```

この一行を入れるだけで
並列化してくれる
注:前提条件を満たしていなくても並列化される
(正しさの保証は自己責任)

SIMD化

- SSE, AVX, SVEなどは全てSIMD命令を実現する命令セット拡張



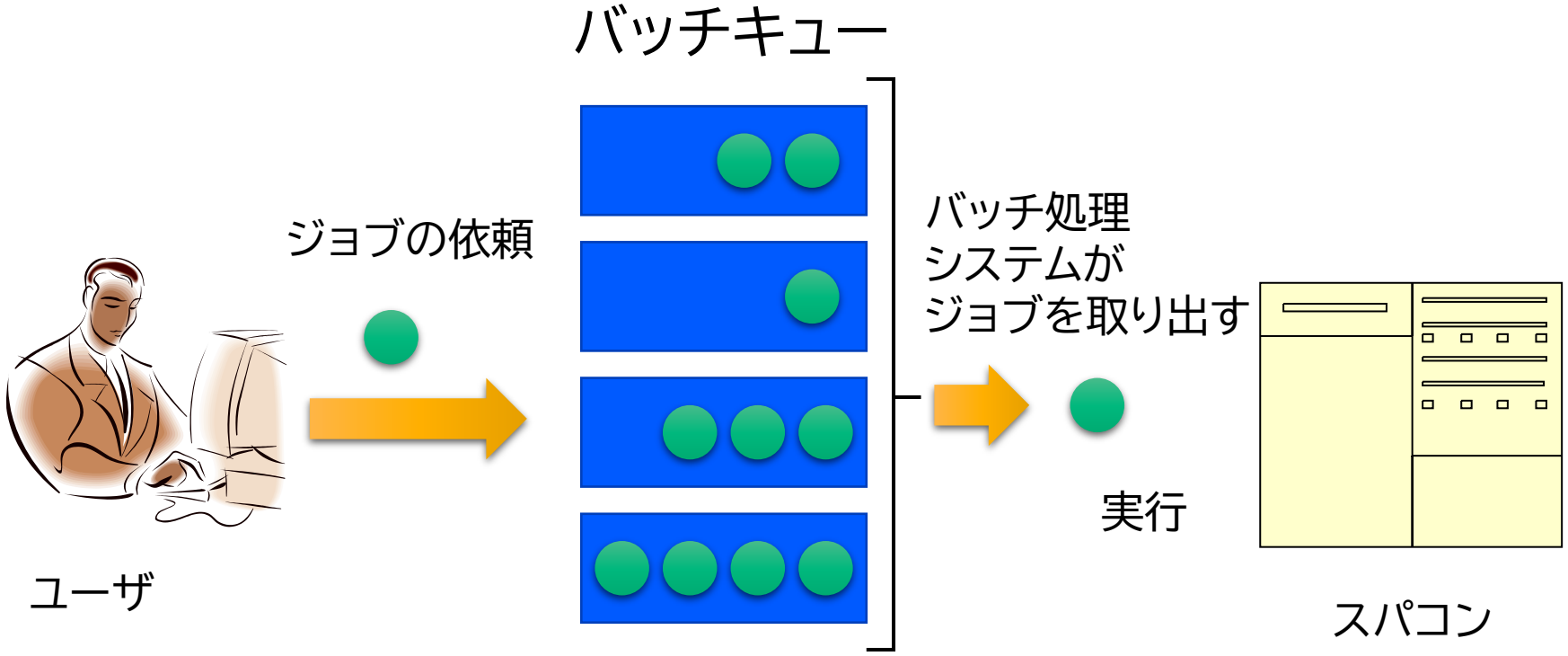
- コンパイラにお伺いをたててSIMD化してもらうのだが, 結構大変
  - 上の図の場合, 4人全員が完全に同じ命令を実行してくれないと困る
    - if文が入っていて各自の処理が分岐するような場合にはSIMD化が効かないこともある
    - mask をあてて空回しして対応, みたいなことをして回避することも
  - コンパイラによっては, 最内ループしかSIMD化の対象にしてくれないことも
- 例えばDirect N-body では, 演算量半減を狙って作用・反作用則を利用した実装にすると(少なくともコンパイラ任せの場合)SIMD化が効かなくなる

OpenMP を使ったCPU向けスレッド並列の練習1

- あらかじめVPN接続しておく
- `$ ssh USERNAME@g00.cfca.nao.ac.jp`
  - ここのUSERNAMEは，皆さん固有のユーザーID
- `$ cd /cfca-work/$USER`
  - ログイン後は，$USER という環境変数に皆さん固有のユーザーIDが入っている
- `$ git clone https://github.com/ymiki-repo/cfca-gpu2026f.git`
- `$ cd cfca-gpu2026f`
- `$ cd 0_vecadd/0_serial`
- `$ make`
- `$ sbatch sh/cfca/run_cpu.sh`

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}
```

OpenMP を使ったCPU向けスレッド並列の練習2

- 先程の結果はCPU 1コアのみを使った場合
- [演習] vecadd.c, Makefile を適切に編集して, スレッド並列化してみる
  - スレッド並列できる・すると良さそうなところに`#pragma omp parallel for`を追加
  - Makefile に OpenMP を使うためのフラグを追加
    - GCC であれば `-fopenmp`
    - NVIDIA HPC SDK であれば `-mp`
- 確認すべきこと:
  - 計算結果は正しいか？(例えば, 単体版の計算結果と比較してみると良い)
  - 並列化前よりも速くなっているか？

- やり方がわからない場合は
  - ヒントをあげるので言ってください
  - 今回の講習会の本題ではないので, `1_omp` の中身をカンニングしてもOK

OpenMP版コードの解説

- 上段が単体版，下段が並列版
  - 指示文の有無だけが違い
    - これだけで速くなるのでお得
- C/C++言語一般の最適化：
  - constやrestrictなどの修飾子は積極的につけておくのが良い
    - const は値を書き換えない場合
    - restrict はアクセスする領域が他と重なっていないことをコンパイラに伝える（言語・コンパイラごとに書き方が違ったりするので注意：__restrict や__restrict__）
- 計算結果の一致の確認例
  - 浮動小数点演算なので，c1[i] != c0[i] はやりすぎな場合もある

```
static inline void vecadd(const int num, const float val, const float*
restrict a, const float* restrict b, float* restrict c) {
  for (int i = 0; i < num; i++) {
    c[i] = a[i] + val * b[i];
  }
}

static inline void vecadd_omp(const int num, const float val, const
float* restrict a, const float* restrict b, float* restrict c) {
#pragma omp parallel for
  for (int i = 0; i < num; i++) {
    c[i] = a[i] + val * b[i];
  }
}
```

```
// validation
int err = 0;
for (int i = 0; i < num; i++) {
  if (fabsf(c1[i] / c0[i] - 1.0F) > 1.0e-6F) {
    err++;
  }
}
if (err == 0) {
  printf("validation: OK¥n");
} else {
  printf("validation: NG (%d errors)¥n", err);
}
```

初日(8/18)の予定

- 10:00--11:00　並列計算の初歩　［講義］
- 11:15--12:15　GPUコンピューティングの基礎概念　［講義］
- 13:30--14:30　簡単なCUDAコードの実装　［講義・演習］
- 14:45--15:45　簡易なプロファイラと性能評価　［講義・演習］
- 16:00--17:00　GPU移植時の罠　［講義・演習］

GPU（Graphics Processing Units）

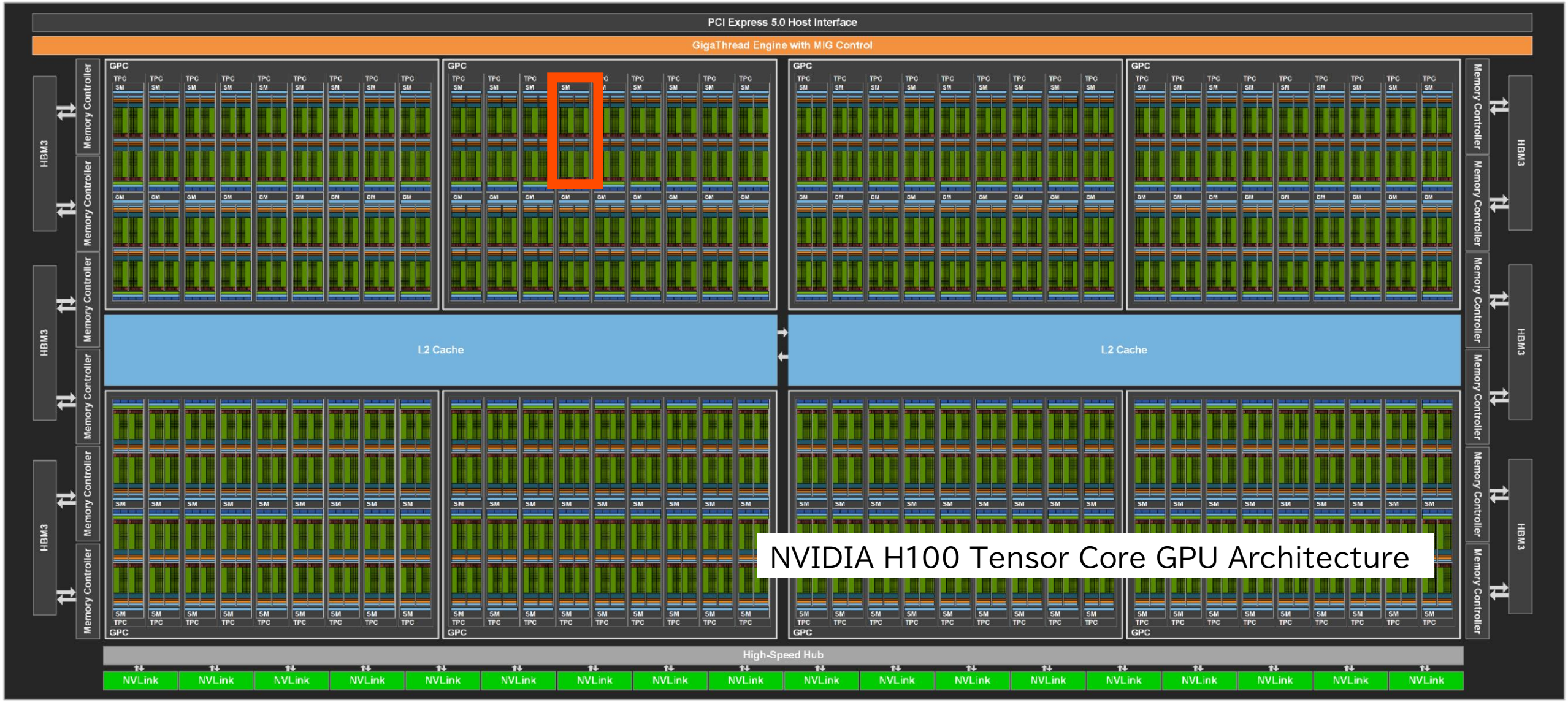
- 元々は画像処理を行うために特化していた専用プロセッサを
  汎用計算にも使えるように拡張
  - GPGPU（General-Purpose computing on GPU）とも呼ばれていた
  - 最近のGPUは（主に深層学習用に）行列積向けユニットも備える
- 高い演算性能，太いメモリバンド幅，消費電力あたり性能も高い
  - 2010年頃のGPUコンピューティング黎明期には安かったが，最近はとても高価
- 多数のコア（最新GPUは1万超）を搭載した並列計算機
  - NVIDIA H100（SXM）：　8448 FP64（16896 FP32）cores
  - NVIDIA B200　　：　9472 FP64（18944 FP32）cores
  - AMD MI300A　　：14592 Stream Processors
  - AMD MI300X　　：19456 Stream Processors
- GPUの多数のコアを使いこなすには
  - 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 2026)

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

	TOP 500	System	Accelerator	Cores	HPL Rmax (Pflop/s)	Power (kW)	GFLOPS/W
1	446	KAIROS, CALMIP/CNRS, France	NVIDIA GH200	13,056	3.05	46	73.282
2	192	ROMEO-2025, ROMEO HPC Center - Champagne-Ardenne, France	NVIDIA GH200	47,328	9.86	160	70.912
3	250	Levante GPU extension, DKRZ, Germany	NVIDIA GH200	35,904	6.75	110	69.426
4	238	Isambard-AI phase1, U. Bristol, UK	NVIDIA GH200	34,272	7.42	117	68.835
5	312	Otus (GPU Only), U. Paderborn, Germany	NVIDIA H100 80GB	19,440	4.66		68.177
6	88	Capella, TU Dresden ZIH, Germany	NVIDIA H100 94GB	85,248	24.06	445	68.053
7	363	SSC-24 Energy Module, Samsung, Korea	NVIDIA H100 80GB	11,200	3.82	69	67.251
8	116	Helios GPU, Cyfronet, Poland	NVIDIA GH200	89,760	19.14	317	66.948
9	451	AMD Ouranos, Atos, France	AMD Instinct MI300A	16,632	2.99	48	66.464
10	85	Portage, HPE, United States	AMD Instinct MI300A	129,024	24.50	370	66.277
31	137	Plasma Simulator Subsystem B, QST/NIFS, Japan	AMD Instinct MI300A	70,560	15.93	269	59.219
37	278	Sirius, U. Tsukuba, Japan	AMD Instinct MI300A	24,192	5.63	114	56.773
50	1	LineShine, NSCS, China	LX2 (汎用CPU最上位)	13,789,440	2,198	42,220	52.070
56	62	TSUBAME 4.0, Science Tokyo, Japan	NVIDIA H100 94GB SXM5	172,800	39.62	816	48.565
60	52	Miyabi-G, JCAHPC, Japan	NVIDIA GH200	221,952	46.80	983	47.588
125	193	Roihu-C, CSC, Finland	AMD EPYC 9965 (x86 CPU最上位)	186,624	13.37	905	13.740

(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)
  - 呼び名は違うが, コアの塊が複数という構成は共通
  - 基礎学習の段階では,
    SMの中の詳細な構造まで理解する必要は無い
- SM内ではL1キャッシュ/シェアードメモリを共有
  (H100ではSMあたり合計256KB)
  - 両者の配分は段階的に調節可能
  - シェアードメモリはCUDAでは扱えるが
    OpenACCでは(明示的には)不可
  - グローバルメモリ(GPUのメモリ)よりも圧倒的に速い
- 処理(GPUスレッド)のSMへの割り振り方法
  (スレッド数, 形状)が性能に強く影響
  - CUDAはユーザが設定
  - OpenACCはコンパイラが設定(ユーザによる設定も可能)

NVIDIA H100のSMの構造

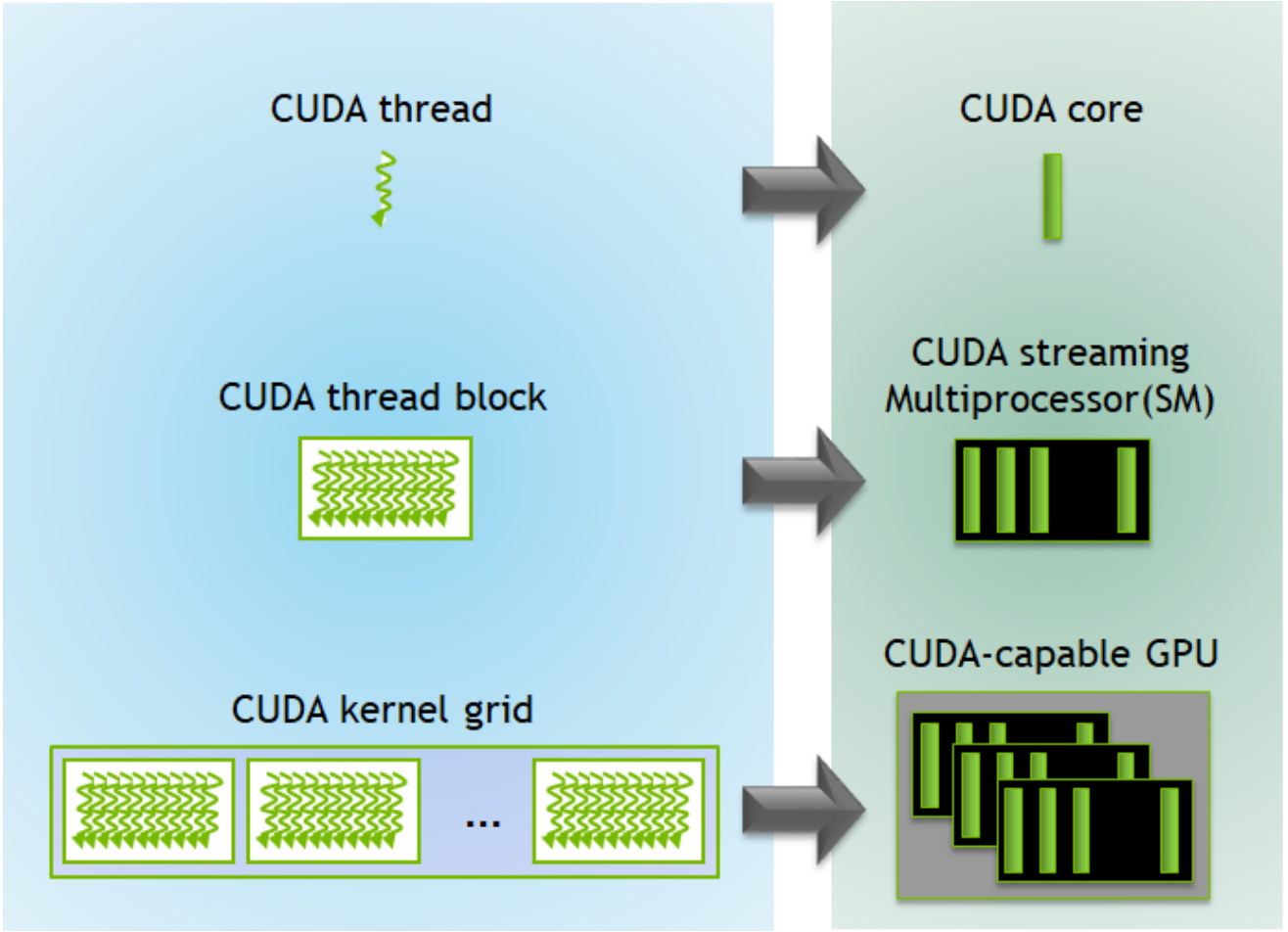
FP32演算コア:128個
FP64演算コア:64個
L1キャッシュ等を共有

GPU単体の理論ピーク性能の導出

- 再掲(CPU):CfCAのXD2000(システムM)の場合
  Intel Xeon CPU Max 9480 (56 cores, 1.9 GHz)
  - $56\ [\mathrm{cores}] \times 1.9\ [\mathrm{GHz}] \times \frac{512\ [\mathrm{bit}]}{64\ [\mathrm{bit}]} \times 2\ (\#\ \mathrm{of\ AVX\ units}) \times 2\ (\mathrm{FMA}) = 3.4\ \mathrm{TFlop/s}$
    - 倍精度(64 bit)変数の理論ピーク性能を計算(単精度であればこの2倍)
- 今回の講習会で用いるNVIDIA A100の場合
  - $1.41\ [\mathrm{GHz}] \times 32\ [\mathrm{FP64\ cores/SM}] \times 108\ [\mathrm{SMs}] \times 2\ (\mathrm{FMA}) = 9.7\ \mathrm{TFlop/s}$ @ FP64
  - $1.41\ [\mathrm{GHz}] \times 64\ [\mathrm{FP32\ cores/SM}] \times 108\ [\mathrm{SMs}] \times 2\ (\mathrm{FMA}) = 19.5\ \mathrm{TFlop/s}$ @ FP32
  - $1.41\ [\mathrm{GHz}] \times 128\ [\mathrm{Flops/SM}] \times 108\ [\mathrm{SMs}] = 19.5\ \mathrm{TFlop/s}$ @ FP64 Tensor Core
  - 動作周波数はブーストクロック
  - 単精度, 倍精度で別の演算コアがついている
  - FMA命令が理論ピーク性能の根拠に入っているのはCPUと同じ
  - 全コアを使わないと性能を出せないのもCPUと同じ, ただしSIMD化ハードルは低減
    - NVIDIA的にはSIMT(Single Instruction, Multiple Threads)という用語がある
      (用語としてあるだけで, 特に意識する必要はない. SIMDよりも簡単, ぐらいの認識で十分)

GPUスレッドとGPU演算コアの対応

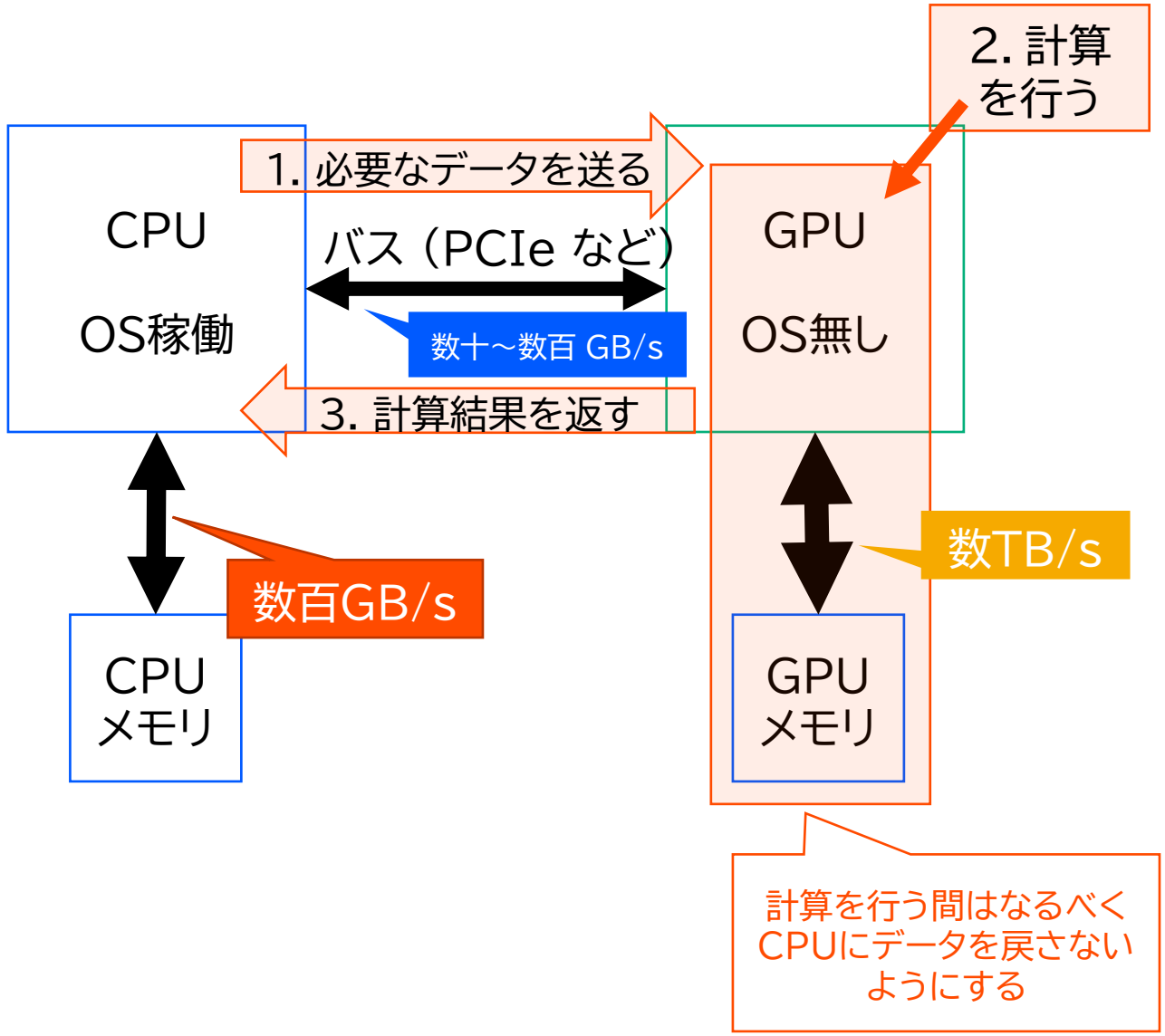
- NVIDIA CUDAアーキテクチャを例に，スレッドと演算コアの対応を説明
- 1スレッド(1要素に相当すること多い)が1コアに対応
- スレッドブロック(64から512程度のスレッド群に分割されたもの)を任意のSMに割り当てて処理させる
  - 多く割り当てた方が性能が出る傾向
  - 各種レイテンシを隠蔽できる
- ブロック全て(＝グリッド)をGPUに割り当てて領域全体を処理する
- 「各種レイテンシの隠蔽」，「ハードの各階層でのキャッシュ等の活用」には適切なGPUスレッドの割り振りが必要



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

CPUとGPUのメモリ空間

- CPUとGPUは, 物理的に独立したメモリを持つことが一般的
  - （AMD MI300Aのような, CPUとGPUが物理的にメモリを共有する製品もある）
- （OSが動作している）CPUがGPU上の演算処理を起動
  - CPUメモリとGPUメモリ間でのデータのやり取りも発生
- CPU-GPU間のデータ転送の帯域はGPUメモリに比べてかなり狭い
  - 性能のボトルネックになりやすい



GPU化で高速化しやすいアプリの特徴

- GPUに搭載されている演算器を動かし続けられるだけの並列度がある
  - コア数の10倍以上の独立に処理可能な並列度があると，
    比較的容易にGPU性能を引き出せる

- GPU側にデータを置きっぱなしにできる
  - CPU-GPU間のデータ転送がこまめに発生すると性能低下要因になる
    - AMD MI300Aでは，CPUとGPUが同じメモリを共有するので，この問題は起きない
    - NVIDIA GH200/GB200では，CPU-GPU間のデータ転送が高速なので問題になりにくい

- 各スレッドの演算処理にデータ依存性がない
  - 並列化可能となり，スレッドを並行して処理することで各種レイテンシを隠蔽できる
  - `if`文などで処理が分裂すると，冗長な処理となりコスト増加
  - 数十スレッド単位で同じ条件に分岐（例：連続する32スレッド全て真）する場合は，
    冗長な処理によるコスト増は抑えられる

GPU化で高速化しやすいアプリ例①（C）

- 拡散方程式コード（CPU版, C言語）の抜粋
  - メモリ律速な問題の代表例

```c
int main(int argc, char *argv[])
    const int nx = 512;
    const int ny = 512;
    const int nz = 512;

    float *f  = (float *)malloc(sizeof(float) * ln);
    float *fn = (float *)malloc(sizeof(float) * ln);

    for (; (icnt < nt) && (time + 0.5 * dt < 0.1); icnt++) {
        diffusion3d(nx, ny, nz, mgn, dx, dy, dz, dt, kappa, f, fn);

        swap(&f, &fn);
        time += dt;
    }

    return 0;
}
```

GPUのコア数よりも十分に大きな問題サイズ

演算をGPUに閉じ込められるので, こまめなCPU-GPU間データ転送は不要

並列化可能（OpenMPでのスレッド並列がkループ軸に設定済み）

```c
void diffusion3d(int nx, int ny, int nz, int mgn, float dx, float
dy, float dz, float dt, float kappa, const float *f, float *fn)
{
#pragma omp parallel for
    for(int k = 0; k < nz; k++) {
        for (int j = 0; j < ny; j++) {
            for (int i = 0; i < nx; i++) {
                const int ix = nx * ny * (k + mgn) + nx * j + i;
                const int ip = i == nx - 1 ? ix : ix + 1;
                const int im = i == 0      ? ix : ix - 1;
                const int jp = j == ny - 1 ? ix : ix + nx;
                const int jm = j == 0      ? ix : ix - nx;
                const int kp = (k == nz - 1) ? ix : ix + nx*ny;
                const int km = (k == 0     ) ? ix : ix - nx*ny;

                fn[ix] = cc*f[ix] + ce*f[ip] + cw*f[im] +
cn*f[jp] + cs*f[jm] + ct*f[kp] + cb*f[km];
            }
        }
    }
}
```

各格子点が共通の処理を実行
i, j, kループ軸にて依存関係も無し
→i, j, k軸でスレッド並列化可能
512^3=134,217,728スレッド並列となり, 十分大きいスレッド数

GPU化で高速化しやすいアプリ例②（C++）

- N体計算コード（CPU版, C++）の抜粋
  - 演算律速な問題の代表例

```
auto main(void) -> int32_t {
  const auto num = 1048576;

  float4 *pos = nullptr;
  float3 *vel = nullptr;
  float4 *acc = nullptr;
  allocate_Nbody_particles(&pos, &vel, &acc, num);

    while (present < snp_fin) {
      // orbit integration
      drift(num, pos, vel, dt);
      calc_acc(num, pos, acc, num, pos, eps2);
      trim_acc(num, acc, pos, eps_inv);
      kick(num, vel, acc, dt);

      // write snapshot
      if (present > previous) {
        write_snapshot(num, pos, vel, acc, file, present, time);
      }
    }
  }

  return (0);
}
```

GPUのコア数よりも十分に大きな問題サイズ

演算をGPUに閉じ込められるので, こまめなCPU-GPU間データ転送は不要

```
static inline void calc_acc(const int32_t Ni, const float4
*const ipos, float4 *__restrict iacc, const int32_t Nj,
const float4 *const jpos, const float eps2) {
#pragma omp parallel for
  for (int32_t i = 0; i < Ni; i++) {
    const auto pi = ipos[i];
    float4 ai = {0.0F, 0.0F, 0.0F, 0.0F};

    for (int32_t j = 0; j < Nj; j++) {
      const auto pj = jpos[j];

      // particle-particle interaction
      const auto dx = pj.x - pi.x;
      const auto dy = pj.y - pi.y;
      const auto dz = pj.z - pi.z;
      const auto r2 = eps2 + dx * dx + dy * dy + dz * dz;
      const auto r_inv = 1.0F / std::sqrt(r2);
      const auto r2_inv = r_inv * r_inv;
      const auto alp = pj.w * r_inv * r2_inv;

      // accumulation
      ai.x += alp * dx;
      ai.y += alp * dy;
      ai.z += alp * dz;
      ai.w += alp * r2;
    }
    iacc[i] = ai;
  }
}
```

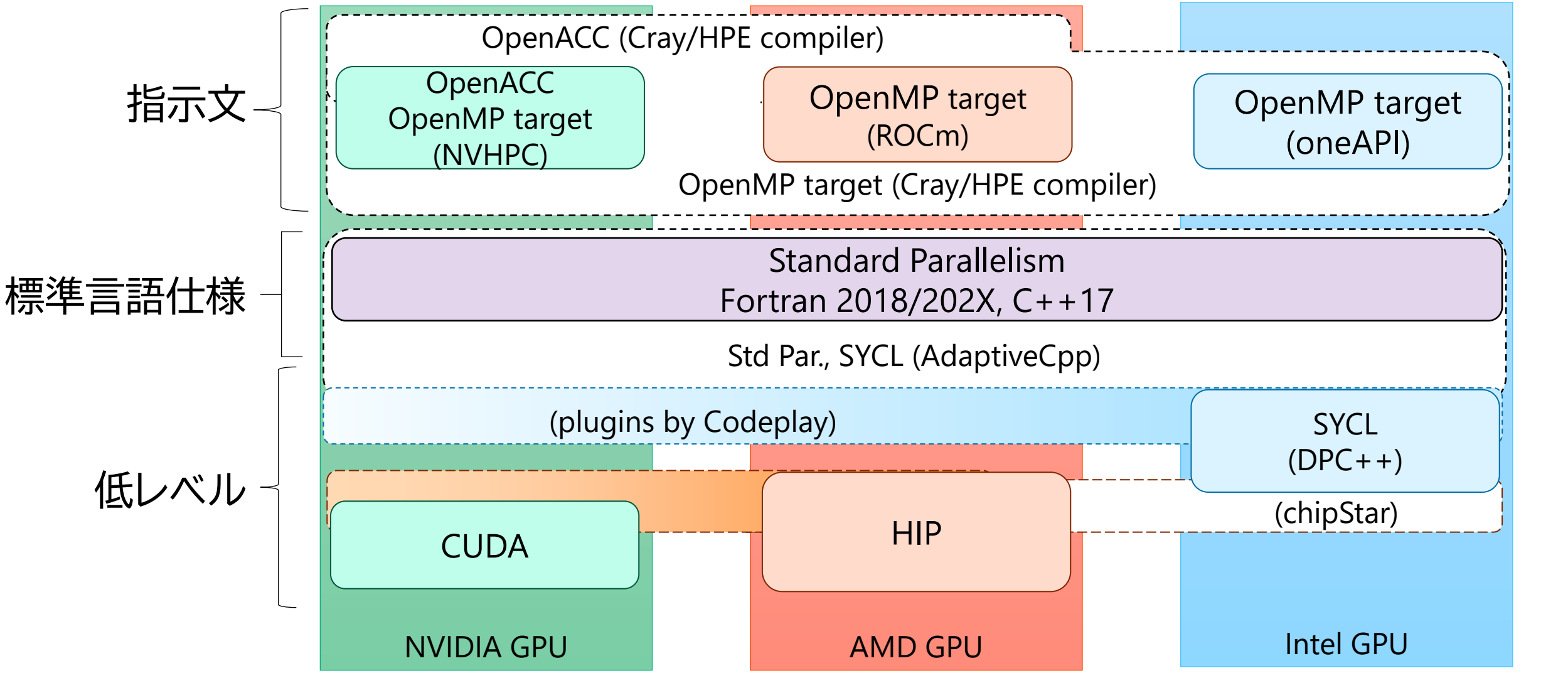
各要素が依存関係無しに共通の処理を実行
・スレッド並列可能
・スレッド数も十分多い

GPUスパコンの近況

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

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

利用可能な機能や性能はプログラミング手法に依存し，調査検討が必要

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

大まかな考え方の違い

- 簡易GPU化(OpenACC, OpenMP target, C++17, …)
  - 並列化可能なループであることをコンパイラに伝える
    (自明な並列度があり, コンパイラが問題なくGPU並列化出来るループが対象)
  - 「どうGPU化するか」はコンパイラ任せ(=GPUのことを分かっていなくても使える)
  - 初心者・初級者は, 細かい指定(ループの分割粒度など)は行わず,
    コンパイラに任せる(自由度を増やしてあげる)方が高速となる傾向
  - 初心者・初級者向け
- 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
  - 演算命令の選択など, 柔軟な記述が可能
  - 簡易GPU化ではGPU化が難しいループであっても実装可能
    - 例:重力ツリーコードの完全GPU実装は指示文では手に負えないが, CUDAなどでは実装可能
  - 中・上級者向け. 簡易GPU化で実装不可又は性能が引き出せない時に検討
    (今回の講習会では, GPU化の概念を理解するためにCUDAベースで進行)

GPUプログラミングに関する資料

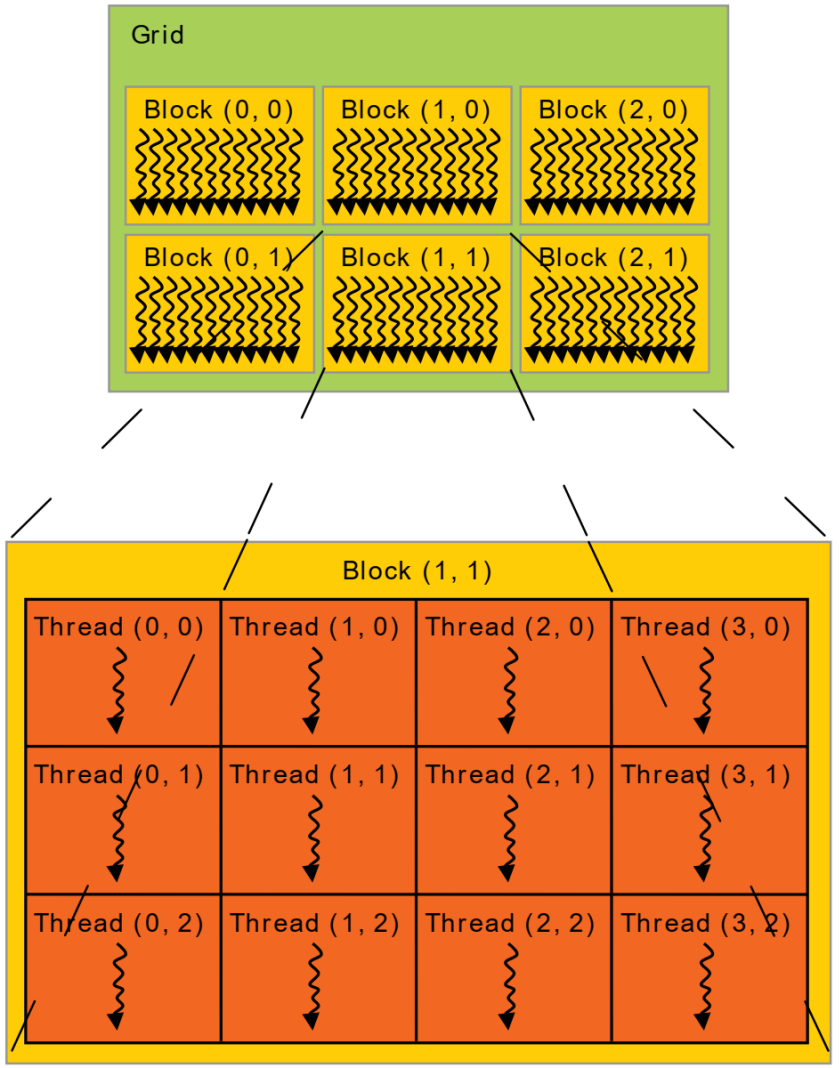
- HAIRDESC GPUプログラミング教材
  - https://hairdesc.jp/gpumaterial/
- 「UTokyo N-Ways to GPU Programming Bootcamp」講習会資料
  - https://www.cc.u-tokyo.ac.jp/events/lectures/207/
  - 講習会の録画も公開されています
  - ISO標準言語, OpenACC, CUDA
- 「GPUプログラミング入門」講習会資料
  - https://www.cc.u-tokyo.ac.jp/events/lectures/251/
  - OpenACC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

初日(8/18)の予定

- 10:00--11:00　並列計算の初歩　［講義］
- 11:15--12:15　GPUコンピューティングの基礎概念　［講義］
- 13:30--14:30　簡単なCUDAコードの実装　［講義・演習］
- 14:45--15:45　簡易なプロファイラと性能評価　［講義・演習］
- 16:00--17:00　GPU移植時の罠　［講義・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64-512スレッド程度
    - 1024スレッドまで使うことはある
  - 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
  - `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
  - ただし1 GPU / MPI プロセスであれば, CUDA_VISIBLE_DEVICESが便利
- 演算処理のGPU化
  - __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>>>())
- メモリ関連の処理
  - (Unified/Managed Memory を使うと, 実装は多少楽になる)
  - 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())
  - 確保したメモリの解放(cudaFree(), cudaFreeHost())

おおまかなGPU化方針

1. まずはループ構造(演算処理)のGPU実装(並列化)のみに注力
   - (マルチコアCPU向けの)OpenMP実装されていれば,
     `#pragma omp parallel for` をGPU化していく
     - メモリ上のデータがCPU側にあるか, GPU側にあるかは一旦忘れて実装
     - Unified/Managed Memoryという機能(後述)に,
       GPU上のメモリ確保, CPU-GPU 間のデータ転送は全てお任せ
2. Unified/Managed Memoryによる自動的なデータ転送のコストが問題となる場合は, 明示的にデータを転送する
   - CPU-GPU間のデータ転送を自動的に行うと, 冗長な転送が発生または最適なタイミングで転送されない場合がある
   - 必要なデータ転送は自分で指示した方が必然的に速くなる
   - GPUDirect系の機能を活用する際には, 指示文でGPU上のデータを明示的に管理する必要あり

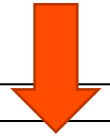
CUDA C++でのコンパイル方法

- `$ 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サポートがなくなった)
  - `-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 < num){…} 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することも多い
      - ブロック内スレッドの全体同期を取った際にデッドロックにならない, 少しでも無駄命令を減らしたい
      - 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

```
static inline void vecadd_omp
(const int num, const float val, const float* __restrict__ a, const float* __restrict__ b, float* __restrict__ c) {
#pragma omp parallel for
  for (int i = 0; i < num; i++) {
    c[i] = a[i] + val * b[i];
  }
}
```



```
__global__ void vecadd_gpu
(const int num, const float val, const float* __restrict__ a, const float* __restrict__ b, float* __restrict__ c) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < num) {
    c[i] = a[i] + val * b[i];
  }
}
```

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 NBLOCKS を使用
      (問題サイズをスレッド数ごとに分割していった時に, 全部で何ブロック必要か？ という数字)

```
#define NTHREADS (128)
#define NBLOCKS(num, threads) ((num) + (threads) - 1) / (threads))
```

  - <<<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)

```
// CPU
vecadd_omp(num, val, a0, b0, c0);

// GPU
vecadd_gpu<<<NBLOCKS(num, NTHREADS), NTHREADS>>>(num, val, a0, b0, c1);
```

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
  - malloc を cudaMallocManaged に変更する
    - cudaMallocManaged で確保したメモリは, cudaFreeで解放する
  - (CPU-GPU間のデータ転送は自分では何もしない)

```
// memory allocation
float *a0, *b0, *c0;
cudaMallocManaged(&a0, sizeof(float) * num);
cudaMallocManaged(&b0, sizeof(float) * num);
c0 = (float*)malloc(sizeof(float) * num);

// free memory
cudaFree(a0);
cudaFree(b0);
free(c0);
```

ここまでの知識でCUDA化はできてしまうので, 演習

- `$ git clone https://github.com/ymiki-repo/cfca-gpu2026f.git`
- `$ cd cfca-gpu2026f`
- `$ cd 0_vecadd`
- まずは自力で行けるところまで行ってみるという意図で, OpenMP版のコード(`1_omp`)をCUDA化してみてください
  - Makefile については, `2_cuda_managed/` に入っているものを使ってください
  - ジョブスクリプトは, `sh/cfca/run_gpu.sh` を使ってください

- ちなみに, `2_cuda_managed/` 内のコードにはバグがあります
  - 初心者が引っかかりがちな落とし穴にわざと引っかかったコード
    (後でここが間違っている, という点を解説します)

2_cuda_managed のコードのバグ(1/2)

- そのまま動かすと, 計算結果が不正と言われる
- GPU上での計算が終わる前にCPU上で validation が走ったのが原因
  - 何枚か前のスライドでコメントしていた部分:
    - 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
      - `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
  - ということで解決策は2つある
    1. CPU上で validation する前に, GPU側の実行を完了させる(`cudaDeviceSynchronize()`)
    2. validation 部分も GPU側に持っていく(同じデフォルトストリームに紐づくので, これでもOK)
- `3_cuda_managed_fix1st` のコードは, `cudaDeviceSynchronize()` を追加することで修正した
  - 動かすと分かるが, 計算結果は正しくなっている
  - ただし, CPUからの速度向上率が異常に高い(実はこれが二番目のバグ)

CUDA C++での実装(お手軽版:4/5)

- 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
  - 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

2_cuda_managed のコードのバグ(2/2)

- GPU上での実行が完了する前に時計を停めたため, 実行時間を過小評価
  - 要は, 先程のバグとほぼ同じ
  - GPU上での実行時間を正しく測定するには, 前後にcudaDeviceSynchronize()を入れてあげる
    - 前の処理が未完了な状態で測定を開始すると, 実行時間を過大評価
    - 注目処理が未完了な状態で測定を開始すると, 実行時間を過小評価

```
cudaDeviceSynchronize();
clock_gettime(CLOCK_MONOTONIC, &ini);

vecadd_gpu<<<NBLOCKS(num, NTHREADS), NTHREADS>>>(num, val, a0, b0, c1);

cudaDeviceSynchronize();
clock_gettime(CLOCK_MONOTONIC, &end);
```

- これで計算結果が正しくなり, 実行時間も正しく評価できるようになった
  - 4_cuda_managed_fix2nd/ がこの修正を適用したコード
  - しかし, 出力を見てみると, CPU実行の方が速いということになっている. なぜ？

初日(8/18)の予定

- 10:00--11:00　並列計算の初歩　［講義］
- 11:15--12:15　GPUコンピューティングの基礎概念　［講義］
- 13:30--14:30　簡単なCUDAコードの実装　［講義・演習］
- 14:45--15:45　簡易なプロファイラと性能評価　［講義・演習］
- 16:00--17:00　GPU移植時の罠　［講義・演習］

なぜ遅いのかを調べるにはどうすれば良いか？

- 時間がかかっている部分がどこかを調べるのが先決,
  その次に理由を調べるという手順になる
- プロファイラを使えば時間がかかっている部分を把握できる
  `$ 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
    （今回のケースは, こちらの出力だけで理由は分かる）
  - `reportN.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
    - 実問題の場合には, こちらの方針で調べることが多い
    - 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

nsys を使う際のトレース対象

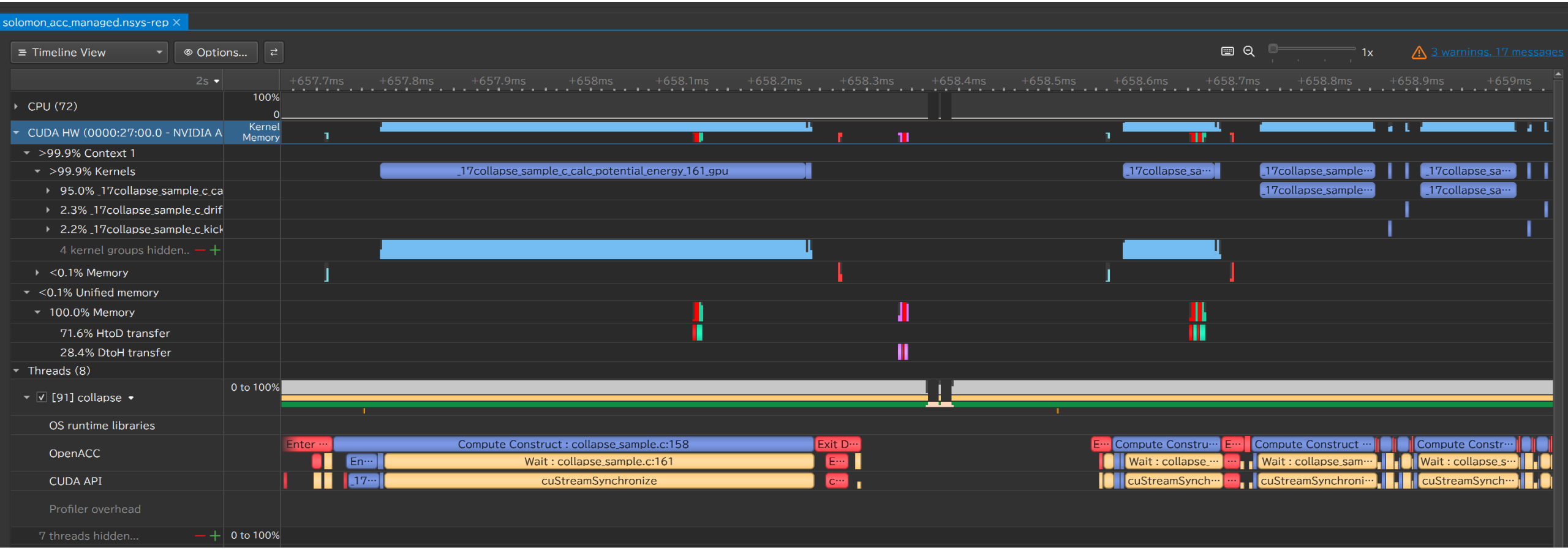
- `$ nsys profile --help` すると下記のトレース対象がリストされる

```
-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’.
```

  - バージョンを新しくすると, 他にも項目が追加されていたりする
- 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）

GUIツールを使って表示した例

- 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



今回の犯人はCPUからGPUへのデータ転送時間

- 現在のコードでは, Managed Memory を用いた実装としている
- Managed Memory 使用時の挙動は以下の通り:
  - GPU(CPU)で配列にアクセスした際に, GPU(CPU)側に最新データがなければページフォルトを起こす
  - ページフォルトが起こった際には, ランタイムがページサイズ(4KBのことが多い, GH200では64KB設定が推奨されていたりする)単位でデータコピー
- コードの実装時には意識していなかったが, 実際にはCPUからGPUへのデータ転送が発生している(でないと, 正しく計算できない)
  - 今回の演習環境では CPUとGPUはPCIe Gen4で接続(片方向32GB/s)
    - 実はこの帯域は, CPU側のメモリバンド幅よりも細い(= メモリ律速な問題ではここが律速)
  - 基本的には, GPU側にデータを置きっぱなしにして, CPU-GPU間のデータ転送をなるべく削るように実装してあげないと高速化されない
- 参考までに, GPU側にデータが移った後ではGPU側の計算の方が高速(`5_cuda_managed_final`)
  - GPUのメモリバンド幅がCPUよりも太いため

CUDA C++での実装(お手軽版:5/5)

- Unified/Managed Memoryを使わない場合は, データ転送も記述
- cudaMalloc でGPU上のメモリを確保
- cudaMemcpy でCPU･GPU間のデータ転送
  - このために用いるCPU側のメモリは, cudaMallocHostで確保することも多い(ピン留めされて, CPU･GPU間のデータ転送が高速化される)

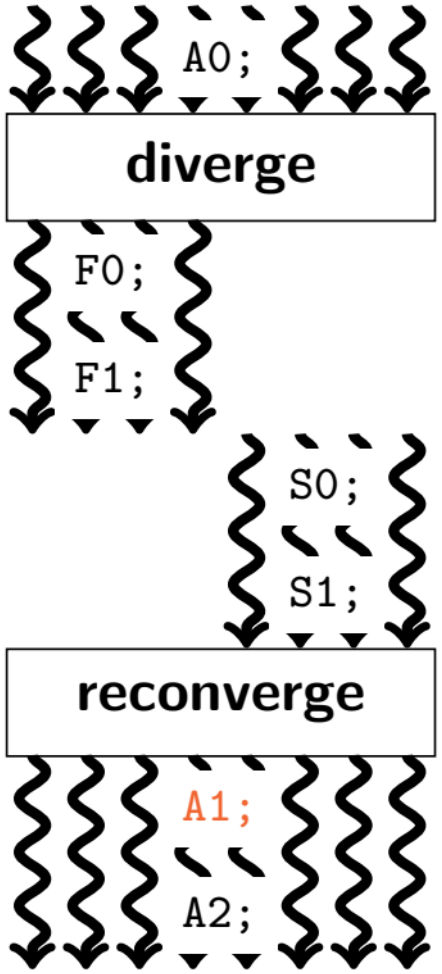
```
// memory allocation
float *a0;
cudaMallocHost(&a0, sizeof(float) * num);
float *a0_dev;
cudaMalloc(&a0_dev, sizeof(float) * num);

// CPU -> GPU transfer
cudaMemcpy(a0_dev, a0, sizeof(float) * num, cudaMemcpyHostToDevice);

// GPU -> CPU transfer
cudaMemcpy(a0, a0_dev, sizeof(float) * num, cudaMemcpyDeviceToHost);

// free memory
cudaFreeHost(a0);
cudaFree(a0_dev);
```

ワープ分裂(warp divergence)

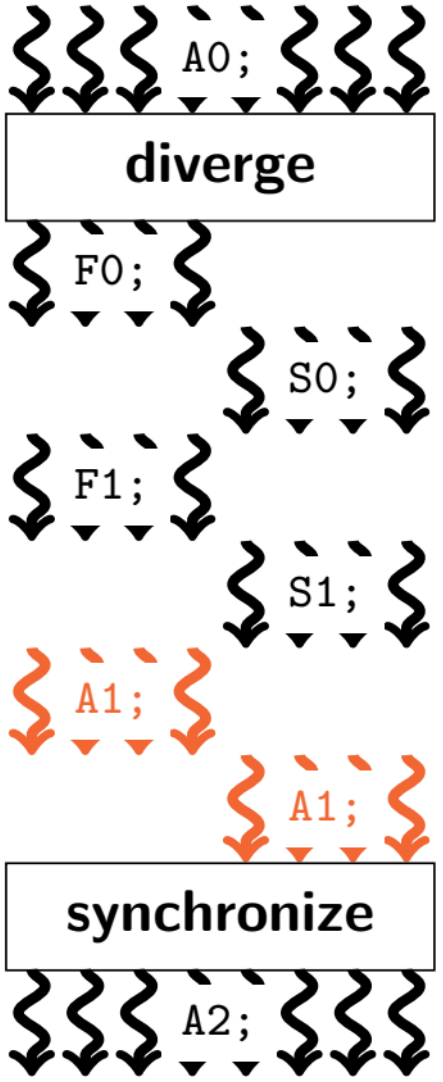
Pseudocode

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

Pascal or earlier

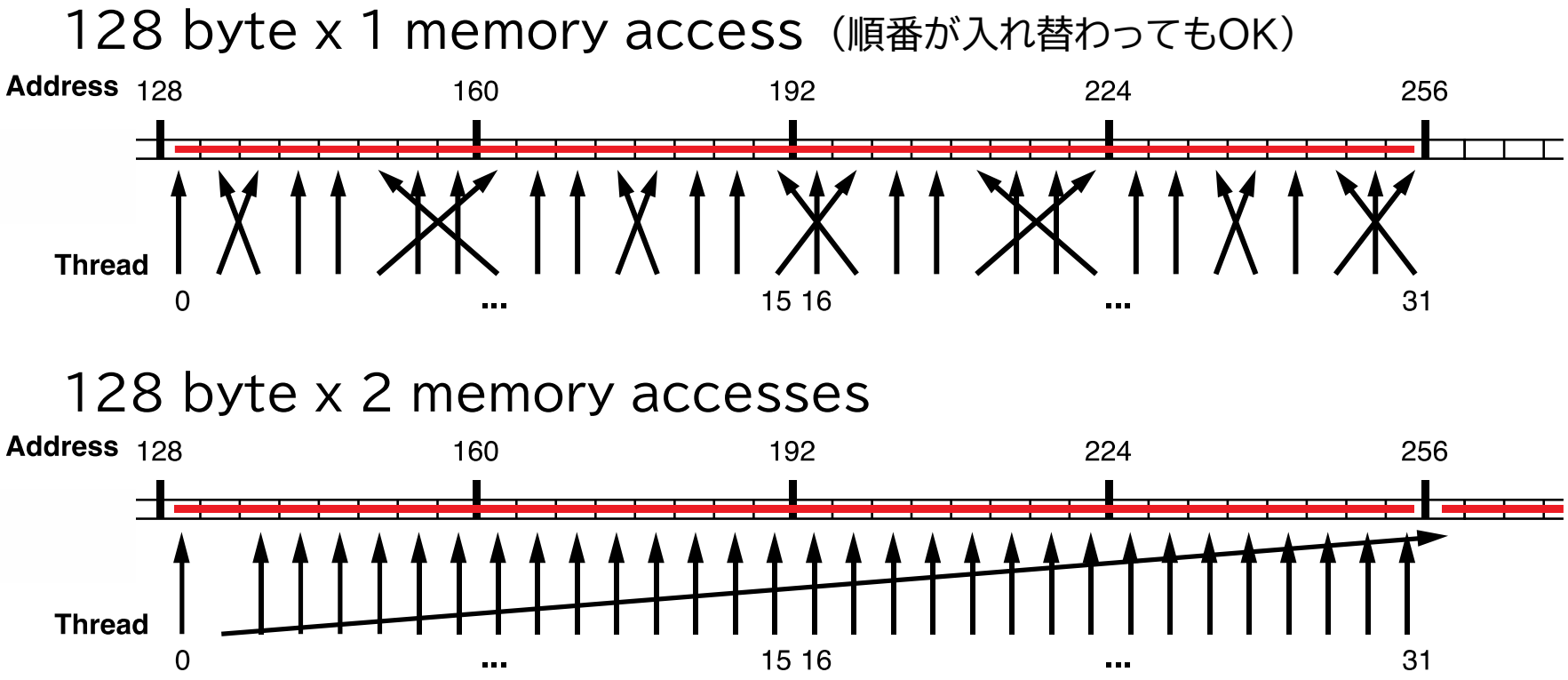


Volta or later



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



初日(8/18)の予定

- 10:00--11:00　並列計算の初歩　[講義]
- 11:15--12:15　GPUコンピューティングの基礎概念　[講義]
- 13:30--14:30　簡単なCUDAコードの実装　[講義・演習]
- 14:45--15:45　簡易なプロファイラと性能評価　[講義・演習]
- 16:00--17:00　GPU移植時の罠　[講義・演習]

N体コードのCUDA化(明日の内容)に向けての準備

- N体コードを眺めて, どういう処理がされているのかを把握しておく
  - まずはCPU実行してみて, どこに計算時間がかかっているかを把握しておく
    - コンパイルとリンクの両方に `-pg` を付ける
    - 実行すると, カレントディレクトリに `gmon.out` が生成される(正常終了が必要)
    - `$ gprof ./a.out gmon.out > profile.txt` で解析結果を出力
    - `flat profile(% time, self seconds)`で重い関数を特定
    - `call graph` で呼び出し関係と累積時間を確認
  - 並列化した方が良さそうな部分, 並列化しやすそうな部分, 並列化しづらそうな部分に目星をつけておく
    - 並列化した方が良い部分 = 高速化した方が良い部分 = 計算時間がかかっている部分
- GPU化にあたって障害になりそうな実装はあらかじめ修正しておく
  - 並列化しづらそうな部分があれば, 何が並列化の障害かを考えておく
  - スレッドごとの負荷バランスが悪くなりそうな部分があれば, 解消しておく
- CUDA化に着手する際には, フォルダごとコピーして別フォルダで作業するのをおすすめします(差分が見やすい, 最終日の指示文との比較も簡単)

計算機へのログインとサンプルの取得

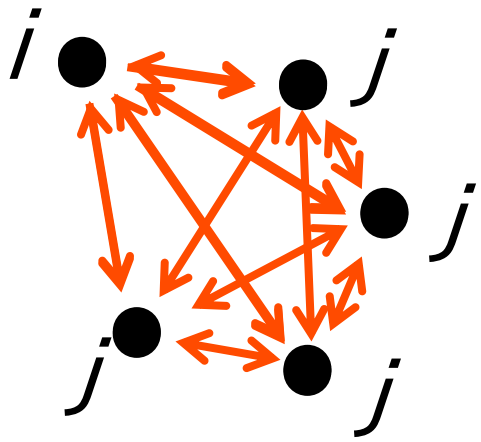
- （事前にVPN接続しておく）
- `$ ssh USERNAME@g00.cfca.nao.ac.jp`
- `$ cd /cfca-work/$USER`
- `$ mkdir 260818gpu-nbody` # 何か適当なフォルダを作る
- `$ cd    260818gpu-nbody` # 先ほど作ったフォルダに入る
- `$ cp /cfca-work/gpuws00/samples/cfca_sample.tar.bz2 .`
  - N体シミュレーション学校の途中成果コード（CPUのみで動作する単純なN体シミュレーションコード）とMakefileだけが入っています
- `$ tar -xvf cfca_sample.tar.bz2`
- `$ cp /cfca-work/gpuws00/samples/run_nvhpc.sh .`

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)

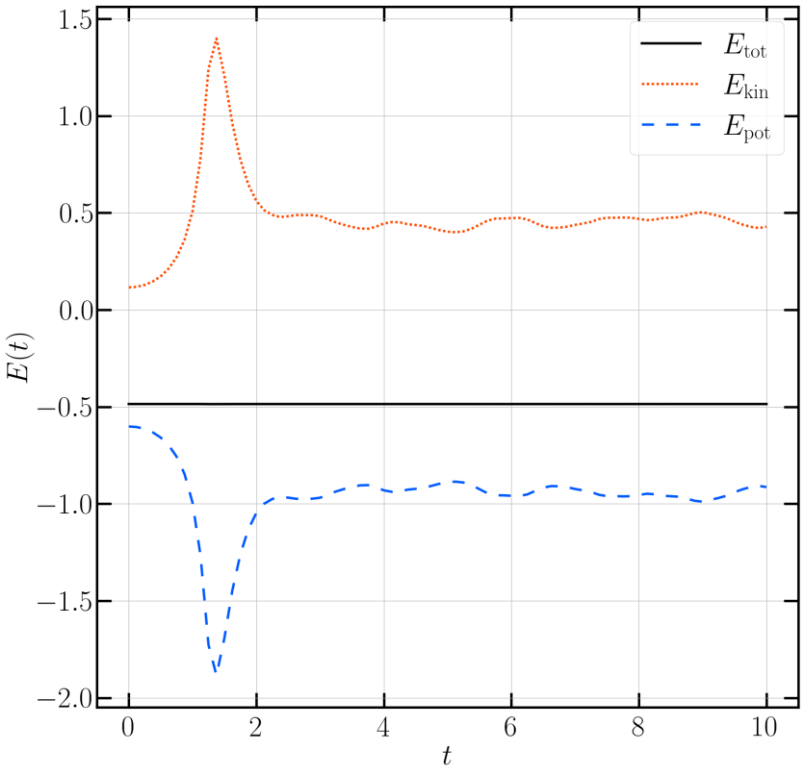
大まかな計算の手続き

- 初期条件はCPU側で生成(今回はcold collapse)
- 軌道積分は時間2次精度のLeapfrog法を用いる
  - Symplectic積分法の代表例(時間刻みΔtが固定というのが大前提)
  - $\boldsymbol{v}_i(t+\Delta t/2) = \boldsymbol{v}_i(t-\Delta t/2) + \Delta t\, \boldsymbol{a_i}(t)$,
  - $\boldsymbol{x}_i(t+\Delta t) = \boldsymbol{x}_i(t) + \Delta t\, \boldsymbol{v}_i(t+\Delta t/2)$.
  - 計算開始時やスナップショット出力時などに粒子速度の時刻を半ステップ分ずらす開始公式, 終了公式と組み合わせて使う
    - $\boldsymbol{v}_i(t+\Delta t/2) = \boldsymbol{v}_i(t) + \frac{\Delta t}{2}\boldsymbol{a_i}(t)$,
    - $\boldsymbol{v}_i(t) = \boldsymbol{v}_i(t+\Delta t/2) - \frac{\Delta t}{2}\boldsymbol{a_i}(t)$.
    - 毎ステップ使うのは損なので, 必要がなければ時刻をずらしたまま計算を進行させる
    - 注:エネルギー保存などをチェックする前には, 必ず全粒子の時刻を揃えておくこと

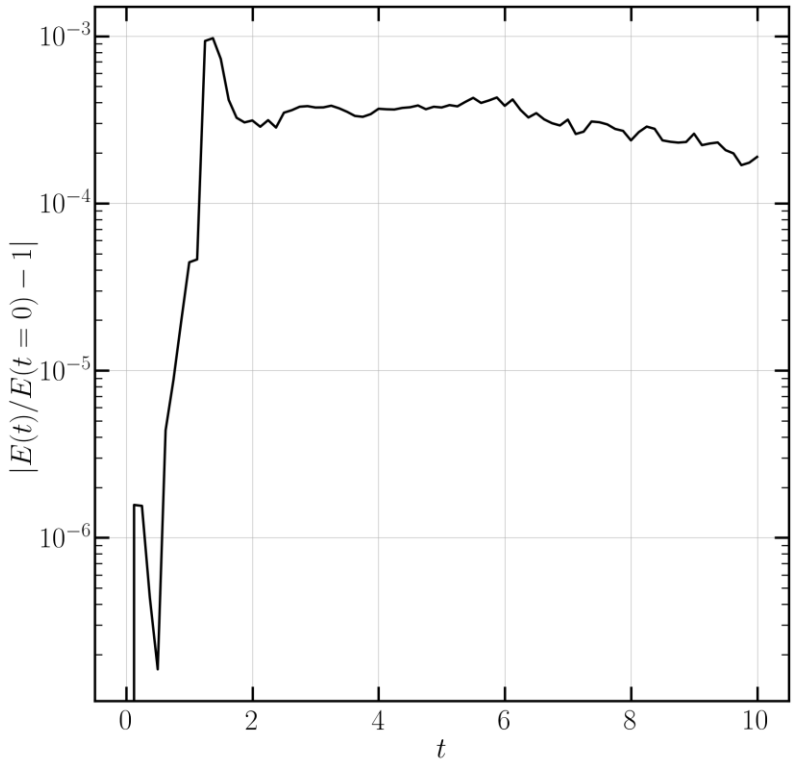
計算結果の可視化例(1/2)

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

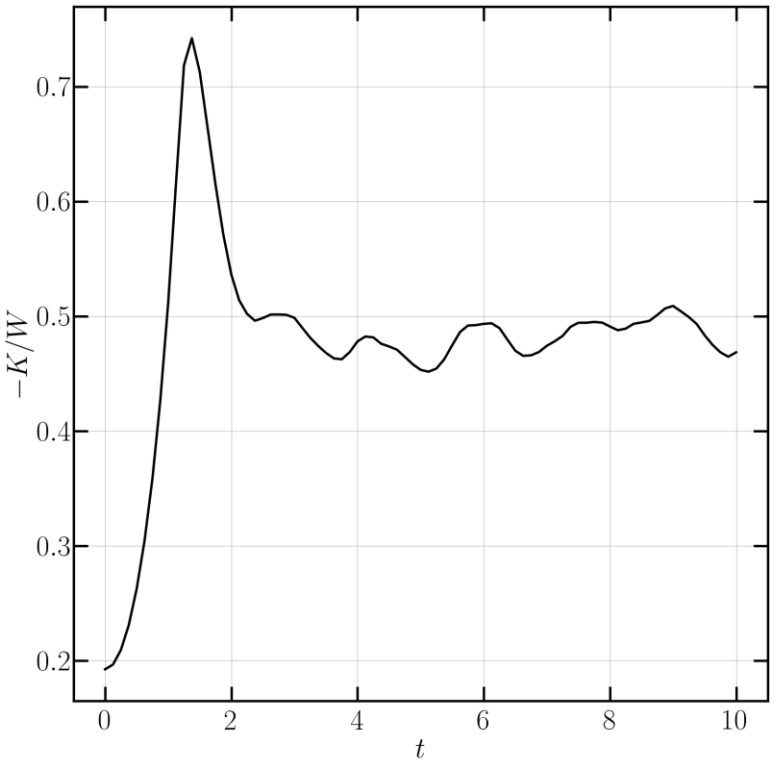
エネルギーの時間進化



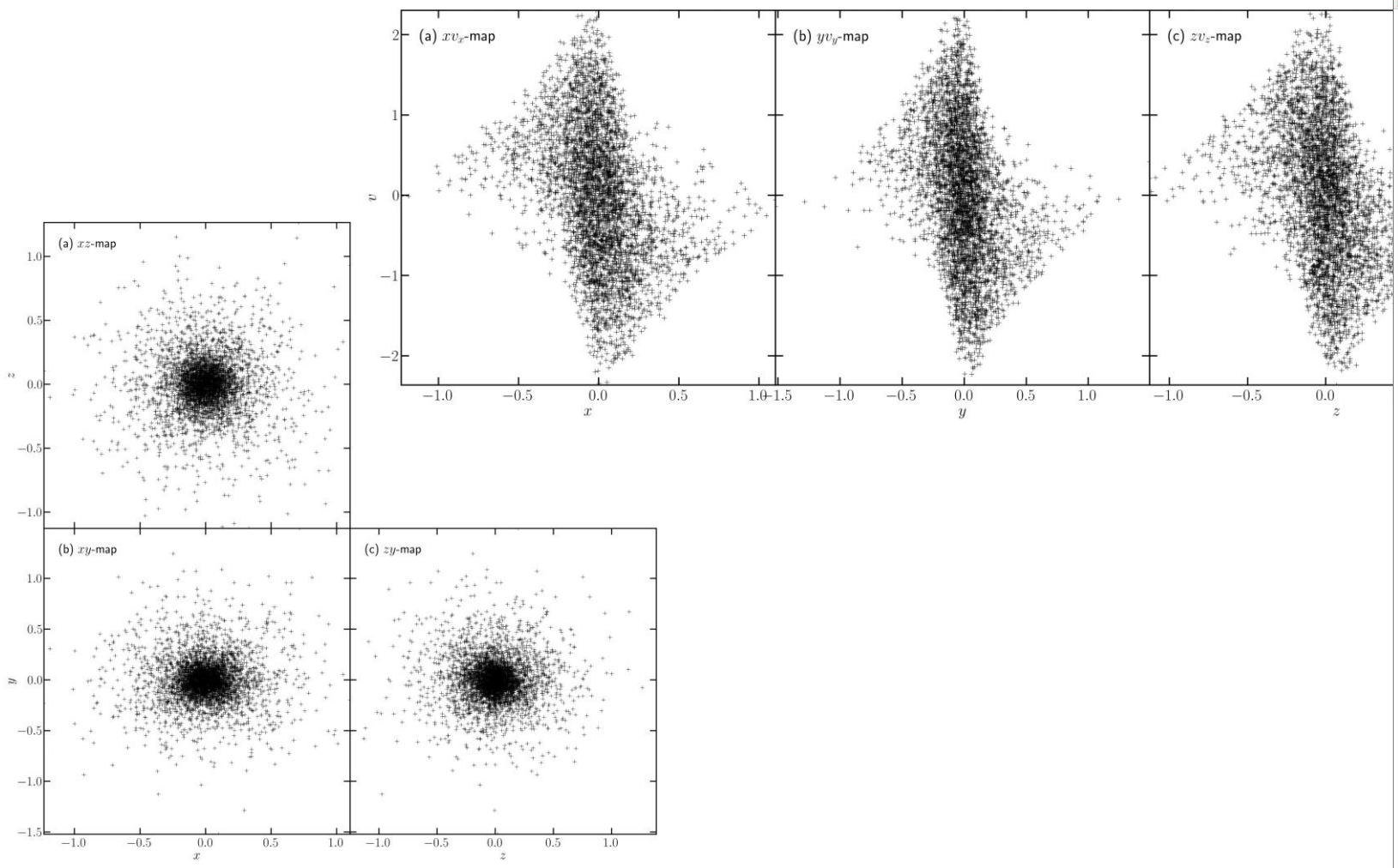
エネルギー保存

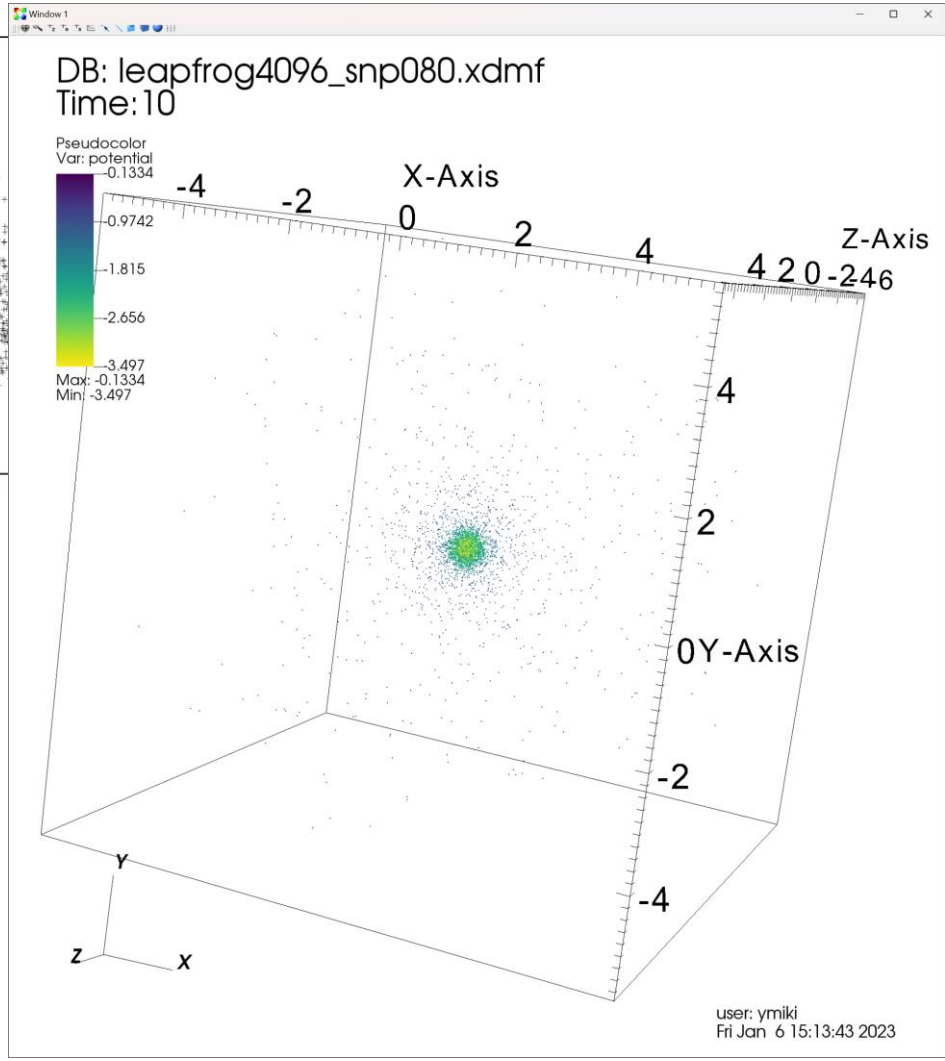


ビリアル比の時間進化



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, 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

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

今のうちに書き換えておくと良い内容

- 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
  - 動的にメモリ確保(malloc, new など)するように書き換えておくと良い
    - これは指示文を用いたGPU化においても共通
- 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
- 作用・反作用を使った実装になっている部分は, そのままCUDA化しようとすると厄介なので今のうちに何とかしておく
  - 配列への書き込みが競合する, スレッド間の負荷バランスが悪い, などの支障がある

Self-interaction を取り除くためのテクニック

- 単純にi-ループを0からN-1まで, j-ループを0からN-1まで回すと, 自分自身との相互作用を計算してしまうので, これを除去しないといけない
  - 元々の作用・反作用則を用いた実装では, i==jということは起こらなかった
- 重力の計算においては, Plummerソフトニングが入っているおかげでself-interactionは自動的に落ちる(分母は非ゼロ, 分子はゼロ)
  - $\boldsymbol{a}_i = \sum_{j=0, j\neq i}^{N-1} \frac{G m_j (\boldsymbol{x}_j - \boldsymbol{x}_i)}{\left(|\boldsymbol{x}_j - \boldsymbol{x}_i|^2 + \epsilon^2 \right)^{3/2}}$
    - 要は, $j \neq i$という条件をわざわざ課さなくても良い
- Self-interactionが問題になるのは重力ポテンシャルの評価
  - $\Phi(x = x_i) = -\sum_{j=0, j\neq i}^{N-1} \frac{G m_j}{\left(|\boldsymbol{x}_j - \boldsymbol{x}_i|^2 + \epsilon^2 \right)^{1/2}}$
  - つまり, self-interactionの項を足してしまうと$-Gm_i/\epsilon$だけオーバーカウント
  - if文を入れて除外するのが正攻法だが, 最後にGm_i/ϵを足して辻褄合わせしても良い
    - 浮動小数点演算の誤差が溜まりやすい処理ではあるが, 本計算の精度に影響ないのでそこまで気を使わなくて良い(と割り切ってしまえば, 条件分岐をなくせる)

C/C++ 向けの(一般的な)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)

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 $*
}
```

おまけ

- GPU実装前に各種落とし穴を回避するために書き換えたコード:
  - `g00:/cfca-work/gpuws00/cfca-gpu2026f/nbody/base`
- これが正解というわけでもないが, 参考実装として眺めてみてください
  - ここ, なぜ書き換えているのだろう？　と疑問に感じられればそれだけですごい（一通り解説するつもりですが, 解説漏れがあれば指摘してください）

- 昔CUDA, OpenACC, OpenMP target, C++17 stdpar の参考実装として作ったN体コード
  - https://github.com/ymiki-repo/nbody
  - 2次Leapfrog, 4次Hermiteの実装例が入っている
  - （はじめにCUDA実装を作って後にCPU化, 指示文付与などとして作ったもの）
- HAIRDESC事業で整備中のサンプルコード群
  - https://github.com/SCD-ITC-UTokyo/gpu-apps-tutorial
  - CUDA実装はまだないが, N体, 拡散方程式, 有限要素法のサンプルが入っている