2026年度GPU講習会（2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

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]

- 今日は演習中心として, CUDA実装に慣れることを目的にします

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]

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)

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

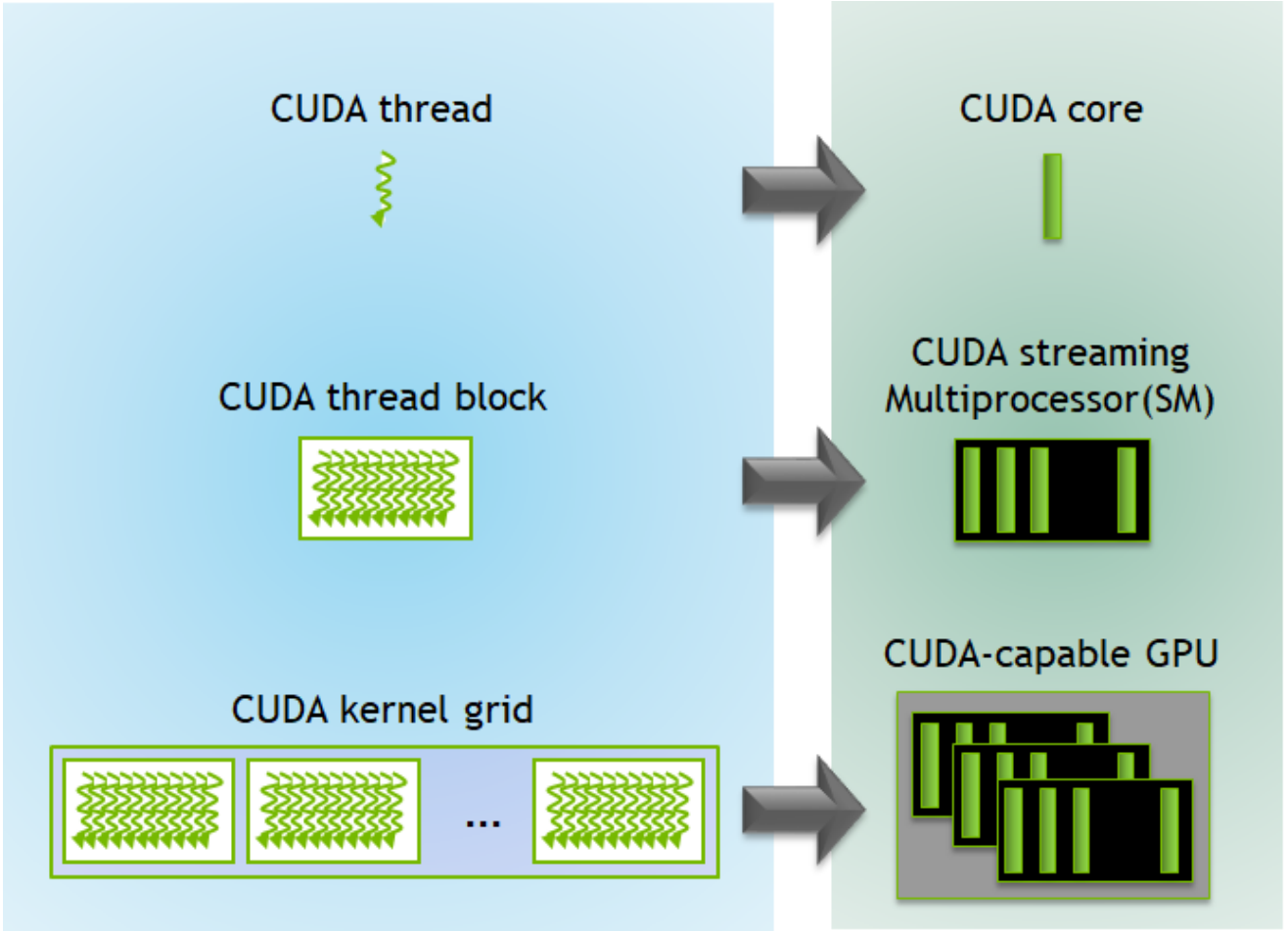
```
static inline void calc_acc(const int32_t Ni, const float4
*const ipos, float4 *__restrict iacc, const int32_t Nj,
const float4 *const jpos, const float eps2) {
#pragma omp parallel for
  for (int32_t i = 0; i < Ni; i++) {
    const auto pi = ipos[i];
    float4 ai = {0.0F, 0.0F, 0.0F, 0.0F};

    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;
  }
}
```

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

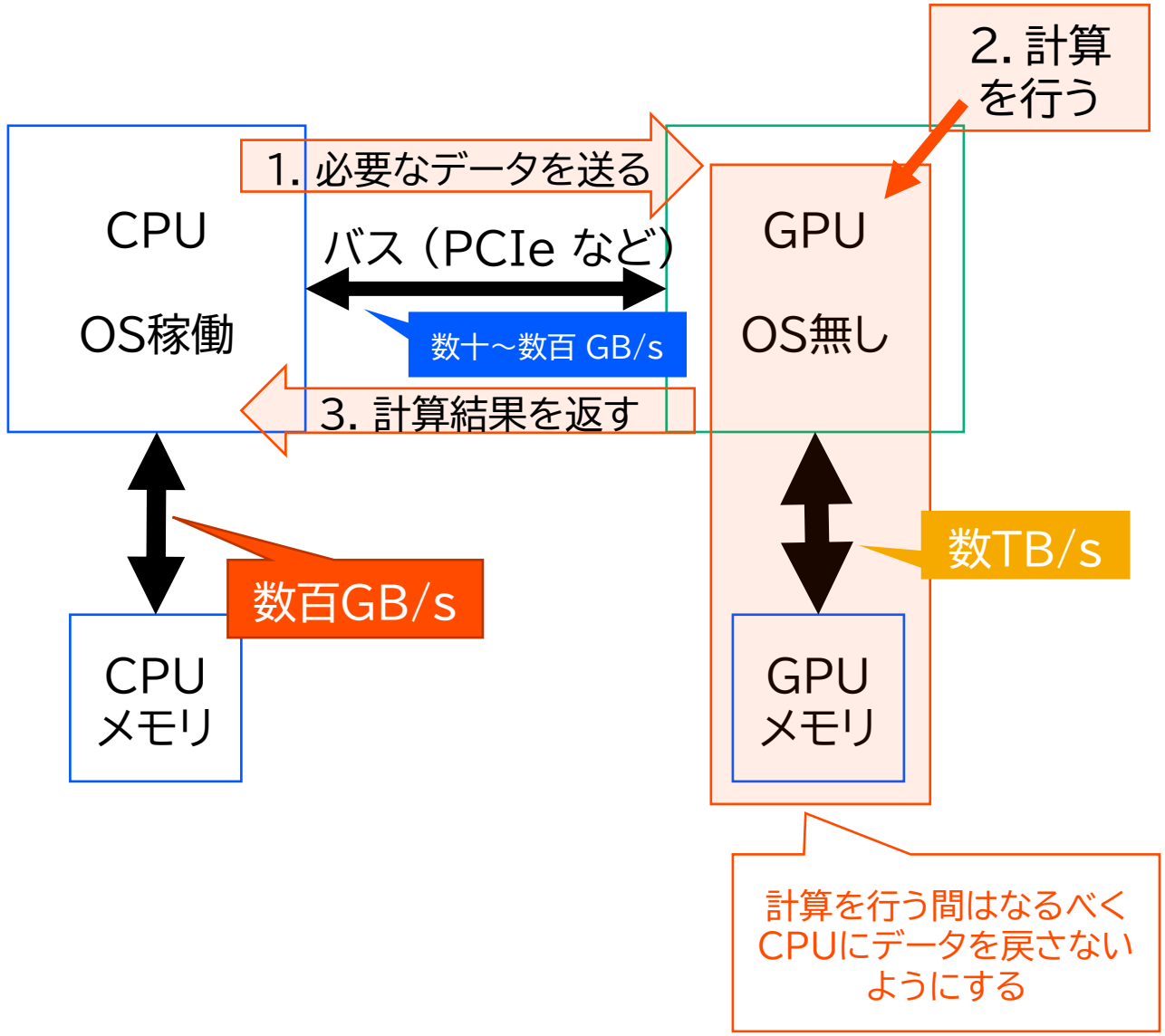
- NVIDIA CUDAアーキテクチャを例に，スレッドと演算コアの対応を説明
- 1スレッド(1要素に相当すること多い)が1コアに対応
- スレッドブロック(64から512程度のスレッド群に分割されたもの)を任意のSMに割り当てて処理させる
  - 多く割り当てた方が性能が出る傾向
  - 各種レイテンシを隠蔽できる
- ブロック全て(＝グリッド)をGPUに割り当てて領域全体を処理する
- 「各種レイテンシの隠蔽」，「ハードの各階層でのキャッシュ等の活用」には適切なGPUスレッドの割り振りが必要



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

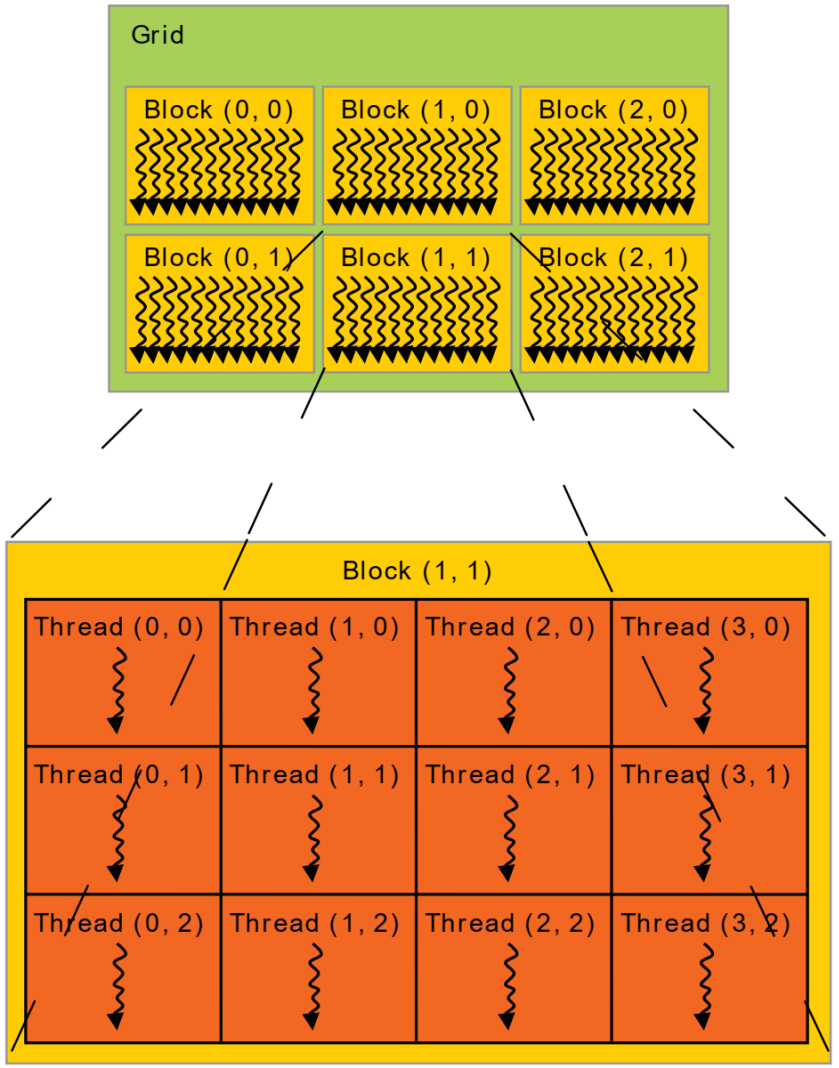
- 各スレッドの演算処理にデータ依存性がない
  - 並列化可能となり，スレッドを並行して処理することで各種レイテンシを隠蔽できる
  - `if`文などで処理が分裂すると，冗長な処理となりコスト増加
  - 数十スレッド単位で同じ条件に分岐（例：連続する32スレッド全て真）する場合は，
    冗長な処理によるコスト増は抑えられる

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)

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)

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

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

N体コードのCUDA化(今日の内容)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
  - 並列化しやすそうな部分, しづらそうな部分に目星をつけておく
- 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

計算機へのログインとサンプルの取得

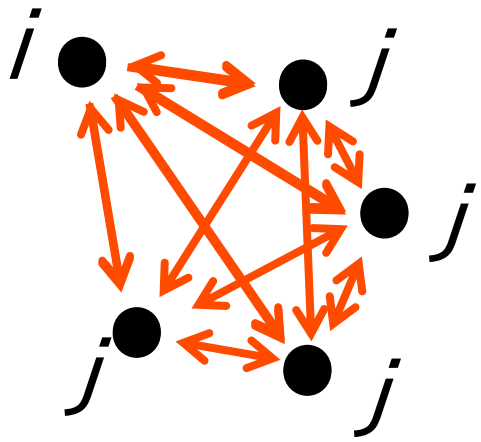
- （事前にVPN接続しておく）
- `$ ssh USERNAME@g00.cfca.nao.ac.jp`
- `$ cd /cfca-work/$USER`
- `$ mkdir 260818gpu-nbody` # 何か適当なフォルダを作る
- `$ cd    260818gpu-nbody` # 先ほど作ったフォルダに入る
- `$ cp /cfca-work/gpuws00/samples/cfca_sample.tar.bz2 .`
  - N体シミュレーション学校の途中成果コード（CPUのみで動作する単純なN体シミュレーションコード）とMakefileだけが入っています
- `$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

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

C/C++ 向けの最適化アドバイス

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

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

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

  - CUDAの場合には, __restrict__ がこれに対応

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

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

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 $*
}
```

2日目(8/19)の予定

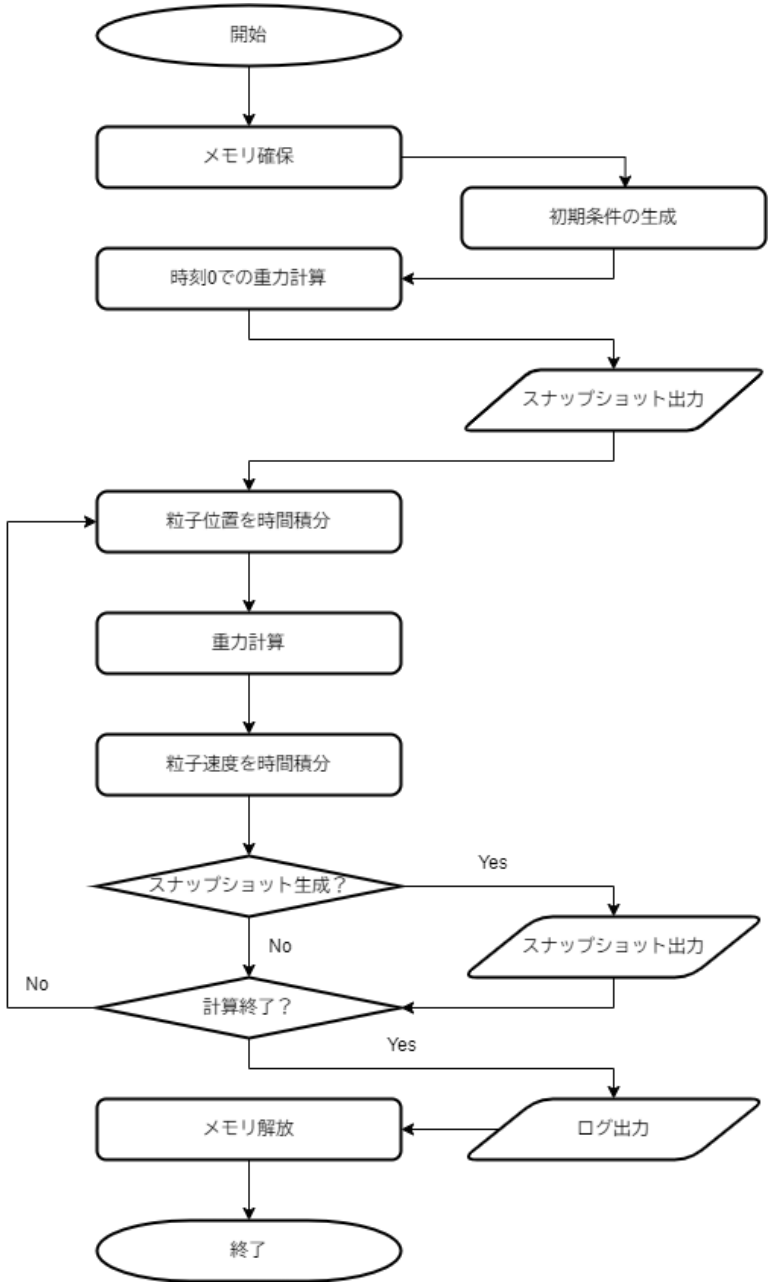
- 9:00--9:45　前日の復習
- 10:00--11:00　無衝突系N体コードのCUDA移植1　[講義・演習]
- 11:15--12:15　無衝突系N体コードのCUDA移植2　[演習]
- 13:30--14:30　ライブラリの利用　[講義・演習]
- 14:45--15:45　衝突系N体コードのCUDA移植　[講義・演習]
- 16:00--17:00　CUDAコードの性能最適化　[講義・演習]

これからの作業

- 昨日の資料を参考に, N体コードのGPU化に取り組んでみましょう
  - 昨日書き換えたコードを別フォルダにコピーしてから作業するのがおすすめ
- まずはGPU上で正しくN体計算できるようになることを目指す
  - 粒子数は少なめにしておく(N=1024, 512など)
  - 意地悪な設定にしても正しく解ければなお良い(N=1031, 521 などの素数)
    - 素数の場合, 通常32の倍数で設定するスレッド数で割り切れないため, 端数処理が発生する
- 正しく計算できるようになったら, 性能を確認してみる
  - スレッド数を変えると性能は変化する？
  - 粒子数を増やしていった時の実行時間の増え方は？
    (1ステップの実行時間を測定して終了するモードを追加すれば,
    N=1Mなどでも測定できる)
    - NVIDIA A100 の場合, 色々がんばって最適化するとN=1Mでは1ステップで2秒を切るぐらい
- 性能最適化については後でも時間を取るので, 今の目標は
  - 正しく動かせるようになること
  - CPU実行よりも速くすること

参考になる(かもしれない)N体コードのGPU実装例

- https://github.com/ymiki-repo/nbody
- 東大の講習会で使っているサンプルコード
  - cpp/cuda/ の下に CUDA C++ で実装したN体コードが入っている
  - nbody_leapfrog2.cu というのが,無衝突系向けN体コードの実装例(cpp/base/0_base/nbody_leapfrog2.cppからのCUDA化の方が実は楽かも)
- 右側のフローチャートのような挙動がゴール
  - 初期条件生成, (必要なら)スナップショット出力はCPU実行でOK
    - CPU-GPU間のデータ転送はこの前後で必要になる
  - 重力計算, 軌道積分をGPU化できれば,必要なデータはGPU側に置きっぱなしにできる



比較的つまづきにくいCUDA移植手順

- ソースコードの拡張子を .c から .cu に変更(あわせてMakefileも編集)
- 以降, 各手順に伴うコード編集をするたびにコンパイル・実行して結果確認
  - 元と結果が不一致であれば, どこが原因かを探す(差分が小さいほど見つけやすい)
  - CUDA化が完了するまでの時点では元のCPU実装より遅いこともある
- メモリ確保＋解放を `cudaMallocManaged` ＋ `cudaFree` に書き換え
  - この時点では, 全ての処理はCPU(ホスト)上で実行している
- 重力計算関数(`calc_force`)だけをCUDA版に書き換え
  - GPU(デバイス)での処理後, CPU(ホスト)実行前に `cudaDeviceSynchronize()` を入れる
- 軌道積分関数をCUDA版に書き換え(位置, 速度の書き換えは個別に)
- 運動エネルギーの計算関数をCUDA版に書き換え
- ポテンシャルエネルギーの計算関数をCUDA版に書き換え
- メモリ確保＋解放を `cudaMalloc`/`cudaMallocHost` ＋ `cudaFree`/`cudaFreeHost` に書き換え, `cudaMemcpy` も追加
  - 不要になった `cudaDeviceSynchronize()` を(一つずつ)外す

Reductionをどう書けば良いか？

- 全体をatomicAddで足し込みとすると遅くなりすぎるので, ブロック内は一工夫
- ブロック内でのreduceは, Cooperative Groupsを使うと楽に書ける
  - ワープシャッフルやシェアードメモリを駆使しても書けるが, この方が楽
- CUBなどではデバイス内reduceも提供されているはずなので, そちらの使い方を調べて使ってみるというのでもOK

```
#include <cooperative_groups.h>
#include <cooperative_groups/reduce.h>

__global__ void calc_kinetic_energy_kernel(const position_t *__restrict__ pos, const
velocity_t *__restrict__ vel, fp_m *__restrict__ K_result) {
  auto block = cooperative_groups::this_thread_block();
  const int i = blockIdx.x * blockDim.x + threadIdx.x;

  fp_m K_sub = AS_FP_M(0.0);
  const velocity_t vi = vel[i];
  K_sub += pos[i].w * (vi.x * vi.x + vi.y * vi.y + vi.z * vi.z);

  // Block-level reduction using thread_block_tile
  auto tile = cooperative_groups::tiled_partition<NTHREADS>(block);
  K_sub = cooperative_groups::reduce(tile, K_sub, cooperative_groups::plus<fp_m>());

  // Each block writes its result atomically
  if (tile.thread_rank() == 0) {
    atomicAdd(K_result, K_sub);
  }
}

fp_m calc_kinetic_energy(const int n, const position_t *pos, const velocity_t *vel,
fp_m *__restrict__ val_dev, fp_m *__restrict__ val_hst) {
  zero_clear<<<1, 1>>>(val_dev);
  calc_kinetic_energy_kernel<<<NBLOCKS(n, NTHREADS), NTHREADS>>>(pos, vel, val_dev);

  cudaMemcpy(val_hst, val_dev, sizeof(fp_m), cudaMemcpyDeviceToHost);
  return (AS_FP_M(0.5) * (*val_hst));
}
```

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]

ライブラリの利用

- GPUコードにおいても, ライブラリを利用するケースは多い
  - ソート
  - FFT(Fast Fourier Transform)
  - BLAS(Basic Linear Algebra Subprograms)
  - 疑似乱数の生成
  - MPI(Message Passing Interface)
- GPU向けだからといってCPUのときと大きく変わる話ではない
  - もちろん, ライブラリごとに呼び出し方が違うことはある
  - Unified/Managed Memory を使っているかには気をつけたほうが良い (ライブラリ関数に配列を渡す際に, CPU側なのかGPU側なのかを意識したい)
    - 実はこのあたりは指示文でのGPU化をしている時の方が一手間増える
- HAIRDESCのGPUプログラミング教材に「ライブラリ編」が追加された
  - https://hairdesc.jp/gpumaterial/

GPU上のソートライブラリの使い方を紹介

- CUB に入っている `cub::DeviceRadixSort::SortPairs` を紹介
  - 以前は CUB として提供されていたが, 最近は CCCL の一部という扱い
  - CCCL（CUDA Core Compute Libraries）
    - Thrust, CUB, libcudacxx がまとまったもの
- ここでソートを例題として扱うのは, 衝突系N体計算の実装で使うため
  - ソート以外の部分は, 衝突系N体コードについても今までの知識だけで実装可能（individual/block time step にしないならばソートは不要ではあるが...）
- 粒子データをソートする際, 全データを構造体に詰めた上でソートする例をよく見かける
  - プログラムとしては, この方が簡単に書ける
  - ただし, データ移動量が無駄に増えるので, 性能的には嬉しくない
- キーのソート完了後に, 粒子データを移し替える実装を紹介（次ページ）

CUBの使用例

- 右側が使用例
  - cub::DeviceRadixSort::SortPairsは, キー(nxt)の値が昇順になるようにtagの値を並べ替えている
  - ソート対象のデータは全てGPUに置いてあるので, そのまま関数に渡してあげている
  - 粒子データのソートは別実行
- この関数は作業配列を確保しておく必要がある
  - 一度空回しして必要なサイズを見積もり, 確保する(↓)

```
#include <cub/device/device_radix_sort.cuh>

__global__ void sort_particles_device(const float4 *const src_pos, ..., float4
*__restrict dst_pos, ..., const int *const tag) {
  const int i = blockIdx.x * blockDim.x + threadIdx.x;
  const auto j = tag[i];
  dst_pos[i] = src_pos[j];
  dst_vel[i] = src_vel[j];
  dst_acc[i] = src_acc[j];
  dst_jrk[i] = src_jrk[j];
  dst_prs[i] = src_prs[j];
  dst_idx[i] = src_idx[j];
}

static inline void sort_particles(const int num, int *__restrict tag0_dev, int
*__restrict tag1_dev, type::nbody *__restrict src, type::nbody *__restrict dst,
void *temp_storage, size_t &temp_storage_size) {
  // sort particle time
  reset_tag_device<<<BLOCKSIZE(num, NTHREADS), NTHREADS>>>(tag0_dev);
  cub::DeviceRadixSort::SortPairs(temp_storage, temp_storage_size, (*src).nxt,
(*dst).nxt, tag0_dev, tag1_dev, num);
  // sort N-body particles
  sort_particles_device<<<BLOCKSIZE(num, NTHREADS), NTHREADS>>>(
(*src).pos, (*src).vel, (*src).acc, (*src).jrk, (*src).prs, (*src).idx,
(*dst).pos, (*dst).vel, (*dst).acc, (*dst).jrk, (*dst).prs, (*dst).idx, tag1_dev);
  // swap SoAs
  const auto _tmp = *src;
  *src = *dst;
  *dst = _tmp;
}
```

```
temp_storage_size = 0U;
*temp_storage = NULL;
cub::DeviceRadixSort::SortPairs(*temp_storage, temp_storage_size, *nxt0_dev, *nxt1_dev, *tag0_dev, *tag1_dev, size);
cudaMalloc(temp_storage, temp_storage_size);
```

GPU向けライブラリを使用する際に気をつけること

- ライブラリに渡すデータが, GPU側に置かれているのか, CPU側に置かれているのかに気を配っておく
  - CUDAで実装していて, 自分でcudaMallocで確保している際には分かりやすい
  - Unified/Managed Memory を用いた実装においては自明ではない
    - 裏で(予期せぬ)CPU･GPU間データ転送が走ることもある
    - 明示的にGPU上のデータを扱わせるために, 専用の配列を確保する場合も見聞きする
- (本講習会ではまだ扱っていないが)指示文でのGPU化ではもう一つ注意点
  - 指示文実装では(Separateの場合であっても)CPU側の配列名とGPU側の配列名が同じになっている➔配列の名前を見ただけではCPU側か, GPU側かは分からない
  - GPU上のアドレスである, ということをコンパイラに伝えるための指示文がある
    - OpenACCでの実装例:

```
#pragma acc host_data use_device(recv_buf)
  MPI_Irecv(recv_buf, recv_count, MPI_DOUBLE, recv_from, tag, MPI_COMM_WORLD, &recv_req);
```

    - OpenMP targetでの実装例:

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

    - Solomonでの実装例:

```
USE_DEVICE_DATA_FROM_HOST(recv_buf)
  MPI_Irecv(recv_buf, recv_count, MPI_DOUBLE, recv_from, tag, MPI_COMM_WORLD, &recv_req);
```

衝突系向けN体計算コードのGPU化

- 時間2次精度のLeapfrog法では正しく粒子軌道の時間進化を追えない問題もある
- そうした場合には, もっと高次精度の積分法を使う
  - 同時に重力計算の精度も上げておく必要もある(今回は直接法なので元々OK)
- 代表的なのは時間4次精度のHermite法
  - 予測子・修正子法の一つであり, 修正子の構成に3次のHermite補間を利用している
  - 予測子(全粒子の時刻を揃えておく必要がある)
    - $\boldsymbol{x}_{j,\mathrm{p}} = \boldsymbol{x}_j + \Delta t_j \boldsymbol{v}_j + \frac{(\Delta t_j)^2}{2!}\boldsymbol{a}_j + \frac{(\Delta t_j)^3}{3!}\boldsymbol{j}_j,\ \ \boldsymbol{v}_{j,\mathrm{p}} = \boldsymbol{v}_j + \Delta t_j \boldsymbol{a}_j + \frac{(\Delta t_j)^2}{2!}\boldsymbol{j}_j,$
  - 修正子(時間積分の対象とする粒子についてだけの処理で良い)
    - $\boldsymbol{x}_{i,\mathrm{c}} = \boldsymbol{x}_{i,\mathrm{p}} + \frac{(\Delta t_i)^4}{4!}\boldsymbol{a}_{0,i}^{(2)} + \frac{(\Delta t_i)^5}{5!}\boldsymbol{a}_{0,i}^{(3)},\ \ \boldsymbol{v}_{i,\mathrm{c}} = \boldsymbol{v}_{i,\mathrm{p}} + \frac{(\Delta t_i)^3}{3!}\boldsymbol{a}_{0,i}^{(2)} + \frac{(\Delta t_i)^4}{4!}\boldsymbol{a}_{0,i}^{(3)},$
  - 加速度の時間1階微分(よくjerkと呼ぶ)も計算に使う
    - $\boldsymbol{j}_i = \dot{\boldsymbol{a}}_i = \sum_{j\neq i} G m_j \left[\frac{\boldsymbol{v}_{ji}}{\left(\epsilon^2 + x_{ji}^2\right)^{3/2}} - \frac{3(\boldsymbol{x}_{ji}\cdot\boldsymbol{v}_{ji})\boldsymbol{x}_{ji}}{\left(\epsilon^2 + x_{ji}^2\right)^{5/2}}\right],\ \text{where } \boldsymbol{x}_{ji} \equiv \boldsymbol{x}_j - \boldsymbol{x}_i, \boldsymbol{v}_{ji} \equiv \boldsymbol{v}_j - \boldsymbol{v}_i.$
    - ここでは重力ソフトニングありの式にしたが, 含めないことの方が多いはず

独立時間刻みの導入

- 粒子ごとに軌道進化のtimescaleが(桁で)異なることが多いため, 全粒子の時間刻みを共通にしておく(shared timestep)と計算量が過大になる
- 粒子ごとに時間刻みを設定する(individual timestep)ことで効率化
  - 並列化の効率を考えて粒子グループごとの時間刻みとして設定することが多い (block timestep, hierarchical timestep)
- どう実装するか？
  - 粒子ごとに適切な時間刻みΔt_iを設定する(Hermite法の場合, 経験的な式がある)
  - 粒子ごとに, 時間積分後の時刻$t_{\mathrm{next},i} \equiv t_i + \Delta t_i$を計算する
  - 粒子データを$t_{\mathrm{next},i}$の昇順でソート
  - 時間積分後の時刻, 軌道積分を実行するべき粒子群が決まる

サンプルコードの取得とコンパイル方法

- `$ git clone https://github.com/ymiki-repo/nbody.git`
- `$ cd nbody`
- `$ module purge`
- `$ module load nvhpc/25.7 cmake`
- `$ module use /cfca-work/gpuws00/opt/modules/lib`
- `$ module load boost hdf5`
- `$ cmake -S . -DUSE_CUDA=ON -DGPU_VENDOR=NVIDIA -DGPU_ARCH=80 -DHERMITE_SCHEME=ON`
- `$ cd build`
- `$ ccmake -S ..`
  - 調整したい設定があればここで調整できる
- `$ cmake --build .`
  - `$ cmake --build . --target cuda_memcpy_shmem` とするとコンパイル対象削減

実行方法と演習モード

- `$ sbatch --export=EXEC="bin/cuda_memcpy_shmem" sh/cfca/run_nvhpc.sh`
  - 実行ファイルを外から指定してジョブを投入する方法
    （デフォルトでは, `EXEC=bin/base` となっている. スクリプトを直接編集してもOK）
- 計算結果は `dat/` に粒子分布が，`log/` にエネルギー誤差や実行時間が出力される

- 演習モードに入る方法:
- `cmake` する際に，`-DEXERCISE_MODE=ON` を付与する
- あるいは, `ccmake` して `EXERCISE_MODE` を `ON` にする

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]

GPU版N体コードの最適化

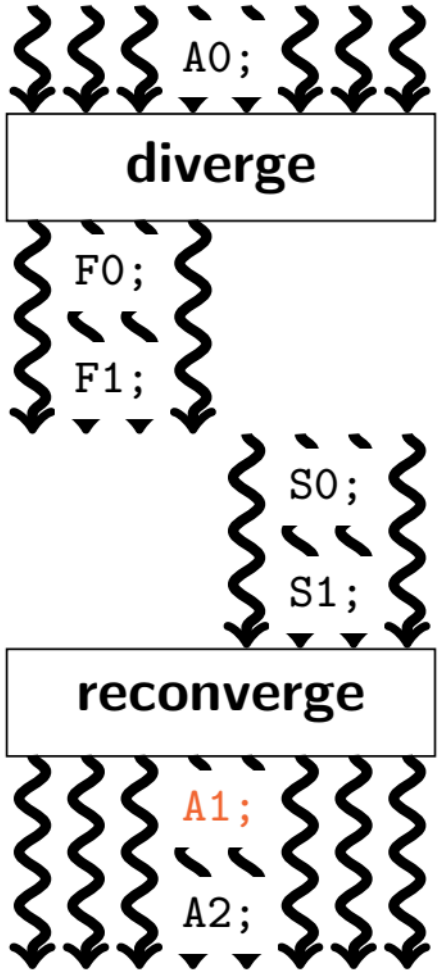
- 最適なスレッド数の選択
  - スレッド数を変えるとどの程度性能が変わるか実験してみてください
- できるだけ条件分岐を削除(CPUコードの場合も同じ)
- FMA(fused multiply-add)命令の発行率を増やす
- 高速な近似逆数平方根命令の使用
  - N体計算の場合は, `rsqrtf()`による高速化が最重要(NVIDIA GPU)
  - 指示文の場合には, コンパイルオプションに `-Mfprelaxed=rsqrt` などを追加)
  - AMD GPUの場合には, `__frsqrtf_rn()`がベスト
- シェアードメモリの活用
  - A100でL2キャッ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

ワープ分裂(warp divergence)

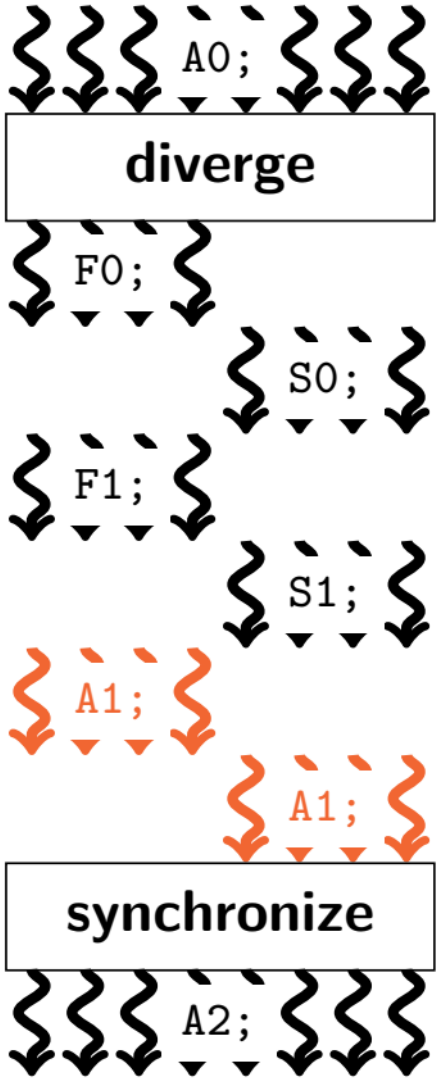
Pseudocode

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

Pascal or earlier



Volta or later



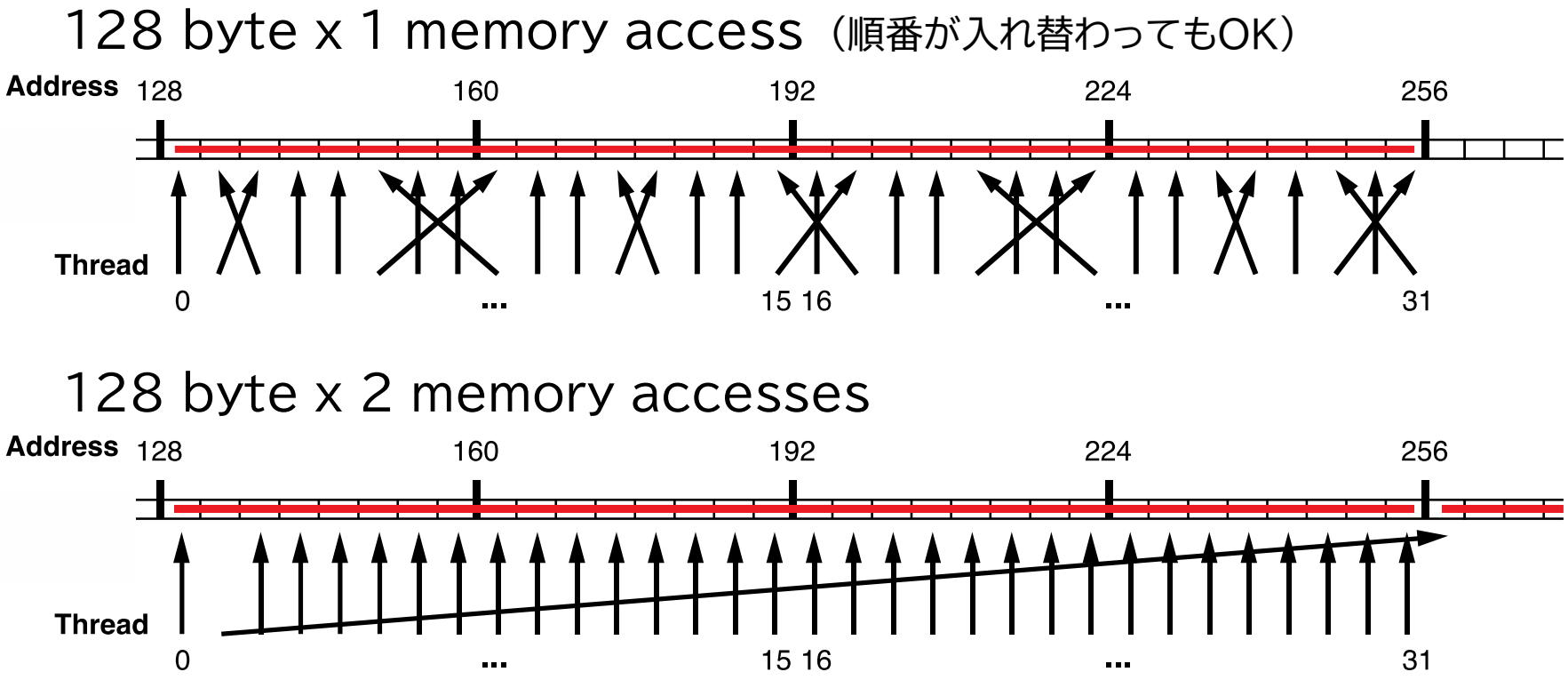
シェアードメモリを使った実装の例

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

```
#include <cooperative_groups.h>

__global__ void calc_force_kernel(...) {
  auto block = cooperative_groups::this_thread_block();
  __shared__ position_t pos_j_shmem[NTHREADS];

  for (int jh = 0; jh < nj; jh += NTHREADS) {
    // load j-particles into shared memory
    const auto pos_j_tmp = pos_j[jh + threadIdx.x];
    block.sync();
    pos_j_shmem[threadIdx.x] = pos_j_tmp;
    block.sync();

    PRAGMA_UNROLL
    for (int j = 0; j < NTHREADS; j++) {
      const position_t pj = pos_j_shmem[j];
      // main calculation
    }
  }

  acc_i[i] = ai;
}
```

コアレスドアクセス(Coalesced access)

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



メモリアクセス命令の発行数削減

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

(超マニアックだが)flush-to-zeroオプションの活用

- 浮動小数点数には, とてもとても小さい数を表現するフォーマットがある
  - 非正規化数(subnormal number, denormalized number)
  - 例えば単精度(FP32)の場合:
    - 最大非正規化数は 1.17549421e-38
    - 最小非正規化数は 1.40129846e-45(`FLT_TRUE_MIN`)
  - 参考までにFP32での他の主要定数
    - 最小正規化数は 1.17549435e-38(`FLT_MIN`)
    - Machine epsilon は 1.19209290e-7(`FLT_EPSILON`)
- この非正規化数を無視してしまって良いのであれば, 計算を高速化できる
  - CUDAの場合, `-ftz=true`をつけてコンパイル(デフォルトは`false`)
    - 注:`-use_fast_math` をつけると自動的に`-ftz=true`を渡したことになっている
  - 無視して問題ないかは, 実際に試してみて計算結果が間違っていないかを見て確認
- 無衝突系N体計算コードの場合には,
  既に最適化されたコードがさらに1割以上速くなった
  - Intel oneAPIではこのオプションがデフォルトで有効だったりする

より高度な内容

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

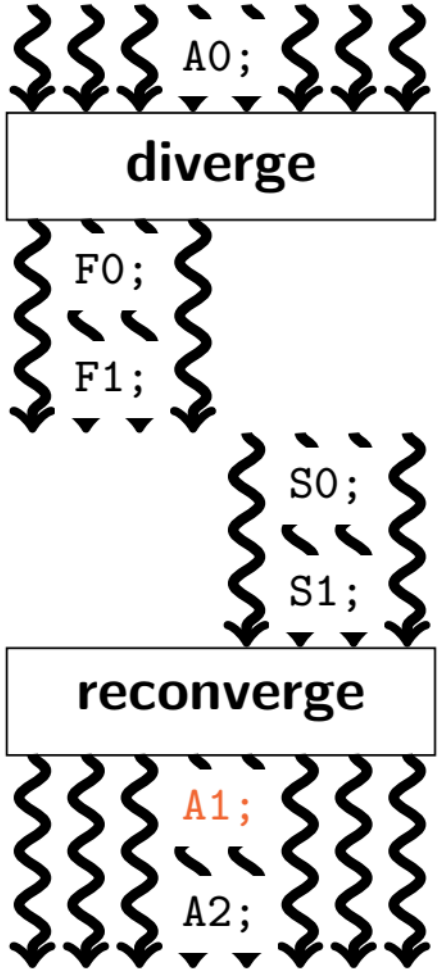
- Volta世代以降のGPUでのワープの挙動
- 最近のGPU上で(CUDAから)使える命令・機能

Independent thread scheduling (1/2)

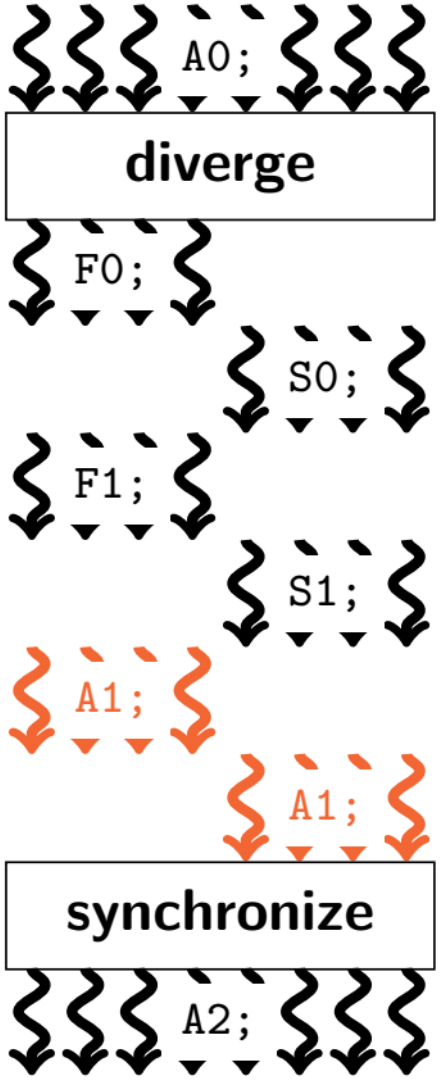
Pseudocode

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

Pascal or earlier



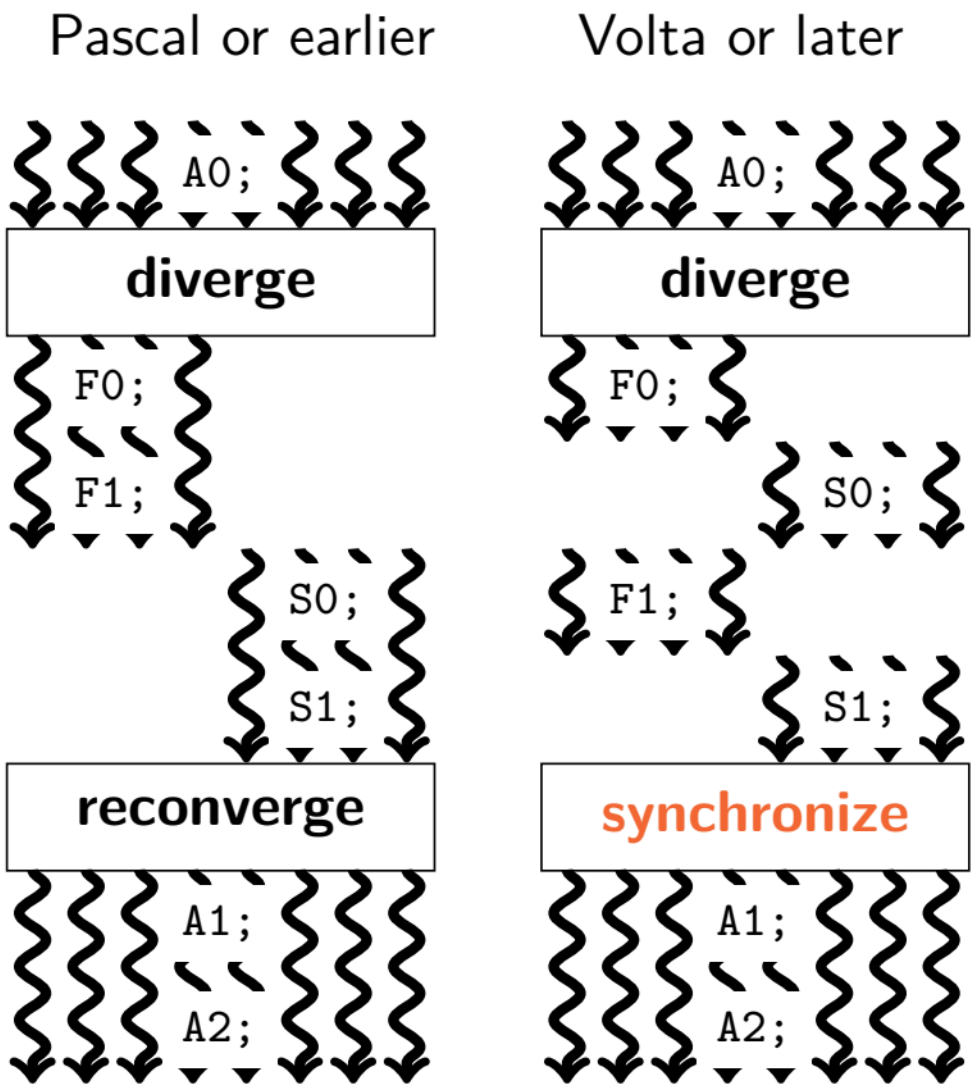
Volta or later



Independent thread scheduling (2/2)

Pseudocode

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



暗黙の同期の挙動変更に対する処方箋

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

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

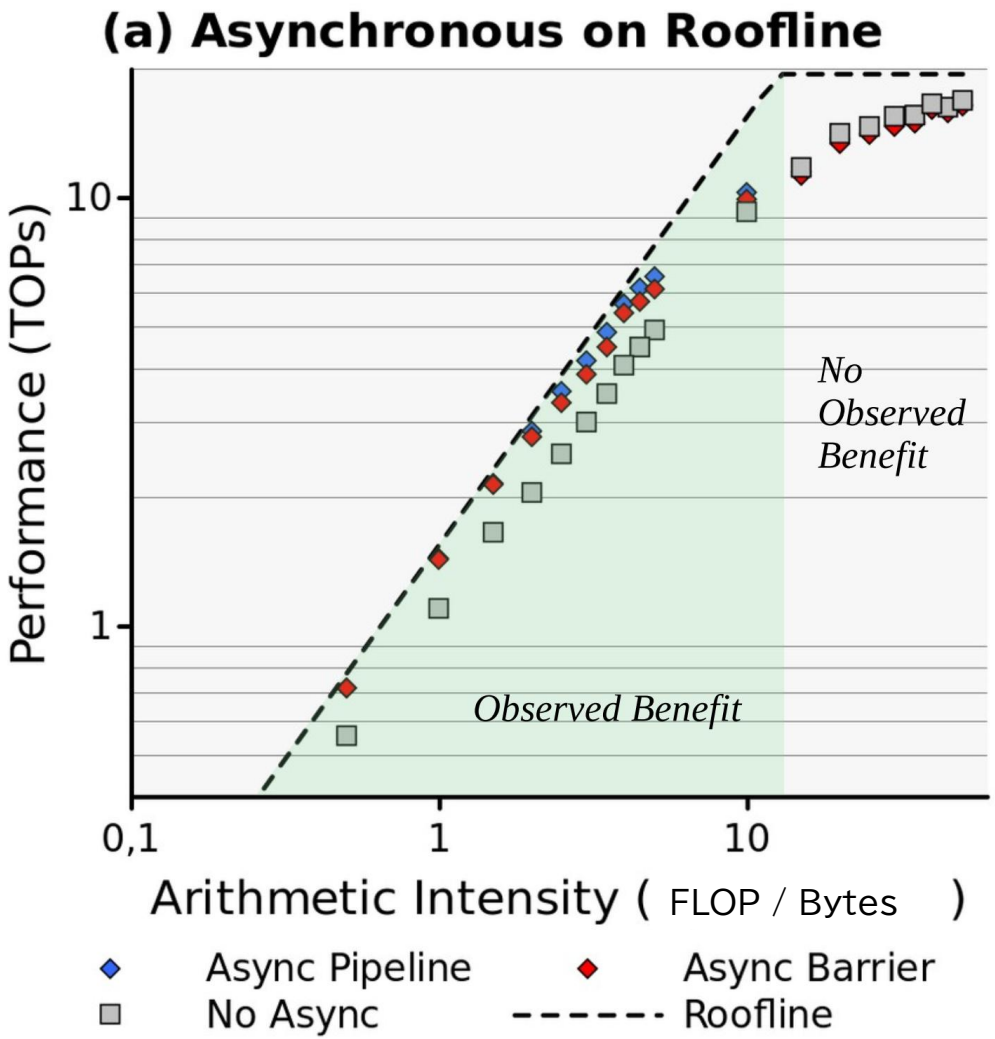
A100/CUDA 11から導入された機能

- 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ードメモリに直接(レジスタを介さずに)データを置ける
- Warp-wide reduction
  - ハードウェア的な高速化が効く
- L2 cache residency control
  - (1つの連続な)データ領域をL2キャッシュに固定しておける

Asynchronous copy (memcpy_async)

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

Svedin et al. 2021



Warp-wide reduction

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

L2 cache residency control

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