重力多体計算のGPU実装
--CUDAでの実装例--

三木 洋平（東京大学 情報基盤センター）

第10回 HPC-Phys 勉強会

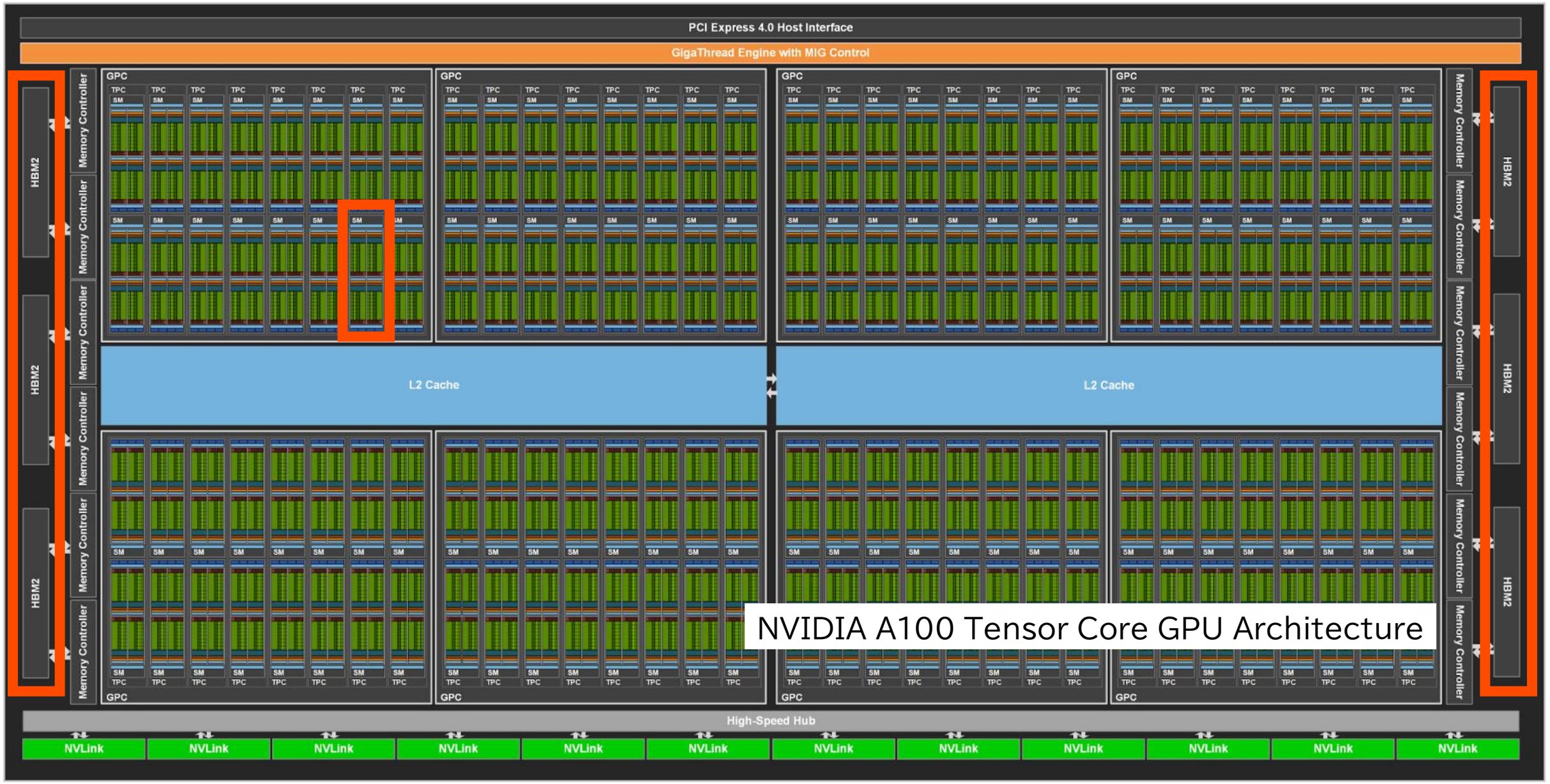
Contents

- GPU，CUDAイントロ
- 直接法での実装紹介
- ツリー法での実装紹介
- （もし時間があれば）A100 PCIeでのpreliminary results
  - この部分は公開するPDFからは消します
- 困ったこと・今後困りそうなこと

Contents

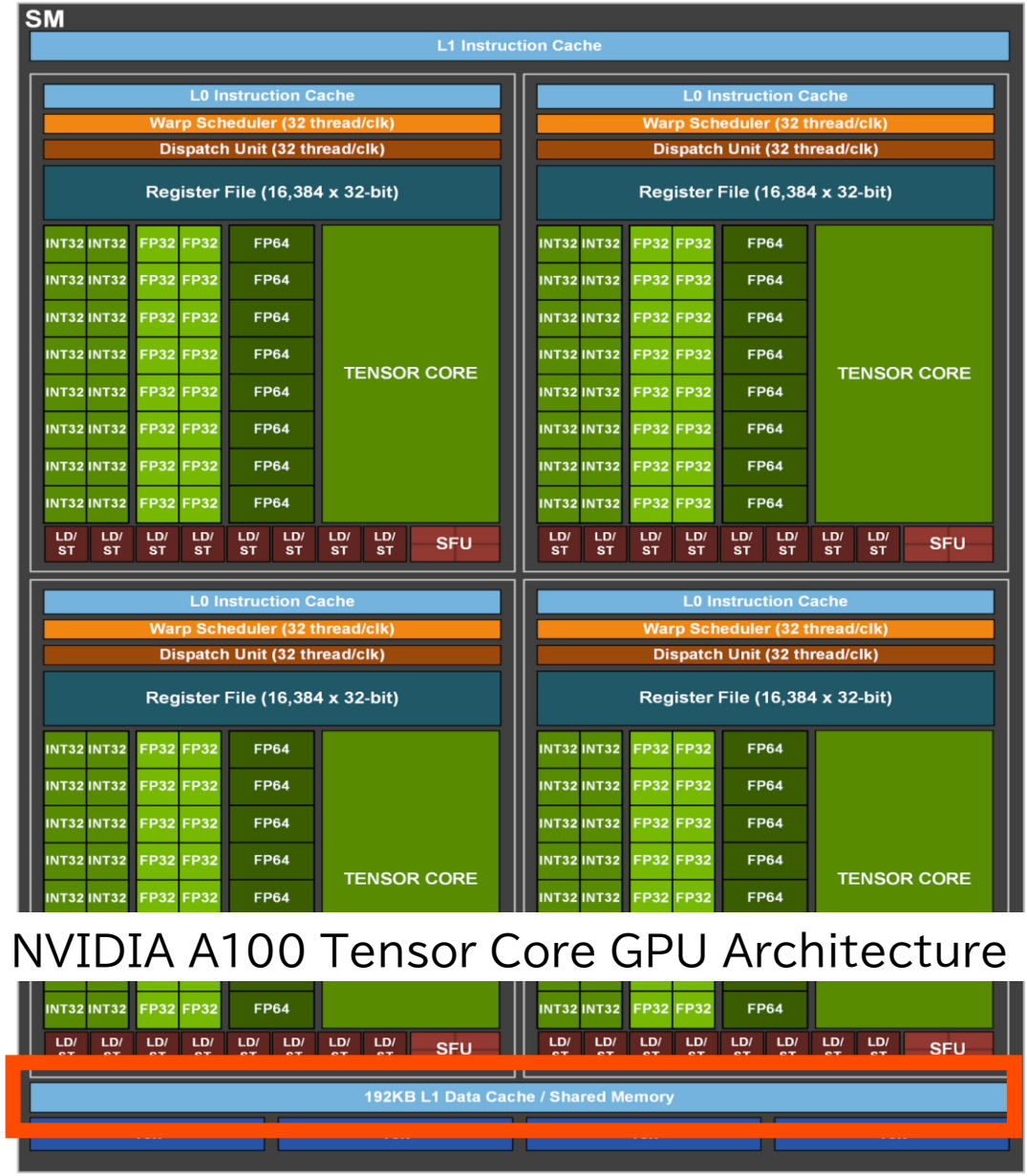
- GPU，CUDAイントロ
- 直接法での実装紹介
- ツリー法での実装紹介
- （もし時間があれば）A100 PCIeでのpreliminary results
  - この部分は公開するPDFからは消します
- 困ったこと・今後困りそうなこと

（NVIDIAの）GPUの構成（1/2）



（NVIDIAの）GPUの構成（2/2）

- SM：Streaming Multiprocessor
- SMの中での構造は気にしなくても性能が出せる
  - 注：32スレッドのグループ（ワープ）内で同じ動作をするように意識しておく
- SM内ではL1キャッシュ/シェアードメモリを共有（A100では192KB）
  - 配分は（ある程度）調節可能
  - シェアードメモリはCUDAでは使えるがOpenACCでは使えない機能
  - グローバルメモリよりも圧倒的に速い



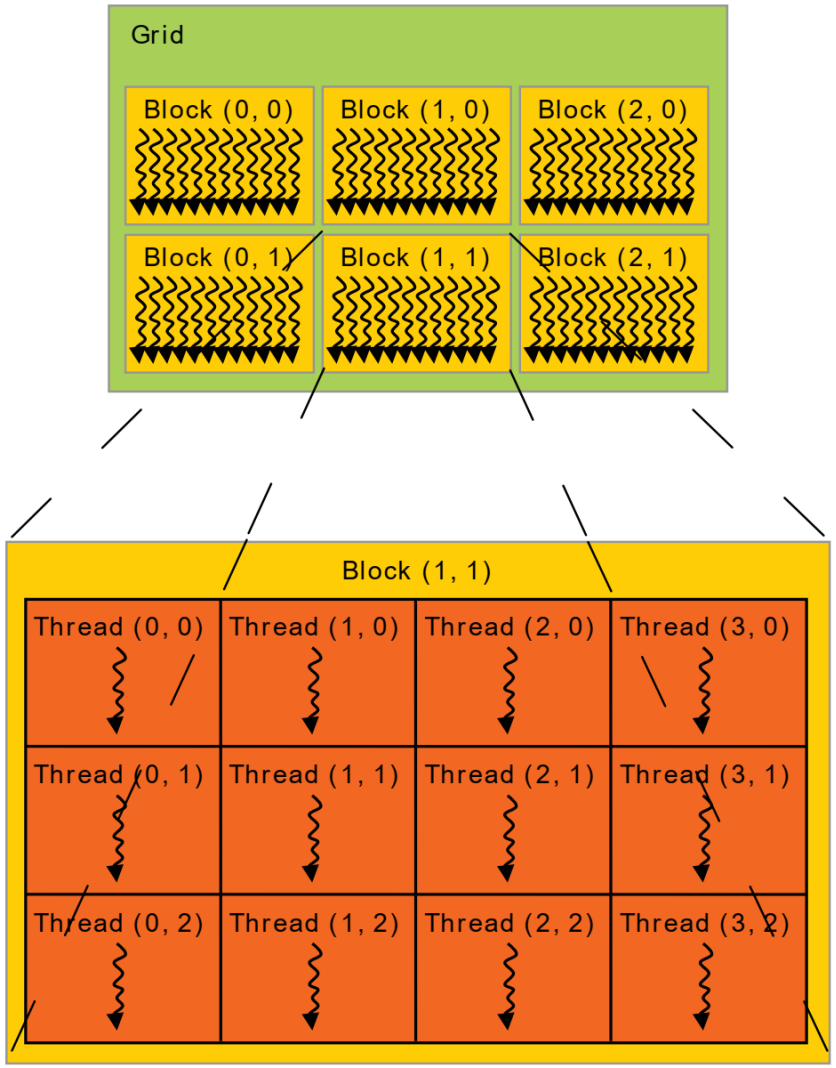
NVIDIA A100 Tensor Core GPU Architecture

GPU向けのプログラミング環境

- 自分で頑張れる人向け
  - CUDA：NVIDIA社製のGPUが対象．今回はこれ
    - 世代が変わるごとに，ホワイトペーパーとProgramming Guideを斜め読みして実装してしまうことが多いです（普通はもうちょっと情報を集めるべきです）
  - HIP：AMD社製のGPUが対象（NVIDIA社製GPUでも動く）
    - HIPIFYでCUDAから変換できる．hipify-clangが推奨とのこと
  - oneAPI ：Intel社製のGPUが対象（CUDAからの変換ツールあり）
    - FPGAなども対象なので，GPU向けという表現は正しくはない
  - 注：CUDAからの変換ツール系は，コードの100%を変換できることは保証しないので，変換後に自分で手直しする必要がある
  - OpenCL
- 指示文ベースでもう少しお手軽に
  - OpenACC
  - OpenMP (for accelerator devices)

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



CUDA C++ Programming Guide (Version 11.2)

CUDAで実装する際に記述する項目

- 以下を詳細に説明すると時間がなくなるので，スキップする
  - このあたりの内容は，OpenACCであればさぼれる部分が多い
- GPUの起動（cudaSetDevice()）
- 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から起動するGPU関数）
- カーネル立ち上げ命令の追加（kernel<<<blck, thrd>>>()）
- 確保したメモリの開放（cudaFree(), cudaFreeHost() ）

Contents

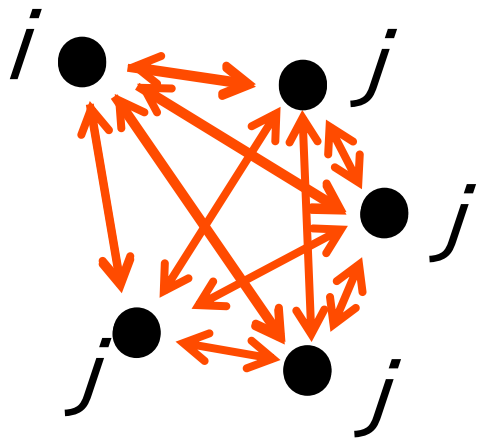
- GPU，CUDAイントロ
- 直接法での実装紹介
- ツリー法での実装紹介
- （もし時間があれば）A100 PCIeでのpreliminary results
  - この部分は公開するPDFからは消します
- 困ったこと・今後困りそうなこと

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: $\mathcal{O}(N)$

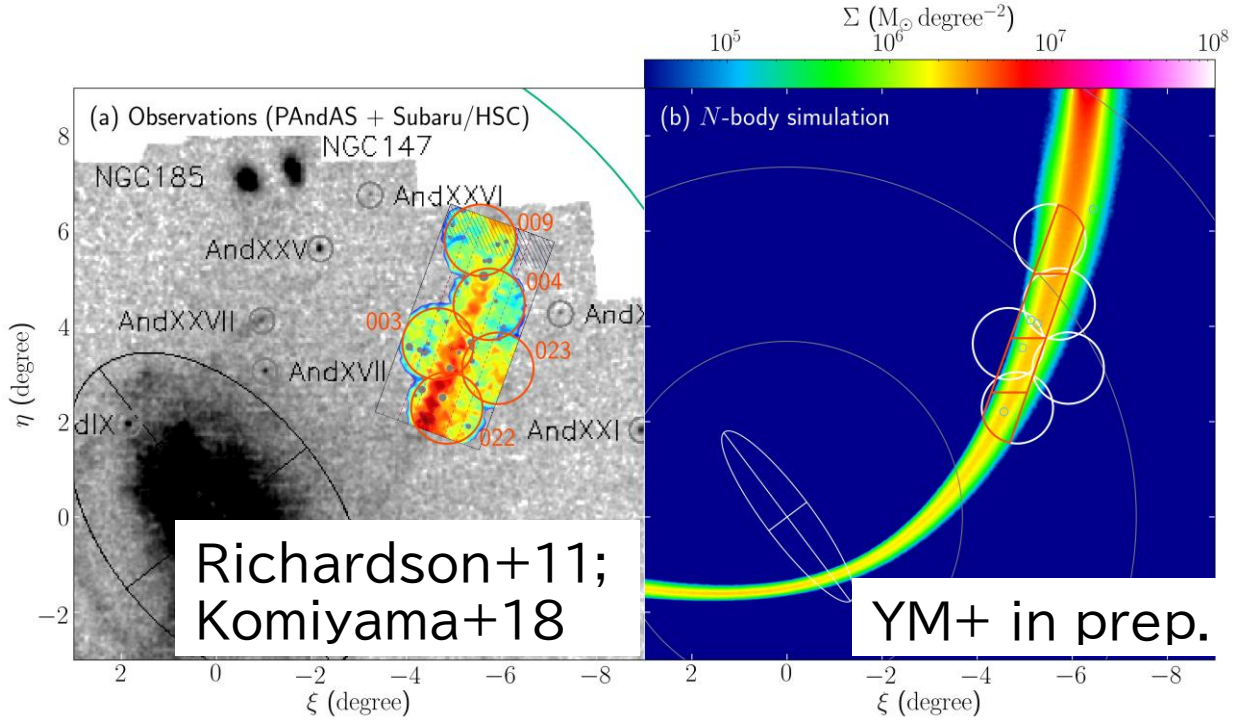
$$\boldsymbol{a}_i = \sum_{\substack{j=0 \\ j\neq i}}^{N-1} \frac{G m_j \left(\boldsymbol{x}_j - \boldsymbol{x}_i\right)}{\left(\left|\boldsymbol{x}_j - \boldsymbol{x}_i\right|^2 + \epsilon^2\right)^{3/2}}$$

- 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



（無衝突）N体計算の代表的な研究対象

- 大規模構造の形成過程
- 銀河どうしの衝突・合体
- 銀河の力学進化（渦状腕の進化過程など）



Direct N-body code (1/3)

- CUDA samples中のnbodyが参考になります
  - 重力計算部分だけを見ましょう
  - GRAPE的なコードが良い
- ブロック当たりのスレッド数は大きめに
  - 後で依存性を見せます
- シェアードメモリをうまく使うのが重要
- 粒子の位置，質量をfloat4型の変数に格納
  - float4は{x, y, z, w}からなるCUDA独自のデータ型
  - 粒子質量をwに置いておく

```
__global__ void calc_gravity
(float4 * __restrict__ ipos, float4 * __restrict__ iacc,
const int Nj, float4 * __restrict__ jpos)
{
  const int tidx = threadIdx.x;
  const int gidx = threadIdx.x + blockIdx.x * blockDim.x;

  __shared__ float4 body[NTHREAD];

  const float4 pi = ipos[gidx];
  float4 ai = {0.0f, 0.0f, 0.0f, 0.0f};

  for(int jh = 0; jh < Nj; jh += NTHREAD){
    float4 pj = jpos[jh + tidx];
    __syncthreads();
    body[tidx] = pj;
    __syncthreads();
```

Direct N-body code (2/3)

- ループアンローリングの段数は，かなり大きめにとる
  - これも依存性を後で紹介します
- 逆数平方根はrsqrtf
  - 単精度で十分（な系を対象にした）
  - CUDAコアとは別ユニットで計算されるので，実行時間が隠蔽される

```
#pragma unroll UNROLL_COUNT
    for(int jj = 0; jj < NTHREAD; jj++){
      pj = body[jj];

      const float rx = pj.x - pi.x;
      const float ry = pj.y - pi.y;
      const float rz = pj.z - pi.z;

      float rinv = rsqrtf(eps2 + rx * rx + ry * ry + rz * rz);
      rinv = rinv * rinv * rinv * pj.w;

      ai.x += rx * rinv;
      ai.y += ry * rinv;
      ai.z += rz * rinv;
    }
  }
  iacc[gidx] = ai;
}
```

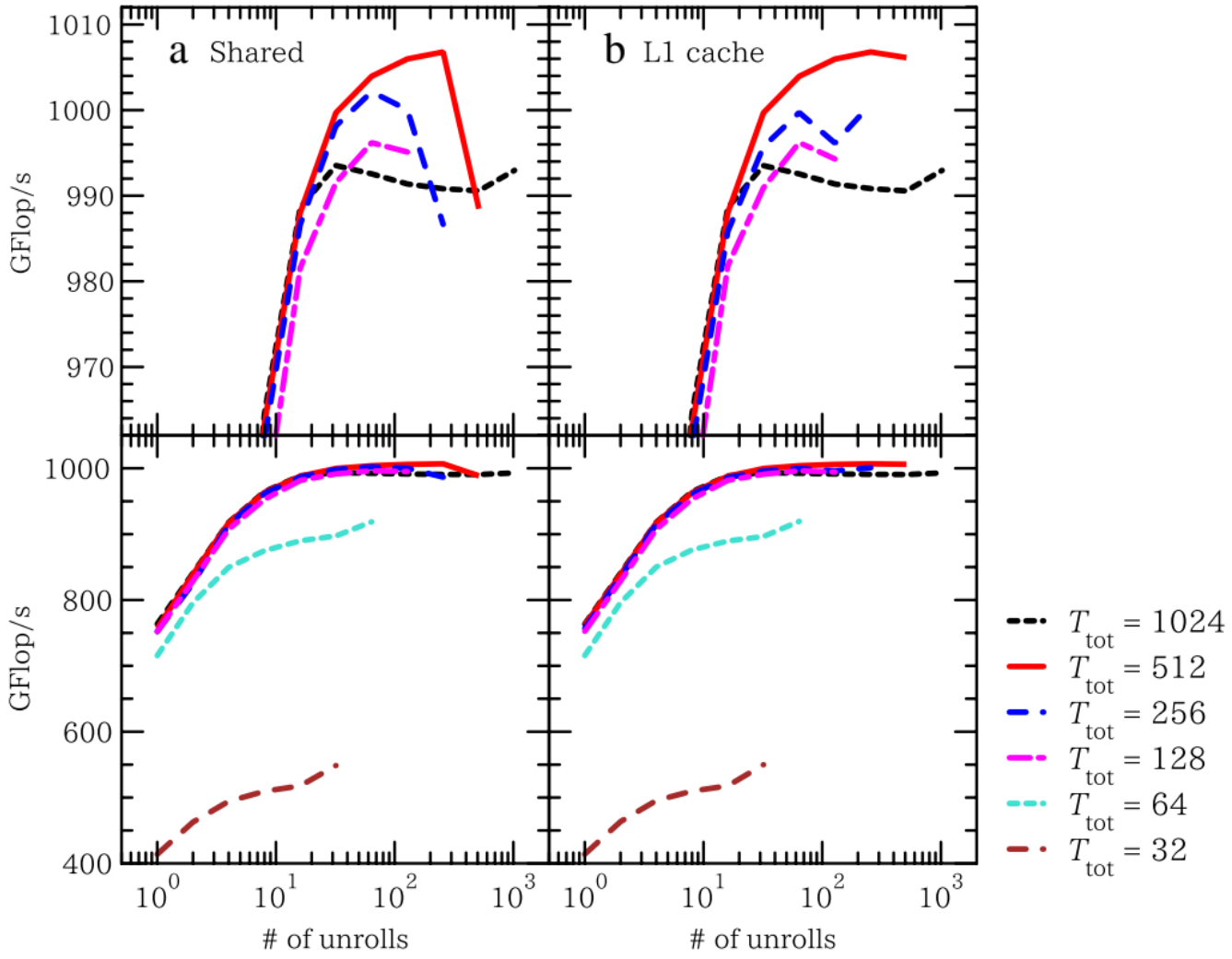
Direct N-body code (3/3)

- カーネル関数の立ち上げだけちょっと特殊
  - これで__global__関数を呼ぶ
- <<<ブロック数，スレッド数，SM容量，ストリーム>>>
  - ブロック構造は多次元でも可
  - 後ろ2つは省略することが多い
  - SM容量は，動的に確保するシェアードメモリの容量（A100では使う場合あり）
  - CUDAストリームを指定して，複数のカーネルを非同期に投げることもできる

```
#include <helper_cuda.h>
#define NB(num, thrd) (((num) + (thrd) - 1) / (thrd))
#define NTHREAD (256)

int main(void){
  const int devIdx = 0;
  checkCudaErrors(cudaSetDevice(devIdx));

  // 途中色々省略

  calc_gravity<<<NB(num, NTHREAD), NTHREAD, 0, 0>>>
              (ipos, iacc, num, jpos);
  getLastCudaError("calc_gravity");

  // 以降も省略
}
```

スレッド数，アンローリング段数依存性

- Miki et al. (2013)
  - M2090（Fermi世代）での測定結果なので，傾向だけに注目してください
  - A100 PCIeでは14.2 TFlop/s
    - 無調整なので，本当はもっと速いはず
    - async copyを使う実装にすればさらに高速化できそう
- スレッド数は128以上が良い
- アンローリング段数を増やしておくとよい
  - ループ長/2が良いことが多い
- 最速を目指すには，こうしたパラメータを色々試す必要あり

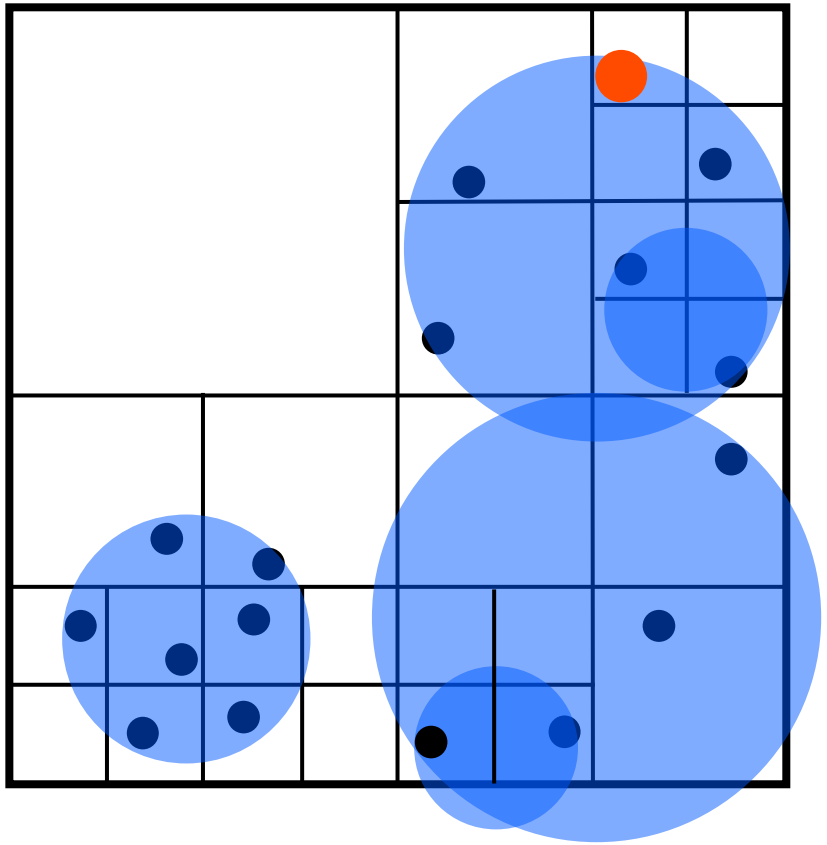


Contents

- GPU，CUDAイントロ
- 直接法での実装紹介
- ツリー法での実装紹介
- （もし時間があれば）A100 PCIeでのpreliminary results
  - この部分は公開するPDFからは消します
- 困ったこと・今後困りそうなこと

N体計算の高速化

- Tree法（Barnes & Hut 1986）がよく使われる
  - 多重極展開を用い，実効的なj-粒子数を減らす
  - $\mathcal{O}(N_i N_j)$ ➔ $\mathcal{O}(N_i \log N_j)$
  - GADGET criterionを採用
    （Springel 2005）
    $\frac{Gm_J}{d_{iJ}^2}\left(\frac{b_J}{d_{iJ}}\right)^2 \leq \Delta_{\mathrm{acc}}|\boldsymbol{a}_i^{\mathrm{old}}|$
    （重力多体系は負の質量がないので，
    誤差のleading termは四重極モーメント）
  - GPU上での高速化も可能
    Nakasato12, Ogiya+13,
    Bedorf+12, 14 etc.



ツリーコードの実装方針，どれにする？

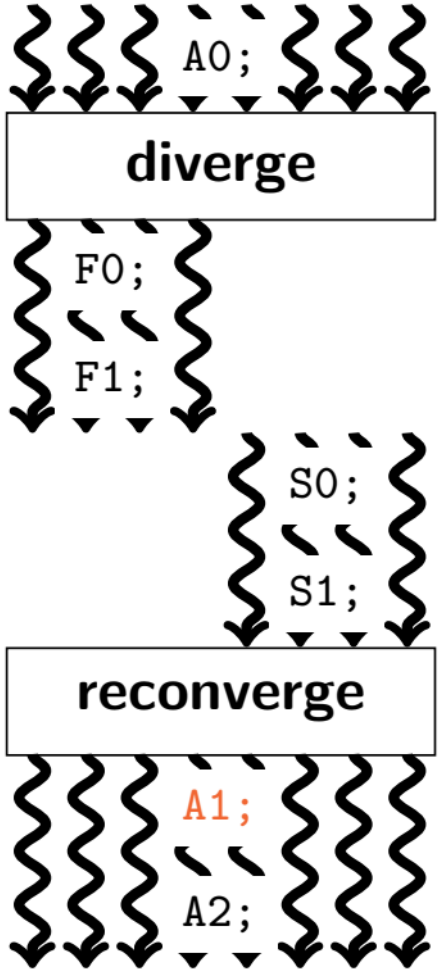
- Direct N-body だと，全ての関数をGPU化するのが簡単
  - C/C++に毛が生えた程度の実装で十分な性能が達成できた
- ツリーコードの場合だと，もう少し考える余地がある
  - CPU側で相互作用リストを作って，GPUで重力計算
    （GPU部分だけを考えれば実質directなので実装がとても楽）
  - ツリー構築から含めて，全てをGPU上で実装
    （実装は面倒だが，CPUとGPUの間のデータ転送が消せる）
- 全関数をGPU上で実装することにしたので，その結果を紹介
  - 実装の詳細は Miki & Umemura (2017)
  - ついでに（というわけではないが）block time stepにした
  - 32スレッドが同期されて動くことを利用した実装
    （Volta世代以降では，この仮定はデフォルトでは無効になった）

Independent thread scheduling

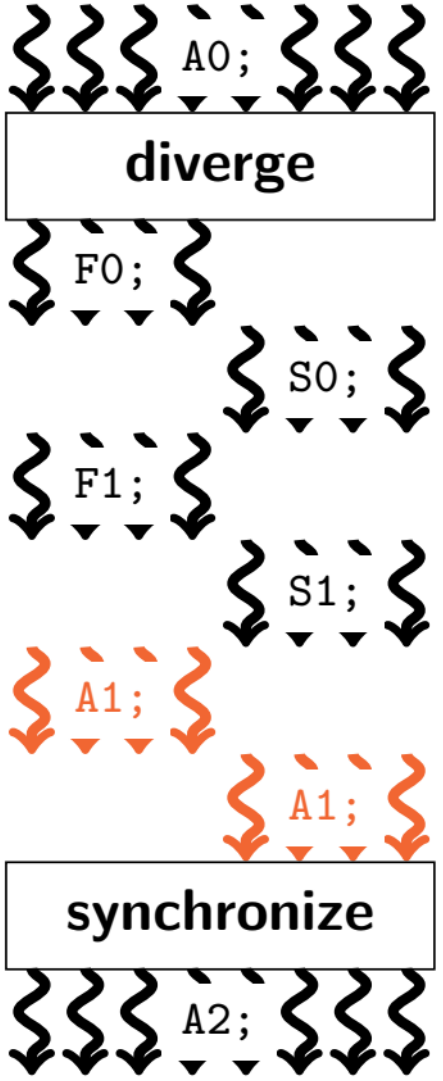
Pseudocode

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

Pascal or earlier



Volta or later

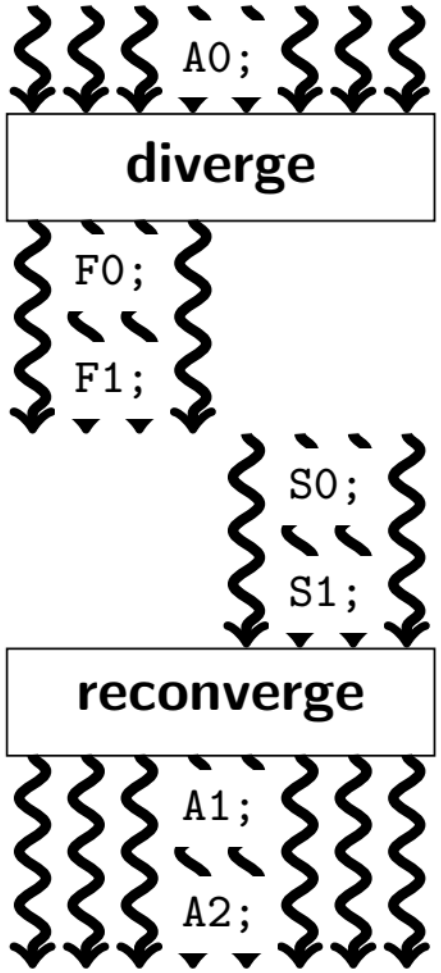


Independent thread scheduling

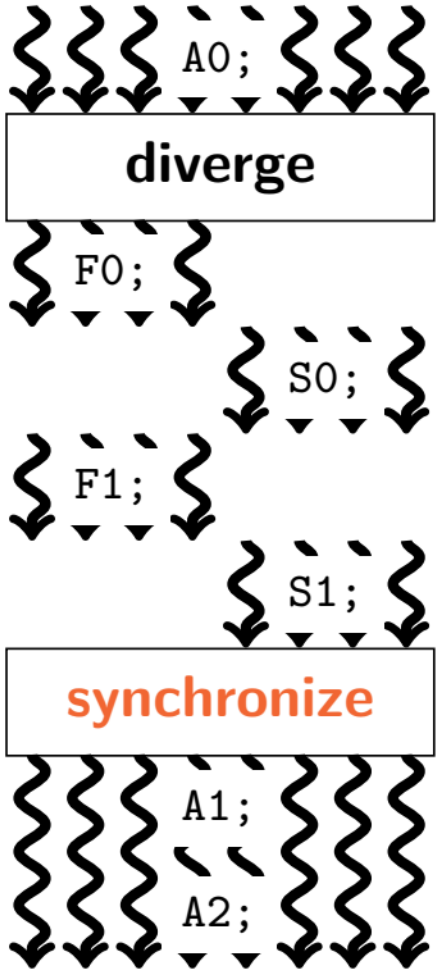
Pseudocode

```
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（のはず）
  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
    （指定方法は変わらず，なのでPascalがCUDAから消えると怖い）
  - 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）

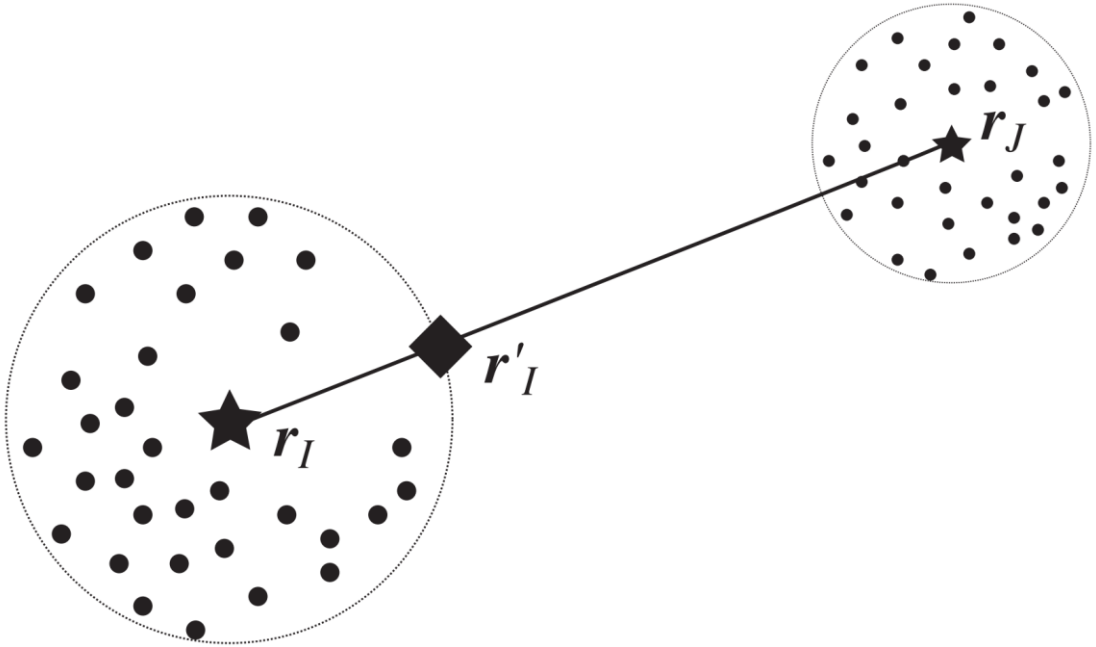
ツリーコードを実装した時の手順

1. GPU向きのアルゴリズムを考える
   - （当時雇用されていたプロジェクト的にGPU実装が前提だった）
   - ワープ内の32スレッドが同時に動くこと（暗黙の同期）が保証されていた時代だったので，これを前提として設計
   - ツリー構築を並列化しやすいので，幅優先探索を採用
   - ツリー探索して小規模な相互作用リストをシェアードメモリに作成・計算するという on-the-fly と相互作用リストの折衷案的実装
2. CPU版のコードを実装して，GPU化するというのは~~面倒~~二度手間だったので，いきなりCUDA実装
   - CPU版を実装する方が大変な設計になっていた
     - 暗黙の同期点に1個ずつ同期命令を書き込まないといけない
     - Volta/Ampereモードでは必要な同期命令を追加した実装になっている
   - （低きに流れた結果）いきなりCUDA実装する人も世の中にはいます..

複数のi-粒子へのツリー判定の同時化

- 1ワープを構成する32スレッド群は複数のi-粒子を担当
- 幅優先探索の下では，複数のj-粒子（候補）との判定は同時実行可能

- 「i-粒子群を含む球」とj-粒子（候補）の距離をツリー判定時の距離として使用
  - 各ペアについて判定し，結果をまとめるような処理が不要に



GPU上での幅優先探索

- 32 threads in a warp が同時に動く（暗黙の同期）
  - 注：前述のとおりVolta以降のGPUでは，状況が変わっています
- 幅優先探索では，距離判定リストに載っているセルとの距離判定は同時に実行可能
  - ”近い”セルは子セルを距離判定リストに追加
  - ”遠い”セルは重力計算リストに追加
  - 距離判定リスト，重力計算リストはともにシェアードメモリに置く
  - Warp内でparallel prefix sum (scan)を何度も実行
  - シェアードメモリ上の距離判定リストに収まらない分は，グローバルメモリ上に確保してあるバッファに退避させる
    （グローバルメモリの大半がこのバッファとして消費されてしまう）

ブロック内・ブロックまたぎのreduce

- 粒子質量の総和などのリダクション演算が必要
  - ワープ内に収まる場合：暗黙の同期が使える
  - ブロック内に収まる場合：__syncthreads() で対応可能
  - 複数ブロックにまたがる場合：ブロックまたぎの同期関数が必要
- 全部自分で実装した
  - 今なら Cooperative GroupsやCUBが使えるはず
  - ブロック間同期もXiao & Feng (2010)のアルゴリズムで実装（Cooperative Groups版よりも速い傾向）
  - ワープシャッフルを使う場合，シェアードメモリ経由にする場合など
    - スカラー変数1つのリダクションだとワープシャッフル版が速い
    - float4のリダクション（重心の計算）ではシェアードメモリ経由が速い

例：scan

- Prefix sum（scan）を1ワープ（32スレッド）内で計算する際の関数
  - warpSizeは32
  - SHFL_MASK_32=0xffffffff
- 暗黙の同期の有無・シェアードメモリ経由かなどをマクロで切り替え
- ブロック内・ブロックまたぎの場合はスレッド数もマクロ渡し

```
template <typename Type>
__device__ __forceinline__ Type prefixSumWarp(Type val, const int lane
#ifndef USE_WARP_SHUFFLE_FUNC_SCAN_INC
 , volatile Type * smem, const int tidx
#endif//USE_WARP_SHUFFLE_FUNC_SCAN_INC
 )
{
#ifdef  USE_WARP_SHUFFLE_FUNC_SCAN_INC
  Type tmp;
  tmp = __SHFL_UP(SHFL_MASK_32, val,  1, warpSize);  if( lane >=  1 )    val += tmp;
#ifndef ENABLE_IMPLICIT_SYNC_WITHIN_WARP
  __syncwarp();
#endif//ENABLE_IMPLICIT_SYNC_WITHIN_WARP
  tmp = __SHFL_UP(SHFL_MASK_32, val,  2, warpSize);  if( lane >=  2 )    val += tmp;
#ifndef ENABLE_IMPLICIT_SYNC_WITHIN_WARP
  __syncwarp();
#endif//ENABLE_IMPLICIT_SYNC_WITHIN_WARP
  tmp = __SHFL_UP(SHFL_MASK_32, val,  4, warpSize);  if( lane >=  4 )    val += tmp;
#ifndef ENABLE_IMPLICIT_SYNC_WITHIN_WARP
  __syncwarp();
#endif//ENABLE_IMPLICIT_SYNC_WITHIN_WARP
  tmp = __SHFL_UP(SHFL_MASK_32, val,  8, warpSize);  if( lane >=  8 )    val += tmp;
#ifndef ENABLE_IMPLICIT_SYNC_WITHIN_WARP
  __syncwarp();
#endif//ENABLE_IMPLICIT_SYNC_WITHIN_WARP
  tmp = __SHFL_UP(SHFL_MASK_32, val, 16, warpSize);  if( lane >= 16 )    val += tmp;
#ifndef ENABLE_IMPLICIT_SYNC_WITHIN_WARP
  __syncwarp();
#endif//ENABLE_IMPLICIT_SYNC_WITHIN_WARP

#else///USE_WARP_SHUFFLE_FUNC_SCAN_INC

  smem[tidx] = val;  if( lane >=  1 )    val += smem[tidx -  1];
#ifndef ENABLE_IMPLICIT_SYNC_WITHIN_WARP
  __syncwarp();
#endif//ENABLE_IMPLICIT_SYNC_WITHIN_WARP
  smem[tidx] = val;  if( lane >=  2 )    val += smem[tidx -  2];
```

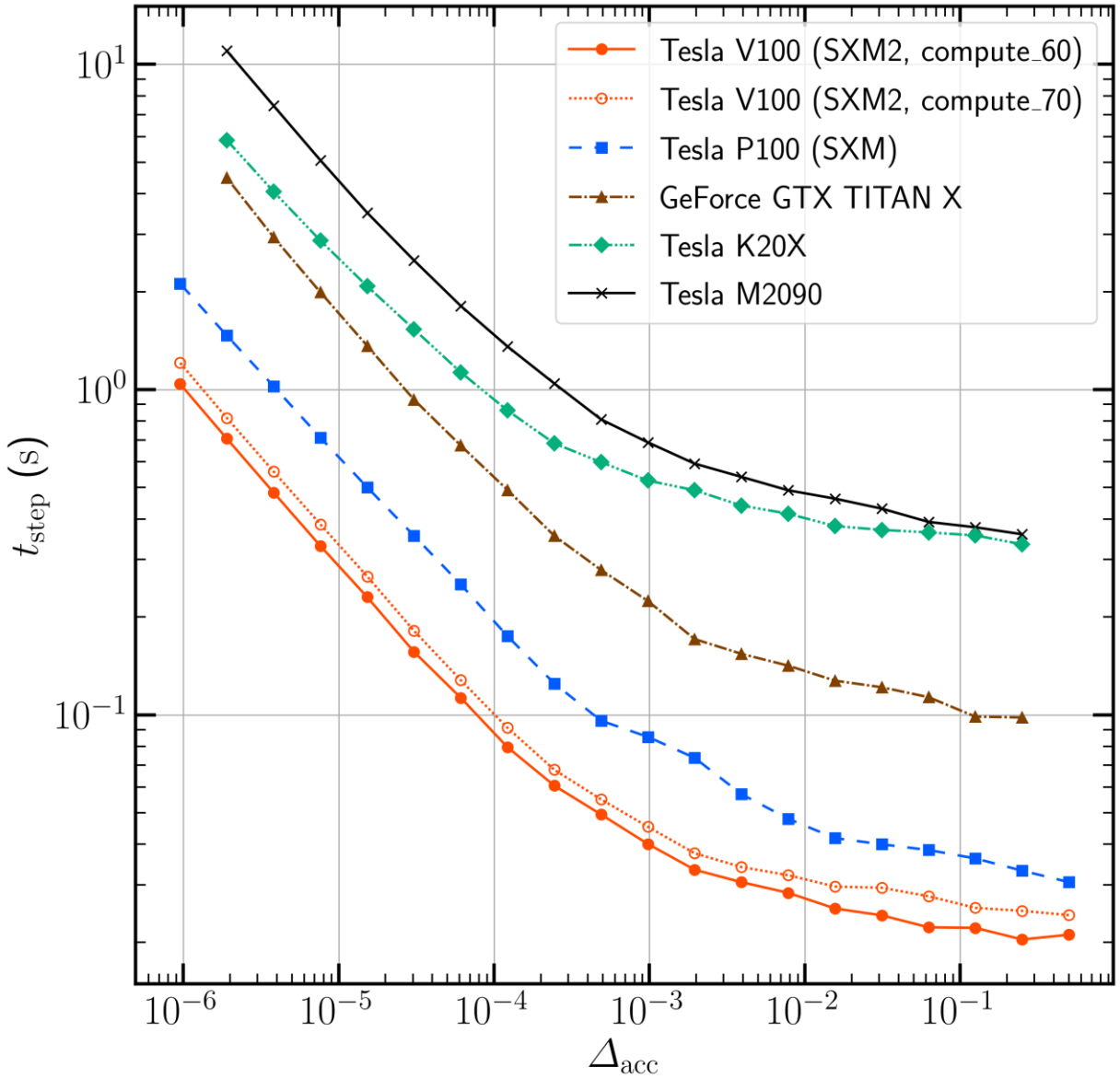
その他の最適化: 木構造の更新頻度

- 木構造（論理構造）を毎ステップ作り直すのは無駄
  - Block time stepを使うとなおさら
  - 注: ツリーノードの重心座標などは毎ステップ更新
- 古い木構造を用いて重力計算をすると…
  - 木構造の再構築コストを削減
  - 論理構造と実際の粒子分布のずれが大きくなると，重力計算のコストが増大
- 両者の和を最小化する更新頻度を動的に設定
  - 実行時間をモデル化し，最適な更新頻度を予測

$$t_{\mathrm{tot}} = t_{\mathrm{make}} + n(t_1 - \Delta t) + \frac{n(n+1)}{2}\Delta t \qquad n^2 = \frac{2}{\Delta t} t_{\mathrm{make}}$$

重力ツリーコードGOTHIC

- Gravitational Oct-Tree code accelerated by HIerarchical time step Contolling
- YM & Umemura (2017), YM (2019)
- Block time stepを採用
- 幅優先探索を採用
- 動的な最適化を実装
- GPUの世代が進むごとにおおむね2倍程度ずつ高速化してきた
  - 右図：M31 model, N = 8M



Contents

- GPU，CUDAイントロ
- 直接法での実装紹介
- ツリー法での実装紹介
- （もし時間があれば）A100 PCIeでのpreliminary results
  - この部分は公開するPDFからは消します
- 困ったこと・今後困りそうなこと

（今回使った）A100とV100の比較（注：テンソルコアは一切考えない）

比較項目	V100 (SXM2)	A100 (PCIe)	比
単精度理論ピーク性能	15.7 TFlop/s	19.5 TFlop/s	1.2x
CUDAコア数	5120 (= 80×64)	6912 (= 108×64)	1.4x
動作周波数（GPU Boost Clock）	1.53 GHz	1.41 GHz	0.92x
グローバルメモリ容量	HBM2 16GB	HBM2 40GB	2.5x
メモリバンド幅	900 GB/s	1555 GB/s	1.7x
L2キャッシュ容量	6 MB	40 MB	6.7x
L2 cache read bandwidth	2048 Bytes/clk	5120 Bytes/clk	2.3x
オンチップメモリ容量（シェアード+L1）	128 KB（sharedは96KBまで）	192 KB（sharedは164KBまで）	1.5x（sharedは1.7x）

- ホワイトペーパー掲載のカタログスペック（Table 4）
- arXivにA100丸裸technical reportが上がることを期待中
  - Jia et al., Dissecting the NVIDIA Ampere GPU Architecture via Microbenchmarking（注：これは予想著者と予想タイトルです）

Contents

- GPU，CUDAイントロ
- 直接法での実装紹介
- ツリー法での実装紹介
- （もし時間があれば）A100 PCIeでのpreliminary results
  - この部分は公開するPDFからは消します
- 困ったこと・今後困りそうなこと

困難その1：ワープ内暗黙同期の仕様変更

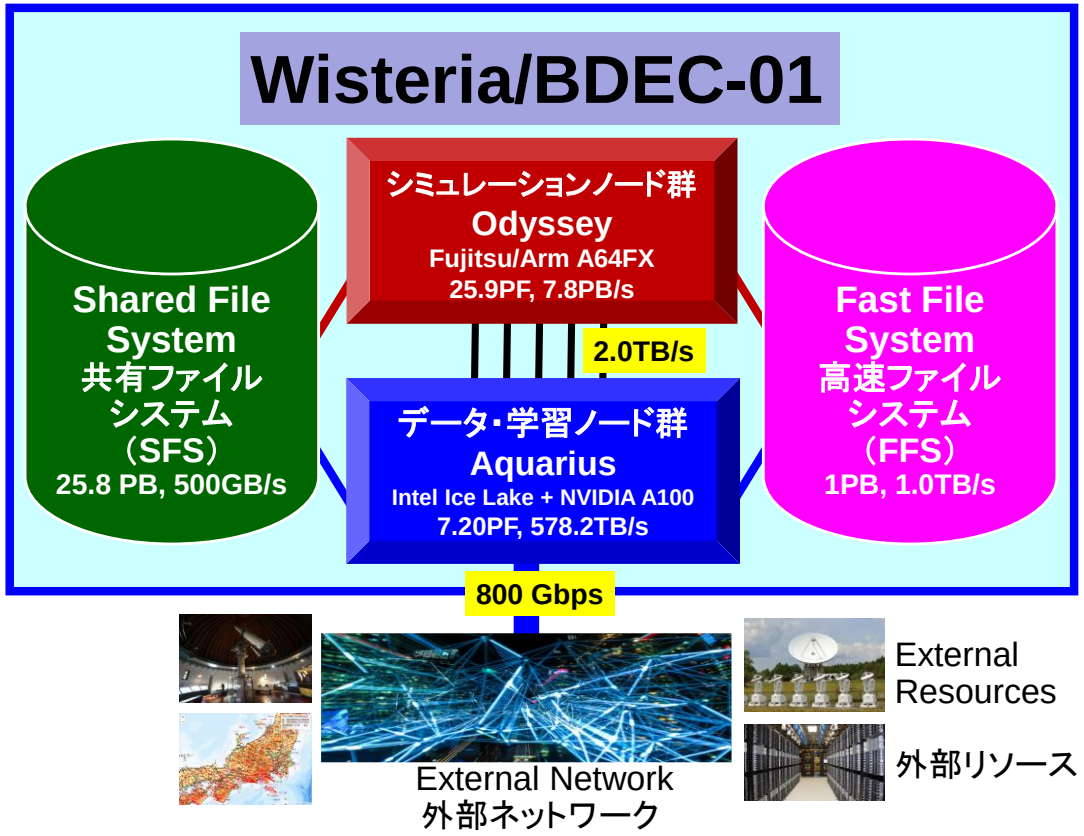
- コード設計時の大前提が崩れたので，精神的ダメージが大きい
  - 幸いにも再設計なしで移行できた（致命傷は回避できた）
  - ホワイトペーパーが出るタイミングでは毎回ドキドキする
- PascalからVoltaへの移行時のバグ取りに案外苦労した
  - ワープシャッフル命令に，maskを追加指定する必要があった
  - 32 threadsの塊が1つ　　入ってくる場合：`mask = 0xffffffff`
  - 16 threadsの塊が1つ　　入ってくる場合：`mask = 0xffff`
  - 16 threadsの塊が2つ同時に入ってくる場合：`mask = 0xffffffff`
  - 挙動にデータ依存性があって事前に決定できない場合は
    `__activemask()`命令を用い，適切なmaskをその場で取得
    （maskの値が事前に分かっているならばべた書きしておけば良い）

困難その2：ベンダーロックイン

- CUDAはNVIDIA社のGPU向けなので，他社GPUで動かない
  - コード論文をarXivに上げた時にコンタクトがあった
  - AMD Vegaで動かしたいと言われたので，対応不可能だった
  - CUDAからの変換ツールはコード全体を移植できるわけではない
- USのエクサスケールスパコンは3機すべてがGPU搭載型だが，NVIDIA製のGPUではない
  - Frontier/El CapitanはAMD GPU（HIP, OpenMPが主流？）
  - AuroraはIntel GPU（oneAPIが主流？ C for Metal?）
- 現時点では日本のGPUスパコンはNVIDIAのGPUが主流
  - USの次期スパコンと同型のシステムが日本に導入されるかは不明
  - 数年しか使わないコードならCUDAで良さそうだが，もっと長く使うコードにしたいならば今から移植性を検討しておいた方が良いかも

最後に東大ITCの新スパコンを宣伝

- Wisteria/BDEC-01
  - Odyssey: FX1000（7680ノード）
  - Aquarius: A100搭載ノード
    （ノード内8 GPUs，45ノード）
- 5月（以降）運用開始予定
  - 詳細は今後東大・情報基盤センターのwebページに記載される予定
    https://www.cc.u-tokyo.ac.jp/
  - 一般利用は電気代相当分だけの料金
    まだ受付開始していませんが，
    https://www.cc.u-tokyo.ac.jp/guide/application/index.php
  - 若手・女性利用（無償，2/22締切）
    https://www.cc.u-tokyo.ac.jp/guide/young/index.php



https://www.cc.u-tokyo.ac.jp/public/pr/pr-wisteria.php

最後に

- CUDAでN体コードを実装した例を紹介した
  - （簡単な例として）直接法
  - （それなりにがんばった例として）ツリー法
  - C/C++に毛が生えた程度の記述で，十分な性能が出せる
- NVIDIA社のGPUは世代ごとにかなり変化する
  - CUDA側で吸収してくれていることが多い（SM内部のコア数など）
  - アルゴリズムの根幹に関わる部分が変更されることもある
- 現時点での日本のGPUスパコンはNVIDIAのGPU搭載なので，CUDAを使っておけば（少なくとも今後数年間は）問題ない
  - 全部CUDAで書かなくても，OpenACC+CUDAなどでも良い
  - ただし次世代のGPUスパコンがどうなるかは未知数であることは留意