GPU向け指示文統合マクロライブラリ Solomon

三木 洋平
（東京大学 情報基盤センター）

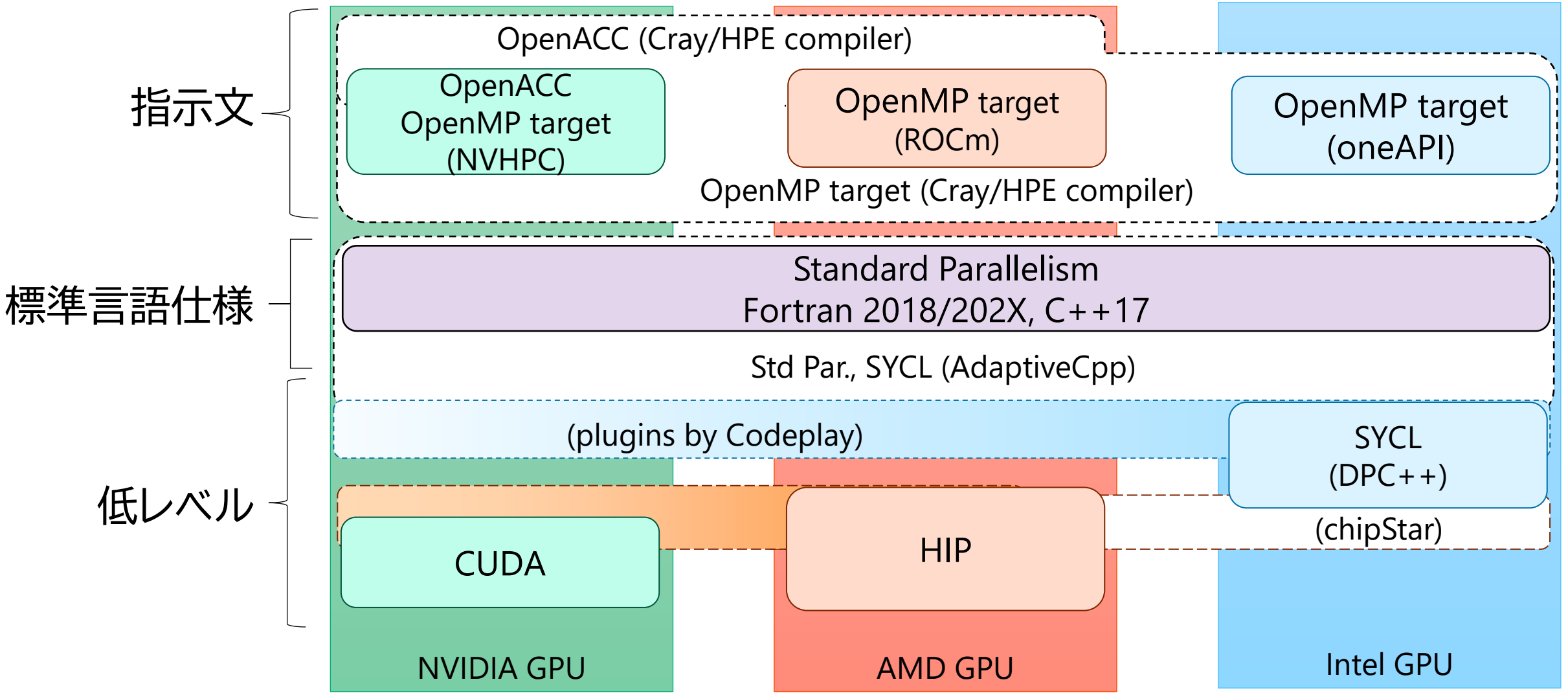
ワークショップ
「アプリ開発者のためのアクセラレータプログラミング最新情報」

GPUスパコンの近況

- GPUは最近のスパコンでのメジャーな演算加速器(特に上位システムで顕著)
- 「GPU = NVIDIAのGPU」と言って良いぐらいの独占状態
  - 国内では, 現在HPCIに資源提供されている全てのGPUスパコンはNVIDIA製
    - 2024年度稼働開始のTSUBAME4.0, 玄界, Miyabi もNVIDIA製GPUを搭載
    - 2025年度はQST/NIFS, 筑波大CCSがそれぞれAMD MI300Aを搭載したシステムを運用開始
  - 海外のハイエンドスパコンではAMD, Intel製GPUも採用
    - Frontier などAMD製GPUを搭載したシステム, AuroraなどIntel製GPUを搭載したシステム
    - TOP500の上位にAMD, Intel, NVIDIA製GPUがランクイン
- GPUベンダー間の競争が活性化
  - NVIDIA製GPUの性能向上: P100, V100, A100の間は各世代$\sqrt{2}$倍➔H100では約3倍
  - 発散するプログラミング環境への対応も必要
- 「富岳NEXT」はNVIDIA製GPUを搭載する
  - HPCI構成機関が今後どのようなシステムを入れてくるかは不明, 拠点ごとの多様性も？
    - 自分たちの開発したコードが特定のシステムでないと動かない, という状況は避けたいというのが本音
  - ➔ ベンダーニュートラルな実装手法を採用したいが, 実現可能か？　十分な性能は出せるか？

GPU向けのプログラミング環境

2024年2月のPCCC AI/HPC OSS活用WSでの講演資料
(https://www.pccluster.org/ja/event/data/240205_pccc_wsAI-HPC-OSS_06_hanawa-miki.pdf)



指示文ベースでGPU化したい場合の選択肢

- OpenACC
  - GPU向けのメジャーな指示文
  - PGIがNVIDIAに買収された結果, NVIDIA色が強くなってしまった
  - AMD, Intelは(きっと)サポートしない
    - HPE Crayコンパイラであれば, AMD GPU向けのOpenACCもサポート
    - IntelはOpenACCからOpenMP targetへの変換ツールを開発中
    - ➔GPUベンダー(AMD, Intel)による直接支援が受けられない
- OpenMPのtarget指示文
  - OpenMP 4.0以降でアクセラレータへのオフロードがサポート
  - OpenMP 5.0で loop 指示節が追加, OpenACC的実装も可能に
  - NVIDIA, AMD, Intel 全てのGPU向けにサポートされる
  - 現時点ではOpenACCの全ての機能に対応できていない
    - 非同期実行の(細やかな)制御など
- 実装時には, 使う指示文についても選択する必要がある
  - 決断をできるだけ後回しにしたいというのが本心だが, そういうわけにもいかない

マクロを用いた指示文のブラックボックス化

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

GPU向け指示文統合マクロSolomon

- OpenACCとOpenMP両方に対応した指示文の追加例(右側)
  - 場合分け(ACC for GPU, OMP for CPUなど)がかなり煩雑
  - `$ nvc++ -acc=multicore -mp=gpu …` も(見かけないが)実は可能
- プリプロセッサマクロを用いてインターフェースを統合するライブラリを開発
  - https://github.com/ymiki-repo/solomon で公開
  - Miki & Hanawa (2024, IEEE Access)
  - 手品の種: `_Pragma()`形式で指示文を記述
  - 対応しているバックエンド:
    - OpenACC, OpenMP target, OpenMP
- やる気が出るのはどちらの手法?
  - 通常の(煩雑な)実装方法➡
  - ⬇ Solomon を用いて簡易化した手法

```
OFFLOAD(AS_INDEPENDENT, NUM_THREADS(NTHREADS))
for(int i = 0; i < N; i++){
```

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

Solomon のインターフェース(指示文)

- 簡易記法, OpenACC的記法, OpenMP的記法の3種を提供

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

Solomon のインターフェース(指示節)

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

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

Solomon 使用にあたっての注意点, 推奨事項など

- 指示節などはカンマ区切りで入力する
  - 各指示節が適用可能かを(Solomon側で)判定し, OKなもののみを有効化する実装
  - OpenMP的記法をOpenACCバックエンドで動かすために実装した機能
    - `_Pragma("omp target teams loop collapse(3) thread_limit(128)")`
      ➔ `_Pragma("acc kernels vector_length(128)") _Pragma("acc loop collapse(3)")`
    - 指示節の適切な振り分けは, Solomon 側で対応すべき機能の一つ
- OpenACCでもOFFLOAD(…), PRAGMA_ACC_[KERNELS PARALLEL]_LOOP(…) の使用を推奨
  - PRAGMA_ACC_[KERNELS PARALLEL](…)とPRAGMA_ACC_LOOP(…)に分けて書くと, OpenMP target への適切な変換が困難になる(LOOP側に入力した指示節が渡らなくなる)
- ~~AS_INDEPENDENT は(複数の)指示節の先頭に置いておく(必須)~~
  - ~~バックエンドを OpenMP にした時には simd に変換されるが, simd は構文の一部分であるため, 中間に他の指示節が紛れ込むとエラーとなってしまう~~
  - ~~指示節候補をソートした後で判定・有効化するよう実装すれば回避できるはず(未対応)~~
- PRAGMA_OMP_TARGET_DATA(…) は非推奨
  - OpenACC における data, host_data と OpenMP target の data が対応
  - バックエンドを OpenACC にした時にエラーとなる場合がある
  - 推奨:PRAGMA_ACC_[DATA HOST_DATA](…), DATA_ACCESS_BY_[DEVICE HOST](…)

Solomon を用いた際の，バックエンド切替方法

- コンパイル時にフラグを渡すだけでOK

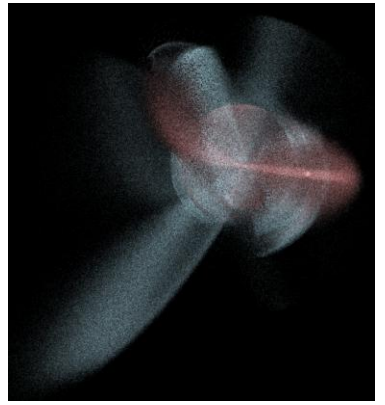
コンパイル時フラグ	バックエンド
-DOFFLOAD_BY_OPENACC	OpenACC を使用，kernels
-DOFFLOAD_BY_OPENACC -DOFFLOAD_BY_OPENACC_PARALLEL	OpenACC を使用，parallel
-DOFFLOAD_BY_OPENMP_TARGET	OpenMP target を使用，loop
-DOFFLOAD_BY_OPENMP_TARGET -DOFFLOAD_BY_OPENMP_TARGET_DISTRIBUTE	OpenMP target を使用，distribute

- 各コンパイラに対する OpenACC / OpenMP target を有効化するためのフラグは別途必要
  - OpenACC 無効化時には，-DOFFLOAD_BY_OPENACCも自動的に無効化される
- -DOFFLOAD_BY_OPENACCと-DOFFLOAD_BY_OPENMP_TARGETを両方指定した場合
  - OpenACC を用いてGPU化
- -DOFFLOAD_BY_OPENACCと-DOFFLOAD_BY_OPENMP_TARGETをどちらも指定しなかった場合
  - マルチコアCPU向けにOpenMPでスレッド並列化

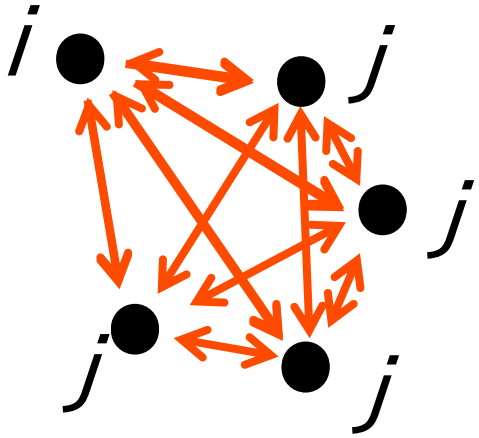
N体計算(重力多体計算)

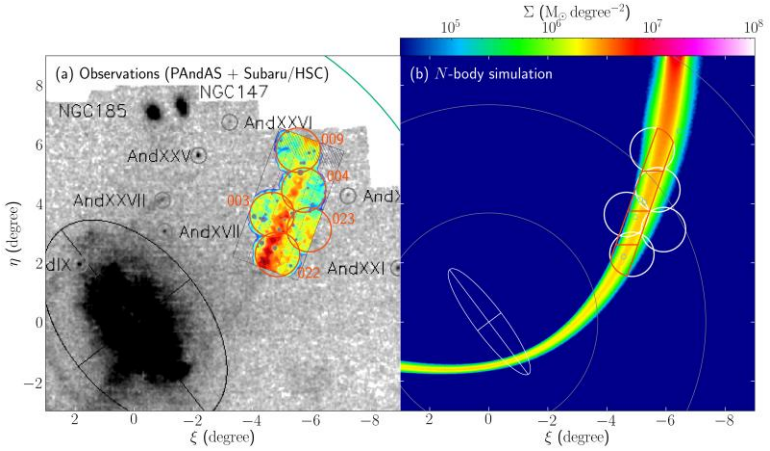
- 粒子どうしに働く自己重力による系の時間進化を，運動方程式に基づいて計算
  - データ量: $\mathcal{O}(N)$
  - 重力計算: $\mathcal{O}(N^2)$
  - 時間積分: $\mathcal{O}(N)$

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



- 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)
- 無衝突系であれば全て単精度浮動小数点演算でも十分な精度が得られる
  - 本研究では，全ての演算に単精度浮動小数点演算を用いた

実装例(N体計算, 簡易記法)

- CPU上で初期条件を生成, GPUに粒子データ転送後に重力計算

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

  - MEMCPY_H2D() でデータ転送

- 重力計算関数(ほぼ省略版)

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

- *i*-ループを並列化
  - スレッド数は NTHREADS
- *j*-ループが並列化されると性能低下の要因となるため, 並列化を抑止(OpenACC)

実装例(N体計算, OpenACC的記法)

- CPU上で初期条件を生成, GPUに粒子データ転送後に重力計算

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

  - PRAGMA_ACC_UPDATE_DEVICE() でデータ転送

- 重力計算関数(ほぼ省略版)

```
void calc_acc(…) {
  PRAGMA_ACC_KERNELS_LOOP(ACC_CLAUSE_INDEPENDENT, ACC_CLAUSE_VECTOR_LENGTH(NTHREADS))
  for (std::remove_const_t<decltype(Ni)> i = 0; i < Ni; i++) {
    // 初期化（省略）
    PRAGMA_ACC_LOOP(ACC_CLAUSE_SEQ)
    for (std::remove_const_t<decltype(Nj)> j = 0; j < Nj; j++) {
      // ループ内は省略
    }
    iacc[i] = ai;
  }
}
```

実装例(N体計算, OpenMP的記法)

- CPU上で初期条件を生成, GPUに粒子データ転送後に重力計算

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

  - PRAGMA_OMP_TARGET_UPDATE_TO() でデータ転送

- 重力計算関数(ほぼ省略版)

```
void calc_acc(…) {
  PRAGMA_OMP_TARGET_TEAMS_LOOP(OMP_TARGET_CLAUSE_SIMD,
OMP_TARGET_CLAUSE_THREAD_LIMIT(NTHREADS))
  for (std::remove_const_t<decltype(Ni)> i = 0; i < Ni; i++) {
    // 初期化（省略）
    PRAGMA_ACC_LOOP(ACC_CLAUSE_SEQ)
    for (std::remove_const_t<decltype(Nj)> j = 0; j < Nj; j++) {
      // ループ内は省略
    }
    iacc[i] = ai;
  }
}
```

性能評価に用いたGPUスパコン

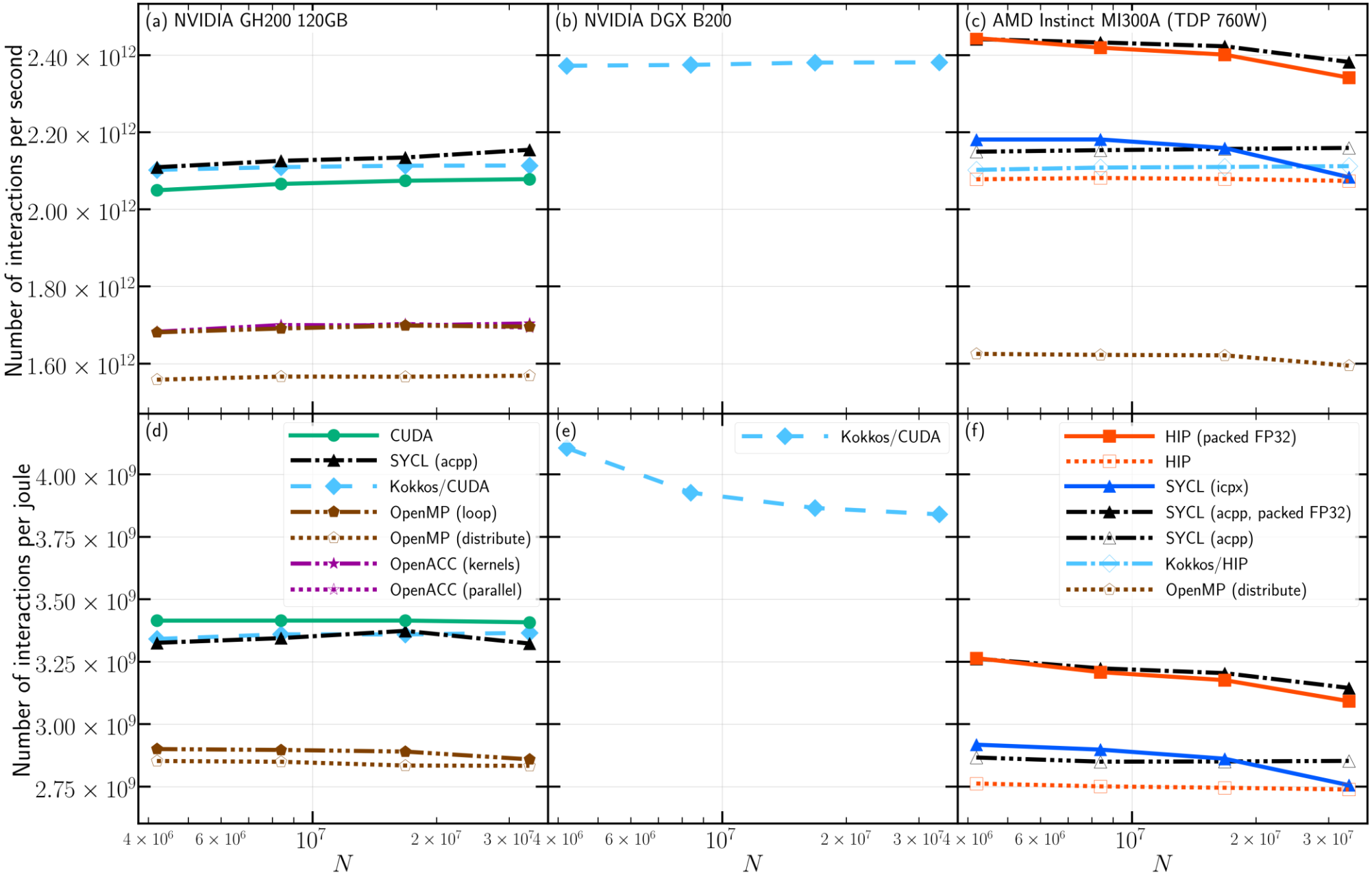
- ただし全ての評価は単体GPU
- Miyabi
  - 最先端共同HPC基盤施設(JCAHPC)
    - 筑波大CCSと東大ITCで共同運用
  - 演算加速ノードMiyabi-Gに
    NVIDIA GH200を全1120基搭載
- 青嵐
  - 東京工科大学 AIテクノロジーセンター
  - NVIDIA B200を全96基(12ノード)搭載
- プラズマシミュレータ“双星”
  - 量子科学技術研究開発機構(QST)および
    核融合科学研究所(NIFS)
  - サブシステムBにAMD MI300Aを
    全280基(70ノード)搭載

https://nsrp.nifs.ac.jp

計算機環境

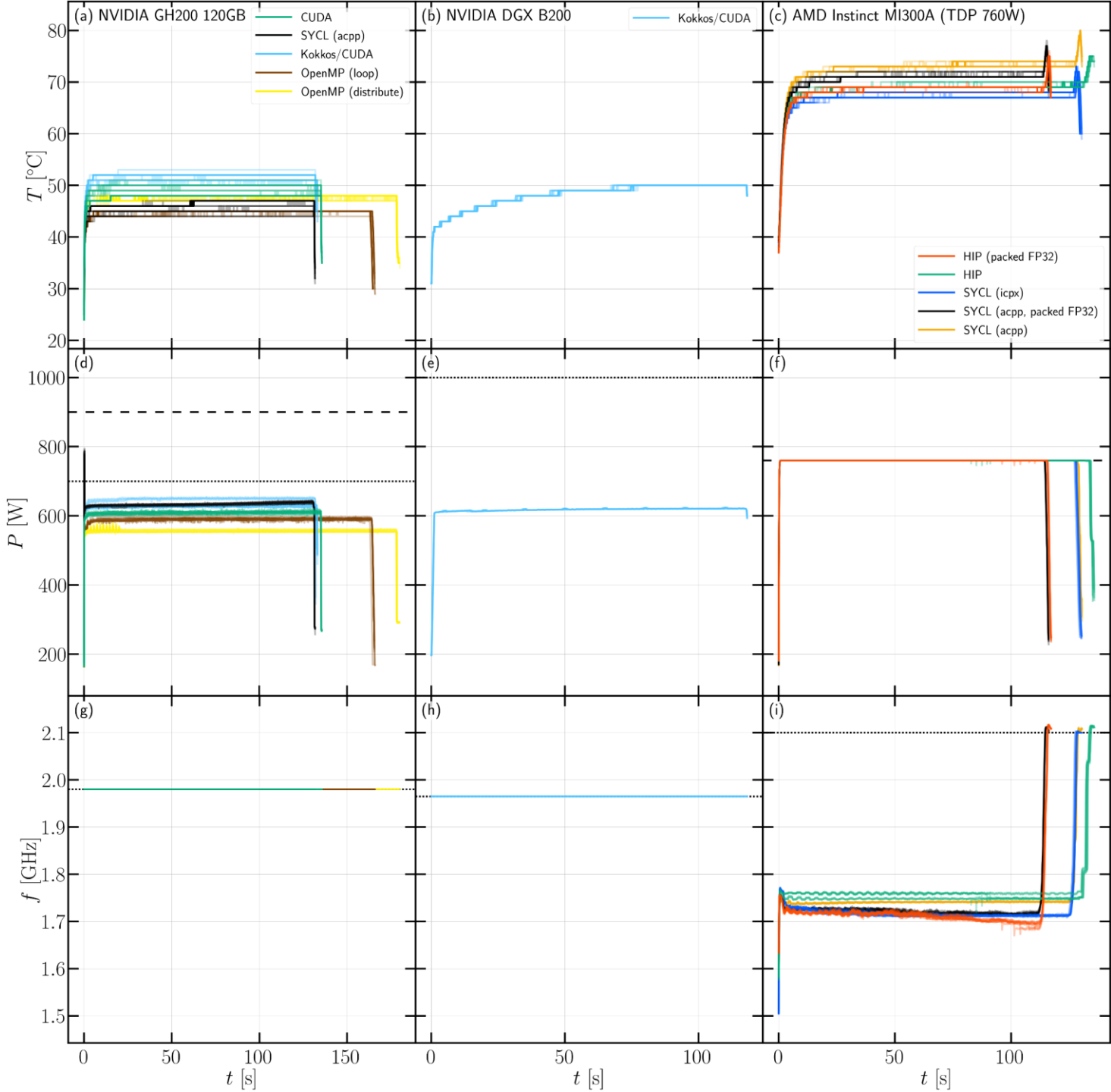
	NVIDIA GH200	NVIDIA B200	AMD MI300A
FP32性能	66.91 TFlop/s	74.45 TFlop/s	122.6 TFlop/s
並列度(FP32)	16896	18944	14592
動作周波数	1980 MHz	1965 MHz	2100 MHz
メモリ容量	HBM3 96GB	HBM3e 192GB	HBM3 128GB
メモリバンド幅	4.0 TB/s	7.7 TB/s	5.3 TB/s
TDP	1000 W (全体)	1000 W	760 W
コンパイラ環境	CUDA 12.9.41	CUDA 13.0	ROCm 6.3.3
	NVIDIA HPC SDK 25.5		Intel oneAPI 2025.2.0
	AdaptiveCpp 25.02.0		AdaptiveCpp 25.02.0
	LLVM 20.1.8		LLVM 20.1.7
	(acppはCUDA 12.8.93利用)		
	Kokkos 5.0	Kokkos 5.0	Kokkos 5.0

性能測定結果（10回測定した結果の最高性能）



GPU状態の監視結果

- 0.1秒間隔でGPUの温度, 消費電力, 動作周波数を測定
  - N=16M
- AMD MI300Aの動作周波数は1.7 GHz程度で推移
  - ブーストクロック(2.1 GHz)からは大幅に低下
  - Packed FP32命令を用いるとさらに低下
  - 供給できる電力(760 W)が不十分なので, 周波数が低下
- NVIDIA GH200/B200では常時ブーストクロックで動作
  - 供給可能な電力に余裕がある



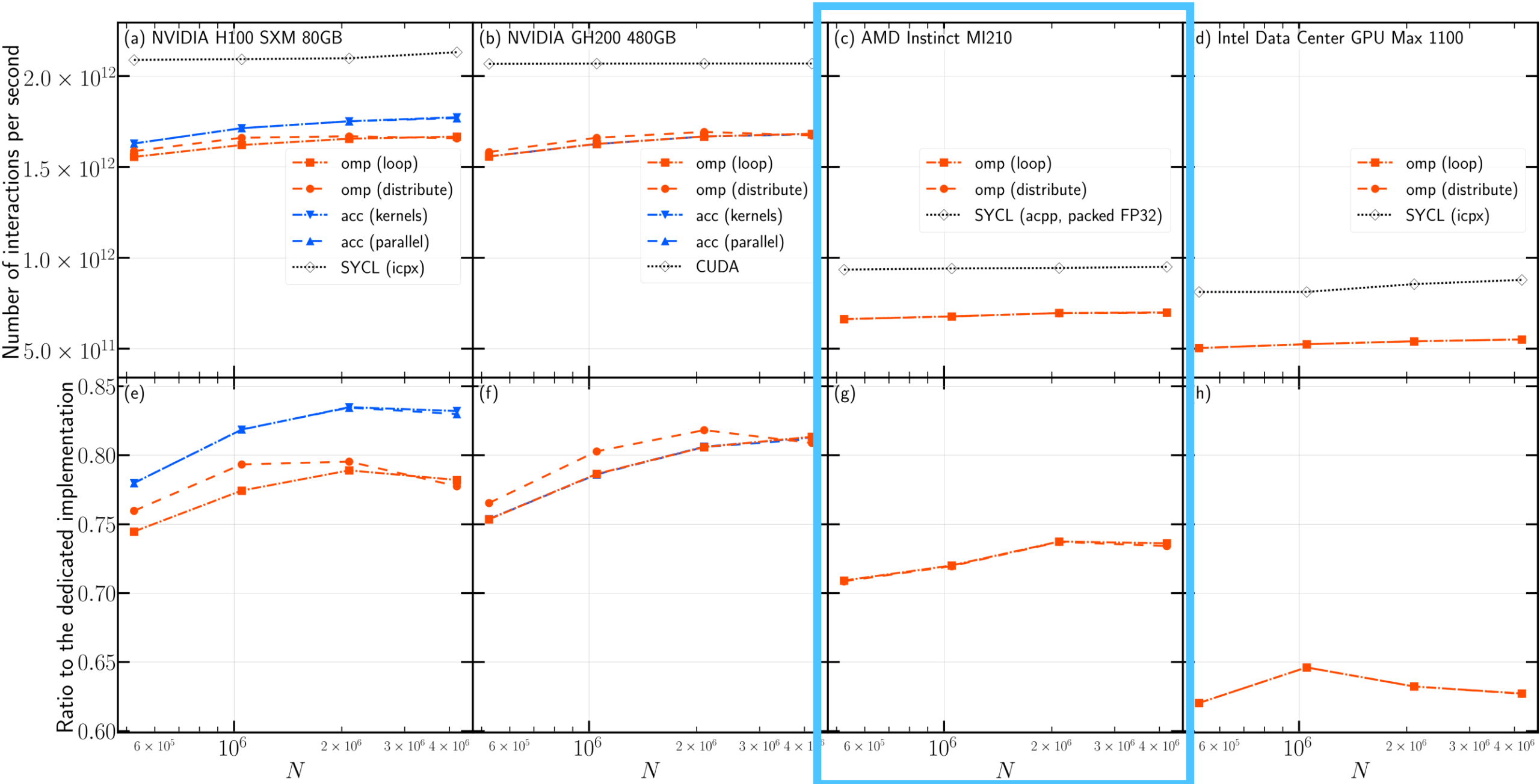
各GPU上で測定した性能の分析

- カタログスペックの比較

	NVIDIA GH200	NVIDIA B200	AMD MI300A
FP32性能	66.91 TFlop/s	74.45 TFlop/s	122.6 TFlop/s
並列度(FP32)	16896	18944	14592
動作周波数	1980 MHz	1965 MHz	2100 MHz
メモリ容量	HBM3 96GB	HBM3e 192GB	HBM3 128GB
メモリバンド幅	4.0 TB/s	7.7 TB/s	5.3 TB/s
TDP	1000 W (全体)	1000 W	760 W
プロセスルール	TSMC 4N	TSMC 4NP	TSMC 5N

- NVIDIA B200は, GH200から理論ピーク性能比相当の高速化
- AMD MI300Aは, GH200からの理論ピーク性能比よりも低い高速化
  - 消費電力あたりの性能はNVIDIA GH200/B200の方が高い
  - TDPがGH200/B200より大幅に増えない限りは, 性能の大幅増も望めない
    - H100のTDPは700 W

参考までに昔AMD MI210で測定したときの結果



指示文実装の性能サマリー

	OpenMP (loop)	OpenMP (distribute)	OpenACC (kernels)	OpenACC (parallel)
NVIDIA GH200 (nvc++)	○	△ (これだけ少し遅い)	○	○
AMD MI300A (amdclang++)	× (遅すぎて問題外)	○	× (そもそも動かない)	× (そもそも動かない)
AMD MI210 (amdclang++)	○	○	× (そもそも動かない)	× (そもそも動かない)

- OpenACCはAMD GPU向けでは動かない
  - 例外はHPE製のコンパイラ. ただしHPEが運用するシステムでないと使えない
    ➔ プラズマシミュレータやSirius の運用はNECのため, 国内GPUスパコンではNG
- 単一コードで全環境に対応しようとすると,
  OpenMP targetのdistribute指示文を選択することになる
  - ×がついていない選択肢を選ぶ, という消去法で解が決まってしまっている
- Solomonの場合はフラグだけでバックエンド切り替え可能なので, 大分マシ
- 指示文実装の場合には, MI300Aの性能はGH200と同程度

まとめと展望

- GPU向け指示文は複数あり, ユーザーはどれか1つを選択する必要がある
  - OpenACC: 機能・資料が充実しているが, ほぼNVIDIA GPU向け
  - OpenMP target: NVIDIA/AMD/Intel 全社に対応, 機能はキャッチアップ中
- GPU向け指示文統合マクロSolomonを開発, GitHubで公開
  - Simple Off-LOading Macros Orchestrating multiple Notations
  - プリプロセッサマクロ経由で指示文を記載するためのマクロ集
  - NVIDIA GPU上ではOpenACCを, AMD/Intel GPU上ではOpenMP targetを選択, ということができる
  - 簡易記法, OpenACC的記法, OpenMP的記法があるため, 学習コストを低減
  - GPU提供ベンダー製のコンパイラをそのまま使える
  - OpenACC と OpenMP target の性能比較も簡単にできる
  - 現状はC/C++版のみだが, Fortran版の開発が進行中
- Solomonのように, 用いるバックエンド(=開発手法)の決断をコンパイル時点にまで先送りできると性能的にも有利になるケースがある(“後出しジャンケン”化)
  - Kokkos, SYCL などC++ラムダベース手法についても同様の仕掛けを作れないか検討中