NVIDIA/AMD/Intel製GPU向けのN体計算コードの実装と性能評価

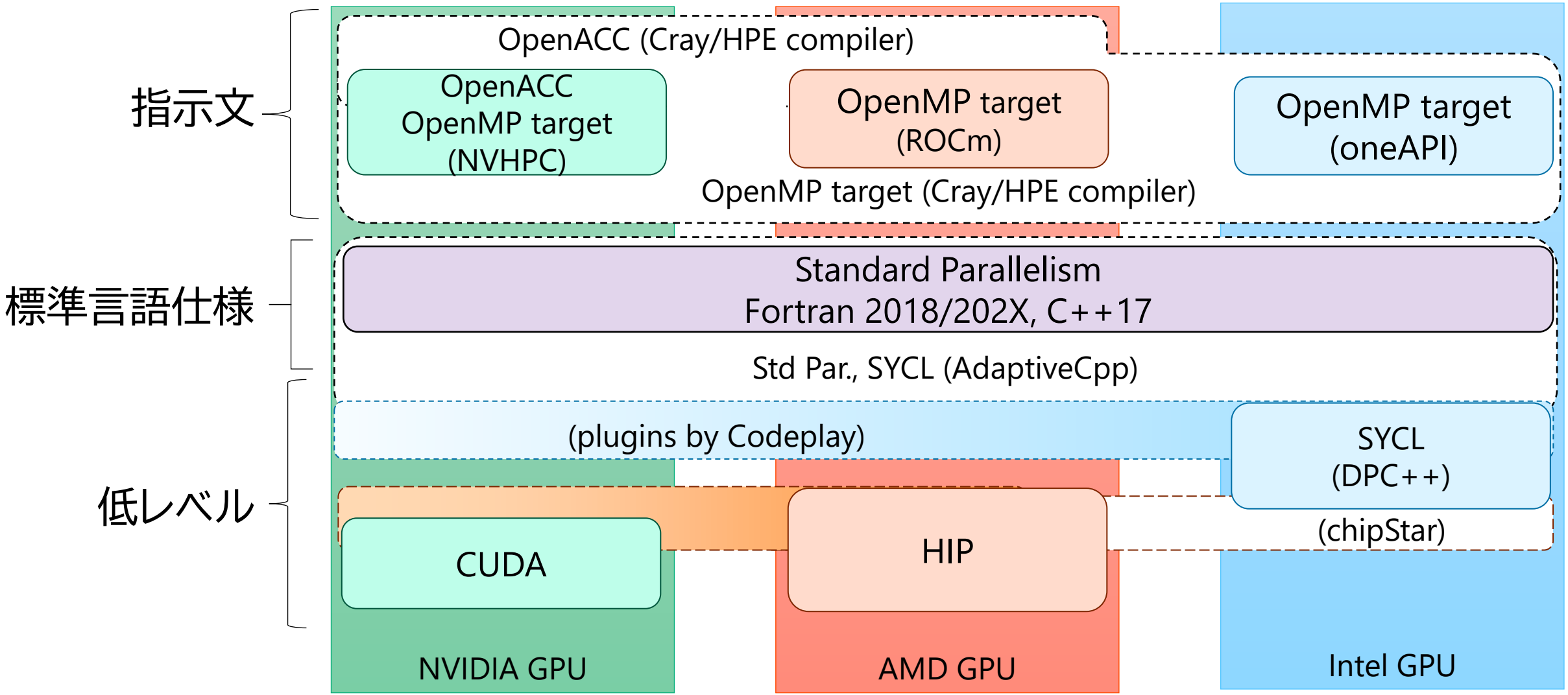
三木 洋平
（東京大学 情報基盤センター）

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
  - GPUは内部に多数のコアを搭載した並列計算機
- 「GPU = NVIDIAのGPU」と言って良いぐらいの独占状態
  - 国内では, HPCIに資源提供されている全てのGPUスパコンはNVIDIA製
    - 今年度稼働開始のTSUBAME4.0, 玄界, Miyabi もNVIDIA製GPUを搭載
  - 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3システムはAMD, Intel, NVIDIA製GPUが分け合う格好
- GPUベンダー間の競争が活性化
  - NVIDIA製GPUはP100, V100, A100の間は各世代$\sqrt{2}$倍の性能向上だったが, H100では約3倍の性能向上
  - (価格は下がってほしいが, 実際には高騰する一方...)
  - それぞれのGPU向けにどういう実装をし, 最適化すれば良いのかを調べる必要
  - 各GPUベンダーに対して中立な立場での検証・性能比較が重要
    - 他社製品向けに, 自社製品向けと同レベルでの最適化ができる(スキルを持っている)保証がない

GPU向けのプログラミング環境

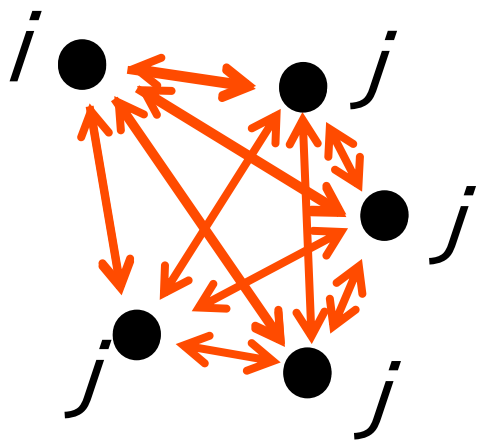
2月のPCCC AI/HPC OSS活用WSでの塙さんの講演資料
(https://www.pccluster.org/ja/event/data/240205_pccc_wsAI-HPC-OSS_06_hanawa-miki.pdf)



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

- 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)

SYCL環境の構築

- Intel oneAPI
  - Codeplay 社提供のプラグインをインストール（for NVIDIA/AMD GPUs）（https://codeplay.com/solutions/oneapi/plugins/）
- AdaptiveCpp（https://github.com/AdaptiveCpp/AdaptiveCpp）
  - ドキュメントにしたがってインストールすれば特に苦労なく使えた
    - Intel GPU 向けのドキュメントが見当たらなかったので, NVIDIA/AMD GPUs 限定の話
    - NVIDIA GH200 向けにもインストールできた（Arm CPU なので, oneAPI が使えない）
      - ただし, nvclangをバックエンドに取ることはできなかった（`llvm-tblgen`が不足）
  - いくつかのインストール方法が提示されているが, 試したのは以下の手順
    1. LLVM を手でインストール（NVPTX or AMDGPU を有効にしておく）
       - ドキュメントでは LLVM は dnf install することを推奨されていたが, 管理者権限なしでインストールできる方法を取ることにした（スパコンに一般ユーザとしてインストールして使うことを想定した予行演習）
    2. AdaptiveCpp のソースを取ってきて, cmake してコンパイル
  - レジスタを大量に消費する場合に計算がこける場合がある
    - スレッド数が512以上かつILP数8以上というかなり極端な条件なので, 普通は出くわさないはず

コードの実装手順・勉強手順

- CUDA C++版
  - 簡単なのでスクラッチから実装
- HIP C++版
  - CUDA版のうちいくつか簡単なものを hipify-clang を用いて変換
    (シェアードメモリの使い方など, ドキュメントを眺めるよりも簡単に使い方が分かる)
  - 感触をつかんだ後は, 普通にスクラッチから実装できるようになる
- SYCL版
  - CUDA版のうちいくつか簡単なものを dpct (今はSYCLomatic)を用いて変換
  - デフォルトでは各queueが Out-of-Order 実行されるので適宜 `wait()` をかける
  - `sycl::property::queue::in_order` を指定して queue を作成すると, CUDAの default stream を使ったときと同じ振る舞い
    - CUDAに慣れている人にとってはこちらの使い方のほうが馴染みやすいと思われる
  - 感触をつかんだ後は, 普通にスクラッチから実装できるようになる
- 注:「簡単」,「普通に」などはあくまでも個人の感想なので保証はしません

チューニング・性能評価環境

GPU	NVIDIA H100 SXM 80GB	NVIDIA GH200 480GB	AMD Instinct MI210	Intel Data Center GPU Max 1100
FP32 peak	66.9 TFlop/s	66.9 TFlop/s	22.6 TFlop/s	22.2 TFlop/s
# of units	132 SMs	132 SMs	104 CUs	448 Eus
FP32 parallelism	16896	16896	6656	7168
Clock frequency	1980 MHz	1980 MHz	1700 MHz	1550 MHz
TDP/TBP	700 W	(全体で)1000 W	300 W	300 W
Host CPU	Intel Xeon Platinum 8468	NVIDIA Grace	AMD EPYC 7713	Intel Xeon Platinum 8468
	48 cores × 2 sockets	72 cores	64 cores × 2 sockets	48 cores × 2 sockets
	2.1 GHz	3.1 GHz	2.0 GHz	2.1 GHz
	CUDA 12.3	CUDA 12.4	ROCm 6.0.2, 5.4.3	
Intel oneAPI	2024.1.0		2024.1.0	2024.1.0
LLVM	18.1.7	18.1.7	18.1.7	
AdaptiveCpp	24.02.0	24.02.0	24.02.0	

NVIDIA製GPU向けの実装

- CUDA, SYCL 実装を測定
  - HIP版はCUDA版と同性能が得られると分かっているので, 今回は割愛
    (管理者権限なしで(= rpmパッケージを使わずに)HIP環境を構築するのは大変)
- 逆数平方根, ブロックあたりのスレッド数, ループアンローリング段数, Instruction Level Parallelism(ILP)数については, パラメータ探査の結果最高性能だったものを採用
  - シェアードメモリの使い方(single or double buffer, ブロック単位 or ワープ単位)や `memcpy_async()`の使用についても同様
  - ただし`memcpy_async()`の使用はCUDA版のみで実装
- パラメータ探査時の粒子数はN=4M
  - 逆数平方根は`rsqrtf()`を使用
    (acppでは`__hipsycl_if_target_cuda()`経由, icpxでは`sycl::rsqrt()`)
  - CUDA: `memcpy_async()`を使ったほうが速くなった(FTZ有効時限定)
  - 最適なパラメータはCUDA, SYCL (icpx), SYCL (acpp) それぞれで異なる
    - スレッド数はそれぞれ 1024, 512, 256 がベストだった

AMD製GPU向けの実装

- HIP, SYCL 実装を測定
- 逆数平方根, ブロックあたりのスレッド数, ループアンローリング段数, Instruction Level Parallelism(ILP)数については, パラメータ探査の結果最高性能だったものを採用
  - シェアードメモリの使い方(single or double buffer, ブロック単位 or ウェーブ単位)の使用についても同様
  - Packed FP32命令の使用・不使用も実験

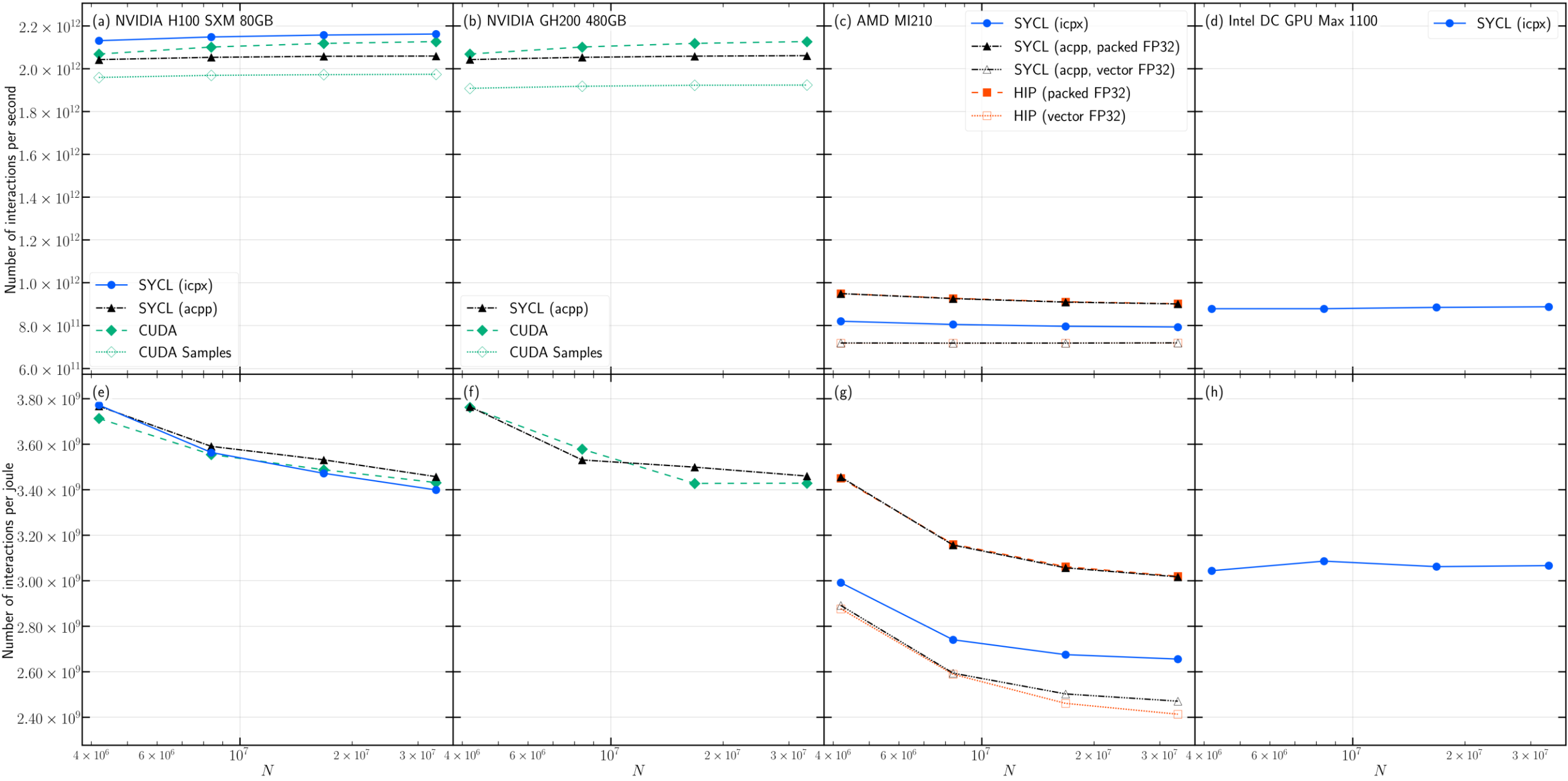
- パラメータ探査時の粒子数はN=4M
  - 逆数平方根は`__frsqrt_rn()`を使用 (acppでは`__hipsycl_if_target_hip()`経由, icpxでは`sycl::rsqrt()`)
  - スレッド数は256, シェアードメモリは不使用

Intel製GPU向けの実装

- SYCL 実装を測定, SYCL も Intel oneAPI のみ
- 逆数平方根, ブロックあたりのスレッド数, ループアンローリング段数, Instruction Level Parallelism(ILP)数については, パラメータ探査の結果最高性能だったものを採用
  - シェアードメモリの使い方(single or double buffer, ブロック単位 or ワープ単位)の使用についても同様

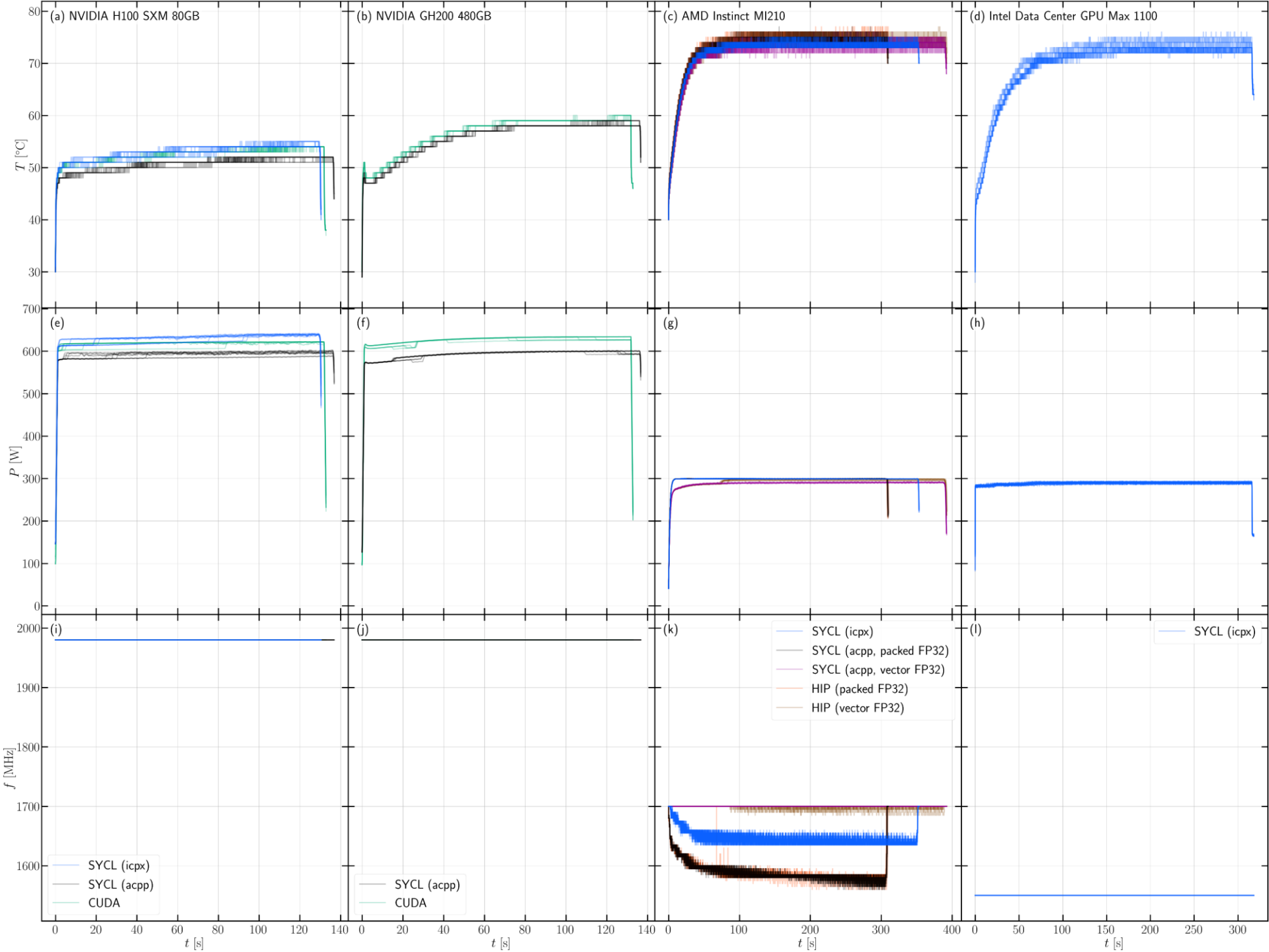
- パラメータ探査時の粒子数はN=4M
  - 逆数平方根は`sycl::rsqrt()`を使用
  - スレッド数は1024
  - シェアードメモリを使用(single buffer)
  - ループアンローリングは2段
  - ILPは使用せず(= 通常の実装)

NVIDIA/AMD/Intel製GPUの性能比較



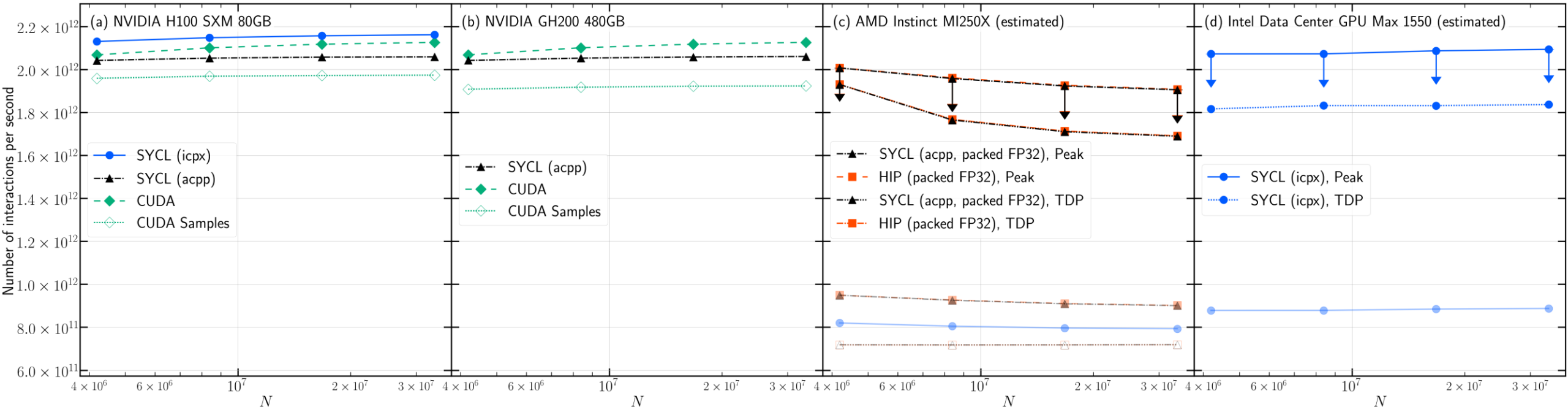
状態変化

- 計算中のGPU状態の変化も監視
  - 10回測定した結果をプロット
  - 電力もGPUの寄与のみを測定
- GPU温度上昇による動作周波数低下は起こらず
- AMD MI210では, packed FP32命令を使用すると供給電力が不足し, 動作周波数が低下



各世代の最上位製品どうしの比較

- AMD MI210, Intel Data Center GPU Max 1100 は最上位製品ではないので, 性能を予測
  - 理論ピーク性能に基づく予測：実際には供給電力が不足するはずなので上限値
    - AMD MI250X: MI210の2.12倍の理論ピーク性能, 1.87倍のTBP(注：MI210でも不足)
    - Intel Data Center GPU Max 1550: 1100の2.36倍の理論ピーク性能, 2倍のTDP
  - 消費電力に基づく予測：電力性能が変わらなければこちらの方が正解に近いはず



まとめ

- CUDA/HIP/SYCL を用いてN体計算コードを実装・最適化し, NVIDIA/AMD/Intel製GPU上での性能を比較した
- SYCL実装の性能が良好
  - 全ベンダー製GPUに対応できるコードできちんと性能を出せる
    - Intel oneAPI と AdaptiveCpp を使い分けられることも大きい
    - NVIDIA GH200 上では AdaptiveCpp が使える
  - NVIDIA H100 上では CUDA よりも速かった(!!)
  - AMD MI210 上では HIP とほぼ同性能
- NVIDIA Hopper, AMD CDNA 2, Intel PVC 世代においては NVIDIA H100 (SYCL, icpx) が最高性能となった
  - 電力性能についても同様(CPU, 冷却などの寄与は入っていない点に注意)
  - 使用電力は600 Wを越えており, こちらも最大
    (電力性能が一番高く, かつ供給電力も最大なので, 性能も最高値となる)