Performance evaluation of *N*-body codes on NVIDIA/AMD/Intel GPUs

Yohei MIKI
(Information Technology Center, The University of Tokyo)

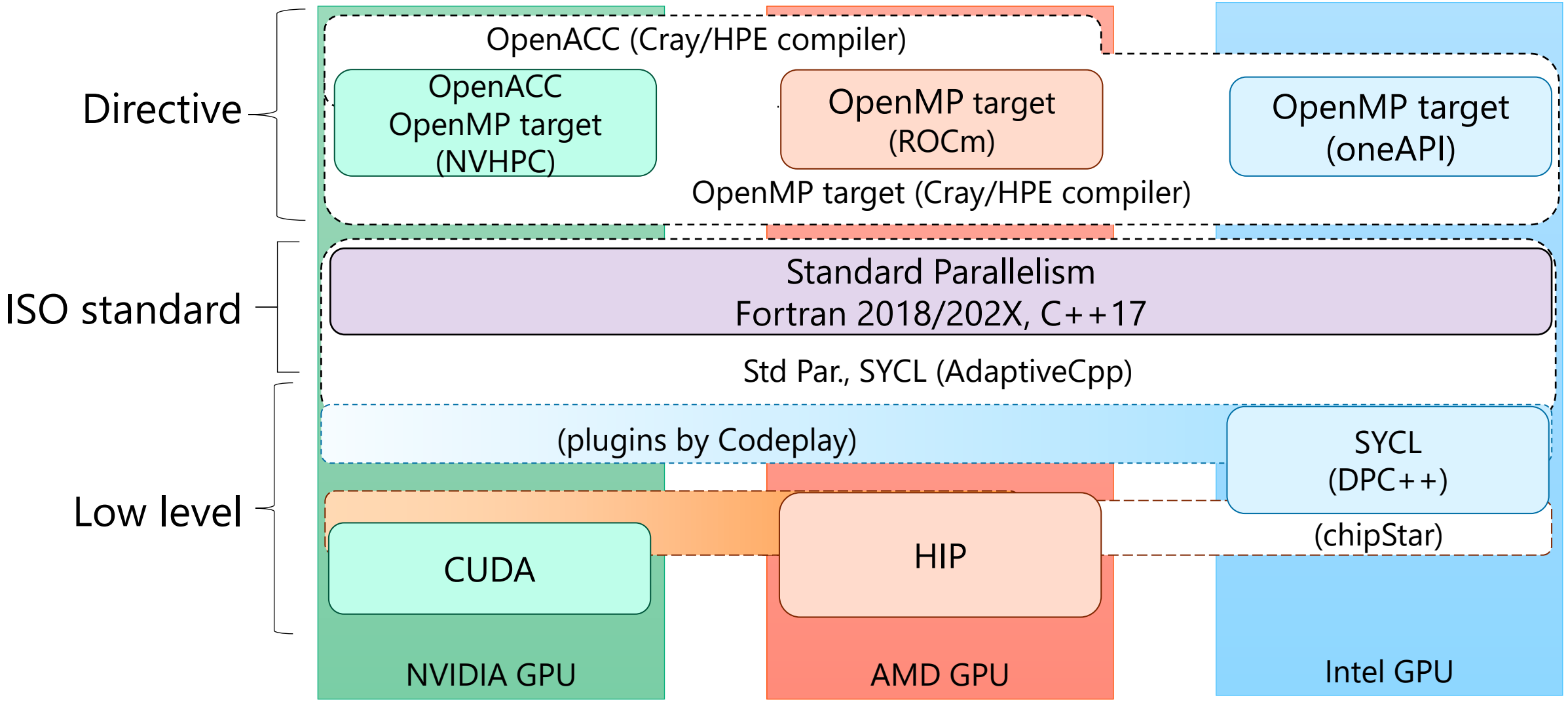
IXPUG Workshop at HPC Asia 2025

GPU-accelerated supercomputers

- GPU is the dominant accelerator on recent supercomputers
  - GPU is a parallel computer equipped with many cores
- Majority of GPU supercomputers are equipped with NVIDIA GPUs; however
  - Rise of AMD GPUs (El Capitan and Frontier: #1 and 2 in TOP500)
  - Intel GPU-powered systems (Aurora: #3 in TOP500)
- Increased competition among GPU vendors
  - Drives performance improvement of NVIDIA GPUs:
    - $\sim \sqrt{2}$x in each generation between P100, V100, and A100
    - ~ 3x in H100
  - Announcement of new GPU sometimes includes the release of additional features (e.g., new function)
  - We need to find out how to optimize for each GPU
  - Importance of “vendor-neutral” programming in GPU computing

Programming environments for GPU

Taken from
https://www.pccluster.org/ja/event/data/240205_pccc_wsAI-HPC-OSS_06_hanawa-miki.pdf



Construction of SYCL environments

- Intel oneAPI
  - We installed plugins (for NVIDIA/AMD GPUs) by Codeplay (https://codeplay.com/solutions/oneapi/plugins/)
- AdaptiveCpp (formerly hipSYCL or Open SYCL) (https://github.com/AdaptiveCpp/AdaptiveCpp)
  - We installed following the installation guide
    - We installed AdaptiveCpp on LLVM built from source files
    - Using dnf installed LLVM is much easier, but requires superuser privilege (we do not have on supercomputers)
  - Succeeded to install on NVIDIA GH200
    - Intel oneAPI is not available because NVIDIA Grace is Arm CPU
    - Failed to set `nvclang` as backend (lack of `llvm-tblgen`)
  - If the register usage is high, jobs might crush
    - My experience: many threads per block (≥ 512) && high ILP (≥ 8)

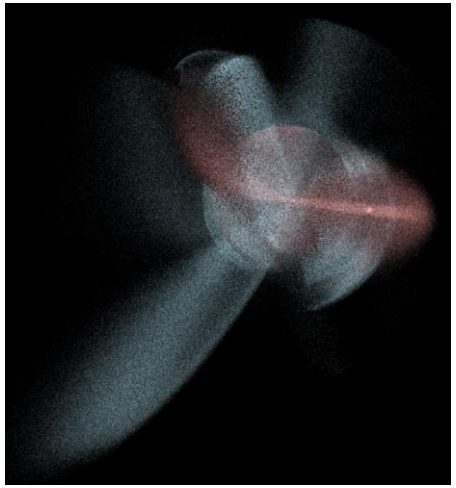
How to learn/develop codes

- CUDA C++
  - Very easy (for me), develop from scratch
- HIP C++
  - Convert some simple CUDA codes by using hipify-clang
    - Easiest way to learn HIP implementation
  - Now, it is straightforward to write codes from scratch
- SYCL
  - Convert some simple CUDA codes by using SYCLomatic (aka dpct)
  - Add `wait()` appropriately (in default, queues are Out-or-Order)
  - Specifying `sycl::property::queue::in_order` would be better for CUDA users (same behavior of default stream in CUDA)
  - Now, it is straightforward to write codes from scratch
- Code converters (hipify-clang, SYCLomatic) can generate personalized sample implementation for your codes

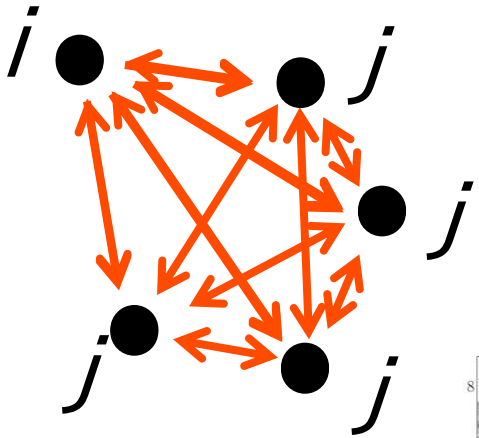
What is N-body simulation?

- Calculating time evolution of the gravitational many-body system by solving Newton's equation of motion
  - Data: $\mathcal{O}(N)$
  - Force calculation: $\mathcal{O}(N^2)$
  - Time integration: $\mathcal{O}(N)$

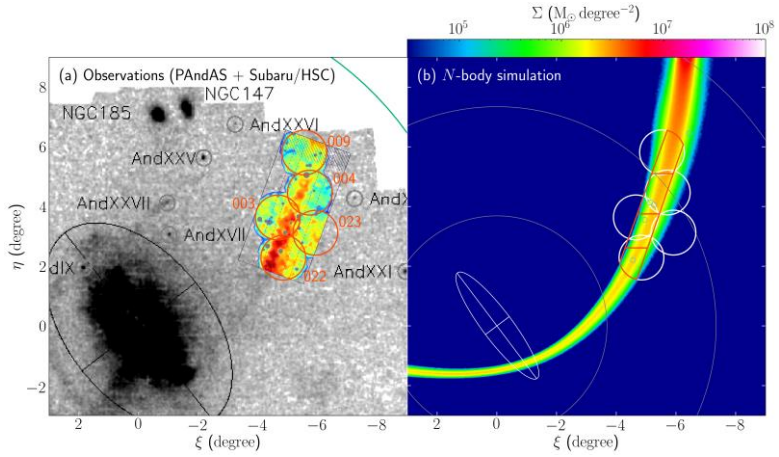
$$\boldsymbol{a}_i = \sum_{\substack{j=0 \\ j \neq i}}^{N-1} \frac{G m_j \left(\boldsymbol{x}_j - \boldsymbol{x}_i\right)}{\left(\left|\boldsymbol{x}_j - \boldsymbol{x}_i\right|^2 + \epsilon^2\right)^{3/2}}$$



- Terminology
  - i-particle: particle feels gravity
  - j-particle: particle causes gravity



- Direct solver is
  - a backend of tree method
  - employed in collisional N-body simulations



HW/SW environments

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	1000 W (total)	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	

Optimal instruction for reciprocal square root (# of interactions per sec. at N=4M)

GPU	Compiler	1.0F/std::sqrt()	rsqrtf()	__frsqrt_rn()	sycl::rsqrt()	sycl::native::rsqrt()
NVIDIA H100	nvcc	8.06×10^{11}	1.51×10^{12}	7.68×10^{11}		
NVIDIA H100	icpx	1.15×10^{12}			1.77×10^{12}	1.77×10^{12}
NVIDIA H100	acpp	7.90×10^{11}			8.08×10^{11}	8.08×10^{11}
NVIDIA GH200	nvcc	8.15×10^{11}	1.51×10^{12}	7.68×10^{11}		
NVIDIA GH200	acpp	8.01×10^{11}			8.08×10^{11}	8.08×10^{11}
AMD MI210	hipcc	2.36×10^{11}	5.28×10^{11}	7.05×10^{11}		
AMD MI210	icpx	7.13×10^{11}			7.13×10^{11}	7.13×10^{11}
AMD MI210	acpp	2.35×10^{11}			7.06×10^{11}	7.06×10^{11}
Intel 1100	icpx	5.52×10^{11}			5.52×10^{11}	5.52×10^{11}

Special treatment for AdaptiveCpp

- We utilized the platform-specific features to exploit the optimal instruction for the reciprocal square root

```
#if defined(__ACPP__) && (defined(__NVPTX__) || defined(__AMDGCN__))
HIPSYCL_UNIVERSAL_TARGET
#endif  // defined(__ACPP__) && (defined(__NVPTX__) || defined(__AMDGCN__))
void calc_acc_kernel(sycl::nd_item<1> nd, const int Ni, position *ipos,
    acceleration *iacc, const int Nj, position *jpos, const float eps, position *
    jpos_shmem) {
     // skip initialization and normal implementation in SYCL

     const auto r2 = sycl::fma(rji.z, rji.z, sycl::fma(rji.y, rji.y, sycl::fma(rji.
        x, rji.x, pi.w)));
#if defined(__ACPP__) && (defined(__NVPTX__) || defined(__AMDGCN__))
#if defined(__NVPTX__)
         __hipsycl_if_target_cuda(rji.w = rsqrtf(r2););
#endif  // defined(__NVPTX__)
#if defined(__AMDGCN__)
         __hipsycl_if_target_hip(rji.w = __frsqrt_rn(r2););
#endif  // defined(__AMDGCN__)
#else   // defined(__ACPP__) && (defined(__NVPTX__) || defined(__AMDGCN__))
         rji.w = sycl::rsqrt(r2);
#endif  // defined(__ACPP__) && (defined(__NVPTX__) || defined(__AMDGCN__))

// skip normal implementation in SYCL and finalization
}
```

Implementation for NVIDIA GPU

- CUDA and SYCL implementations
  - HIP is skipped in this study (same performance with CUDA)
- Reciprocal square root: `rsqrtf()`
- Parameter survey to find the best configuration:
  - number of threads per block
  - number of unrolling counts
  - number of instruction level parallelism (ILP)
  - shared memory usage (single or double buffer, per-block or per-warp)
  - `memcpy_async()` usage
  - adoption of FTZ (flush-to-zero) option

Implementation for AMD GPU

- HIP and SYCL implementations
- Reciprocal square root: `__frsqrt_rn()`
- Parameter survey to find the best configuration:
  - number of threads per block
  - number of unrolling counts
  - number of instruction level parallelism (ILP)
  - shared memory usage (single or double buffer, per-block or per-warp)
  - packed FP32 usage: doubles the FMA, FADD, FMUL performance
  - adoption of FTZ (flush-to-zero) option

Implementation for Intel GPU

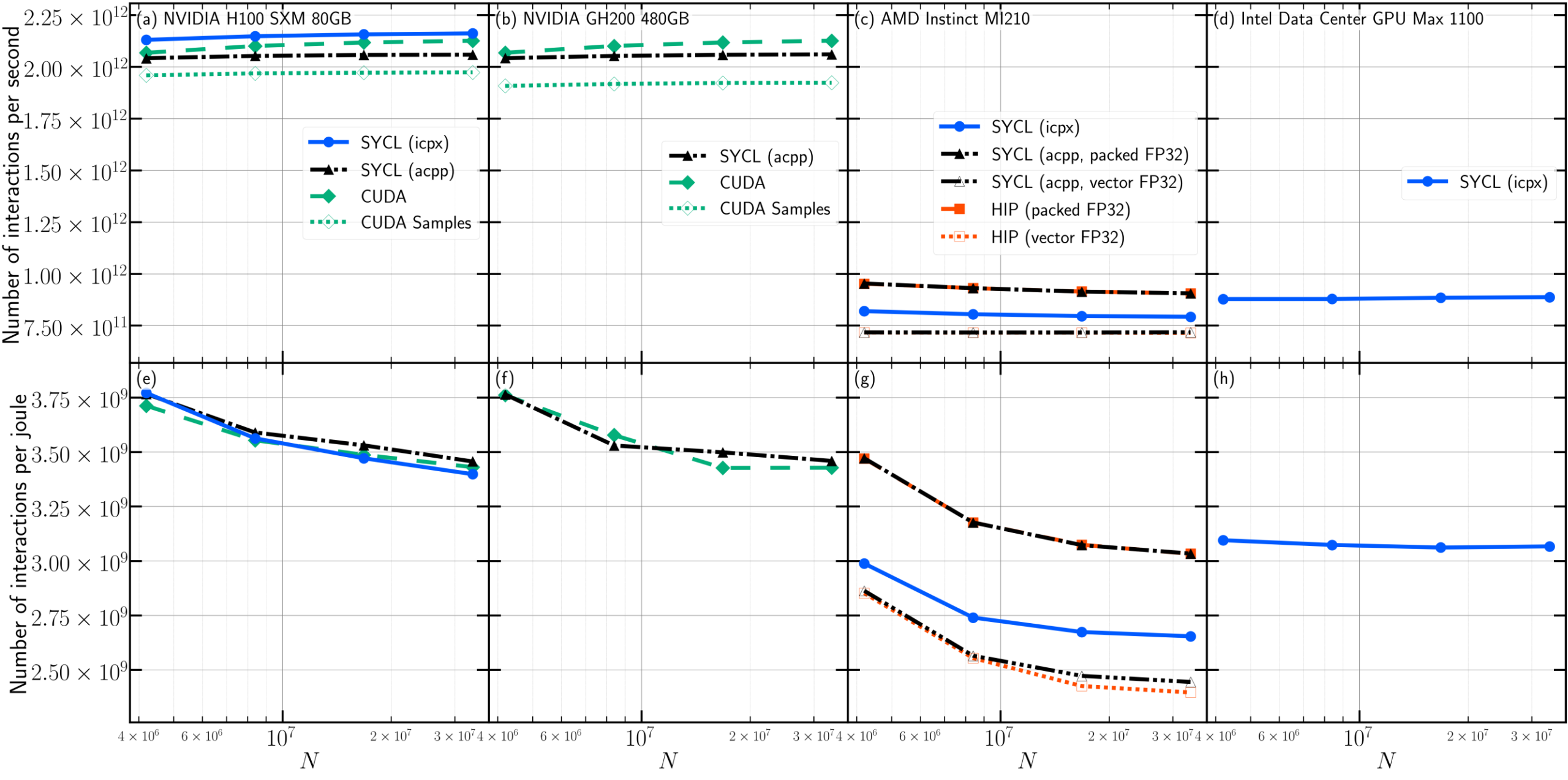
- SYCL implementation
- Reciprocal square root: `sycl::rsqrt()`
  (Intel oneAPI uses the best instruction)
- Parameter survey to find the best configuration:
  - number of threads per block
  - number of unrolling counts
  - number of instruction level parallelism (ILP)
  - shared memory usage (single or double buffer, per-block or per-warp)
  - adoption of FTZ (flush-to-zero) option

How to monitor GPU status

- On-the-fly monitoring:
  - temperature
  - clock frequency
  - power
- NVIDIA GPU: NVML
- AMD GPU: ROCm SMI
- Intel GPU: Level-Zero Sysman API
  - The root account is required for some metrics (e.g., temperature)
- We utilized “unused cores” to monitor GPU status in 0.1 sec interval
  - Task clause in OpenMP
  - No change in the main code

```
static bool repeat;
repeat = true;
#pragma omp parallel num_threads(2)
{
#pragma omp single
  {
#pragma omp task
    {
      while (repeat) {
        observe_GPU();
        sleep();
      }
    }
#pragma omp task
    {
      run_simulation();
      repeat = false;
    }
#pragma omp taskwait
  }
}
```

Measured performance

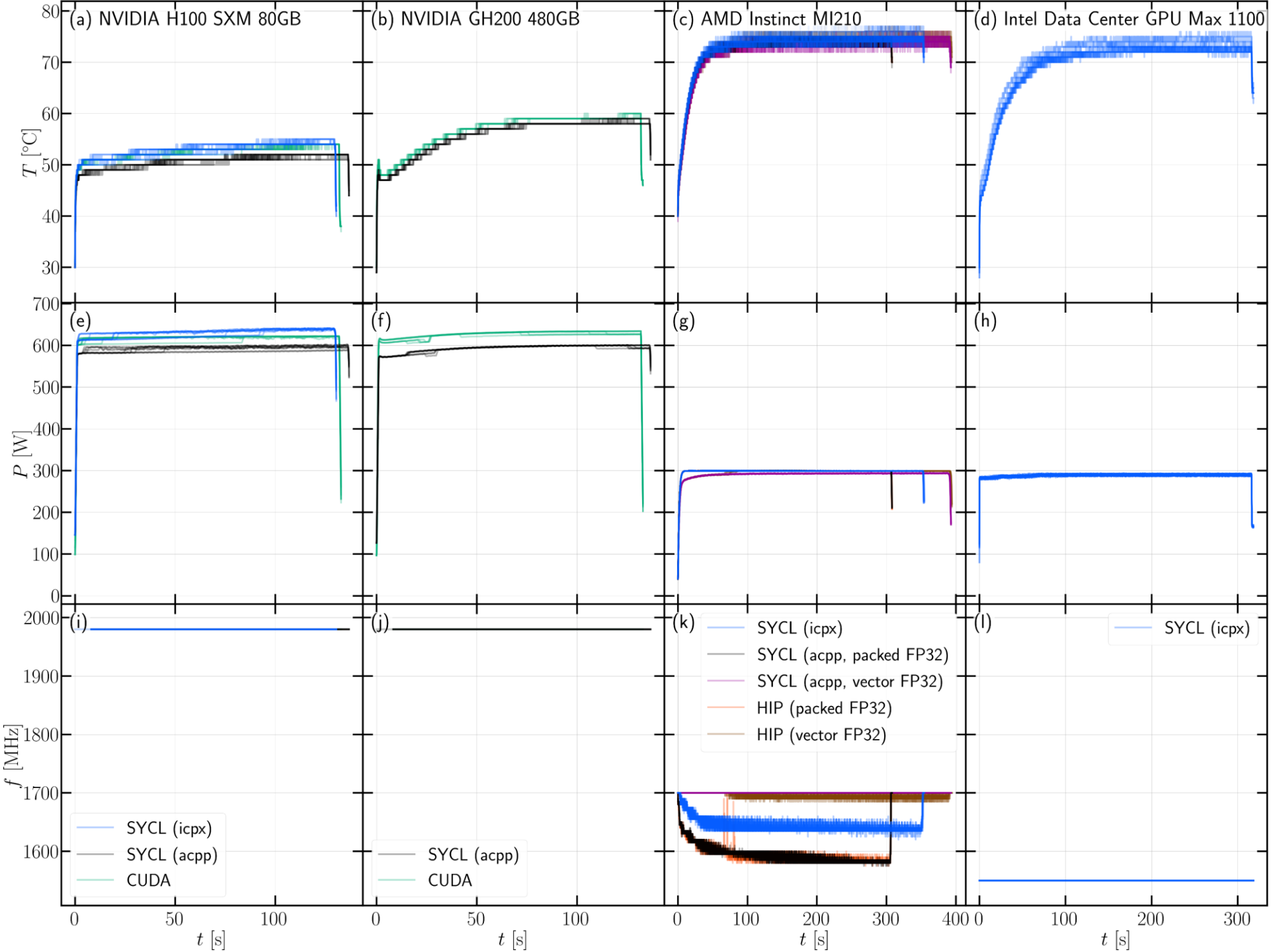


What we observed

- SYCL covers all three GPU vendors
  - On NVIDIA GPU, SYCL (compiled by Intel oneAPI) is the fastest (although not available on NVIDIA GH200)
  - On AMD GPU, SYCL (compiled by AdaptiveCpp) and HIP provide almost the same performance
    - Importance of packed FP32 instructions
- Intel PVC achieves similar performance to AMD MI210
- Measured performances on AMD MI210 and Intel Data Center GPU Max 1100 are around half of those on NVIDIA H100 SXM and GH200
  - The ratio of the theoretical peak performance is around 1/3
- NVIDIA GPUs show the best energy efficiency
  - Perhaps, due to the finest fabrication process

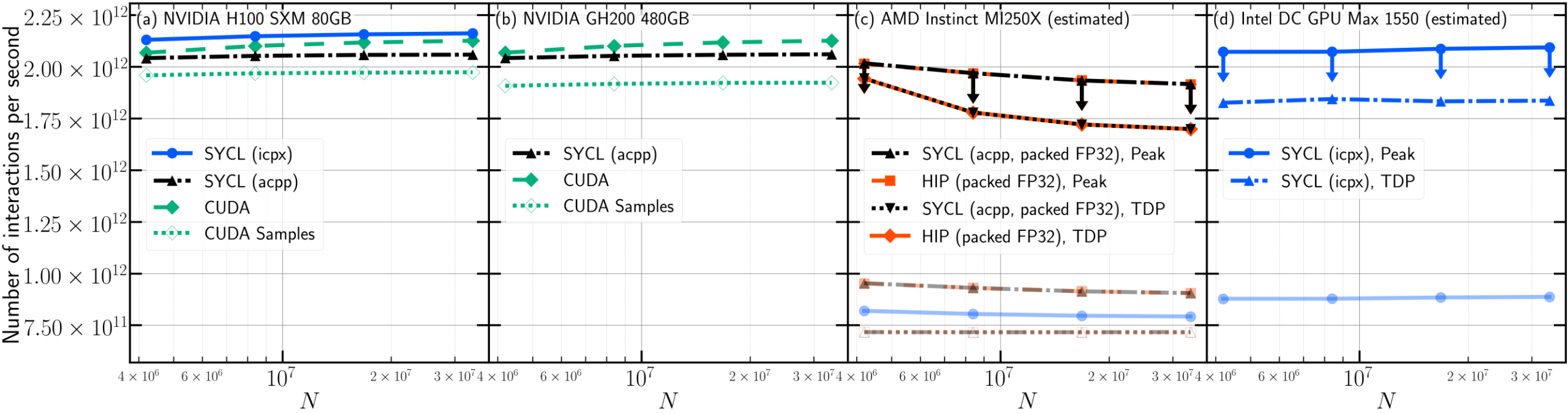
GPU status

- Temperature, power, frequency of GPU
- No thermal throttling
- On AMD MI210, clock frequency decreases due to insufficient power supply if we use packed FP32 instructions



Performance projection

- AMD MI210 and Intel Data Center GPU Max 1100 are not the flagship GPUs (flagships have ~2x higher performance)
  - Estimation based on the ratio of theoretical peak performance
    - (Estimation with lower arrows)
    - AMD MI250X: 2.12x performance of MI210; however, 1.87x TBP
    - Intel Data Center GPU Max 1550: 2.36x performance of 1100, 2x TDP
  - Estimation based on the energy efficiency



Summary

- We have developed and optimized N-body codes written in CUDA C++, HIP C++, and SYCL
- We measured the performance on NVIDIA H100 SXM, NVIDIA GH200, AMD MI210, and Intel DC GPU Max 1100
- SYCL was quite performance-portable
  - Fastest on NVIDIA H100 (Intel oneAPI)
  - 97% of CUDA performance on NVIDIA GH200 (AdaptiveCpp)
  - Almost identical performance with HIP on AMD MI210 (AdaptiveCpp)
  - Reasonable performance on Intel Data Center GPU Max 1100 (Intel oneAPI)
- NVIDIA GPUs achieved the best performance and the best energy efficiency in this generation
  - What about the next generation?: AMD MI300, NVIDIA B200, …

Comparison with Kokkos (preliminary)

- Kokkos is a widely employed performance-portable framework
  - The most time-consuming part of code development for me was the CMake part
- Measurement on Miyabi-G (JCAHPC)
  - NVIDIA GH200 120GB
  - CUDA 12.6
  - AdaptiveCpp 24.10.0
    - LLVM 19.1.7
  - Kokkos 4.5.99
    - with trivial fix in `nvcc_wrapper`
- SYCL (acpp) is fastest(!!)