

# CUDA Programming

---

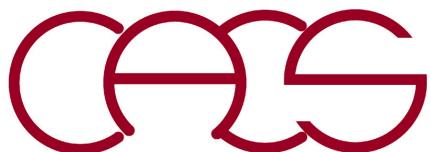
---

Aiichiro Nakano

*Collaboratory for Advanced Computing & Simulations  
Department of Computer Science  
Department of Physics & Astronomy  
Department of Quantitative & Computational Biology  
University of Southern California*

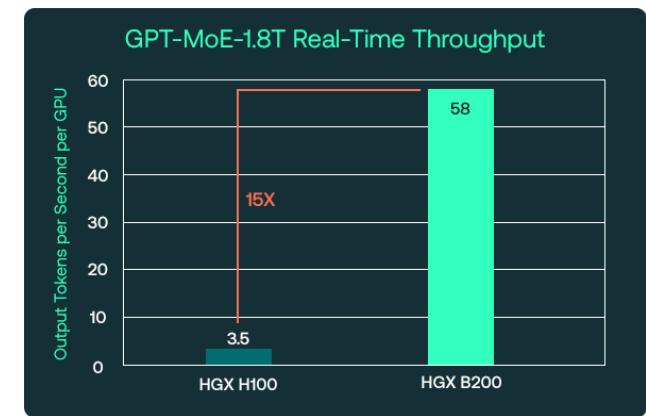
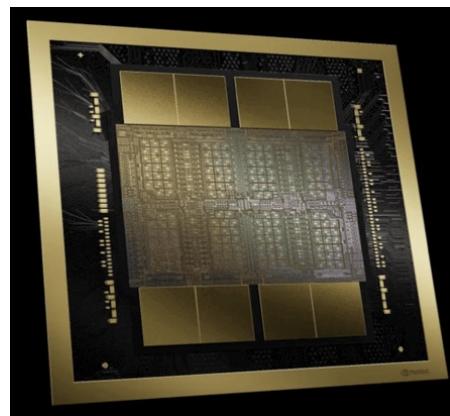
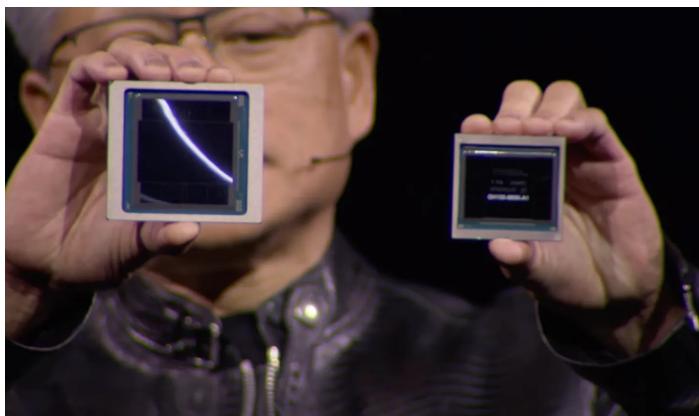
Email: [anakano@usc.edu](mailto:anakano@usc.edu)

Goal: Multithreading on graphics processing units (GPUs);  
heterogenous device concept



# Graphics Processing Unit (GPU)

- **GPU:** A specialized processor that offloads 3D graphics rendering from the central processing unit (CPU)
- **GPGPU:** General-purpose computing on GPU, by using a GPU to perform computation traditionally handled by the CPU; GPU is considered as a multithreaded, massively data parallel co-processor (accelerator)
- NVIDIA Quadro, Tesla & newer GPUs are capable of general-purpose computing with the use of Compute Unified Device Architecture (CUDA)



NVIDIA B200 (16,896 CUDA cores & 528 tensor cores)

# CUDA

---

---

## How to program GPGPU?

- **Compute Unified Device Architecture**
- **Integrated host (CPU) + device (GPU) application programming interface based on C language, developed at NVIDIA**
- **CUDA homepage**  
[http://www.nvidia.com/object/cuda\\_home.html](http://www.nvidia.com/object/cuda_home.html)
- **Widely used in the deep-learning community**  
<https://www.deeplearningbook.org/contents/applications.html>

# Using CUDA on Discovery

---

- Add the following commands in .bashrc in your home directory

```
module purge
module load usc
module load cuda
```

- Compilation

```
nvcc -o pi pi.cu
```

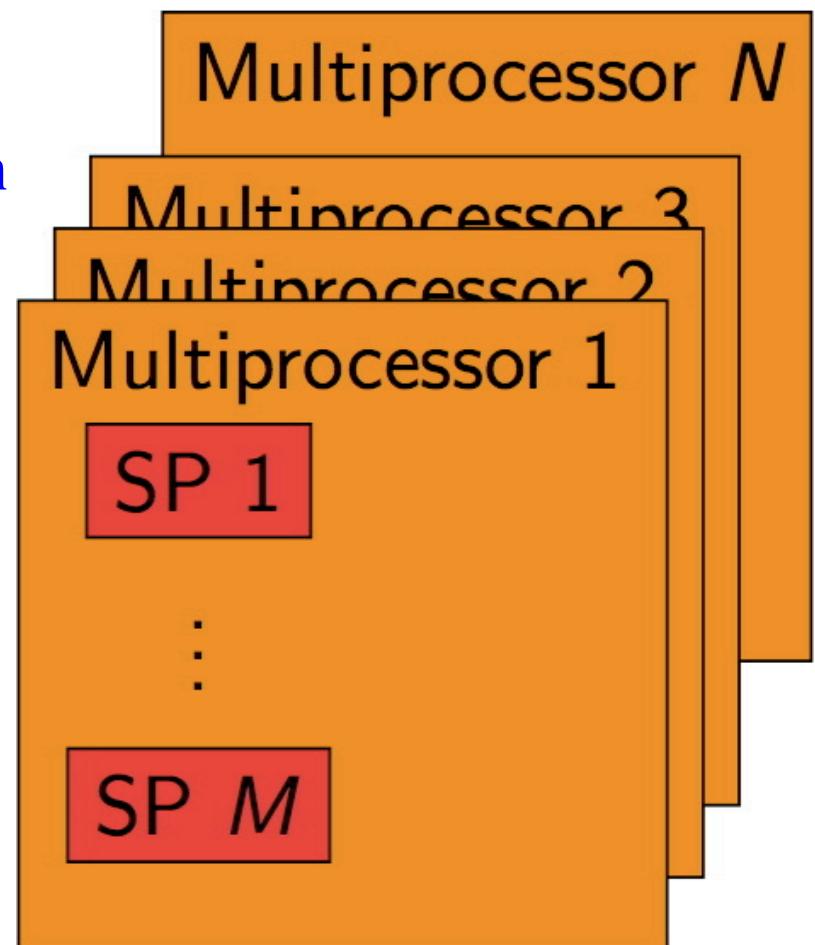
- Submit a Slurm script

```
#!/bin/bash
#SBATCH --partition=gpu
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --gpus-per-task=1
#SBATCH --time=00:00:59
#SBATCH --output=pi.out
#SBATCH -A anakano_429
./pi
```

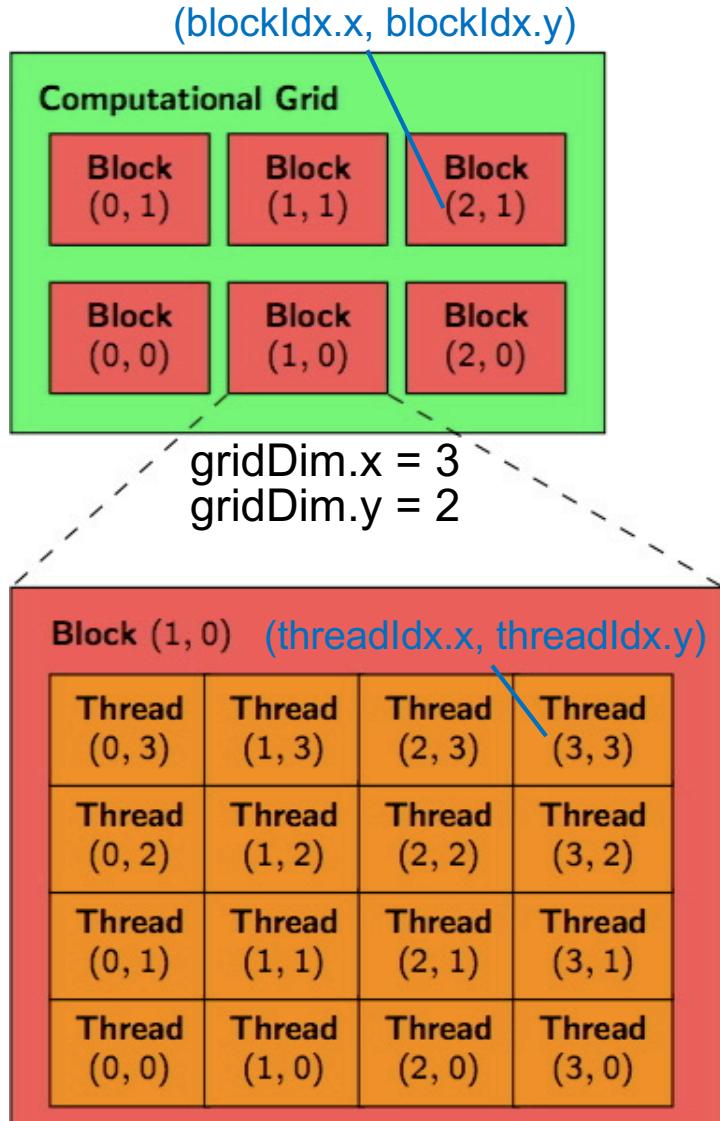
<https://aiichironakano.github.io/cs653/src/parallel/pi.cu>

# Example of NVIDIA GPU at CARC

- Host (CPU)
  - > Dual octacore ( $2 \times 8 = 16$ ) Intel Xeon
  - > Clock rate: 2.4 GHz
  - > Memory: 64 GB
- Device (GPU): Dual NVIDIA Tesla K20m
  - > Number of streaming multiprocessors (SMs) per GPU: 13
  - > Number of cores (or streaming processors, SPs) per SM: 192
  - > Total number of cores:  $13 \times 192 = 2496$
  - > Clock rate: 706 MHz
  - > Global memory: 5 GB
  - > Shared memory per SM: 48 KB



# Grid, Blocks & Threads



- **Computational grid** = a 1 or 2D grid of **thread blocks** (*cf.* SMs); each **block** = a 1, 2 or 3D array of  $\leq 512$  **threads** (*cf.* SPs); the application specifies the grid & block dimensions
  - **gridDim** provides dimension of grid; 1 or 2 element struct: “**.x**” & “**.y**”
  - **blockDim** provides dimension of block; 1, 2 or 3 element struct: “**.x**”, “**.y**” & “**.z**”
- All threads within a block execute the same kernel (SPMD) & cooperate *via* shared memory, atomic operations & barrier synchronization
- Each block has a unique **block ID**
  - **blockIdx** is 1 or 2 element struct
- Each thread has a unique ID within the block
  - **threadIdx** is a struct with up to 3 elements: “**.x**”, “**.y**” (in 2 or 3D) & “**.z**” (in 3D) for the innermost, intermediated & outermost index
- Each thread uses the block & thread IDs to decide what data to work on (*i.e.*, SPMD)

*cf. vproc[3], vthrd[3], vid[3], vtd[3] in hmd.c*

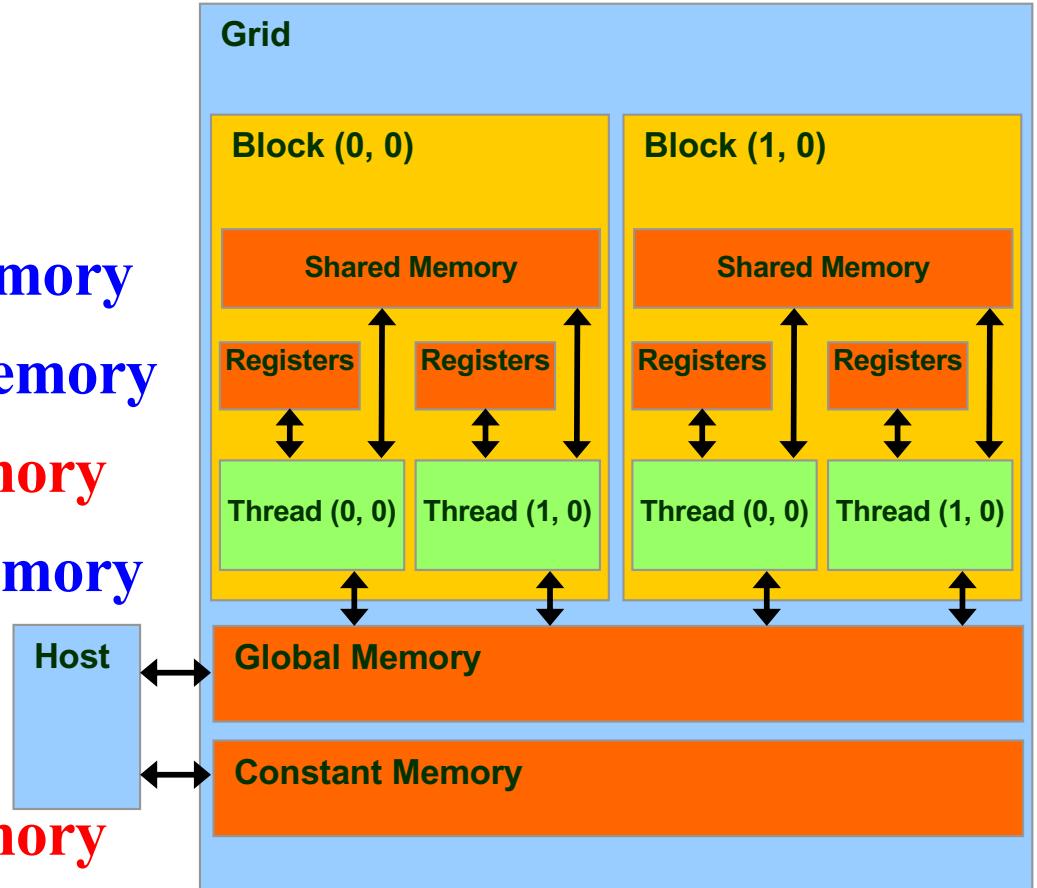
# Hierarchical Device Memory

Each thread can:

- Read/write **per-thread registers**
- Read/write **per-thread local memory**
- Read/write **per-block shared memory**
- Read/write **per-grid global memory**
- Read only **per-grid constant memory**

Host code can:

- Read/write **per-grid global memory**
- Read/write **per-grid constant memory**



We will only use global device memory in assignment

# Device Memory Allocation

## cudaMalloc()

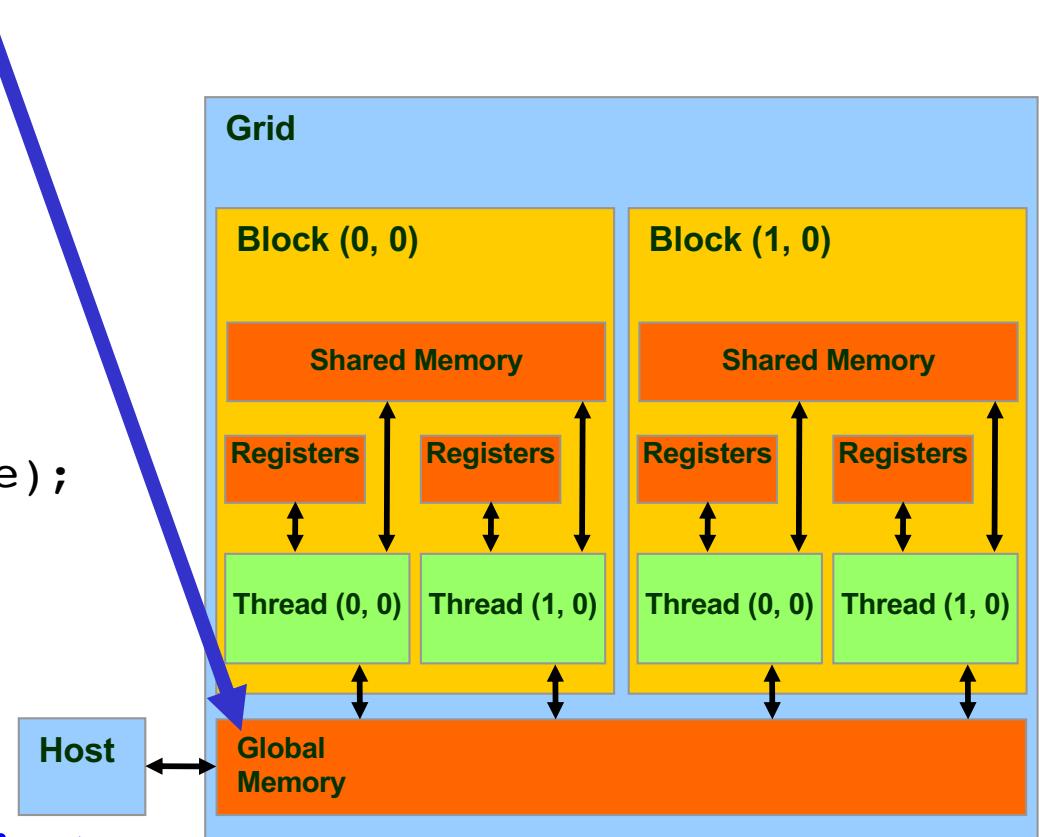
- Allocates object in the device global memory
- Requires two parameters:
  - Address of a pointer to the allocated object
  - Size of allocated object

```
cudaMalloc( void ** )&sumDev, size);
```

## cudaFree()

- Frees object from device global memory
- Parameter: Pointer to freed object

```
cudaFree( sumDev );
```

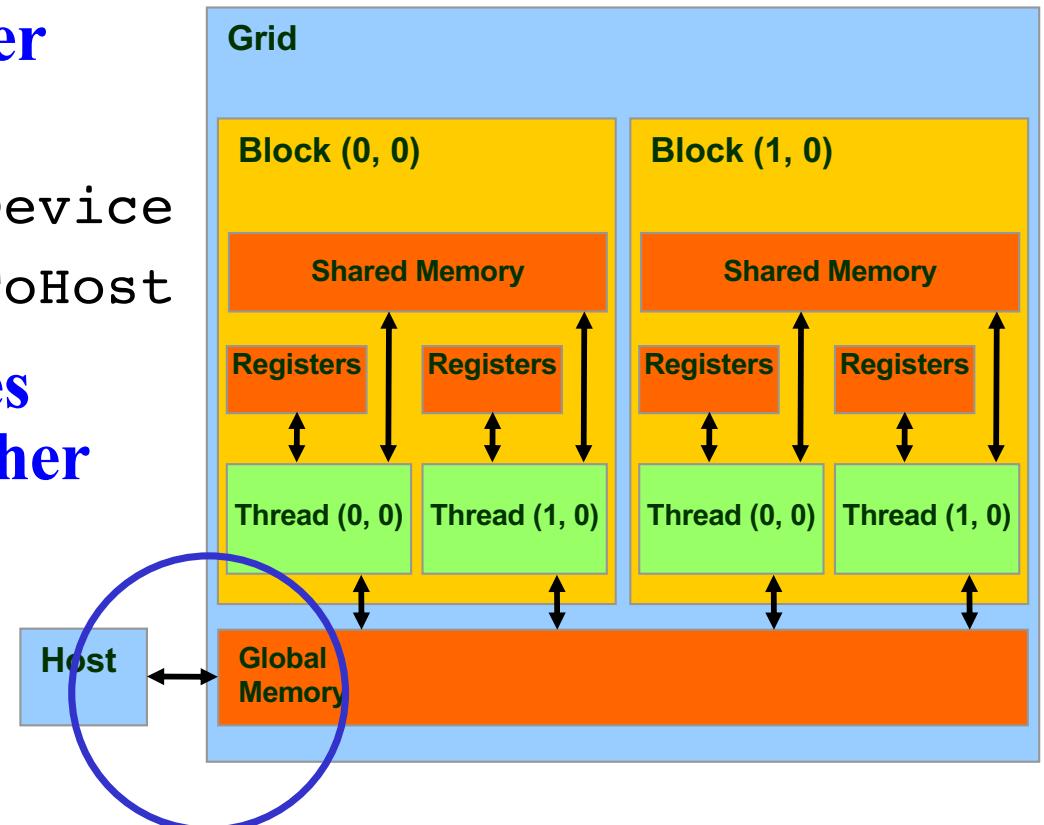


# Host-Device Data Transfer

**cudaMemcpy(dest, src, size, cmd)**



- **Arguments**
  - dest = pointer to array to receive data
  - src = pointer to array to source data
  - size = # of bytes to transfer
  - cmd = transfer direction
    - > cudaMemcpyHostToDevice
    - > cudaMemcpyDeviceToHost
- **Transfer specified # of bytes from one memory to the other in direction specified**



cudaMemcpy(sumHost, sumDev, size, cudaMemcpyDeviceToHost);

# Kernel Program for Device

- Set of threads triggered by invocation of a single kernel

- Definition

```
__global__ void kernel_fun(argument_list)
```

Kernel that can be called from a host function

- Invocation

```
kernel_fun <<<execution configuration>>> (operands)
```

- Range specifies set of values for thread grid

```
host_fun() {  
    ...  
    dim3 dimGrid(4, 2, 1)  
    dim3 dimBlock(2, 2, 2)  
    kernel_fun <<<dimGrid, dimBlock>>> (operands)  
    ...  
}
```

4×2 grid (3<sup>rd</sup> dimension not used)

2×2×2 block

*cf. omp\_set\_num\_threads()*

3-element struct accessed by `dimGrid.x`, `dimGrid.y`, `dimGrid.z`

# Built-in Variables

---

- `dim3 gridDim;`

**Dimensions of the grid in blocks (gridDim.z unused)**

- `dim3 blockDim;`

**Dimensions of the block in threads**

*cf. vproc[3] & vthrd[3] in hmd.c*

- `dim3 blockIdx;`

**Block index within the grid**

- `dim3 threadIdx;`

**Thread index within the block**

*cf. vid[3] & vtd[3] in hmd.c*

# Calculate Pi with CUDA: pi.cu (1)

```
// Using CUDA device to calculate pi
#include <stdio.h>
#include <cuda.h>

#define NBIN 10000000 // Number of bins
#define NUM_BLOCK 13 // Number of thread blocks
#define NUM_THREAD 192 // Number of threads per block
int tid;
float pi = 0;

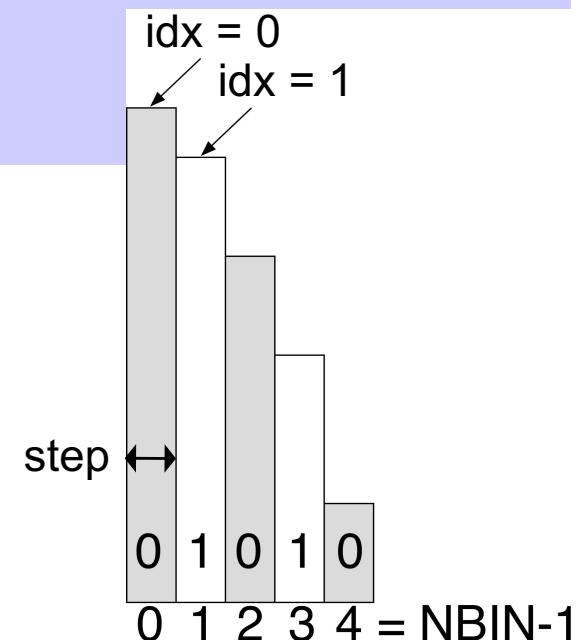
// Kernel that executes on the CUDA device
__global__ void cal_pi(float *sum, int nbin, float step, int nthreads, int nblocks) {
    int i;
    float x;
    int idx = blockIdx.x*blockDim.x+threadIdx.x; // Sequential thread index across blocks
    for (i=idx; i< nbin; i+=nthreads*nbblocks) { // Interleaved bin assignment to threads
        x = (i+0.5)*step;
        sum[idx] += 4.0/(1.0+x*x); // Data privatization
    }
}
```

blockIdx.x:	0	1	2
threadIdx.x:	0 1 2 ... 191	0 ... 191	0 ...
idx:	0 1 2 ... 191	192 ... 383	384 ...

1D grid & block

gridDim.x|y = 13|1  
blockDim.x|y|z = 192|1|1

Total number of threads =  $13 \times 192 = 2,496$



# Calculate Pi with CUDA: pi.cu (2)

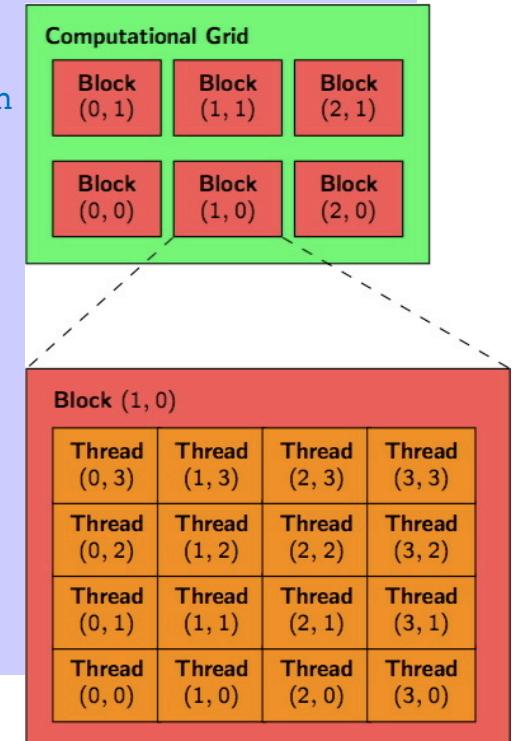
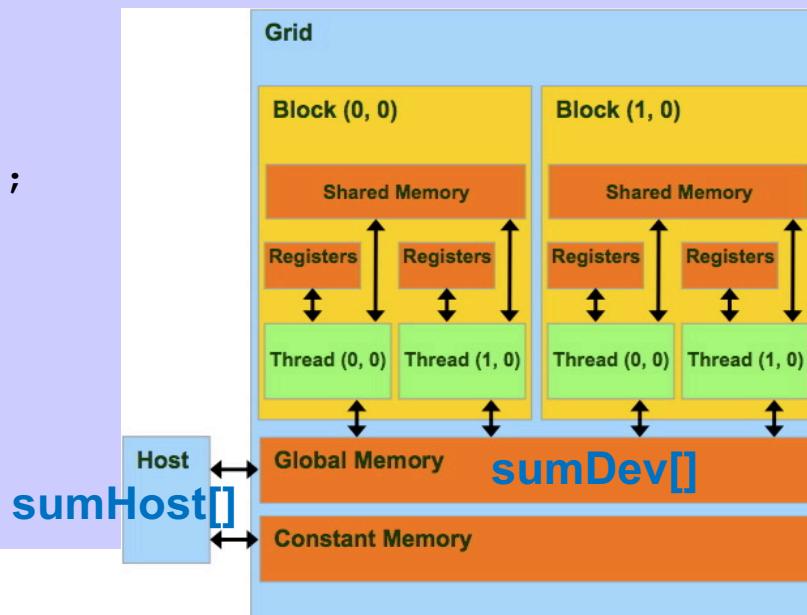
```
// Main routine that executes on the host
int main(void) {    13      192
    dim3 dimGrid(NUM_BLOCK,1,1); // Grid dimensions
    dim3 dimBlock(NUM_THREAD,1,1); // Block dimensions
    float *sumHost, *sumDev; // Pointer to host & device arrays

    float step = 1.0/NBIN; // Step size
    size_t size = NUM_BLOCK*NUM_THREAD*sizeof(float); // Array memory size
    sumHost = (float *)malloc(size); // Allocate array on host
    cudaMalloc((void **) &sumDev, size); // Allocate array on device
    // Initialize array in device to 0
    cudaMemset(sumDev, 0, size);
    // Do calculation on device by calling CUDA kernel
    cal_pi <<<dimGrid, dimBlock>>> (sumDev, NBIN, step, NUM_THREAD, NUM_BLOCK);
    // Retrieve result from device and store it in host array
    cudaMemcpy(sumHost, sumDev, size, cudaMemcpyDeviceToHost);
    for(tid=0; tid<NUM_THREAD*NUM_BLOCK; tid++) // Thread reduction
        pi += sumHost[tid];
    pi *= step;

    // Print results
    printf("PI = %f\n",pi);

    // Cleanup
    free(sumHost);
    cudaFree(sumDev);

    return 0;
}
```



# Summary: CUDA Computing

copy: host → device  
input

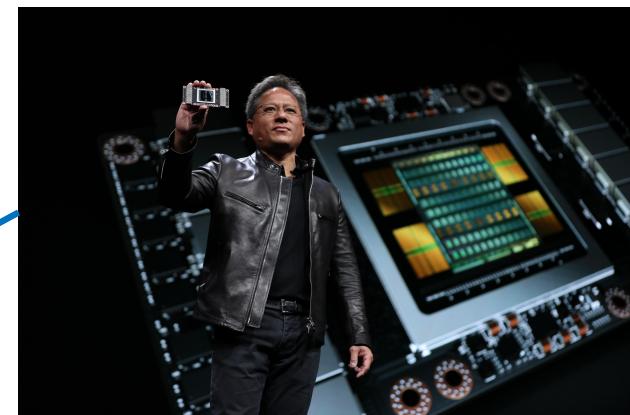
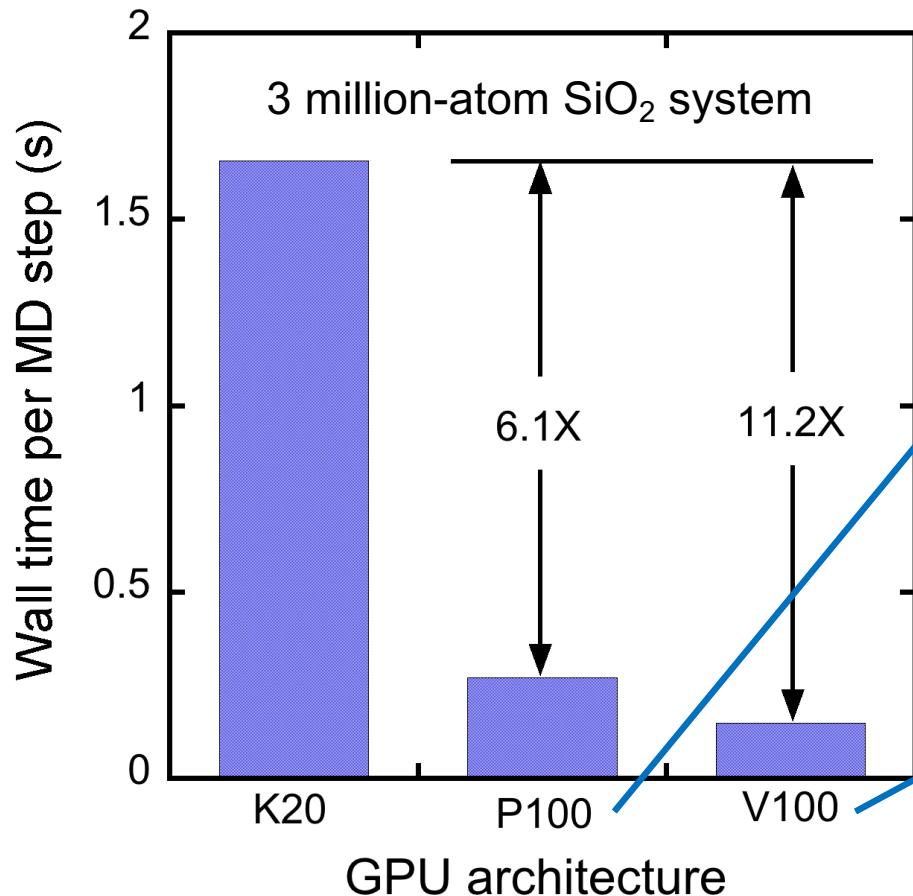
Multithreading  
(SPMD<sup>\*</sup>):  
big loop

copy: host ← device  
output

- \* Single program multiple data we have learned is called **single instruction multiple threads (SIMT)** in GPU terminology

# New Generations of GPUs

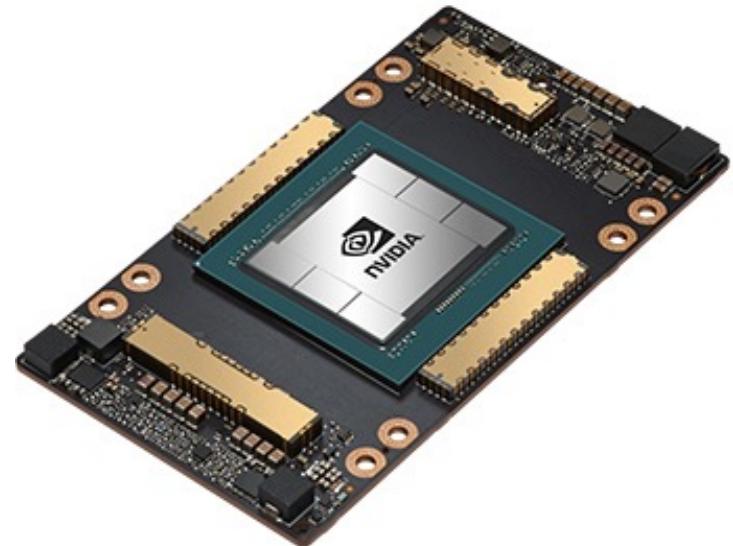
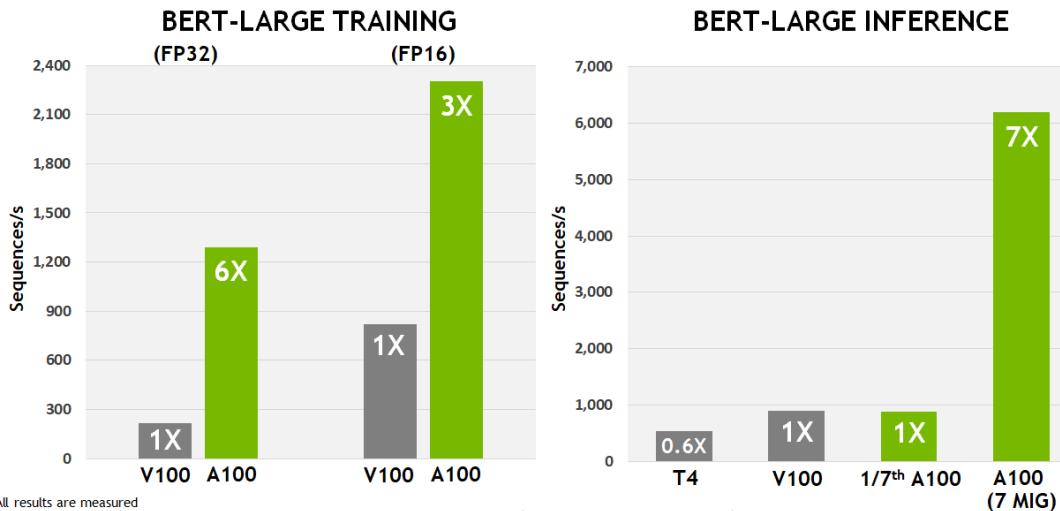
- Running time per molecular dynamics (MD) step on Kepler (K20), Pascal (P100) & Volta (V100) GPUs



# New Generations of GPUs (2)

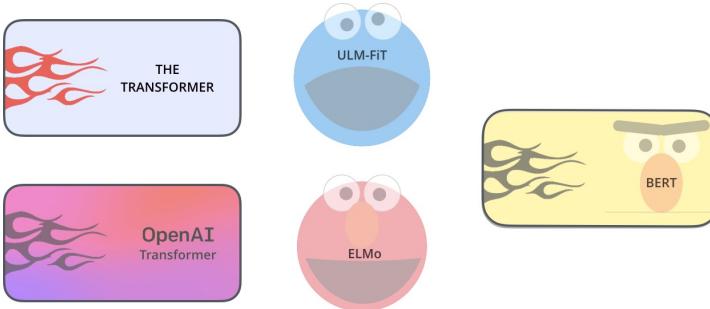
- A100 is at CARC

## UNIFIED AI ACCELERATION

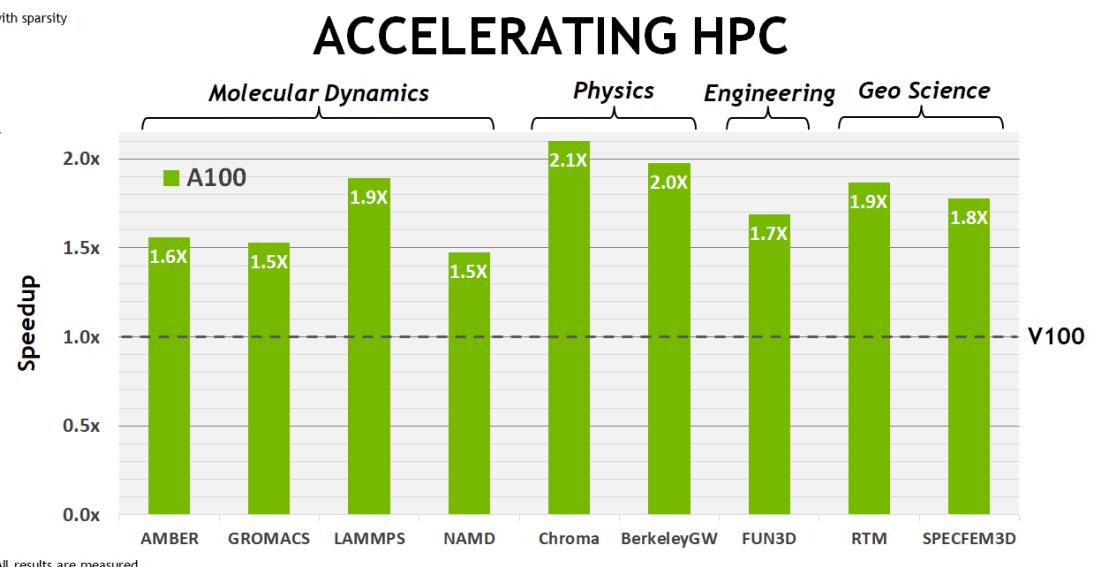


All results are measured  
BERT Large-Training (FP32 & FP16) measures Pre-Training phase, uses PyTorch including (2/3) Phase1 with Seq Len 128 and (1/3) Phase 2 with Seq Len 512,  
V100 is DGX1 Server with 8xV100, A100 is DGX A100 Server with 8xA100, A100 uses TF32 Tensor Core for FP32 training.  
BERT Large-Inference uses TRT 7.1 for T4/V100, with INT8/FP16 at batch size 256. Pre-production TRT for A100, uses batch size 94 and INT8 with sparsity

**BERT: Bidirectional Encoder Representations from Transformers used in natural language processing (NLP)**



cf. Pytorch GPU engine

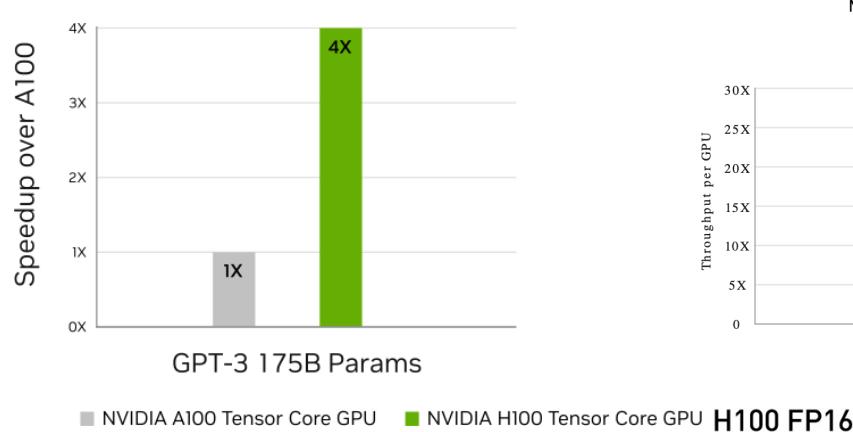


All results are measured  
Except BerkeleyGW, V100 used is single V100 5XM2, A100 used is single A100 5XM4  
More apps detail: AMBER based on PME-Cellulose, GROMACS with STMV (h-bond), LAMMPS with Atomic Fluid LJ-2.5, NAMD with v3.0a1 STMV\_NVE  
Chroma with sszcd1\_24\_128, FUN3D with dpw, RTM with isotropic Radius 4 1024^3, SPECFEM3D with Cartesian four material model  
BerkeleyGW based on Chi Sum and uses 8xV100 in DGX-1, vs 8xA100 in DGX A100

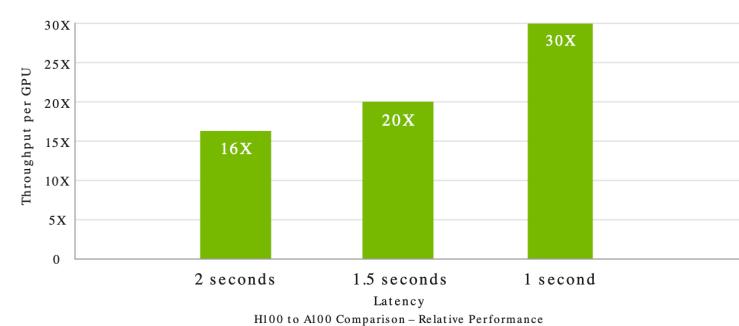
# New Generations of GPUs (3)

- H100 is faster: 14,592 CUDA cores & 456 tensor cores

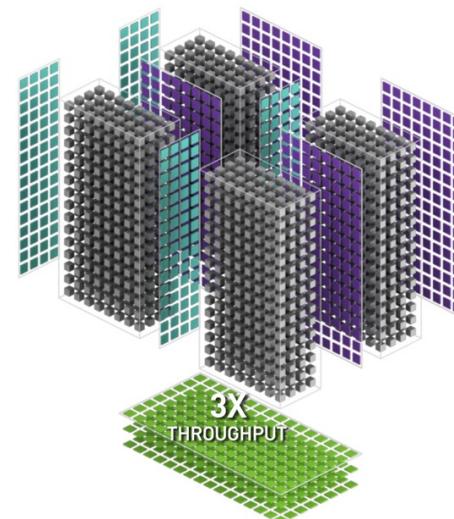
Up to 30X higher AI inference performance on the largest models



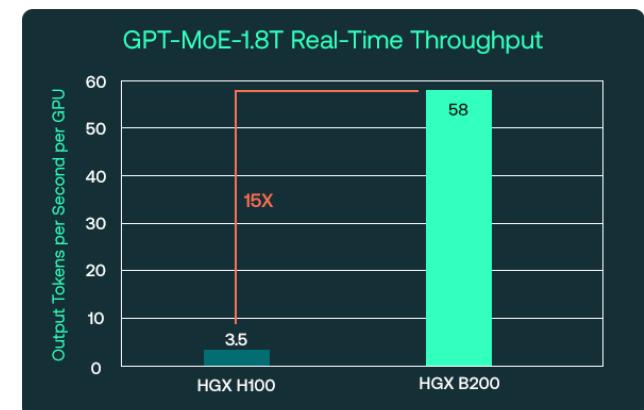
Megatron chatbot inference (530 billion parameters)



- Unlike general-purpose CUDA cores, tensor cores are specialized processing units designed for (mixed-precision) matrix operations in deep learning



- Now B200:  
16,896 CUDA cores & 528 tensor cores



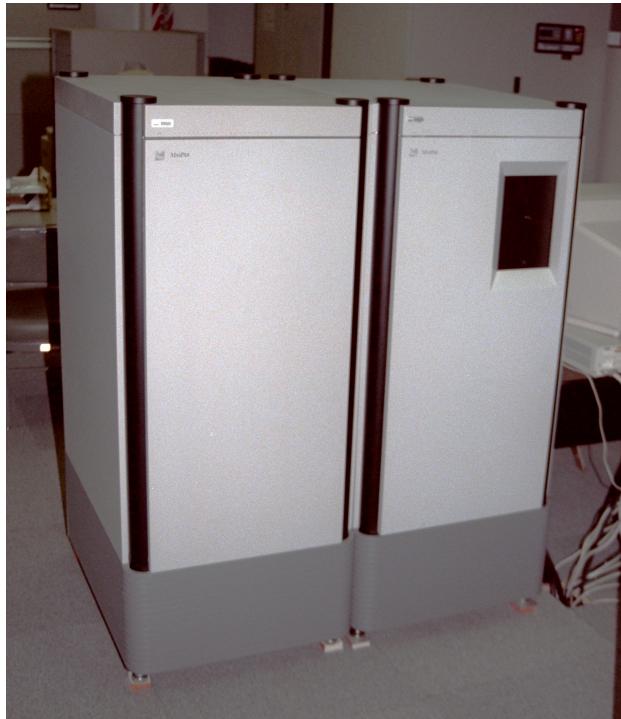
# Warp & Control Divergence

---

- Threads in a block are subdivided into warps (e.g. consisting of 32 threads)
- Warps are executed in SIMD (single-instruction multiple-data) fashion, *i.e.*, multiple threads concurrently perform the same operation
- CUDA provides warp-level primitives for efficient warp-level programming
- Single instruction multiple thread (SIMT) execution model penalizes control divergence, where different threads execute different instructions
- Warp voting: All threads (e.g. particles) within a warp vote on which computation to perform, with an overhead of unnecessary computations, for example:

if (any thread in a warp wants to compute) all threads do

# Massive SIMD Data-Parallel Accelerator



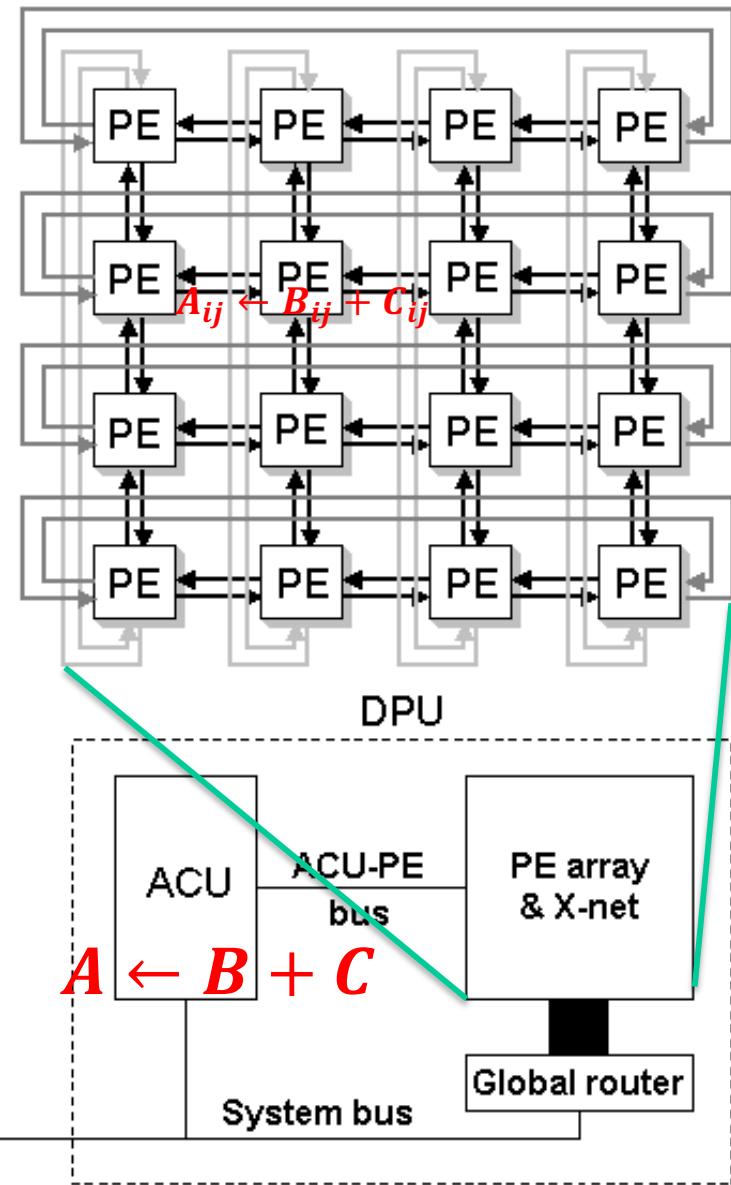
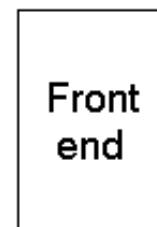
8,192-  
processor  
MasPar



6,912-  
core  
A100

**SIMD: single-instruction multiple data**  
**Quantum dynamics on 8,192-processor**  
**(128 × 64) MasPar 1208B**

Nakano,  
*Comput. Phys. Commun.*  
83, 181 ('94)



See lecture on [pre-Beowulf parallel computing](#)

# Final Projects on GPU

---

---

- L. Peng *et al.*, “Parallel lattice Boltzmann flow simulation on emerging multi-core platforms,” [Proc. Euro-Par, 763 \('08\)](#)
- P. E. Small *et al.*, “Acceleration of dynamic  $n$ -tuple computations in many-body molecular dynamics,”  
[Proc. IEEE HPC Asia \('18\)](#)
- Sasan Tavakkol’s final project became a [poster](#) in [GPU Technology Conference](#) (see nice videos [1](#) & [2](#))
- C. Rizzo *et al.*, “PAR2: parallel random walk particle tracking method for solute transport in porous media,”  
[Comput. Phys. Commun. 239, 265 \('19\)](#)

# Final Project on GPU-MD?

- J. C. Phillips *et al.*, “Quantum-based molecular dynamics simulations using tensor cores,” *J. Chem. Phys* **153**, 044130 ('20)

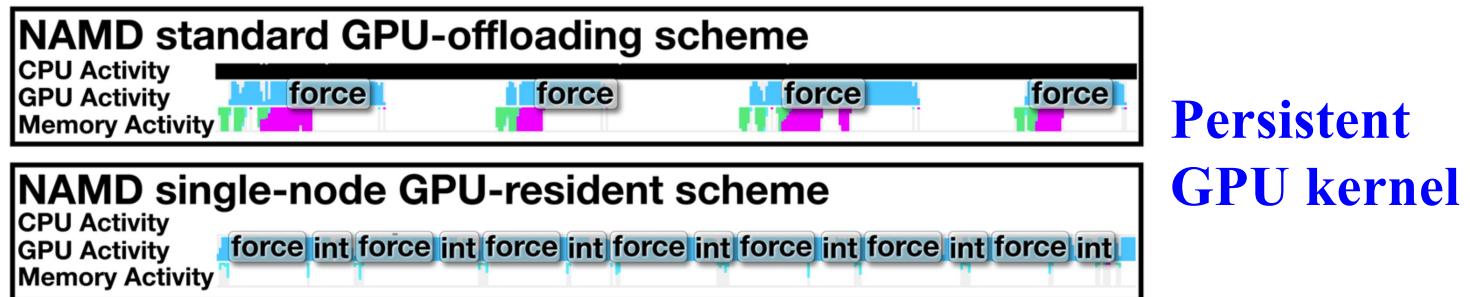
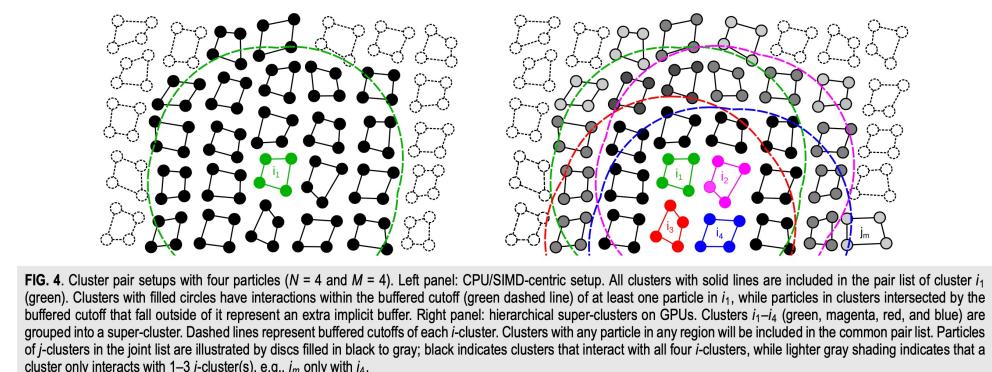
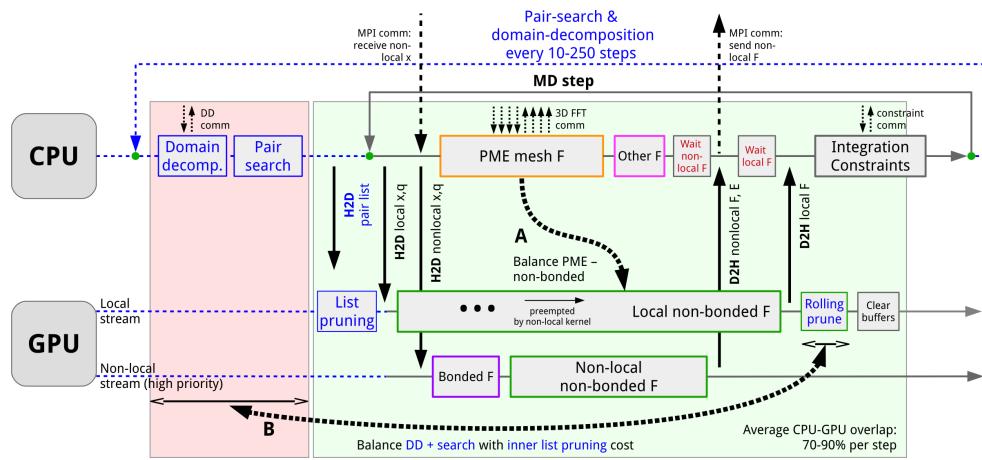


FIG. 5. Standard GPU offload approach compared against new GPU-resident execution scheme for a single-node NAMD simulation of apolipoprotein 1 (ApoA1) in water, consisting of 92 224 atoms. The light blue line tracks GPU activity, while the black strip tracks CPU activity. GPU force calculations are labeled “force,” and GPU integration calculations are labeled “int.”

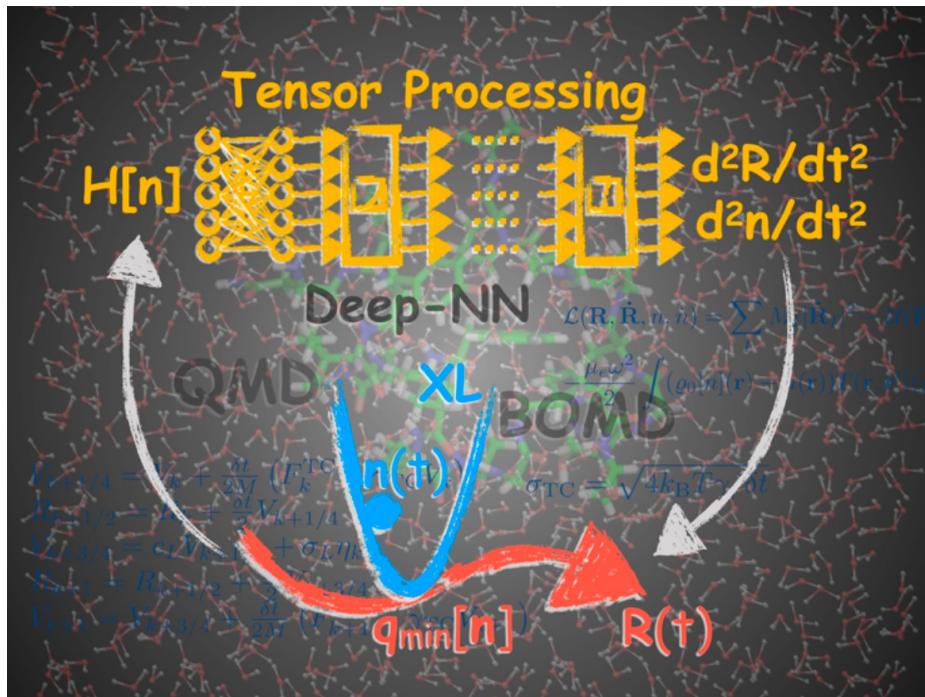
- S. Pall *et al.*, “Heterogeneous parallelization and acceleration of molecular dynamics simulations in GROMACS,” *J. Chem. Phys.* **153**, 134110 ('20)



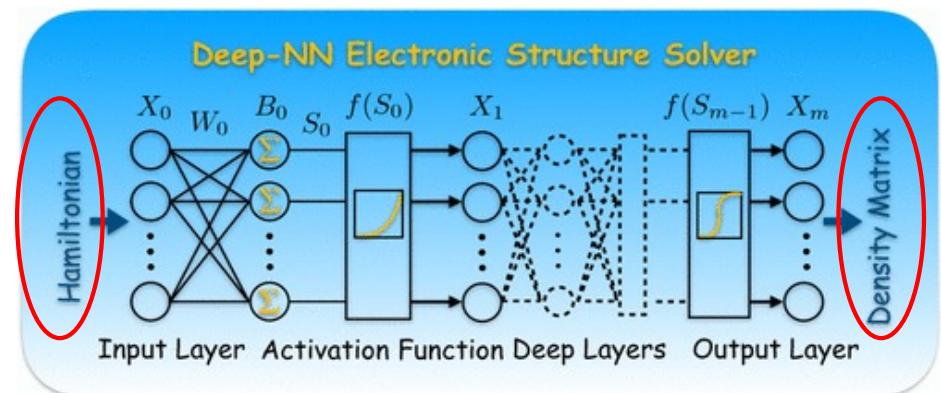
Thread blocking

# Final Project on GPU-MD?

- J. Finkelstein *et al.*, “Quantum-based molecular dynamics simulations using tensor cores,” *J. Chem. Theo. Comput.* doi: 10.1021/acs.jctc.1c00726 ('21); Python code for an associated paper is available at [https://pubs.acs.org/doi/suppl/10.1021/acs.jctc.1c00057/suppl\\_file/ct1c00057\\_si\\_001.zip](https://pubs.acs.org/doi/suppl/10.1021/acs.jctc.1c00057/suppl_file/ct1c00057_si_001.zip)



“computational structure naturally takes advantage of the exceptional processing power of the tensor cores (utilizing FP16) and allows for high performance in excess of 100 Tflops on a single Nvidia A100 GPU.”



# Precision vs. Flop/s Performance

Tale of three Gordon Bell prize finalists in 2025: Dramatic effect of floating-point (FP) and brain-float (BF) precisions on Flop/s (floating-point operations per second) performance on the Aurora supercomputer

- “Advancing quantum many-body GW calculations on exascale supercomputing platforms,” B. Zhang (USC) et al.

**0.71 Exaflop/s on 9,600 nodes (FP64)**

- “Multiscale light-matter dynamics in quantum materials: from electrons to topological superlattices,” T. Razakh (USC) et al.

**1.87 Exaflop/s on 10,000 nodes (hybrid FP64/FP32/BF16)**

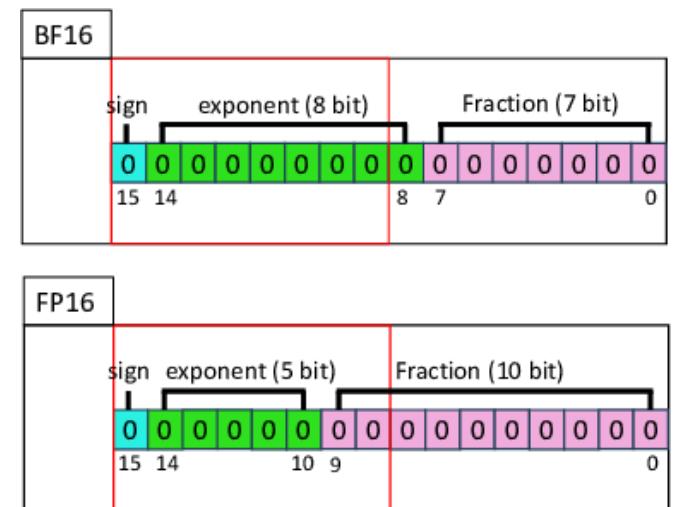
- “AERIS: Argonne earth systems model for reliable and skillful predictions,” V. Hatanpaa (Argonne) et al.

**10.2 Exaflop/s on 10,080 nodes (FP32/BF16)**

Hybrid-precision accumulation/  
matrix-multiplication

$$\overbrace{\mathbf{M} += \sum_i}^{\text{FP32}} \overbrace{\mathbf{L}_i \times \mathbf{R}_i}^{\text{BF16}}$$

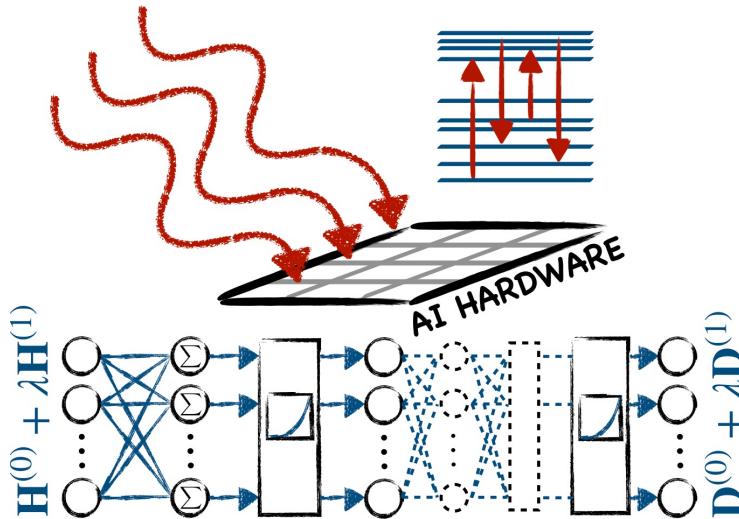
BF16 provides a wider  
dynamic range with  
lower precision



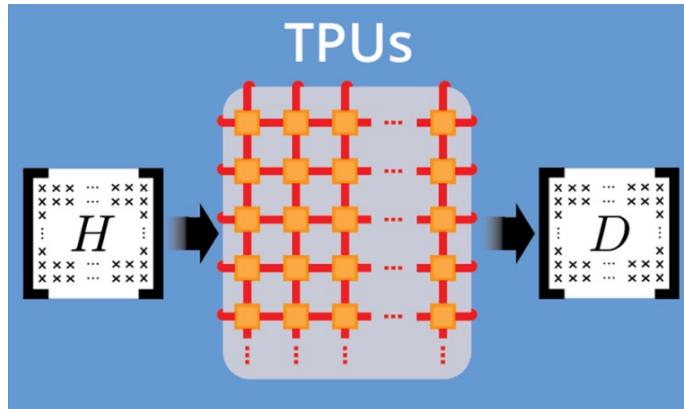
# Scientific Tensor Computing?

NVIDIA tensor cores to Google tensor processing unit (TPU) & beyond

- Joshua Finkelstein *et al.*, “Quantum perturbation theory using tensor cores and a deep neural network,” *J. Chem. Theo. Comput.* **18**, 4255 ('22)



- Ryan Pederson *et al.*, “Large scale quantum chemistry with tensor processing units,” *J. Chem. Theo. Comput.* **19**, 25 ('23)



Tensor processing unit (TPU) is an AI accelerator developed by Google for neural-network machine learning, using Google's own TensorFlow software

Google Cloud Says TPU-Powered Machine Learning Cluster Delivers 9 Exaflops Aggregate Power

May 12, 2022 by [Doug Black](#)

<https://insidehpc.com>

# Where to Go from Here

---

---

- CUDA is a proprietary language for NVIDIA GPUs
- Several open languages are available
  - > High-level, directive-based languages  
OpenACC: <https://www.openacc.org>
  - > Low-level, comprehensive languages  
OpenMP 4.5 and later: <https://www.openmp.org/specifications>
  - > Low-level, comprehensive languages  
OpenCL: <https://www.khronos.org/opencl>
  - > Low-level, comprehensive languages  
SYCL: <https://software.intel.com/content/www/us/en/develop/tools/oneapi.html>