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

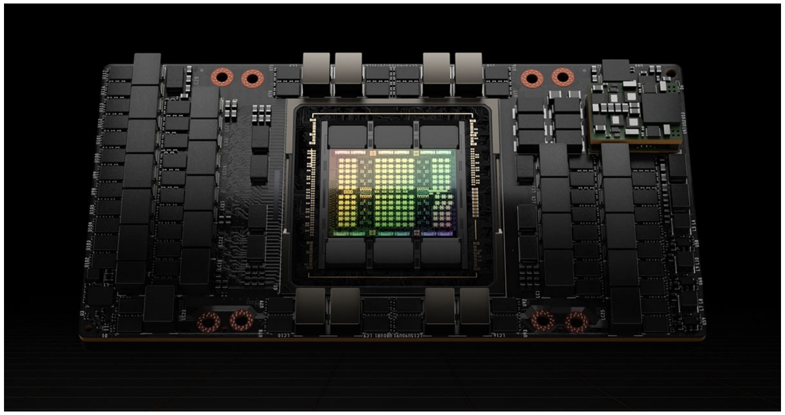
Email: anakano@usc.edu

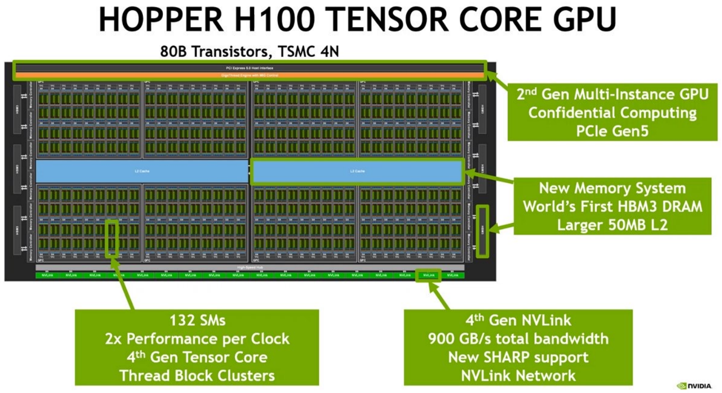
Goal: Multithreading on graphics processing units (GPUs); heterogenous device concept

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 H100 (18,432 CUDA cores & 640 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

- **Widely used in the deep-learning community**

  https://www.deeplearningbook.org/contents/applications.html

The Nobel Prize in Physics 2024 was awarded to John J. Hopfield and Geoffrey E. Hinton "for foundational discoveries and inventions that enable machine learning with artificial neural networks"

The Nobel Prize in Chemistry 2024 was divided, one half awarded to David Baker "for computational protein design", the other half jointly to Demis Hassabis and John M. Jumper "for protein structure prediction"

Nvidia and Competitors

- **CUDA was developed by Nvidia**

Forbes

June 18, 2024

BREAKING

Nvidia Now World's Most Valuable Company—Topping Microsoft And Apple

Jensen Huang

- **World's fastest supercomputers are accelerated by AMD, Intel & Nvidia GPUs** [https://www.top500.org]

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	

Using CUDA on Discovery

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

```
module purge
module load usc/8.3.0
module load cuda
```

- **Compilation**

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

- **Submit a Slurm script**

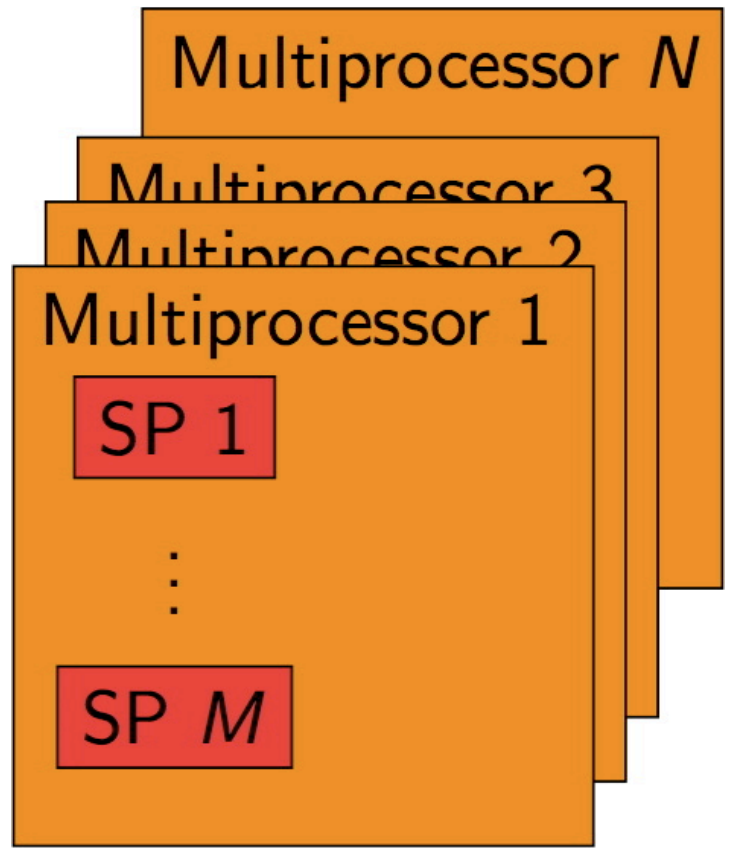
```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --gres=gpu:1
#SBATCH --time=00:00:59
#SBATCH --output=pi.out
#SBATCH -A anakano_429
./pi
```

https://aiichironakano.github.io/cs596/src/cuda/pi.cu

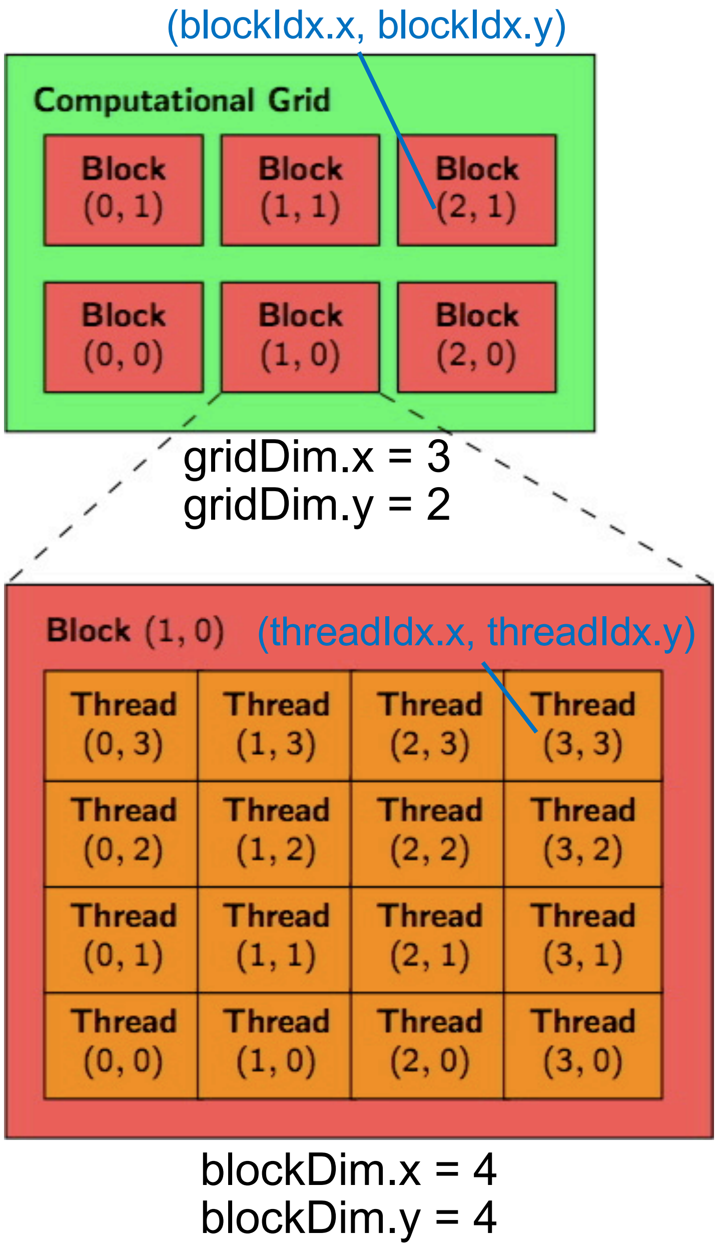
Example of NVIDIA GPU at CARC

- 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 ≤ 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**

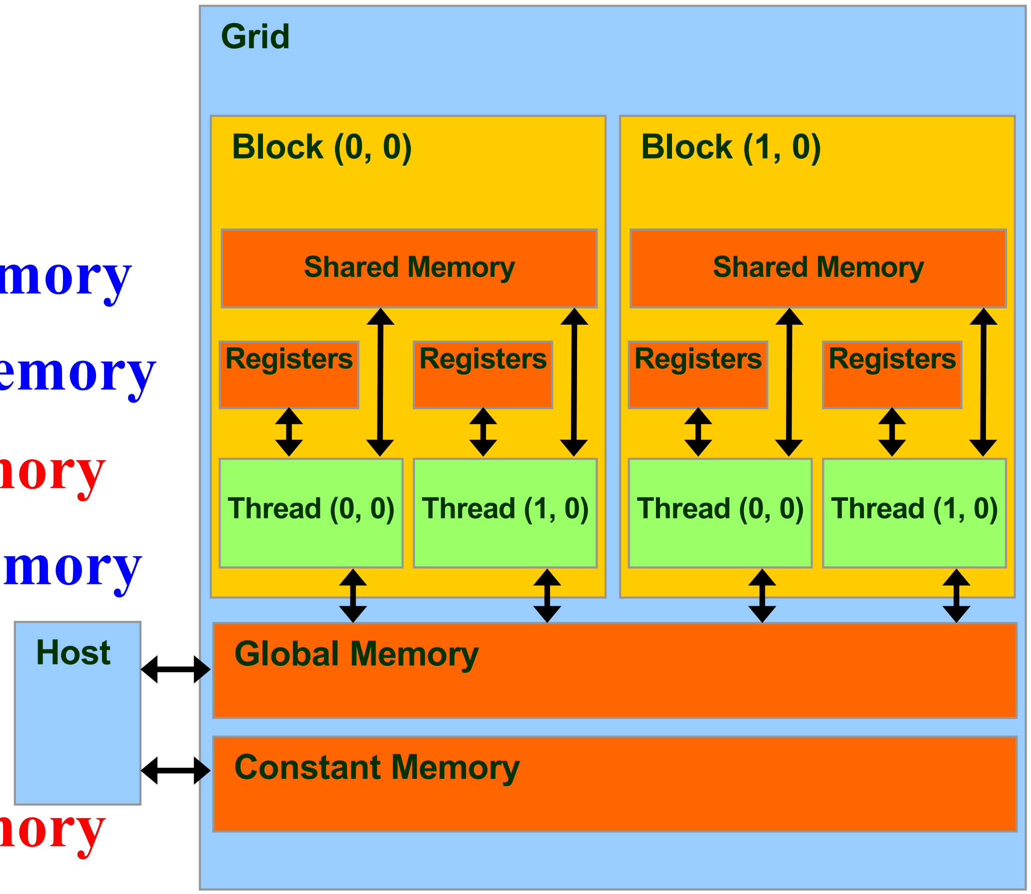
Hierarchical Device Memory

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()

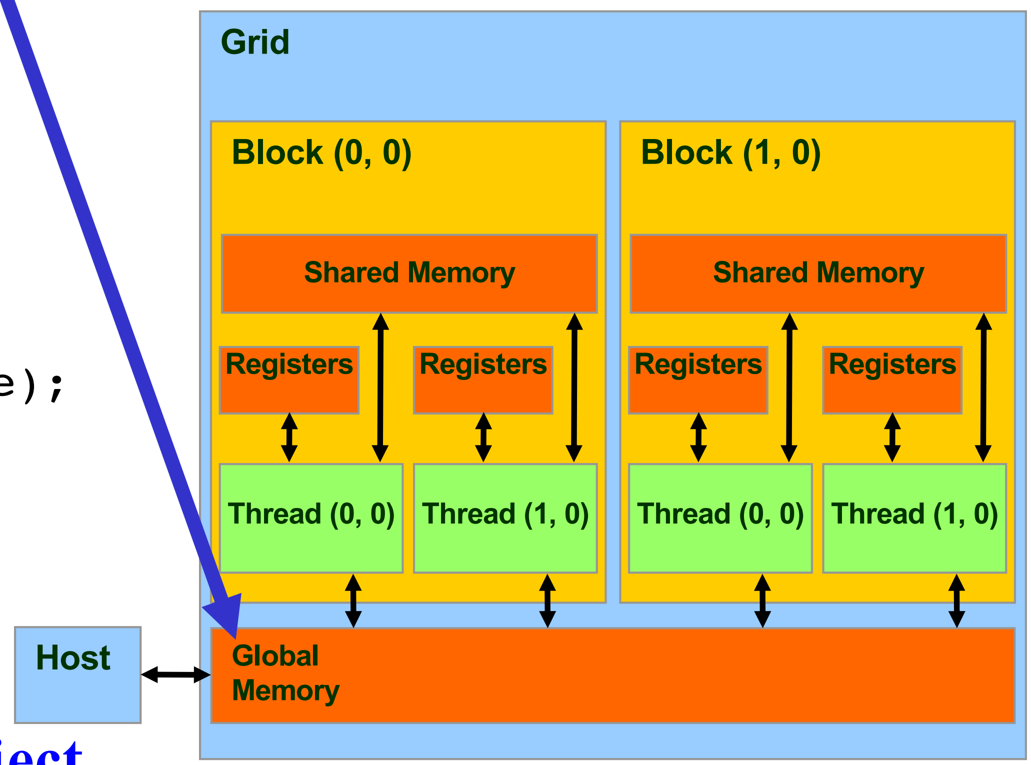
- Allocates object in the device global memory
- Requires two parameters:
  - —Address of a pointer to the allocated object
  - —Size of of allocated object

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

cudaFree()

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

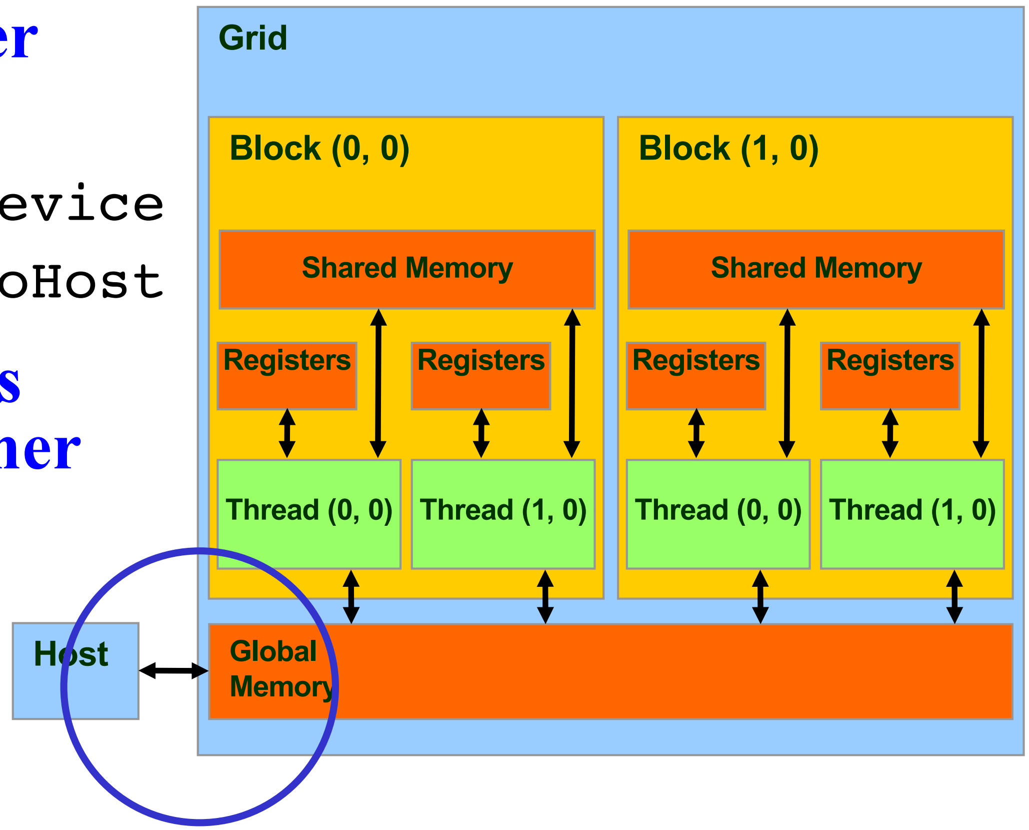
```
cudaFree(sumDev);
```



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

  Two underscores

  `__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

4×2 grid (3rd dimension not used)

2×2×2 block

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

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)

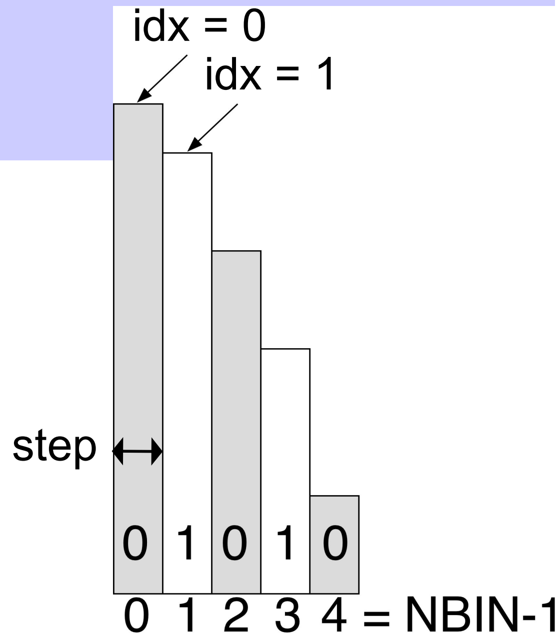
```
// Using CUDA device to calculate pi
#include <stdio.h>
#include <cuda.h>

#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;
```

NBIN NUM_THREAD NUM_BLOCK

```
// 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;
```

Offset: how many threads before this block

```
   int idx = blockIdx.x*blockDim.x+threadIdx.x;  // Sequential thread index across blocks
   for (i=idx; i< nbin; i+=nthreads*nblocks) {   // 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	...

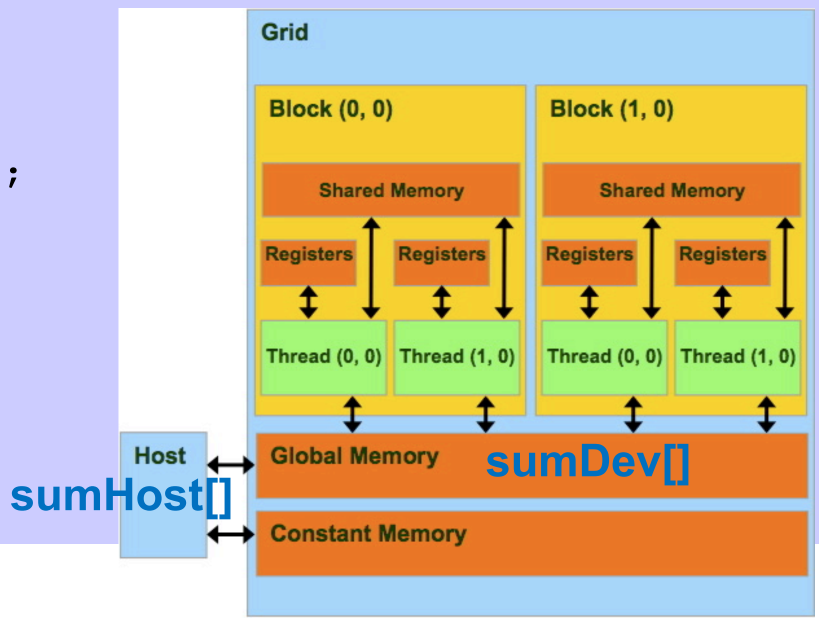
1D grid & block

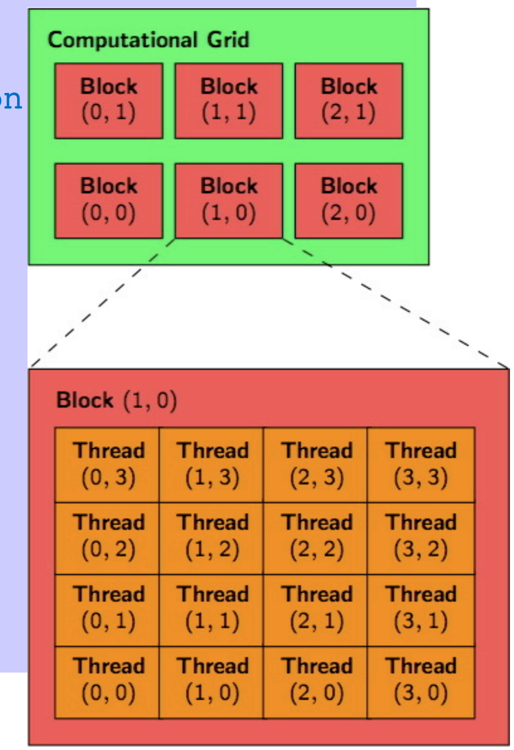
gridDim.x|y = 13|1
blockDim.x|y|z = 192|1|1

Total number of threads = 13×192 = 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

`cudaMemcpy(,,,cudaMemcpyHostToDevice)`

copy: host → **device**
input

Multithreading (SPMD*): big loop

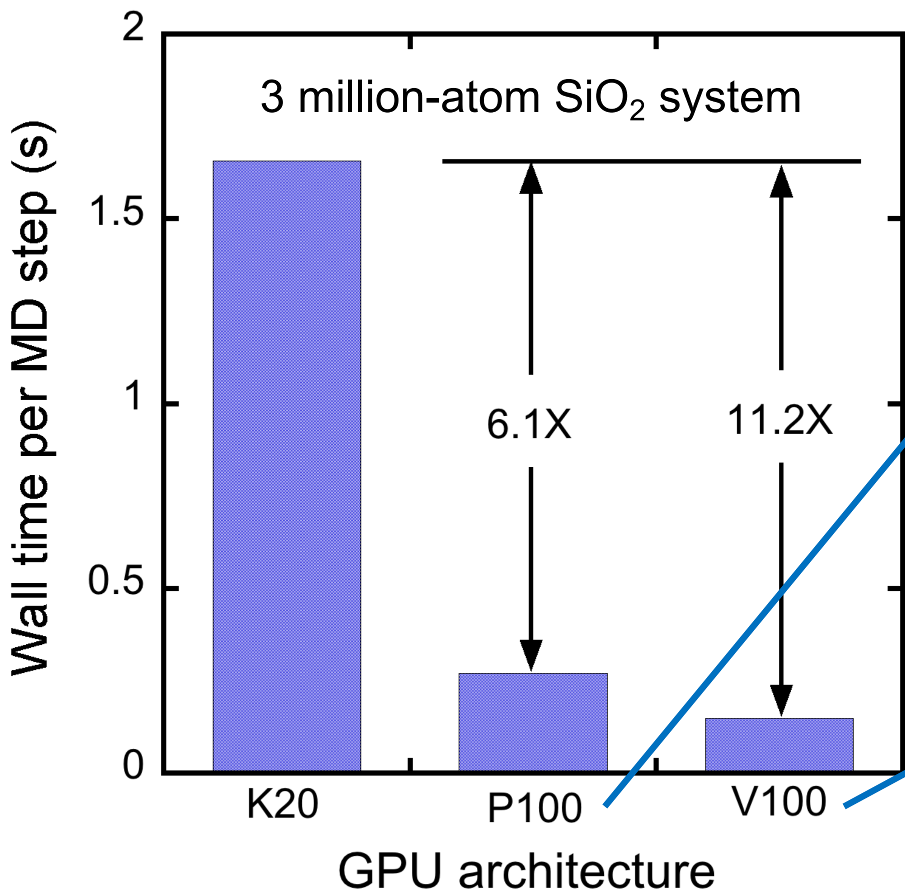
copy: host ← **device**
output

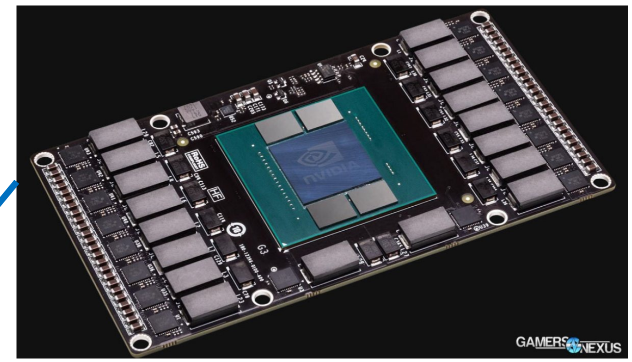
`cudaMemcpy(,,,cudaMemcpyDeviceToHost)`

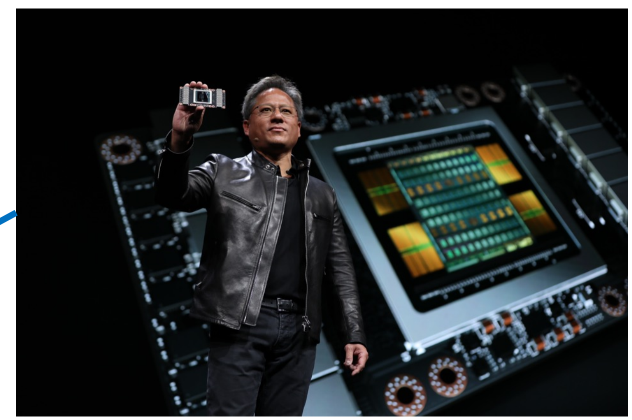
* **Single program multiple data we have learned is called single instruction multiple threads (SIMT) in GPU terminology**

New Generations of GPUs

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

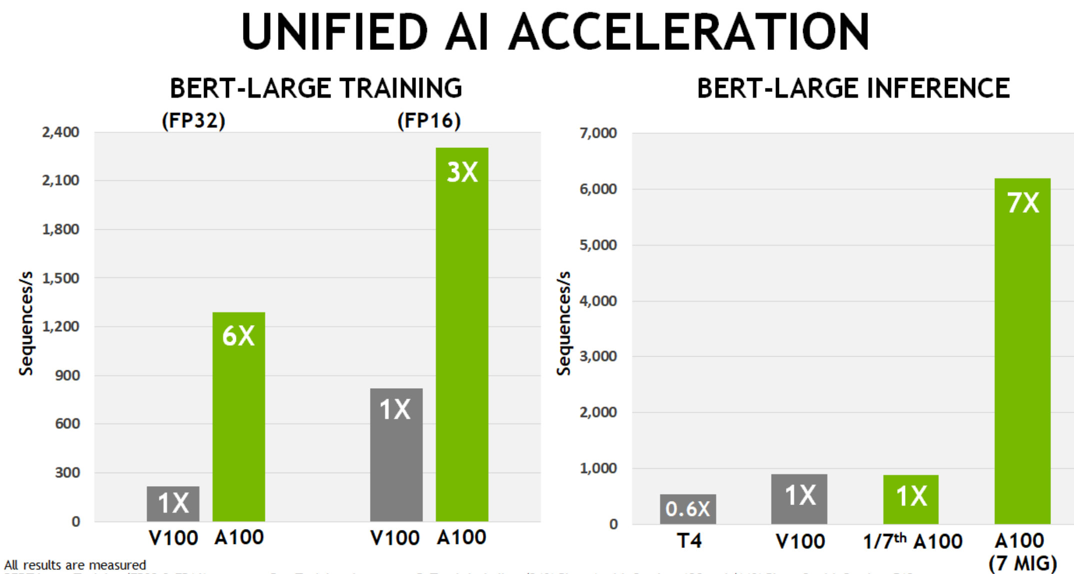






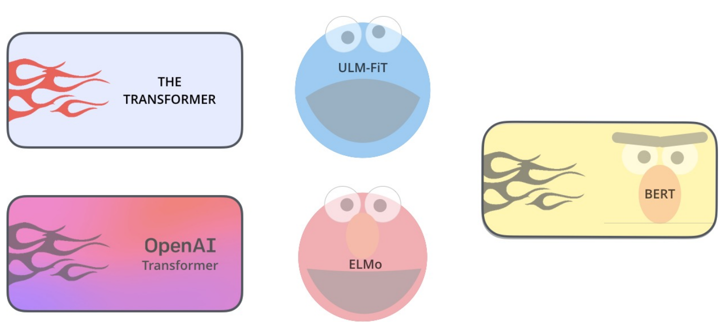
New Generations of GPUs (2)

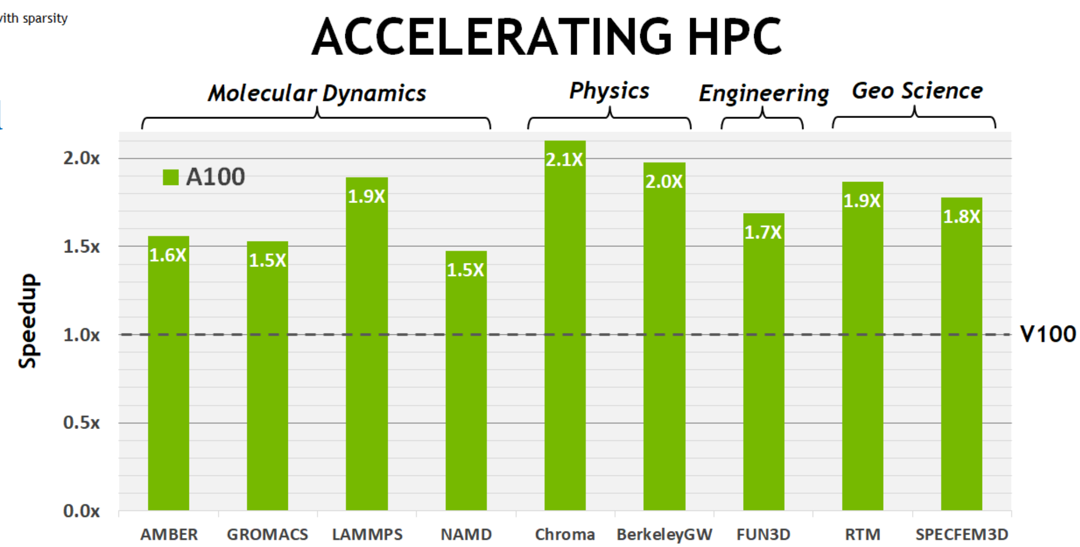
- **Use A100 at CARC**



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)



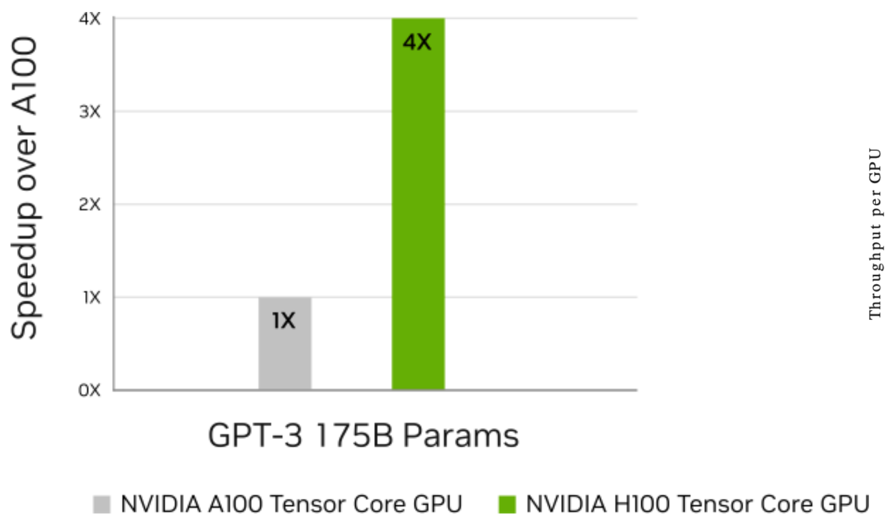


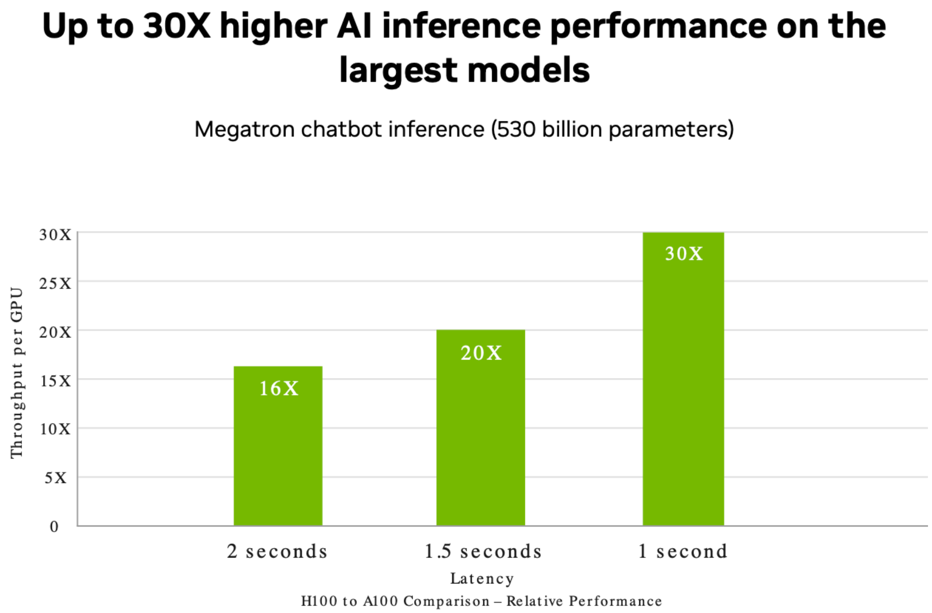
All results are measured
Except BerkeleyGW, V100 used is single V100 SXM2. A100 used is single A100 SXM4
More apps detail: AMBER based on PME-Cellulose, GROMACS with STMV (h-bond), LAMMPS with Atomic Fluid LJ-2.5, NAMD with v3.0a1 STMV_NVE
Chroma with szscl21_24_128, FUN3D with dpw, RTM with Isotropic 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

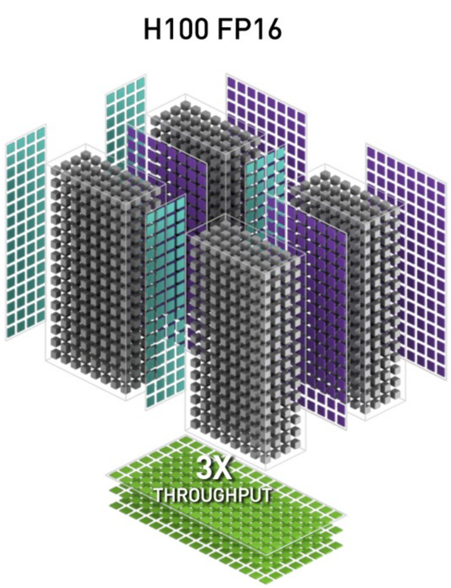
cf. Pytorch GPU engine

New Generations of GPUs (3)

- **H100 is here: 18,432 CUDA cores & 640 tensor cores**







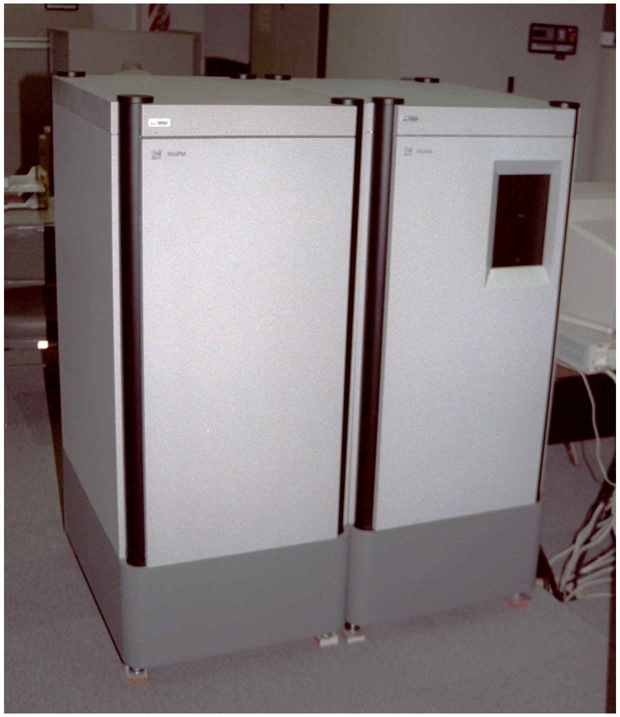
- **Unlike general-purpose CUDA cores, tensor cores are specialized processing units designed for (mixed-precision) matrix operations in deep learning**

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
```

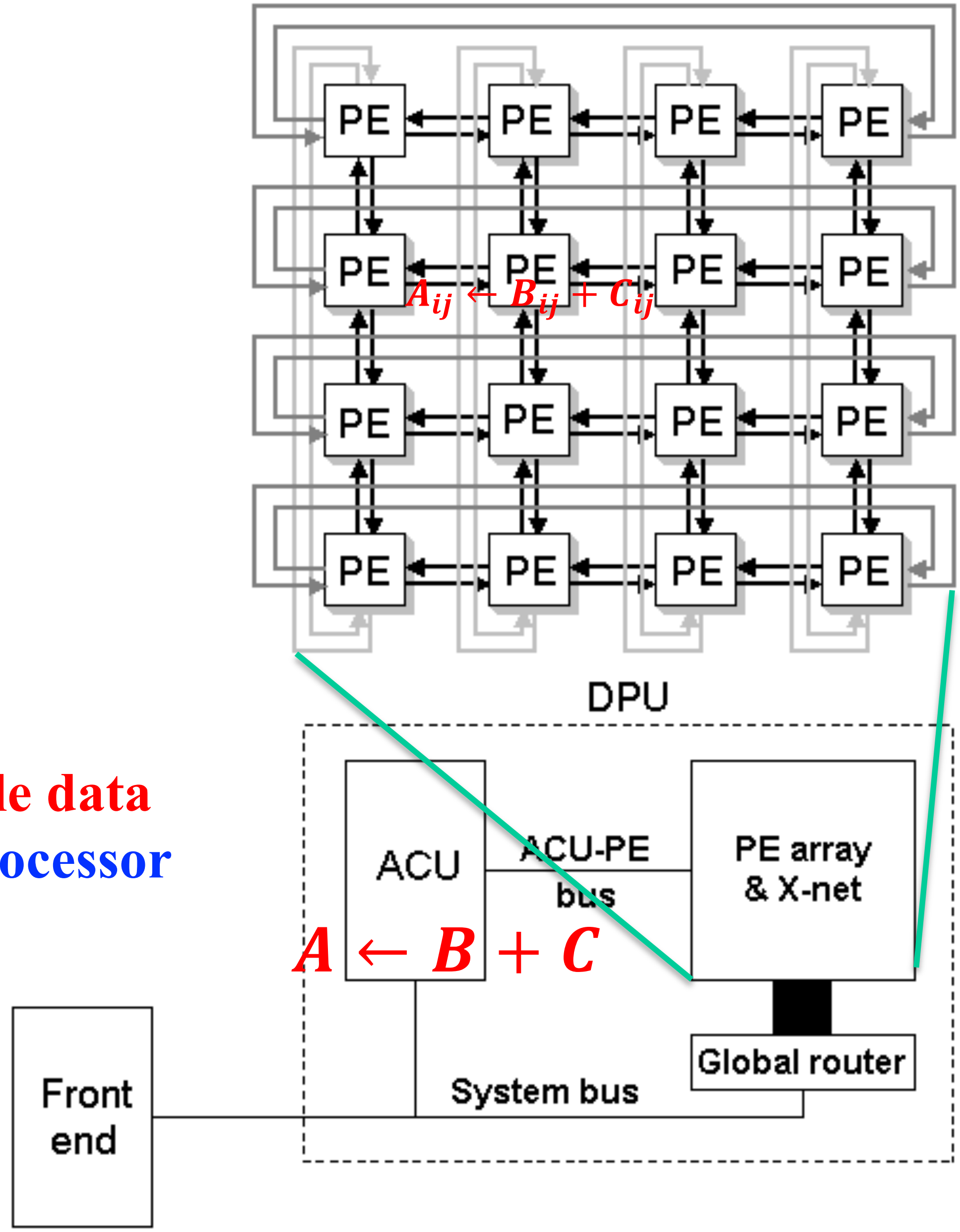
Massive SIMD Data-Parallel Accelerator



SIMD: single-instruction multiple data

Quantum dynamics on 8,192-processor (128 × 64) MasPar 1208B

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



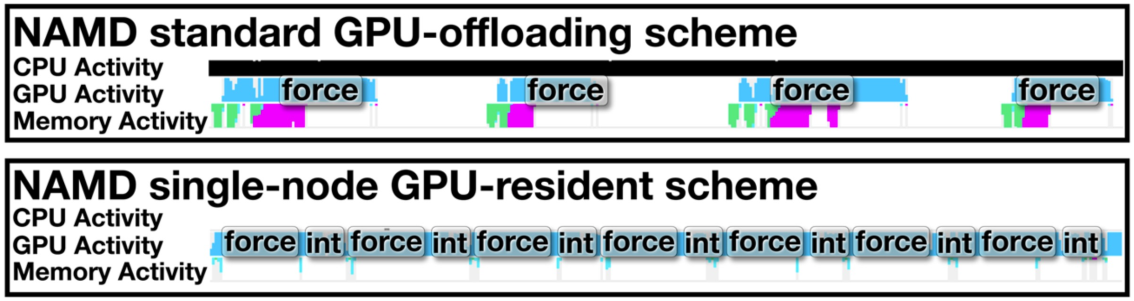
See lecture on pre-Beowulf parallel computing

CSCI 596 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)
- **S. 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?

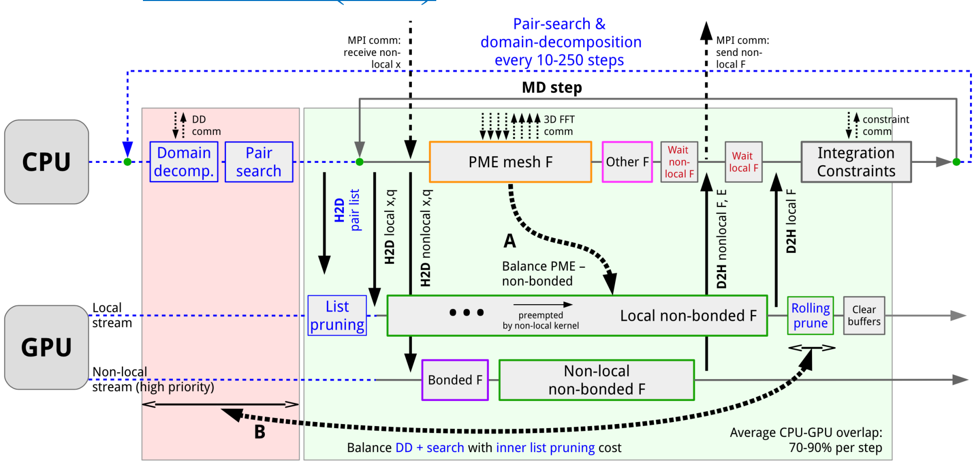
- J. C. Phillips *et al.*, "Quantum-based molecular dynamics simulations using tensor cores," *J. Chem. Phys* **153**, 044130 ('20)



Persistent GPU kernel

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)



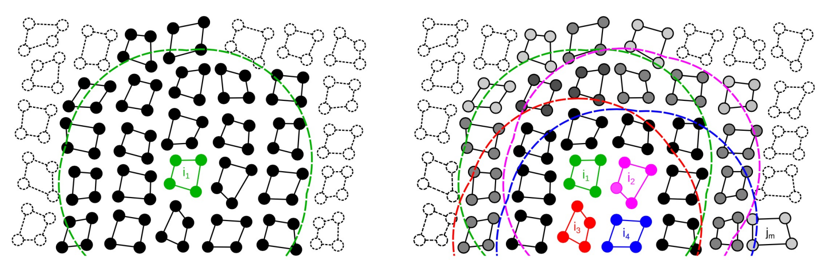


FIG. 4. Cluster pair setups with four particles (N = 4 and M = 4). Left panel: CPU/SIMD-centric setup. All clusters with solid lines are included in the pair list of cluster i_1 (green). Clusters with filled circles have interactions within the buffered cutoff (green dashed line) of at least one particle in i_1, while particles in clusters intersected by the buffered cutoff that fall outside of it represent an extra implicit buffer. Right panel: hierarchical super-clusters on GPUs. Clusters i_1–i_4 (green, magenta, red, and blue) are grouped into a super-cluster. Dashed lines represent buffered cutoffs of each i-cluster. Clusters with any particle in any region will be included in the common pair list. Particles of j-clusters in the joint list are illustrated by discs filled in black to gray; black indicates clusters that interact with all four i-clusters, while lighter gray shading indicates that a cluster only interacts with 1–3 i-cluster(s), e.g., j_m only with i_4.

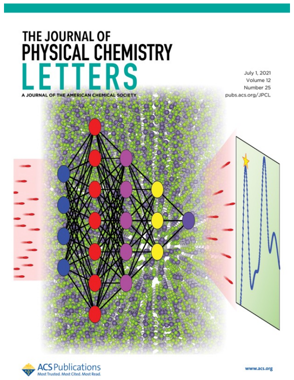
Thread blocking

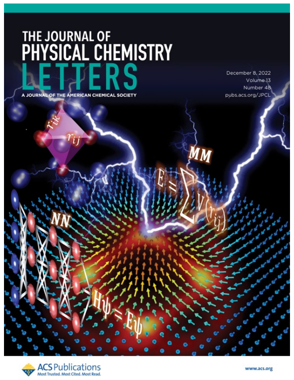
Final Project on GPU-MD? (2)

- **Machine learning (ML) interatomic potentials take full advantage of tensor cores & other ML accelerators**
  - **> W. Jia *et al.*, "Pushing the limit of molecular dynamics with ab initio accuracy to 100 million atoms with machine learning," Gordon Bell prize,** *SC* ('20)
  - **> K. Nguyen-Cong *et al.*, "Billion atom molecular dynamics simulations of carbon at extreme conditions and experimental time and length scales," Gordon Bell finalist,** *SC* ('21)
  - **> A. Musaelian *et al.*, "Scaling the leading accuracy of deep equivariant models to biomolecular simulations of realistic size (Allegro model)," Gordon Bell finalist,** *SC* ('23)
  - **> H. Ibayashi *et al.*, "Allegro-Legato: scalable, fast, and robust neural-network quantum molecular dynamics via sharpness-aware minimization,"** *ISC* ('23)

Allegro-Legato

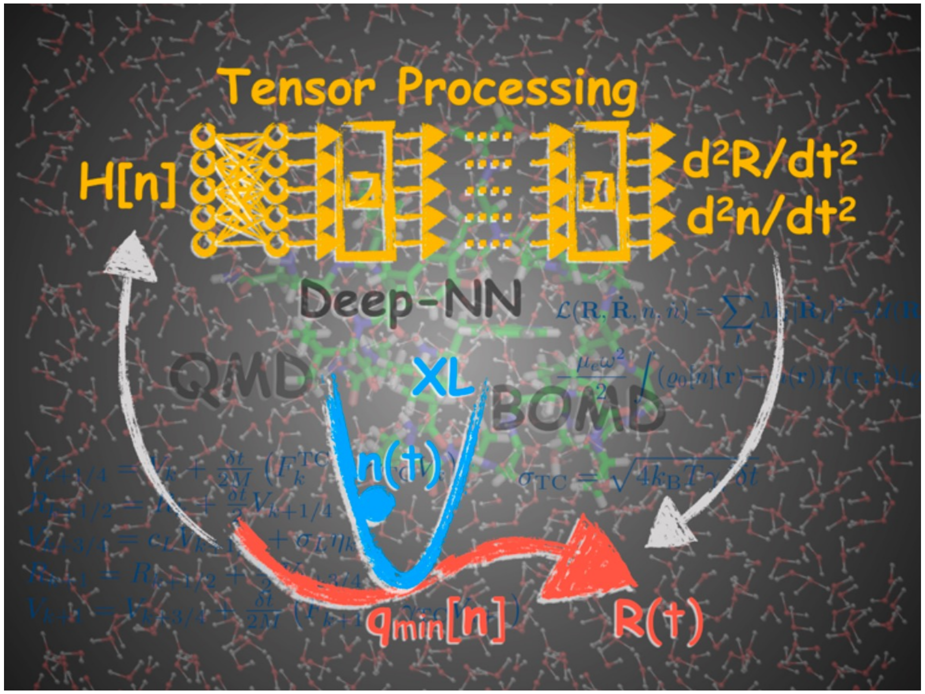
https://github.com/mir-group/allegro





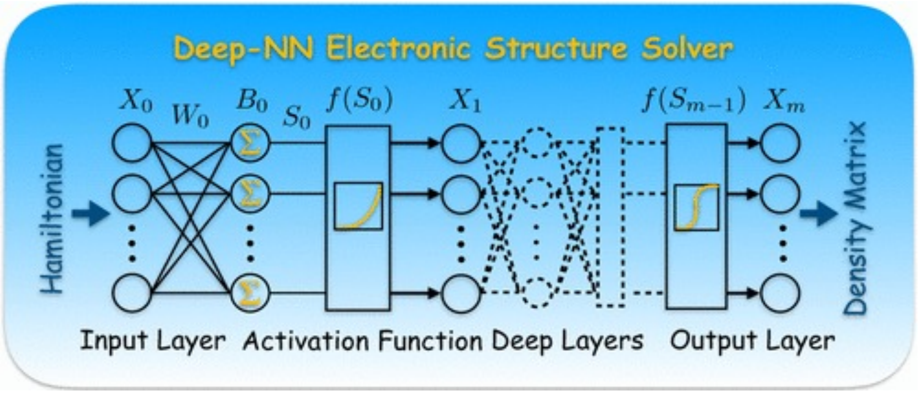
Final Project on GPU-MD? (3)

- **J. Finkelstein *et al.*, "Quantum-based molecular dynamics simulations using tensor cores,"** [*J. Chem. Theo. Comput.* **17**, 6180 ('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



"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."

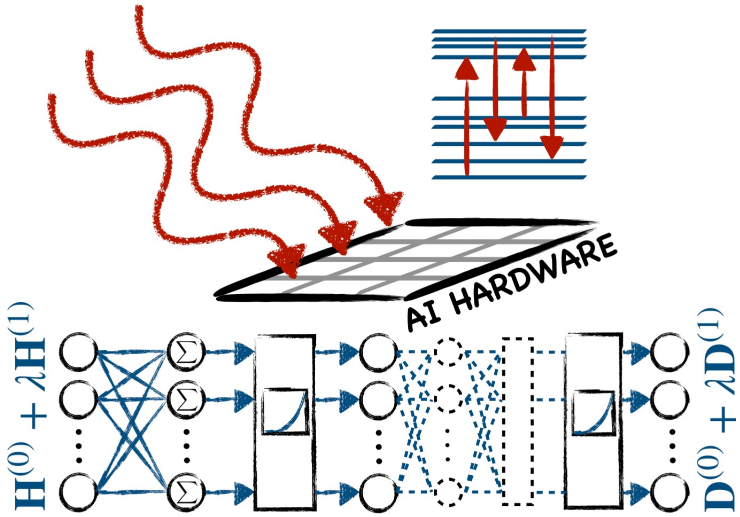
Map scientific computation to mixed-precision tensor processing!



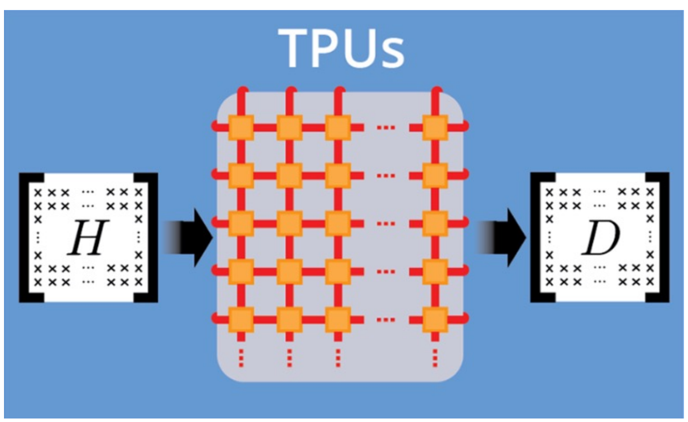
Scientific Tensor Computing?

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)



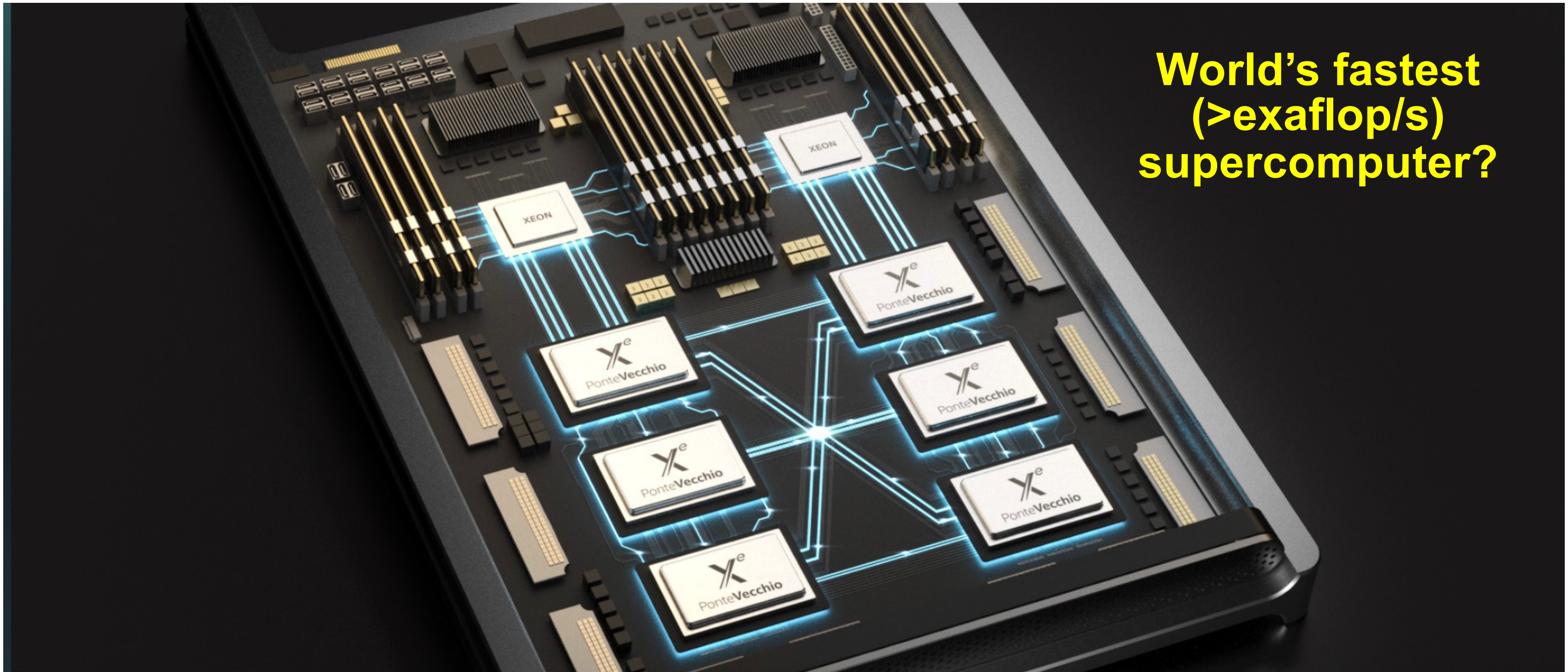
Tensor processing unit (TPU) is an AI accelerator developed by Google for neural-network machine learning, using Google's own TensorFlow software

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

May 12, 2022 by Doug Black

https://insidehpc.com

Aurora: Heterogeneous Future



Aurora's compute nodes will be equipped with two Intel Xeon Scalable processors and six general-purpose GPUs based on Intel's X^e architecture.
Image: Intel Corporation

GPU Architecture

X^e arch-based "Ponte Vecchio"
GPUTile-based, chiplets, HBM
stack, Foveros 3D integration, 7nm

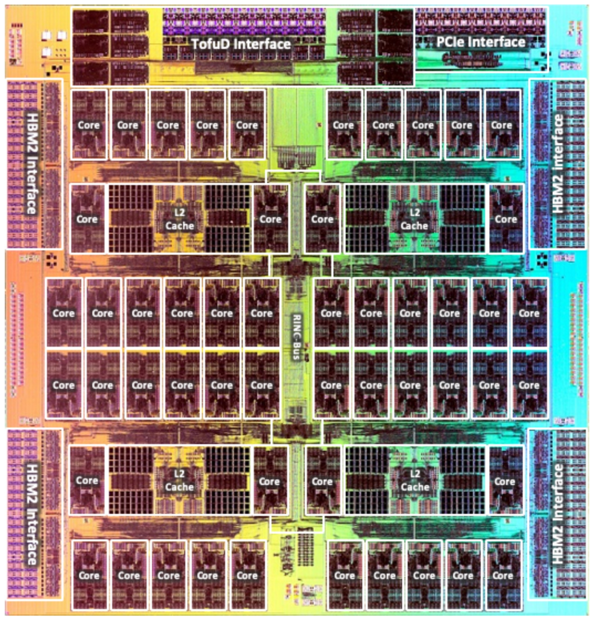
On-Node Interconnect

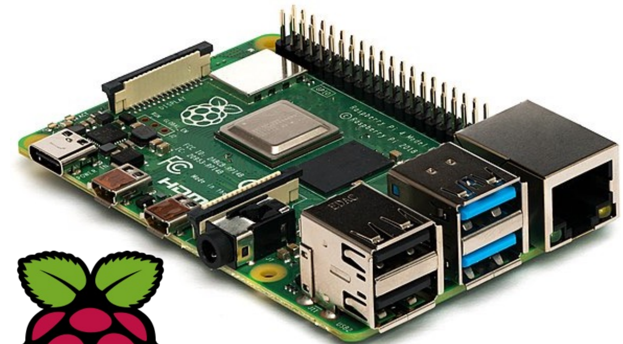
CPU-GPU: PCIe
GPU-GPU: X^e Link

Homogeneous Alternative: ARM

- **ARM:** Advanced RISC (Reduced Instruction Set Computer) Machine
- **Big ARM:** The world's fastest supercomputer in 2021, Fugaku (442 petaflop/s) consists of 7.3 million ARM A64FX (2.2. GHz) cores
- **Little ARM:** Do-it-yourself Raspberry Pi 4 cluster can be built with 1.5 GHz quadcore ARM Cortex-A72 processors

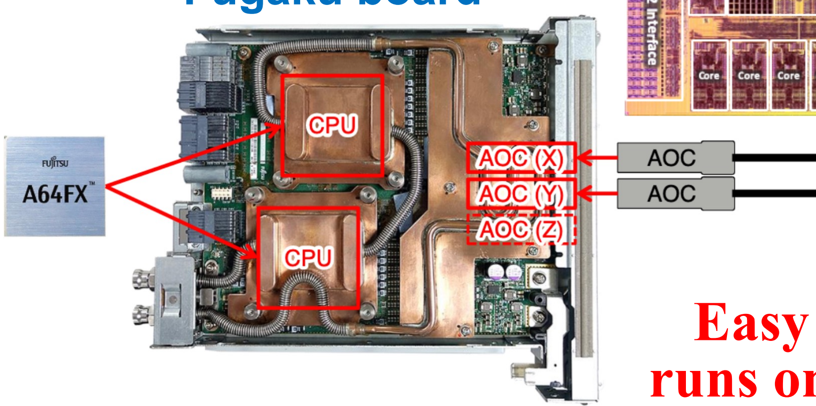
48-core A64FX processor

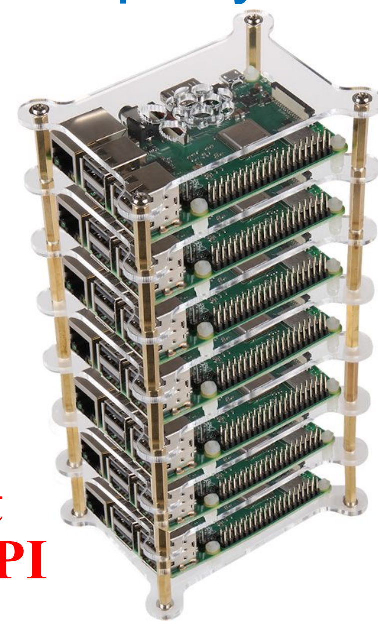




Raspberry Pi 4

Water-cooled 2-socket Fugaku board





Easy to use: Any language that runs on commonplace CPU + MPI

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

  OpenMP 4.5 and later: https://www.openmp.org/specifications

  > Low-level, comprehensive languages

  OpenCL: https://www.khronos.org/opencl

  SYCL: https://software.intel.com/content/www/us/en/develop/tools/oneapi.html