

# SDSC Summer Institute 2024

## GPU Computing and Programming

Instructor: Andreas W. Götz

Date: August 8, 2024, 1:45 AM – 4:30 PM

Location: Main Room Session



# Training overview

## We will cover the following topics

- GPU hardware overview
- Accessing SDSC Expanse GPU nodes
- GPU accelerated software examples
- Programming GPUs – Overview for Nvidia GPUs
- GPU enabled libraries
  - 15 minutes break –
- CUDA C programming basics
- Hands-on exercises on SDSC Expanse – CUDA
- OpenACC introduction
- Hands-on exercises on SDSC Expanse – OpenACC

# Training overview

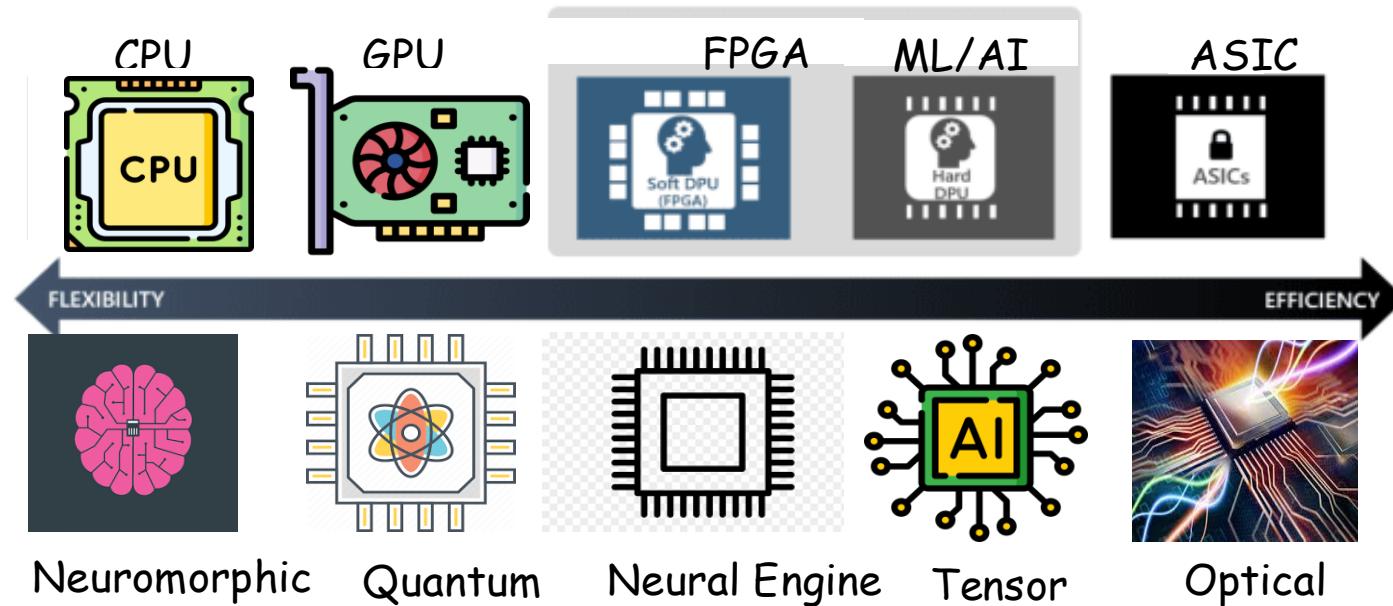
We will cover the following topics

- **GPU hardware overview**
- Accessing SDSC Expanse GPU nodes
- GPU accelerated software examples
- Programming GPUs – Overview for Nvidia GPUs
- GPU enabled libraries
  - 15 minutes break –
- CUDA C programming basics
- Hands-on exercises on SDSC Expanse – CUDA
- OpenACC introduction
- Hands-on exercises on SDSC Expanse – OpenACC

# What is a GPU?

## Accelerator

- Specialized hardware component to speed up some aspect of a computing workload.
- Examples include floating point co-processors in older PCs, specialized chips to perform floating point math in hardware rather than software.
- Many types of accelerators, including GPUs, Field Programmable Gate Arrays (FPGAs), ML/AI processors, Quantum processing units (QPUs)
- Different types of accelerators for different types of applications / compute workloads



# What is a GPU?

## Graphics processing unit

- “Specialist” processor to accelerate the rendering of computer graphics.
- Development driven by \$150 billion gaming industry.
- Originally fixed function pipelines.
- Modern GPUs are programmable for general purpose computations (GPGPU).
- Simplified core design compared to CPU
  - Limited architectural features, e.g. branch caches
  - Partially exposed memory hierarchy



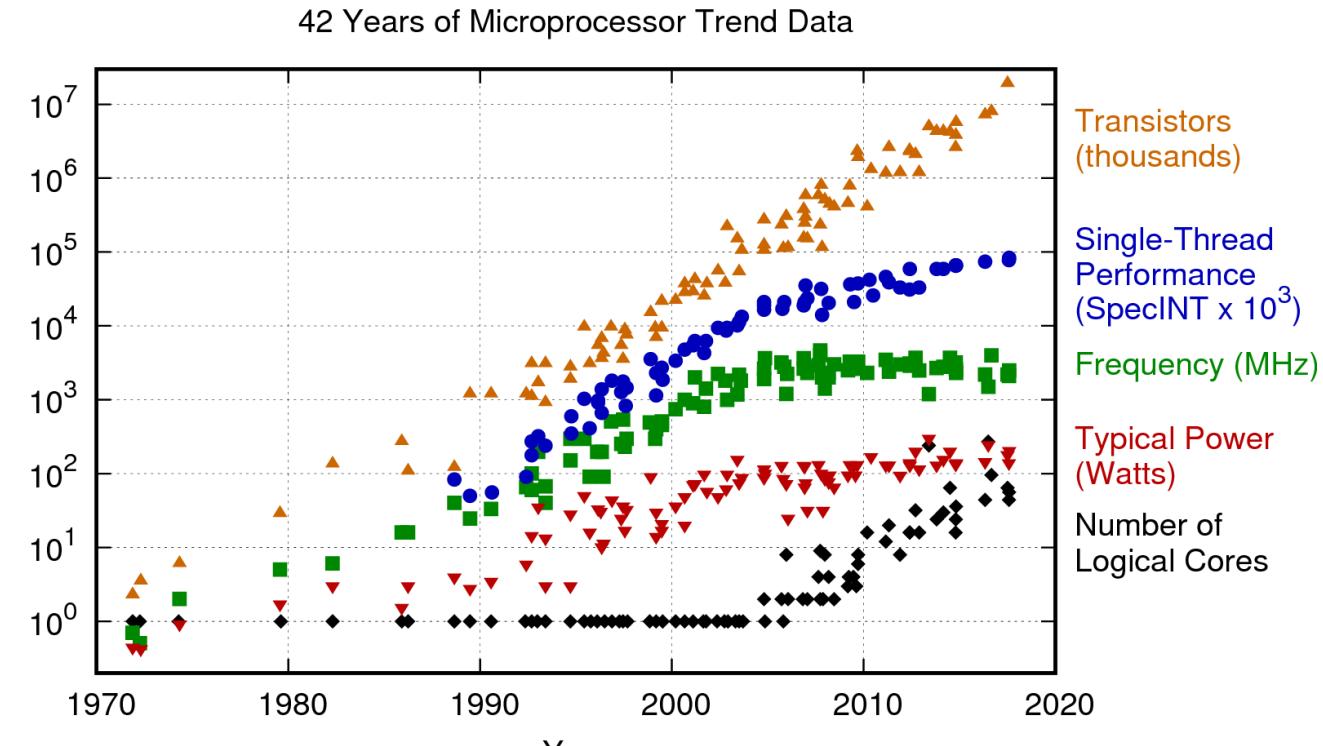
# Why is there such an interest in GPUs?

## Moore's law

- Transistor count in integrated circuits doubles about every two years.
- Exponential growth still holds (see figure).
- However...

## Trends since mid 2000s

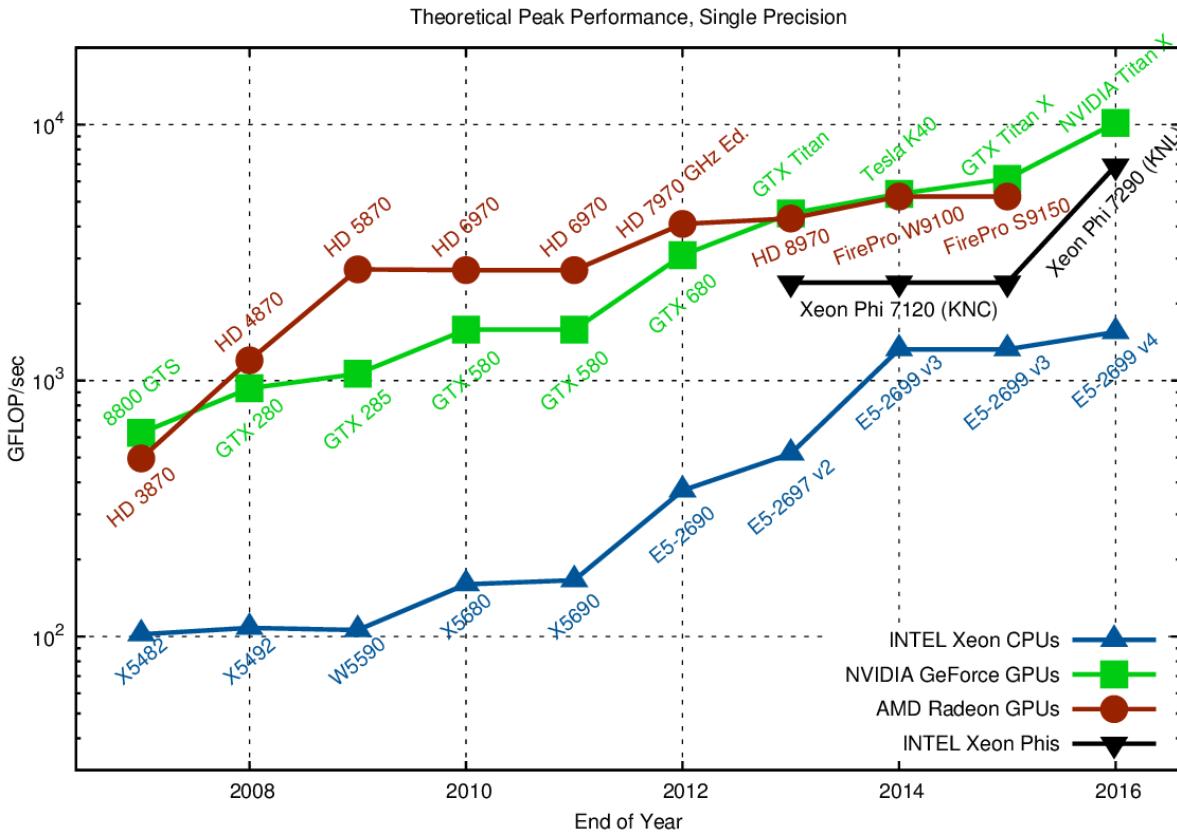
- Clock frequency constant.
- Single CPU core performance (serial execution) roughly constant.
- Performance increase due to increase of CPU cores per processor.
- Cannot simply wait two years to double code execution performance.
- Must write parallel code.



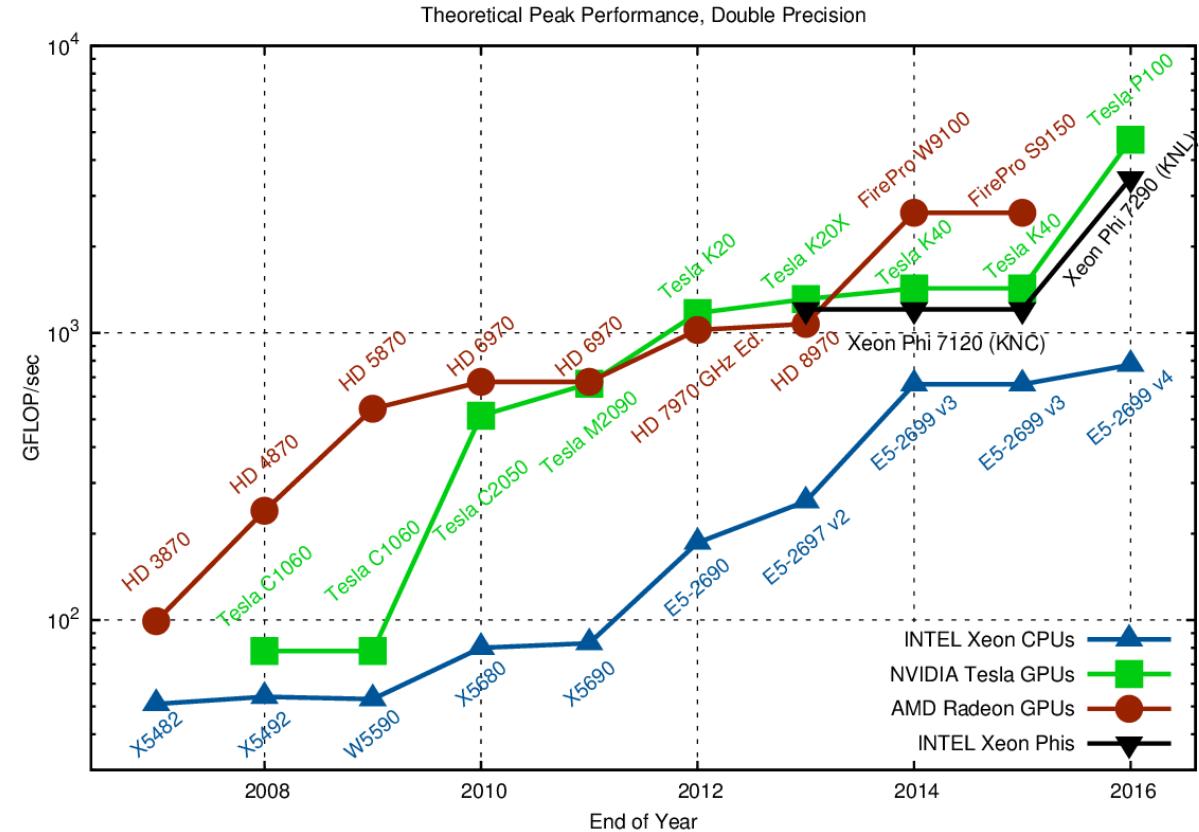
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2017 by K. Rupp

Source:  
<https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>

# Why is there such an interest in GPUs?



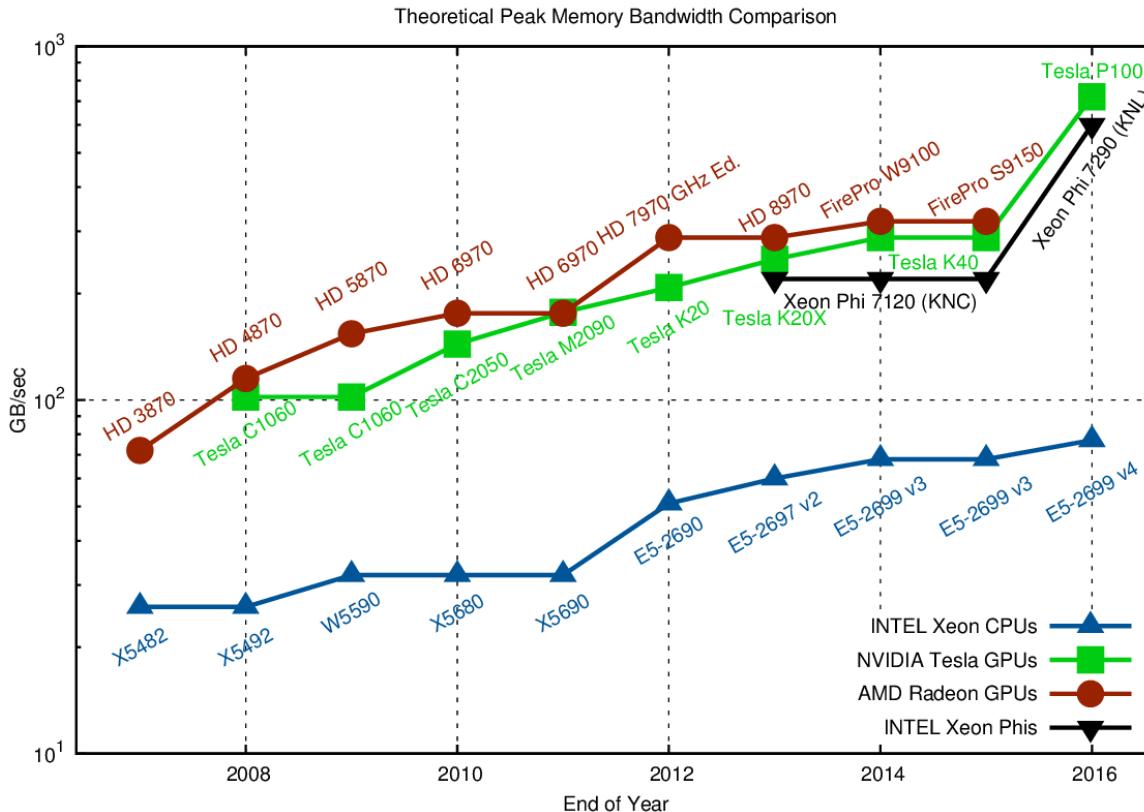
- GPUs offer significantly higher 32-bit floating point performance than CPUs.



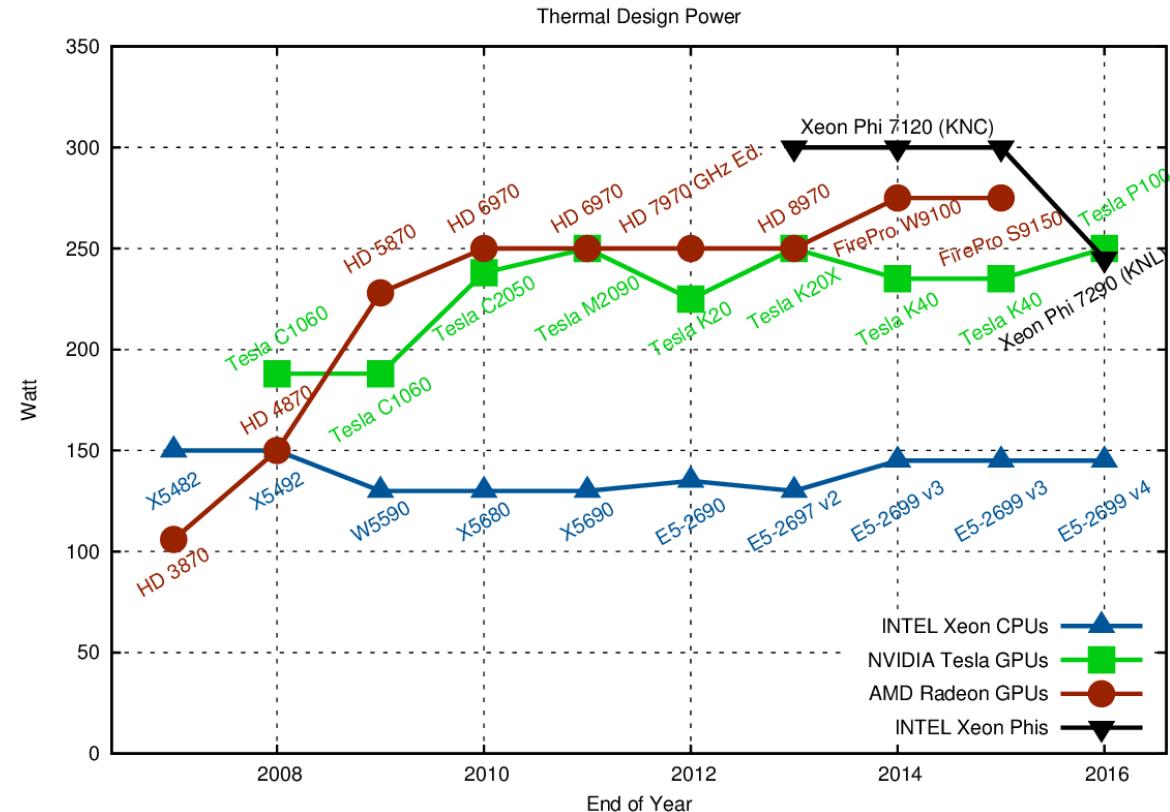
- Datacenter GPUs also offer significantly higher 64-bit floating point performance than CPUs.

Figures source: <https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/>

# Why is there such an interest in GPUs?



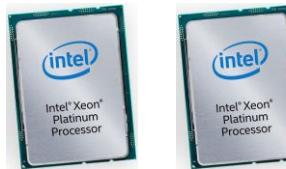
- GPUs have significantly higher memory bandwidth than CPUs.



- Given power consumption, a fair comparison would be a single GPU to 1- to 2-socket CPU server.

Figures source: <https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/>

# Comparison of top (in 2018) X86 CPU vs Nvidia V100 GPU



Aggregate performance numbers (FLOPs, BW)	<b>Dual socket Intel 8180 28-core (56 cores per node)</b>	<b>Nvidia Tesla V100, dual cards in an x86 server</b>
<b>Peak DP FLOPs</b>	4 TFLOPs	14 TFLOPs (3.5x)
<b>Peak SP FLOPs</b>	8 TFLOPs	28 TFLOPs (3.5x)
<b>Peak HP FLOPs</b>	N/A	224 TFLOPs
<b>Peak RAM BW</b>	~ 200 GB/sec	~ 1,800 GB/sec (9x)
<b>Peak PCIe BW</b>	N/A	32 GB/sec
<b>Power / Heat</b>	~ 400 W	2 x 250 W (+ ~ 400 W for server) (~ 2.25x)
<b>Purchase cost</b>	\$20,000 USD	\$20,000 USD
<b>Code portable?</b>	Yes	Yes (OpenACC, OpenCL, OpenMP)

# A supercomputer in a desktop?



## ASCI White (LLNL)

- **12.3 TFLOP/sec** – #1 Top 500, November 2001.
- Cost – \$110 Million USD (in 2001!)

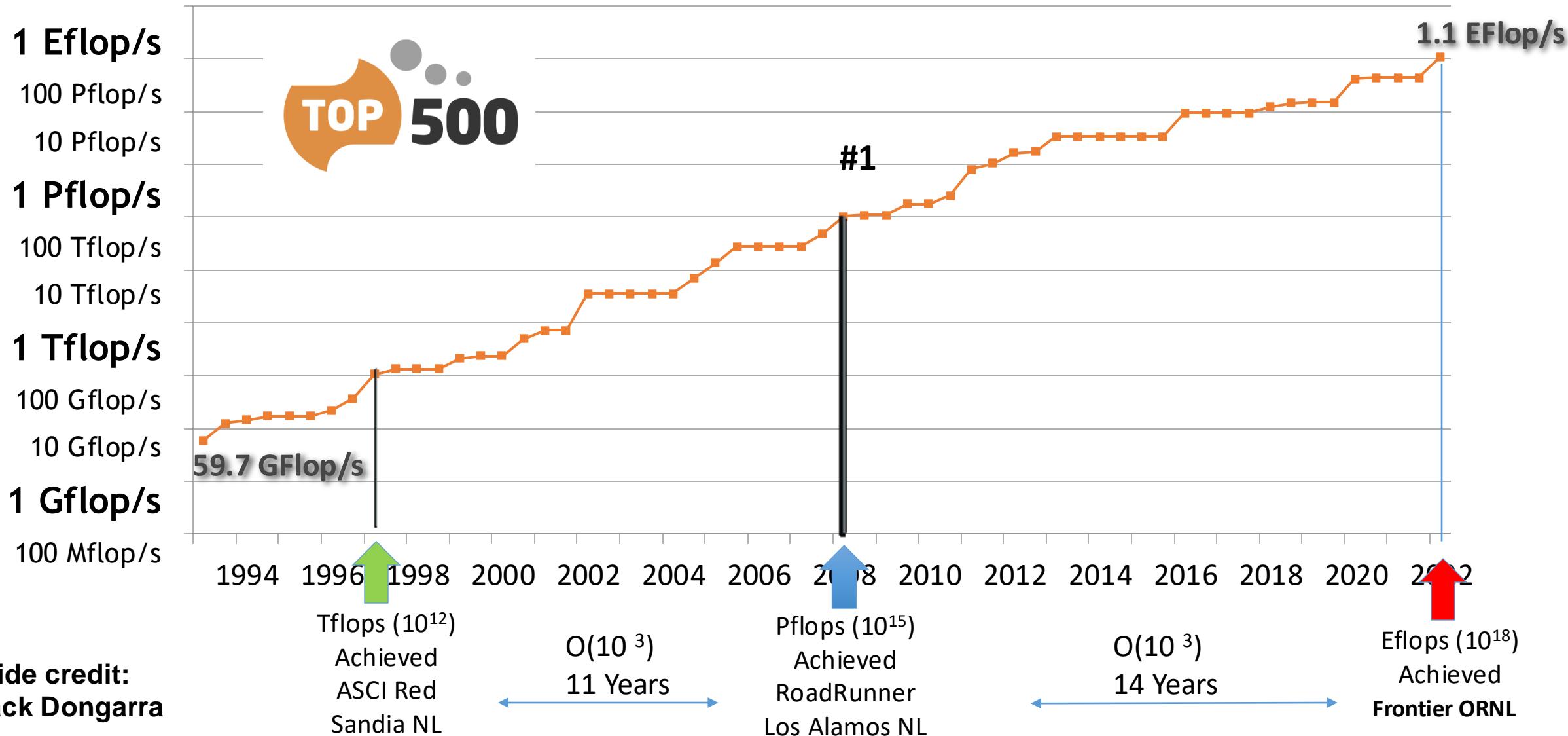
## SDSC Expanse

- 728 CPU nodes with 4.6 TFLOP/sec (each node)  
**3.4 PFLOP/sec (aggregate CPU)**
- 52 GPU nodes 4 x Nvidia V100 (Volta arch)  
31.3 TFLOP/sec DP, 62.7 TFLOP/sec SP (each node)  
**1.6 PFLOP/sec DP, 3.3 PFLOP/sec SP (aggregate GPU)**
- Hardware Cost – \$10 Million USD

## DIY 4 x Nvidia RTX 3080 box (2020) (Ampere arch)

- 1.9 TFLOP/sec DP
- **119.0 TFLOP/sec SP**
- Cost – ~ \$4 Thousand USD

# Performance Development



Slide credit:  
Jack Dongarra

# Current HPC Environment for Scientific Computing

## DOE Exascale Supercomputers



AMD APU Based  
(Planned)



Intel CPU + GPU Based  
(Recently installed)



AMD CPU + GPU Based  
(Up & running)

## SDSC Expanse



AMD + Intel CPUs  
Nvidia GPUs

### Highly parallel

- Distributed memory
- MPI + Open-MP programming model

### Heterogeneous

- Commodity processors + GPU accelerators

### Communication

- Special interconnects
- Moving data between parts very expensive compared to floating point operations

### Floating point hardware

- 64, 32, 16, & 8 bit precision

#### ORNL Frontier: 2 Eflop/s

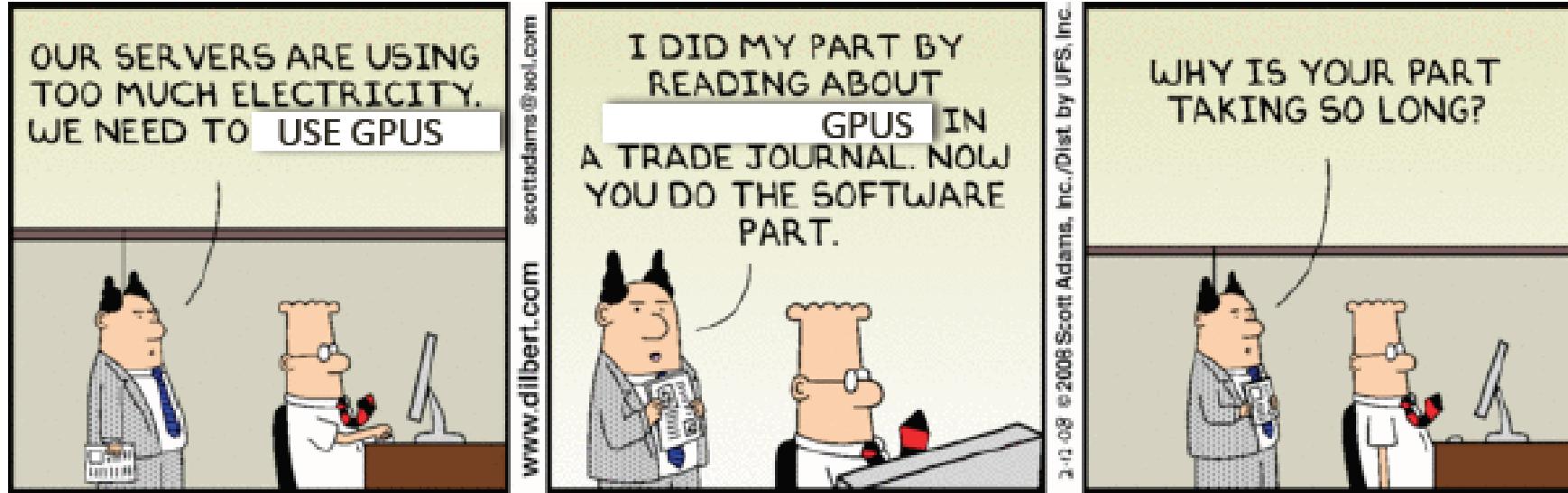
8.8 x 10<sup>6</sup> Cores, 9408 nodes, 30 MW  
(node = 1 AMD CPU + 4 AMD GPUs)  
**> 98% of performance from GPUs**



# HPCG Benchmark Top 10, November 2022

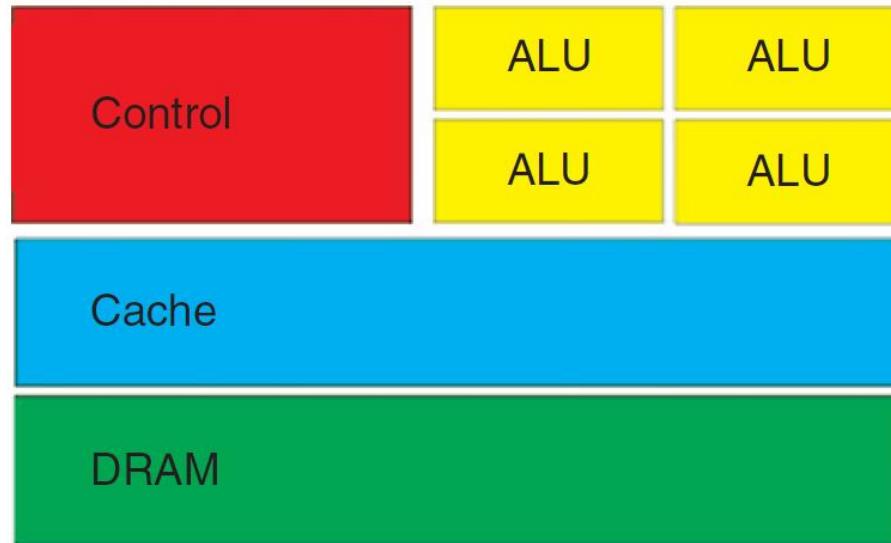
Rank	Site	Computer	Cores	HPL Rmax (Pflop/s)	TOP500 Rank	HPCG (Pflop/s)	Fraction of Peak
1	RIKEN Center for Computational Science Japan	Fugaku, Fujitsu A64FX 48C 2.2GHz, Tofu D, Fujitsu	7,630,848	442	2	16.0	3.0%
2	DOE/SC/ORNL USA	Frontier, HPE Cray EX235a, AMD 3 <sup>rd</sup> EPYC 64C, 2 GHz, AMD Instinct MI250X, Slingshot 10	8,730,112	1,102	1	14.1	0.8%
3	EuroHPC/CSC Finland	LUMI, HPE Cray EX235a, AMD Zen-3 (Milan) 64C 2GHz, AMD MI250X, Slingshot-11	2,174,976	304	3	3.41	0.8%
4	DOE/SC/ORNL USA	Summit, AC922, IBM POWER9 22C 3.7GHz, Dual-rail Mellanox FDR, NVIDIA Volta V100, IBM	2,414,592	149	5	2.93	1.5%
5	EuroHPC/CINECA Italy	Leonardo, BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 40 GB, Quad-rail NVIDIA HDR100 Infiniband	1,463,616	175	4	2.57	1.0%
6	DOE/SC/LBNL USA	Perlmutter, HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10	761,856	70.9	8	1.91	2.0%
7	DOE/NNSA/LLNL USA	Sierra, S922LC, IBM POWER9 20C 3.1 GHz, Mellanox EDR, NVIDIA Volta V100, IBM	1,572,480	94.6	6	1.80	1.4%
8	NVIDIA USA	Selene, DGX SuperPOD, AMD EPYC 7742 64C 2.25 GHz, Mellanox HDR, NVIDIA Ampere A100	555,520	63.5	9	1.62	2.0%
9	Forschungszentrum Juelich (FZJ) Germany	JUWELS Booster Module, Bull Sequana XH2000 , AMD EPYC 7402 24C 2.8GHz, Mellanox HDR InfiniBand, NVIDIA Ampere A100, Atos	449,280	44.1	12	1.28	1.8%
10	Saudi Aramco Saudi Arabia	Dammam-7, Cray CS-Storm, Xeon Gold 6248 20C 2.5GHz, InfiniBand HDR 100, NVIDIA Volta V100, HPE	672,520	22.4	20	0.88	1.6%

# What's the catch?

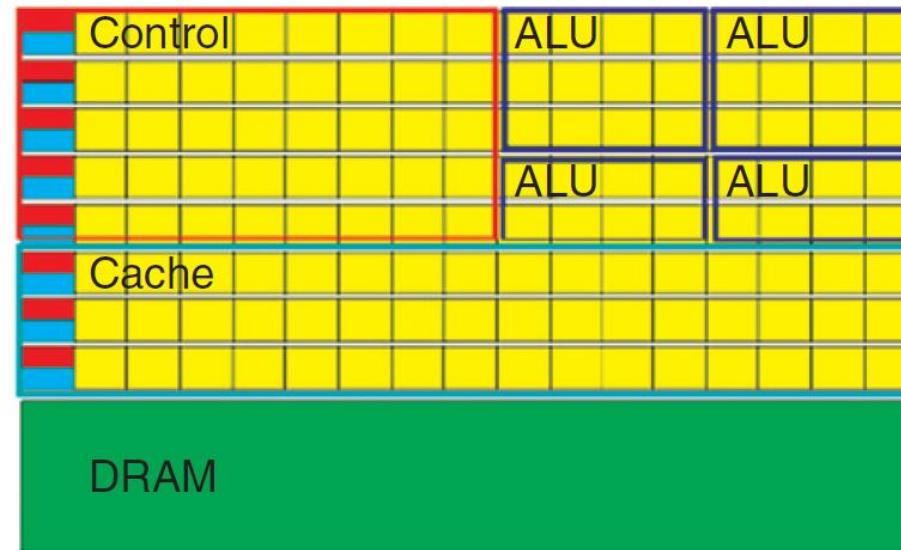


# GPU vs CPU architecture

(a) CPU



(b) GPU



## CPU

- Few processing cores with sophisticated hardware
- Multi-level caching
- Prefetching
- Branch prediction

## GPU

- Thousands of simplistic compute cores (packaged into a few multiprocessors)
- Operate in lock-step
- Vectorized loads/stores to memory
- Need to manage memory hierarchy

# GPU architecture

## CUDA Computing with Tesla T10

- 240 SP processors at 1.45 GHz: 1 TFLOPS peak
- 30 DP processors at 1.44Ghz: 86 GFLOPS peak
- 128 threads per processor: 30,720 threads total

The diagram illustrates the Tesla T10 architecture across four levels of abstraction:

- Physical Components:** A die shot of the GPU chip, a physical PCIe graphics card, and a server rack containing multiple Tesla T10 cards.
- System Architecture:** Shows the Tesla T10 card connected to a Host CPU, Bridge, and System Memory via a bus. It also includes a Host Distribution section and an Interconnection Network.
- Processor Level:** A detailed view of one of the 10 multiprocessors. Each multiprocessor contains:
  - 10 ROP units (red)
  - 10 L2 cache blocks (green)
  - 10 DRAM blocks (blue)
  - An interconnection network (yellow)
- Multiprocessor Internal Structure:** A zoomed-in view of a single multiprocessor showing:
  - SP (Single Precision) compute cores (green)
  - SPU (Special Function Unit) cores (light green)
  - DP (Double Precision) compute core (blue)
  - Instruction cache (I-Cache)
  - Memory cache (C-Cache)
  - Shared memory
  - MT Issue (Multiple Thread Issue) unit

© NVIDIA Corporation 2008

## Nvidia GPU architecture in 2009

- Tesla T10, a server with early C1060 datacenter GPU
- Basic architecture still looks similar

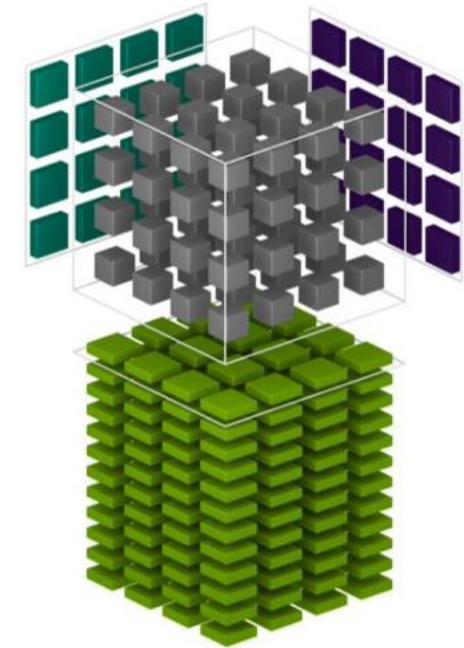
## Multiprocessor

- SP compute cores
- DP compute core(s)
- Special function units
- Instruction cache
- Shared memory / data cache
- Handles many more threads than processing cores

# Tensor cores

## Accelerating MMA for FP64, TF32, Bfloat15, and FP16

- Tensor Cores are specialized hardware for deep learning that help accelerate matrix multiply and accumulate operations
- Nvidia Volta architecture introduced Tensor Cores with FP16 data types
- Nvidia Ampere GPUs introduce Tensor Core support for FP64, TF32, and Bfloat16 data types
- Deep learning operations that benefit from tensor cores are
  - Fully connected / linear / dense layers
  - Convolutional layers
  - Recurrent layers
- Tensor Cores are also used for mixed precision matrix operations (CUBLAS library)



Tensor core 4x4 matrix multiply and accumulate, Volta architecture

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}_{\text{FP16 or FP32}} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}_{\text{FP16}} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}_{\text{FP16 or FP32}}$$

# GPU architecture

## Hardware characteristics change across GPU models and generations

- FP32 / FP64 floating point performance
- Memory bandwidth
- Number of compute cores and multiprocessors
- Number of threads that the hardware can execute
- Number of registers and cache size
- Available GPU memory, device / shared

Data Center GPUs	P100	V100	A100
#Multi Proc	56	80	108
SP Cores per MP	64	64	64
#Cores	3,584	5,120	6,912
Warp Size	32	32	32
FP64 Gflop/s	4,763	7,066	9,746

## Memory hierarchy needs to be explicitly managed

- CPU memory, GPU global / shared / texture / constant memory
- Unified memory helps, but the memory hierarchy still exists

## Different hardware vendors work in different ways

- Nvidia vs AMD vs Intel GPUs

# Nvidia GPU models

**Nvidia compute capabilities determine features available on Nvidia GPUs**

- E.g. double precision support since version 1.3

## **Hardware Version 7.0 (Volta V100)**

- Titan-V
- V100

## **Hardware Version 6.1 (Pascal GP102/104)**

- Titan-XP [aka Pascal Titan-X]
- GTX-1080Ti / 1080 / 1070 / 1060
- Quadro P6000 / P5000
- P4 / P40

## **Hardware Version 6.0 (Pascal P100/DGX-1)**

- Quadro GP100 (with optional NVLink)
- P100 12GB / P100 16GB / DGX-1

## **Hardware Version 8.0 (Ampere)**

- RTX 3050, 3060, 3060Ti, 3070, 3080, 3090 (gaming)
- RTX A4000, A5000, A6000 (workstations)
- A10, A16, A30, A40, A100 (datacenter)

## **Hardware Version 7.5 (Turing)**

- RTX 2060, 2070, 2080, 2080Ti (gaming)
- RTX A4000, A5000, A6000 (workstations)
- Quadro RTX 4000, 5000, 6000, 8000
- Tesla T4 (datacenter)

# What this means for your program

## Threads

- Never write code with any assumption for how many threads it will use.
- Use functions (CUDA calls in the case of Nvidia) to query the hardware configuration at runtime.
- Launch many more threads than processing cores.  
This helps hide memory access latency.

## Data types

- Avoid using double precision where not specifically needed.

You don't have to worry about any of this if you use An ML framework such as PyTorch or Tensorflow, which will automatically compile your ML model for the targeted hardware



# SDSC Expanse

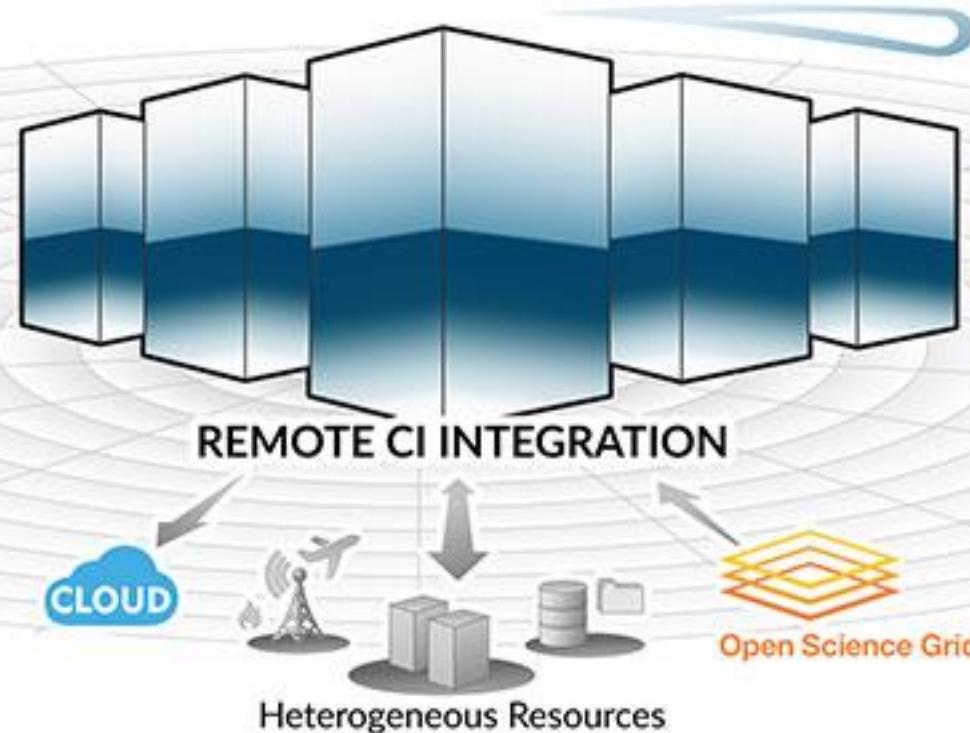
Launched in Fall 2020

## HPC RESOURCE

13 Scalable Compute Units  
728 Standard Compute Nodes  
52 GPU Nodes: 208 GPUs  
4 Large Memory Nodes

## DATA CENTRIC ARCHITECTURE

12PB Perf. Storage: 140GB/s, 200k IOPS  
Fast I/O Node-Local NVMe Storage  
7PB Ceph Object Storage  
High-Performance R&E Networking



## LONG-TAIL SCIENCE

Multi-Messenger Astronomy  
Genomics  
Earth Science  
Social Science

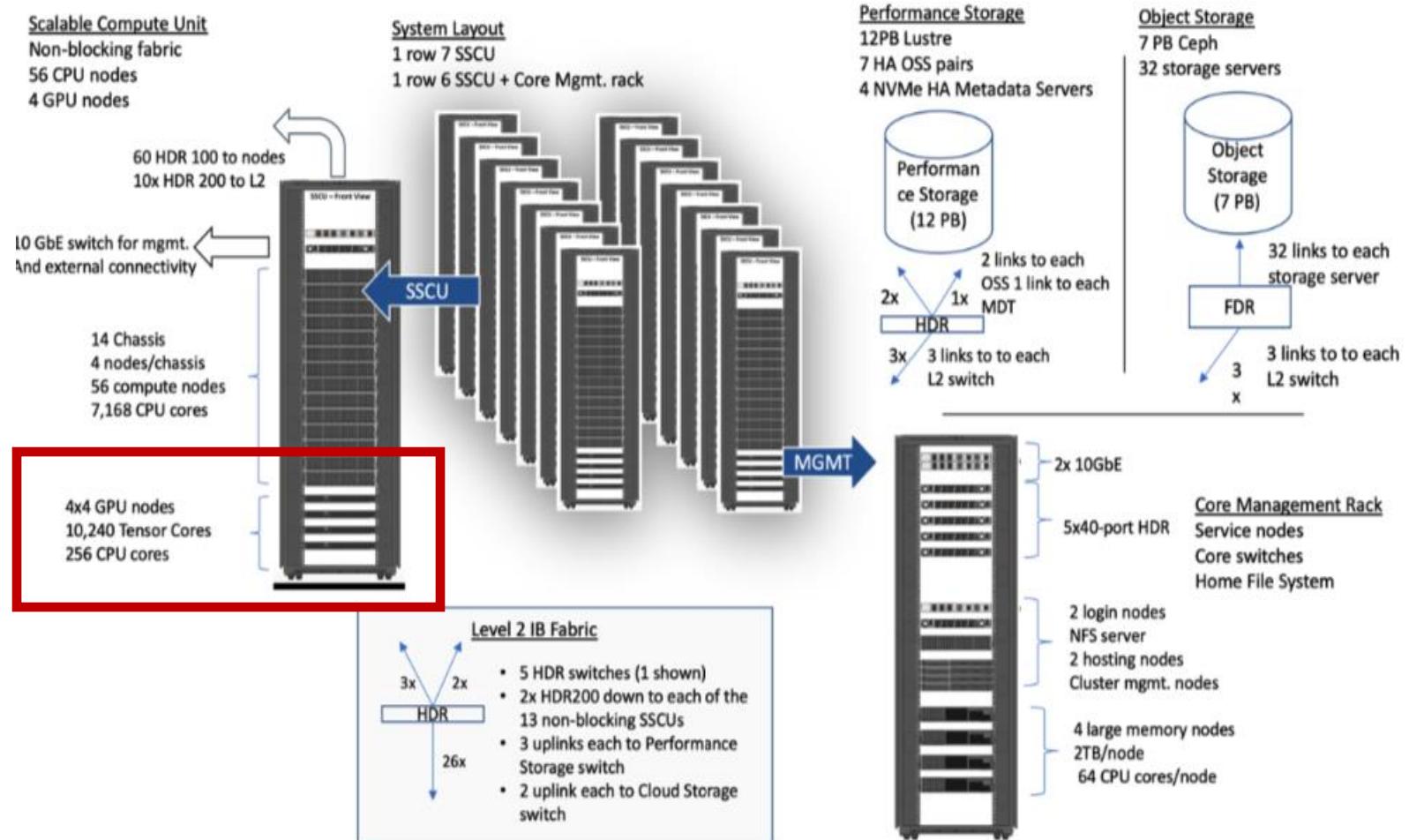
## INNOVATIVE OPERATIONS

Composable Systems  
High-Throughput Computing  
Science Gateways  
Interactive Computing  
Containerized Computing  
Cloud Bursting

# Expanse Heterogeneous Architecture

## System Summary

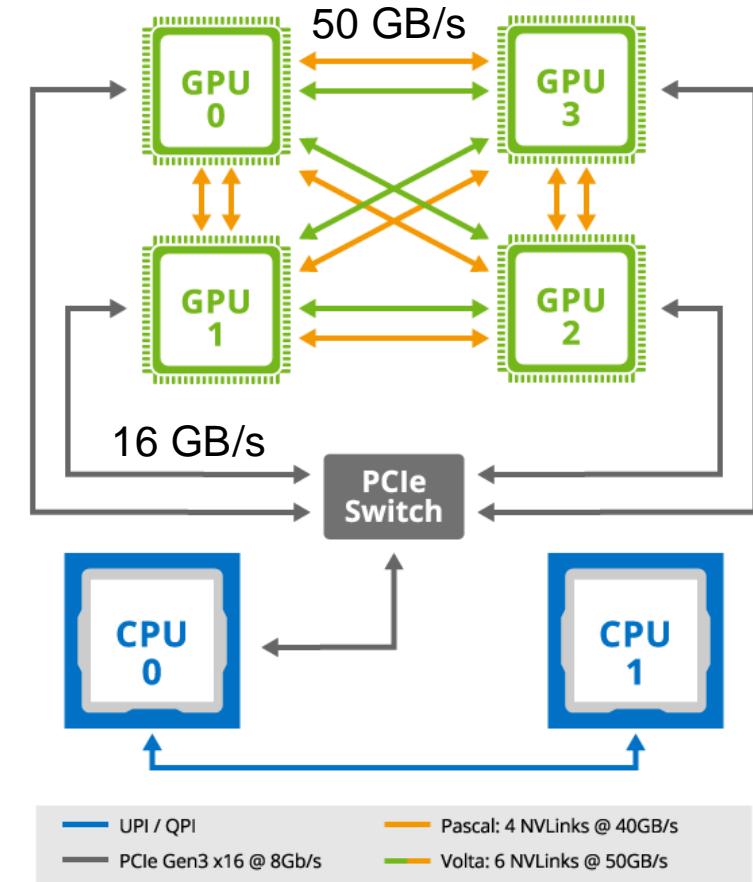
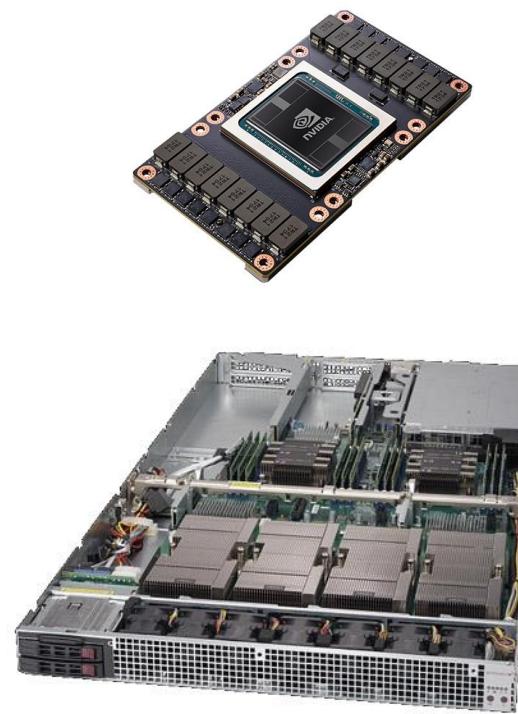
- 13 SDSC Scalable Compute Units (SSCU)
- 728 Standard Compute Nodes
- 93,184 Compute Cores
- 200 TB DDR4 Memory
- 52x 4-way GPU Nodes w/NVLINK
- **208 V100 GPUs**
- 4x 2TB Large Memory Nodes
- HDR 100 non-blocking Fabric
- 12 PB Lustre High Performance Storage
- 7 PB Ceph Object Storage
- 1.2 PB on-node NVMe
- Dell EMC PowerEdge
- Direct Liquid Cooled



# SDSC Expanse GPU nodes with Nvidia V100 SXM2

## 52 GPU nodes

- 2 x 20-core Intel Xeon Gold 6248 (Cascade Lake) CPUs
- 384 GB RAM (131 GB/s)
- 4 x Nvidia V100 SXM2 GPUs
- 32 GB HBM2 RAM per GPU (897 GB/s)
- 1.6 TB NVMe/node



## User guide:

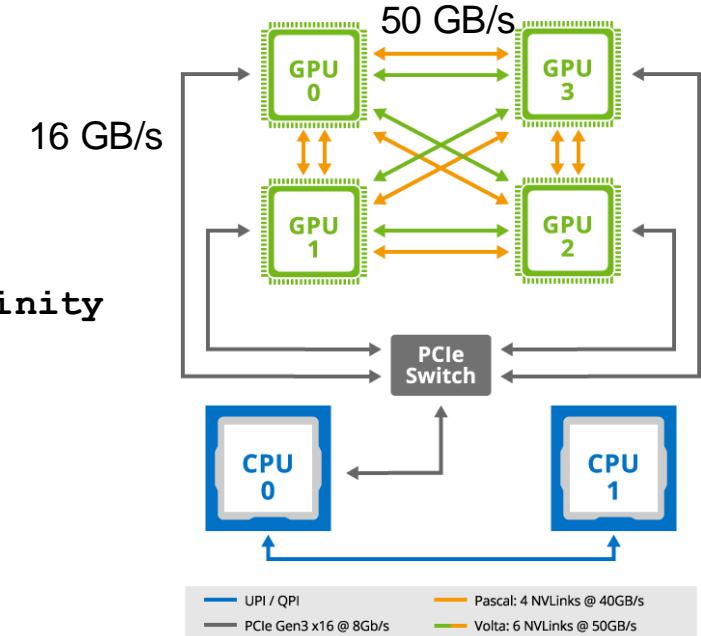
[https://www.sdsc.edu/support/user\\_guides/expanse.html](https://www.sdsc.edu/support/user_guides/expanse.html)

# SDSC Expanse GPU nodes with Nvidia V100 SXM2

## Node topology

Output of `nvidia-smi topo -m`

	GPU0	GPU1	GPU2	GPU3	mlx5_0	CPU Affinity	NUMA Affinity
GPU0	X	NV2	NV2	NV2	NODE	0,2,4,6,8,10	0
GPU1	NV2	X	NV2	NV2	NODE	0,2,4,6,8,10	0
GPU2	NV2	NV2	X	NV2	SYS	1,3,5,7,9,11	1
GPU3	NV2	NV2	NV2	X	SYS	1,3,5,7,9,11	1
mlx5_0	NODE	NODE	SYS	SYS	X		



## Legend:

- X = Self
- SYS = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
- NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
- PHB = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
- PXB = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
- PIX = Connection traversing at most a single PCIe bridge
- NV# = Connection traversing a bonded set of # NVLinks

# SDSC Expanse – V100 SXM2 GPUs



TECHPOWERUP

# SDSC Expanse – V100 SXM2 GPUs



# SDSC Expanse – V100 SXM2 GPUs

## Each GPU

- 32 GB HBM2 RAM (897 GB/s)
- 80 SMs (Streaming Multiprocessors)
- 64 FP32 cores / SM (5120 total)
- 32 FP64 cores / SM (2560 total)
- 8 Tensor cores / SM (640 total)
- 300 Watt TDP

## Peak performance

- 7.8 FP64 TFLOPs
- 15.7 FP32 TFLOPs
- 31.3 FP16 TFLOPs
- 125 Tensor TFLOPs



# Training overview

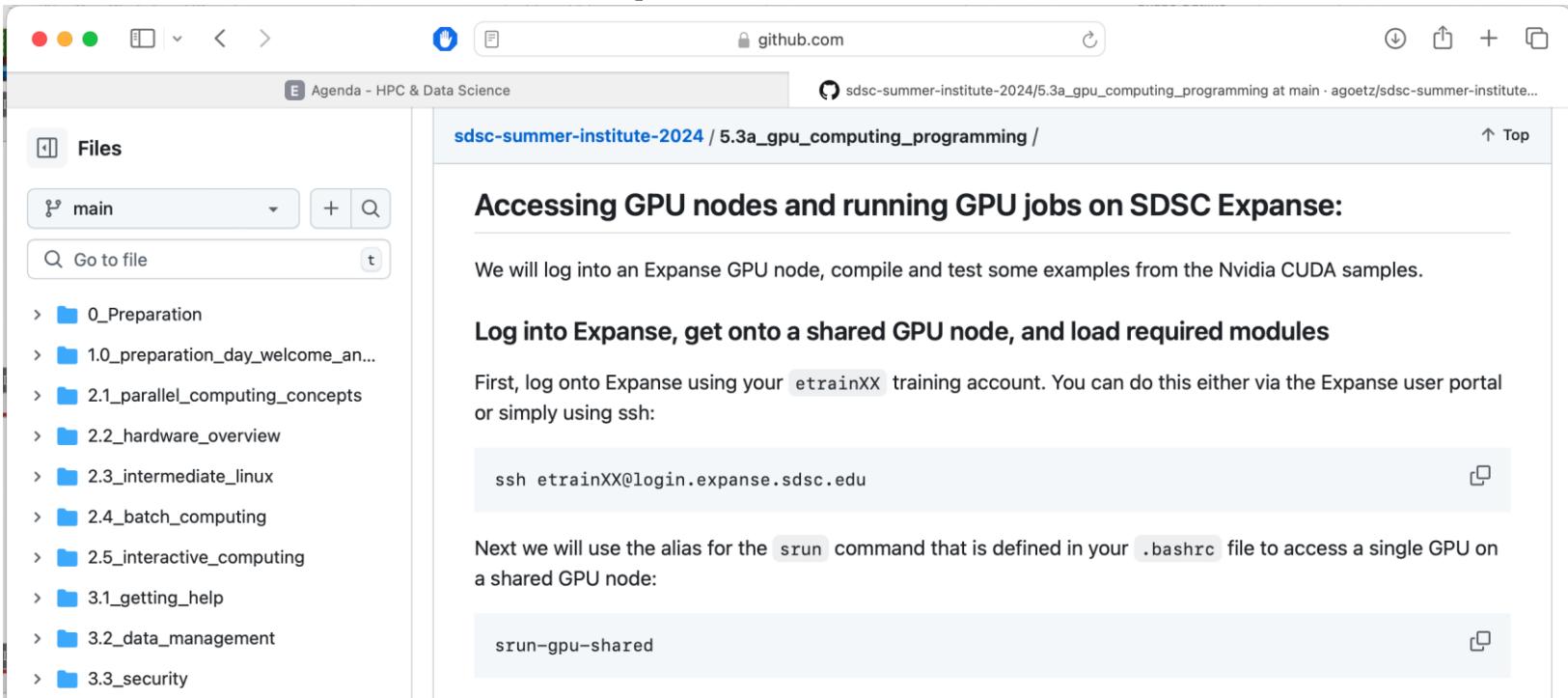
## We will cover the following topics

- GPU hardware overview
- **Accessing SDSC Expanse GPU nodes**
- GPU accelerated software examples
- Programming GPUs – Overview for Nvidia GPUs
- GPU enabled libraries
  - 15 minutes break –
- CUDA C programming basics
- Hands-on exercises on SDSC Expanse – CUDA
- OpenACC introduction
- Hands-on exercises on SDSC Expanse – OpenACC

# Hands-on session

- In SI 2024 Github repository  
<https://github.com/sdsc/sdsc-summer-institute-2024>  
directory 5 . 3a \_gpu \_computing \_and \_programming

Follow the instructions in the Github repo README.md



The screenshot shows a GitHub repository page for 'sdsc-summer-institute-2024/5.3a\_gpu\_computing\_programming'. The left sidebar lists several sub-directories: 0\_Preparation, 1.0\_preparation\_day\_welcome\_an..., 2.1\_parallel\_computing\_concepts, 2.2\_hardware\_overview, 2.3\_intermediate\_linux, 2.4\_batch\_computing, 2.5\_interactive\_computing, 3.1\_getting\_help, 3.2\_data\_management, and 3.3\_security. The main content area displays the README.md file's text:

**Accessing GPU nodes and running GPU jobs on SDSC Expanse:**

We will log into an Expanse GPU node, compile and test some examples from the Nvidia CUDA samples.

**Log into Expanse, get onto a shared GPU node, and load required modules**

First, log onto Expanse using your `etrainXX` training account. You can do this either via the Expanse user portal or simply using ssh:

```
ssh etrainXX@login.expanse.sdsc.edu
```

Next we will use the alias for the `srun` command that is defined in your `.bashrc` file to access a single GPU on a shared GPU node:

```
srun-gpu-shared
```

# SDSC Expanse login

## Login

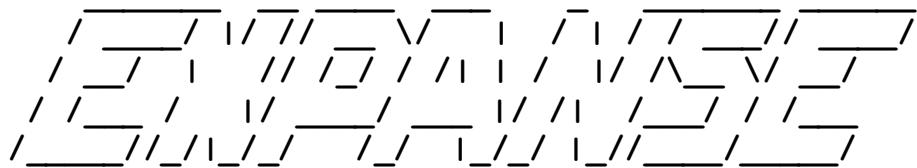
```
$> ssh agoetz@expanse.sdsc.edu  
Welcome to Bright release
```

9.0

Based on CentOS Linux 8  
ID: #000002

---

WELCOME TO



---

Use the following commands to adjust your environment:

```
'module avail'           - show available modules  
'module add <module>' - adds a module to your environment for this session  
'module initadd <module>' - configure module to be loaded at every login
```

---

```
Last login: Tue Apr 27 01:58:53 2021 from 136.26.112.138  
[agoetz@login01 ~]$
```

# SDSC Expanse GPU node access

- The GPU nodes can be accessed via two different partitions "gpu" (entire nodes with 4 GPUs) and "gpu-shared" (individual GPUs).

```
#SBATCH --partition=gpu
```

or

```
#SBATCH --partition=gpu-shared
```

- In addition to the partition name (required), the number of GPUs must be specified.

```
#SBATCH --gpus=n
```

- For example, to obtain access to a single GPU for 30 minutes in an interactive session

```
srun --partition=gpu-shared --nodes=1 --gpus=1 --ntasks-per-node=1 --cpus-per-task=10 \
--mem=92G --time=00:30:00 --wait=0 --pty /bin/bash
```

```
srun: job 2210010 queued and waiting for resources
```

```
srun: job 2210010 has been allocated resources
```

```
[agoetz@exp-8-59 ~]$
```

# SDSC Expanse GPU node access

- Note to make requests proportional to the number of available resources.
  - 4 x V100 GPUs
  - 40 CPU cores
  - 374 GB RAM
- Do not request more than 10 CPU cores and 93GB RAM per GPU, otherwise you will be charged for proportionally more time.

# SDSC Expanse GPU nodes

- For this course we have an alias to get access to a single GPU on the shared queue for 4 hours:

**srun-gpu-shared**

**Please try to get access to an Expanse GPU node now**

- Reset the modules environment to activate GPU modules, then load GPU related modules

```
# reset environment  
module reset
```

```
# Load CUDA Toolkit  
module load cuda12.2/toolkit
```

```
# Or load Nvidia HPC SDK  
module load nvhpc/21.9
```

# SDSC Expanse GPU nodes

- For information only: Interactive access to GPU nodes requesting 1 GPU for 4 hours

```
[agoetz@login01 ~] srun --partition=gpu-shared --nodes=1 --gpus=1 \
--ntasks-per-node=1 --cpus-per-task=10 --mem=92GB \
--time=04:00:00 --pty --wait=0 /bin/bash
```

- Check available GPUs using Nvidia system management interface

```
[agoetz@exp-8-59 ~]$ nvidia-smi
Tue Apr 27 02:45:26 2021
+-----+
| NVIDIA-SMI 450.51.05      Driver Version: 450.51.05      CUDA Version: 11.0      |
+-----+
| GPU  Name      Persistence-M| Bus-Id      Disp.A  | Volatile Uncorr. ECC  | |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M.  |
|                               |             |           | MIG M.               |
+-----+
|  0  Tesla V100-SXM2...  On   | 00000000:18:00.0 Off  |                0 | |
| N/A   45C     P0    67W / 300W |        0MiB / 32510MiB |      0%     Default |
|                               |             |           | N/A                 |
+-----+
...
...
```

# SDSC Expanse GPU nodes

- Processes running on the GPU are also listed.

```
...
+-----+
| Processes:
| GPU   GI   CI      PID   Type   Process name          GPU Memory |
|        ID   ID
|=====|
| No running processes found
+-----+
```

- There should be no jobs running on the GPU assigned to you
- The nodes of the shared GPU queue are configured for the CUDA runtime to use only the requested number of GPUs.

# SDSC Expanse GPU nodes

- Check Nvidia CUDA C compiler

```
[agoetz@exp-8-59 ~]$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2020 NVIDIA Corporation
Built on Wed_Jul_22_19:09:09_PDT_2020
Cuda compilation tools, release 11.0, v11.0.221
Build cuda_11.0_bu.TC445_37.28845127_0
```

- Check PGI C compiler / NVHPC C compiler

```
[agoetz@exp-8-59 ~]$ pgcc --version
pgcc (aka nvc) 21.9-0 64-bit target on x86-64 Linux -tp skylake
PGI Compilers and Tools
Copyright (c) 2021, NVIDIA CORPORATION & AFFILIATES. All rights reserved.
```

```
[agoetz@exp-8-59 ~]$ nvc --version
nvc 21.9-0 64-bit target on x86-64 Linux -tp skylake
NVIDIA Compilers and Tools
Copyright (c) 2021, NVIDIA CORPORATION & AFFILIATES. All rights reserved.
```

# Training overview

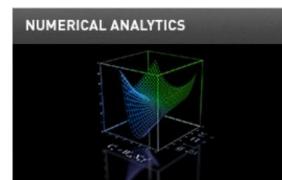
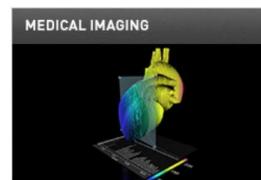
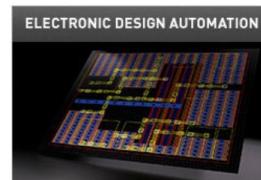
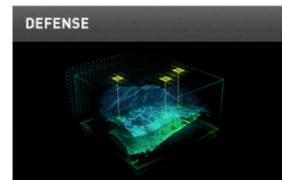
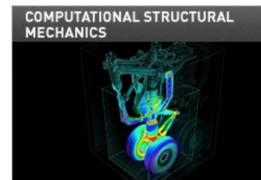
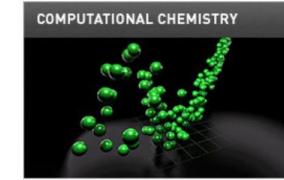
## We will cover the following topics

- GPU hardware overview
- Accessing SDSC Expanse GPU nodes
- **GPU accelerated software examples**
- Programming GPUs – Overview for Nvidia GPUs
- GPU enabled libraries
  - 15 minutes break –
- CUDA C programming basics
- Hands-on exercises on SDSC Expanse – CUDA
- OpenACC introduction
- Hands-on exercises on SDSC Expanse – OpenACC

# GPU accelerated software

## Examples from virtually any field

- Exhaustive list on <https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/>
- Chemistry
- Life sciences
- Bioinformatics
- Astrophysics
- Finance
- Medical imaging
- Natural language processing
- Social sciences
- Weather and climate
- Computational fluid dynamics
- **Machine learning**, of course
- etc...



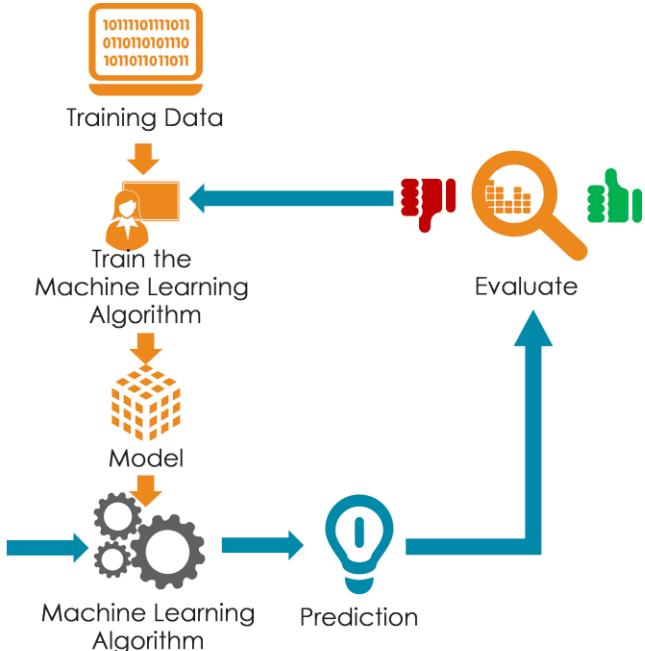
# Machine learning and GPUs

## Machine learning

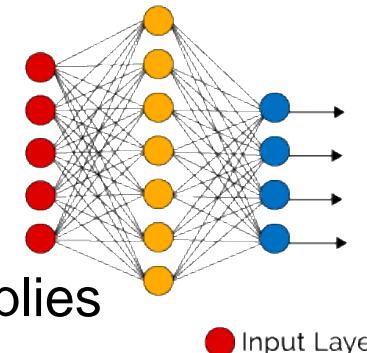
- Estimate / predictive model based on reference data.
- Many different methods and algorithms.
- GPUs are particularly well suited for deep learning workloads

## Deep learning

- Neural networks with many hidden layers.
- Tensor operations (matrix multiplications).
- GPUs are very efficient at these (4x4 matrix algebra is used in 3D graphics)
- Half-precision arithmetic can be used for many ML applications, at least for inference.
- Nvidia Volta architecture introduced tensor cores, dedicated hardware for mixed-precision matrix multiplies
- ML frameworks provide GPU support (E.g. PyTorch, TensorFlow)

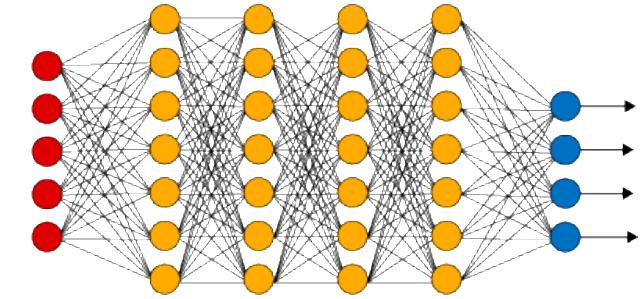


Simple Neural Network



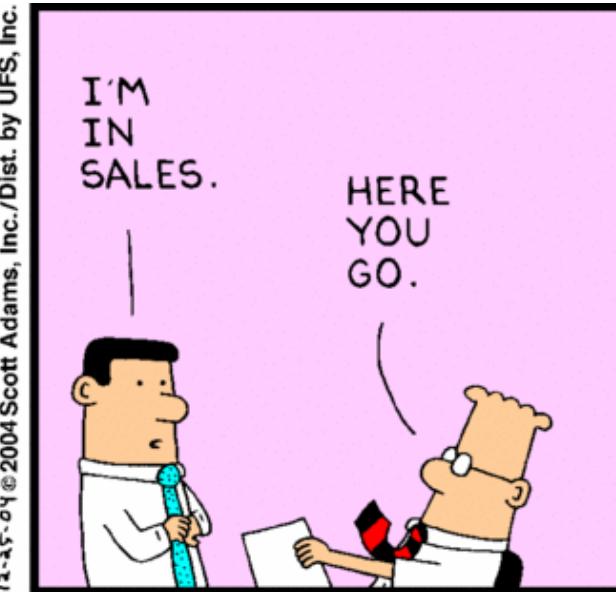
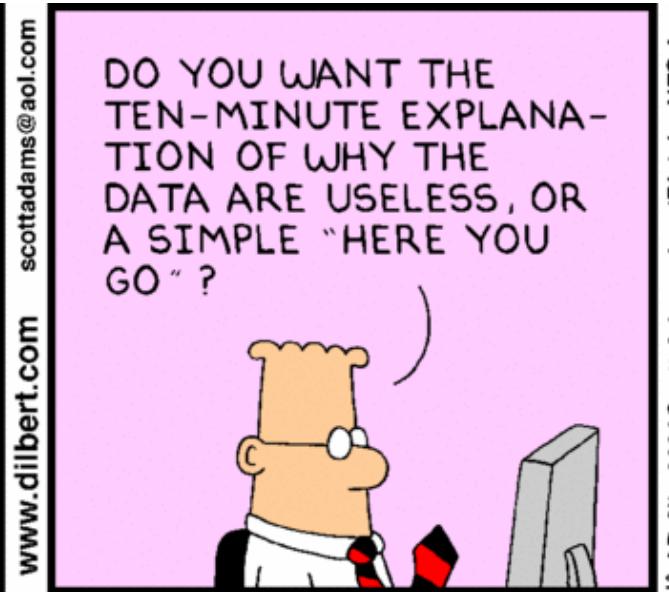
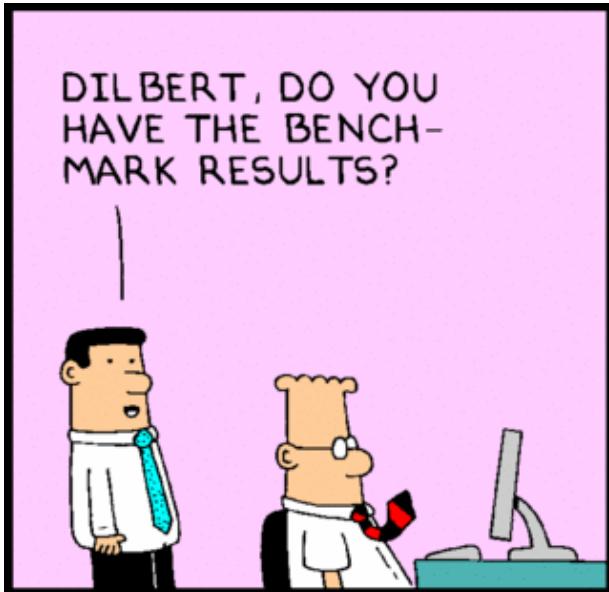
● Input Layer

Deep Learning Neural Network



● Hidden Layer      ● Output Layer

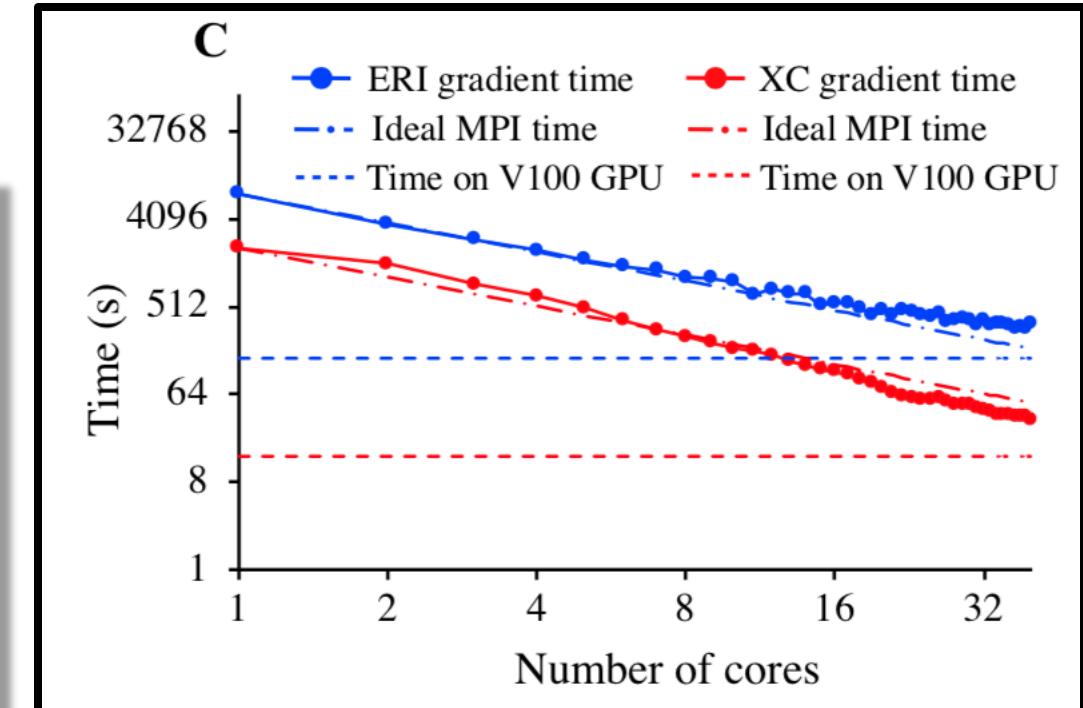
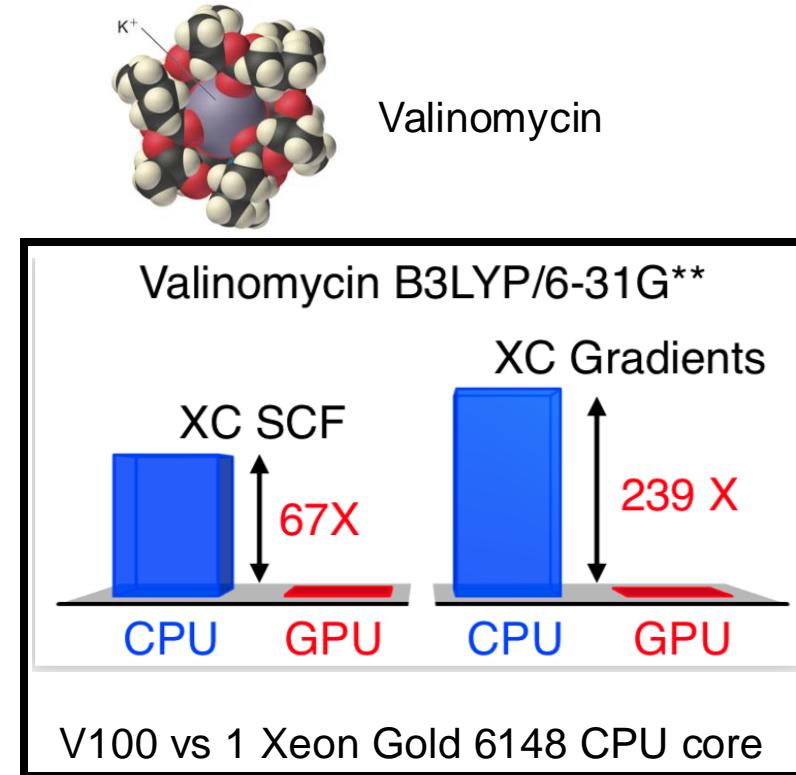
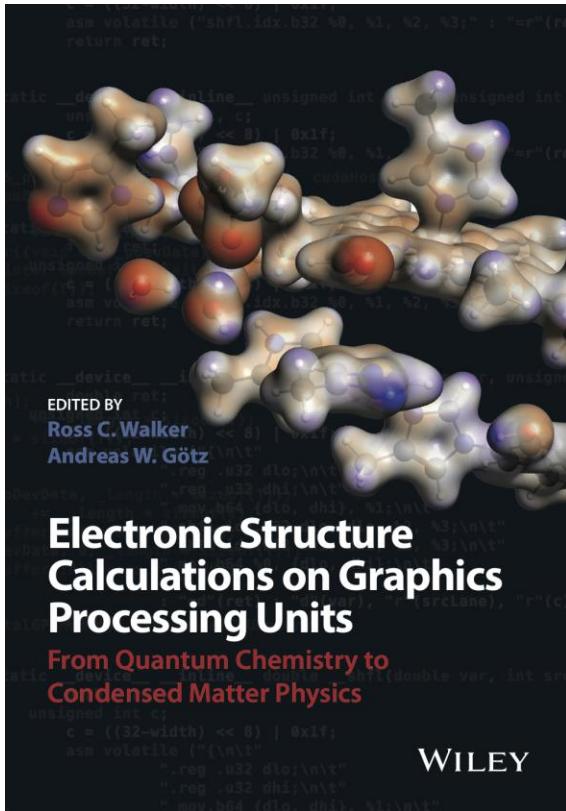
# Benchmark examples



# Quantum Chemistry

## Example: Open source code QUICK

- Compute molecular properties from quantum mechanics
- <https://github.com/merzlab/QUICK> (developed by Merz and Goetz labs)



See *J. Chem. Theory Comput.* **16**, 4315-4326 (2020) <https://dx.doi.org/10.1021/acs.jctc.0c00290>

# Quantum Chemistry

$$E[\rho] = T_s[\rho] + \int d\mathbf{r} \rho(\mathbf{r}) v_{\text{ext}}(\mathbf{r}) + \frac{1}{2} \int d\mathbf{r} d\mathbf{r}' \frac{\rho(\mathbf{r})\rho(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} + E_{\text{xc}}[\rho]$$

## Analytical electron repulsion integrals (ERIs)

- Coulomb interaction between electrons (3<sup>rd</sup> term)
- Exact exchange (not shown in equation above)

## Numerical quadrature of XC potential and energy

- XC energy has complicated form
- Cannot be computed analytically
- Thousands of grid points per atom

$$E^{xc} = \int f(\rho_\alpha, \rho_\beta, \gamma_{\alpha\alpha}, \gamma_{\alpha\beta}, \gamma_{\beta\beta}) dr,$$

$$\int d\mathbf{r} f(\mathbf{r}) \approx \sum_i \omega_i f(\mathbf{r}_i)$$

Get QUICK for free at  
<https://github.com/merzlab/quick>

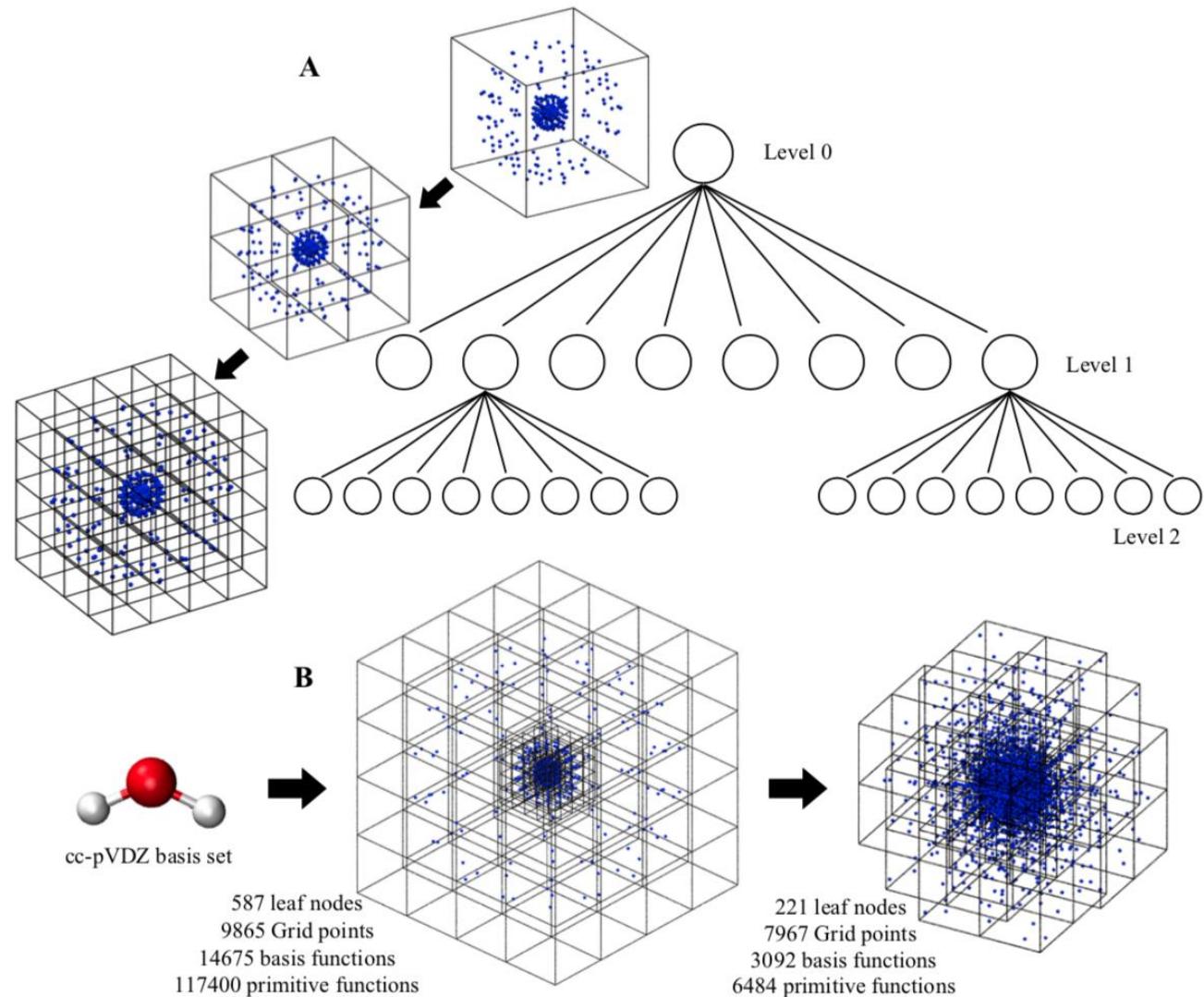
### Relevant publications

- **DFT octree-based quadrature on GPUs**  
Manathunga, Miao, Mu, Götz, Merz, *JCTC* **16** (2020) 4315.
- **Multi-GPU parallelization**  
Manathunga, Jin, Cruzeiro, Miao, Mu, Arumugam, Keipert, Aktulga, Merz, Götz, *JCTC* **17** (2021) 3955.
- **QM/MM implementation**  
Cruzeiro, Manathunga, Merz, Götz, *JCIM* **61** (2021) 2109.
- **Nvidia and AMD GPU QM/MM + optimization**  
Manathunga, Aktulga, Merz, Götz, *JCIM* **63** (2023) 711.

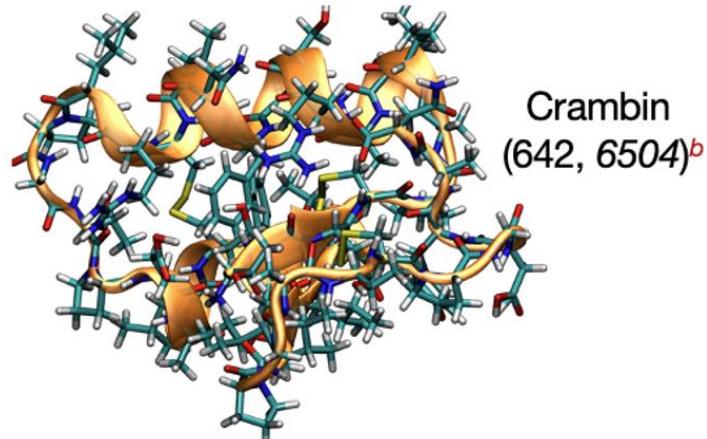
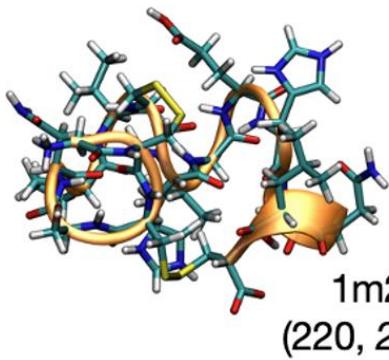
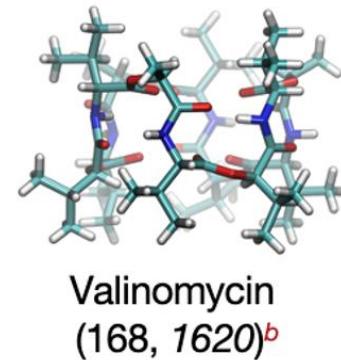
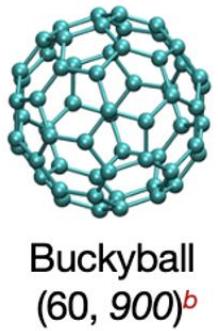
# Quantum Chemistry

## Parallel numerical quadrature in QUICK

- Octree based partitioning of 3D grid points
- Prescreening of function values on grid point batches leads to linear scaling for large molecules
- Grid point batches are processed in parallel on CPU cores via MPI or GPUs via CUDA.
- See *J. Chem. Theory Comput.* **16**, 4315-4326 (2020)  
<https://dx.doi.org/10.1021/acs.jctc.0c00290>



# Quantum Chemistry



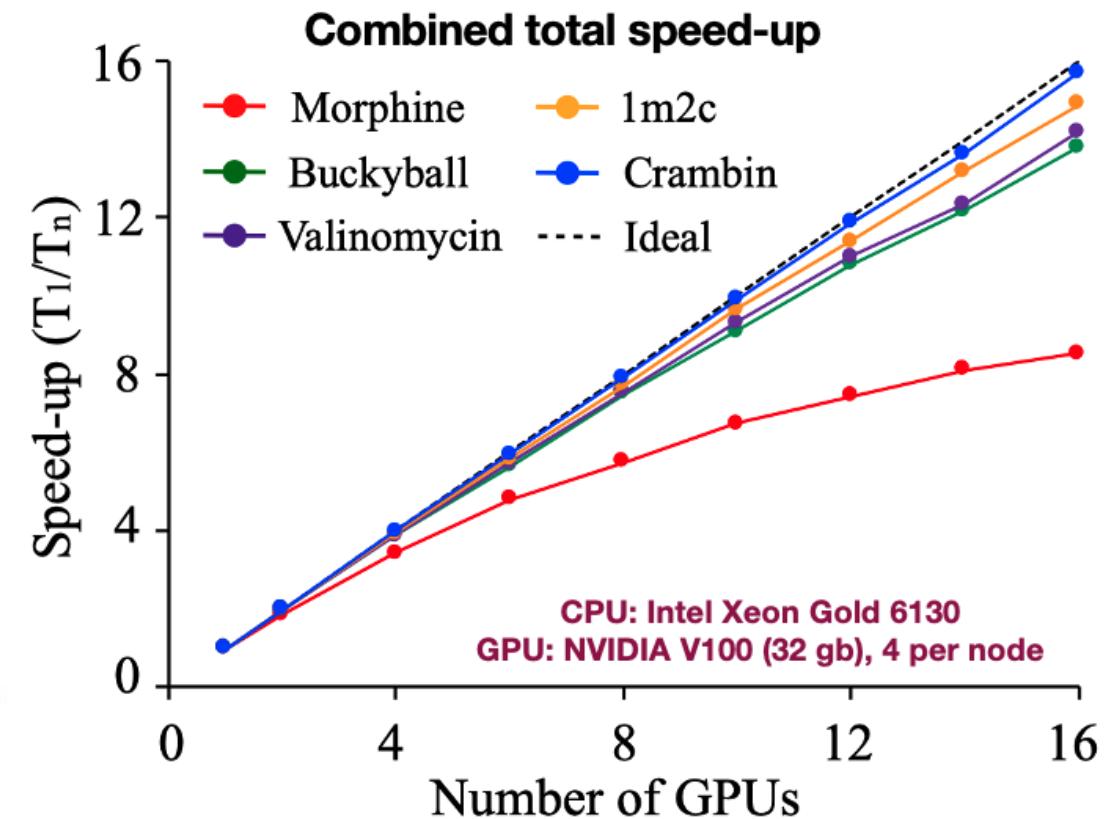
*b.* The number of atoms and number of basis functions are given in brackets.

Benchmarks: QUICK 20.06

Manathunga *et al.*, JCTC 17 (2021) 3955.

QUICK scaling across multiple GPUs (MPI + CUDA)

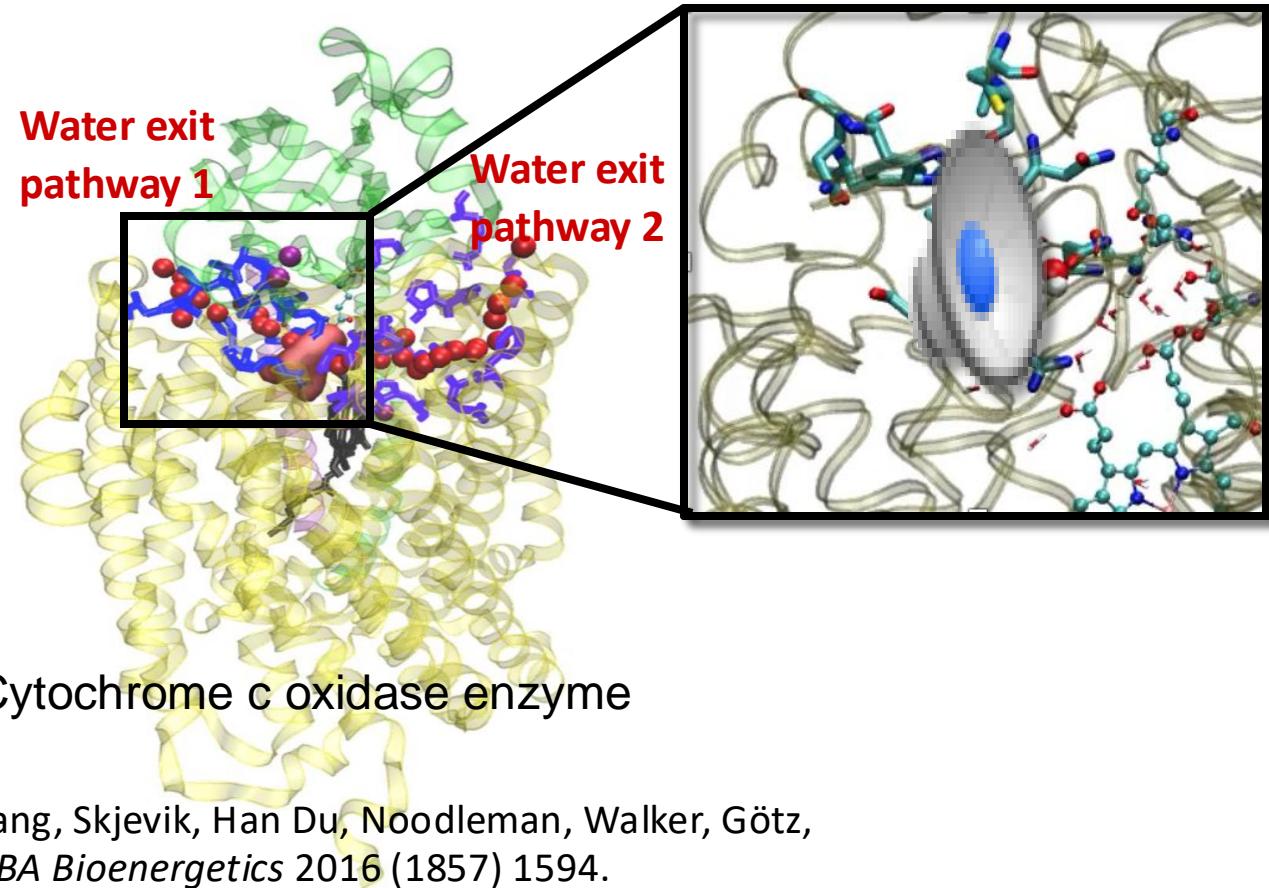
B3LYP/6-31G\*\* gradient calculations



# Molecular Dynamics

## Classical Molecular dynamics

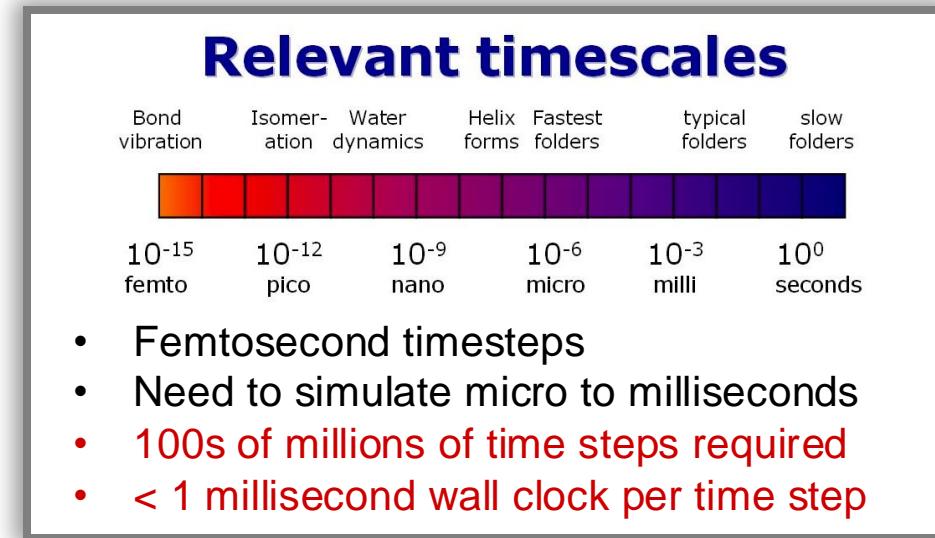
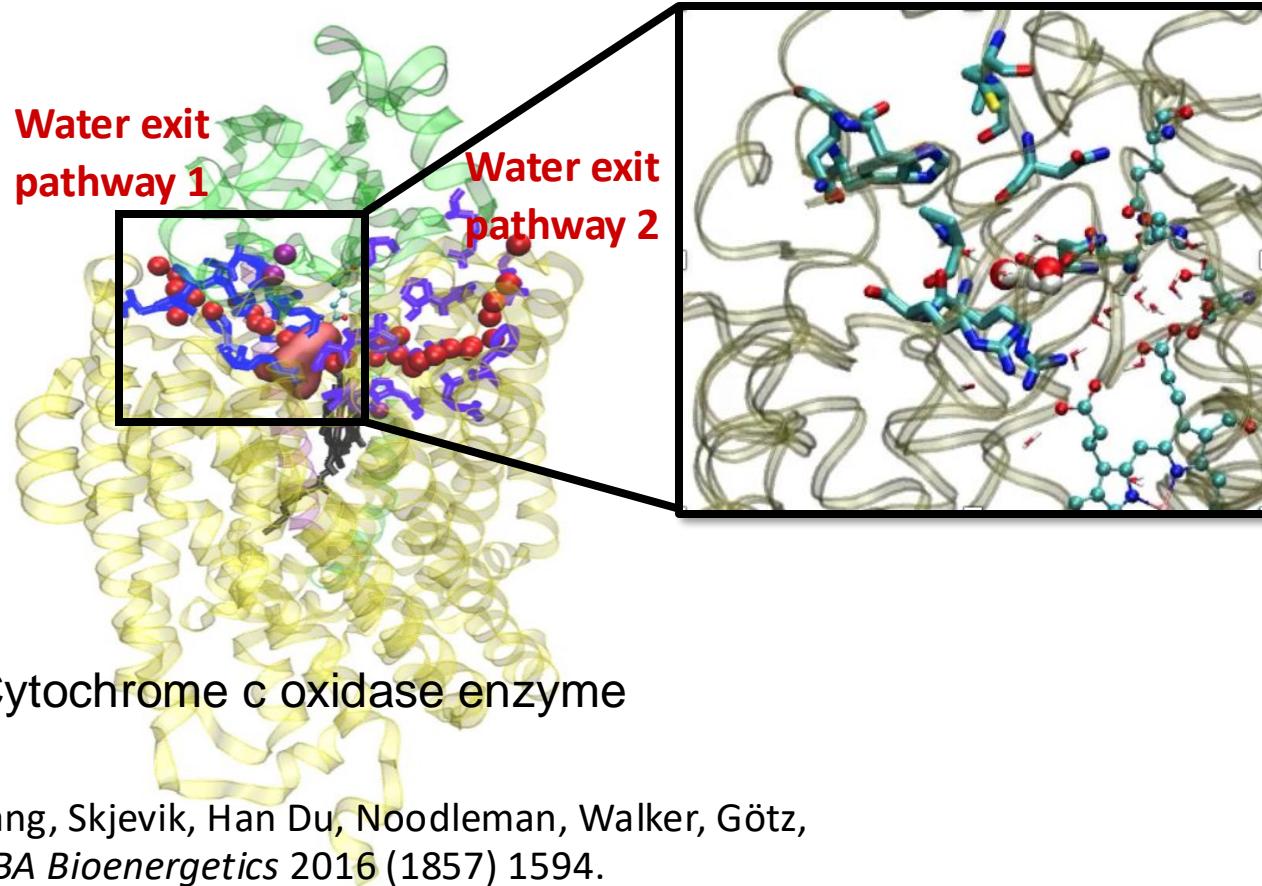
- Amber code: Atomistic simulations of condensed phase biomolecular systems



# Molecular Dynamics

## Classical Molecular dynamics

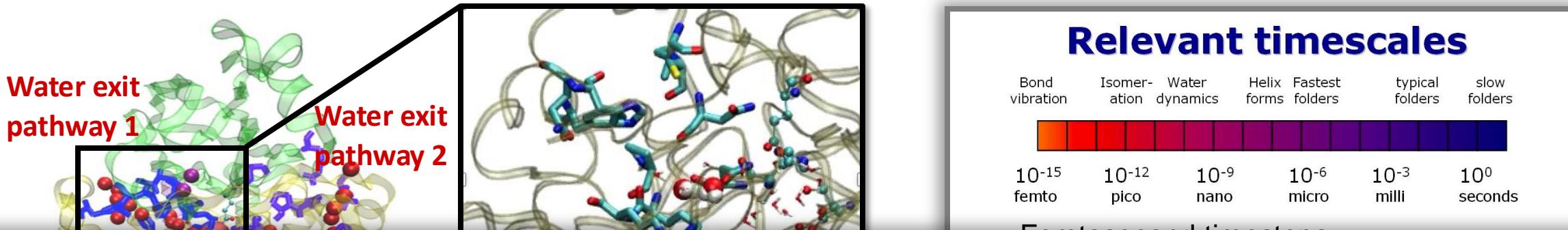
- Amber code: Atomistic simulations of condensed phase biomolecular systems



# Molecular Dynamics

## Classical Molecular dynamics

- Amber code: Atomistic simulations of condensed phase biomolecular systems

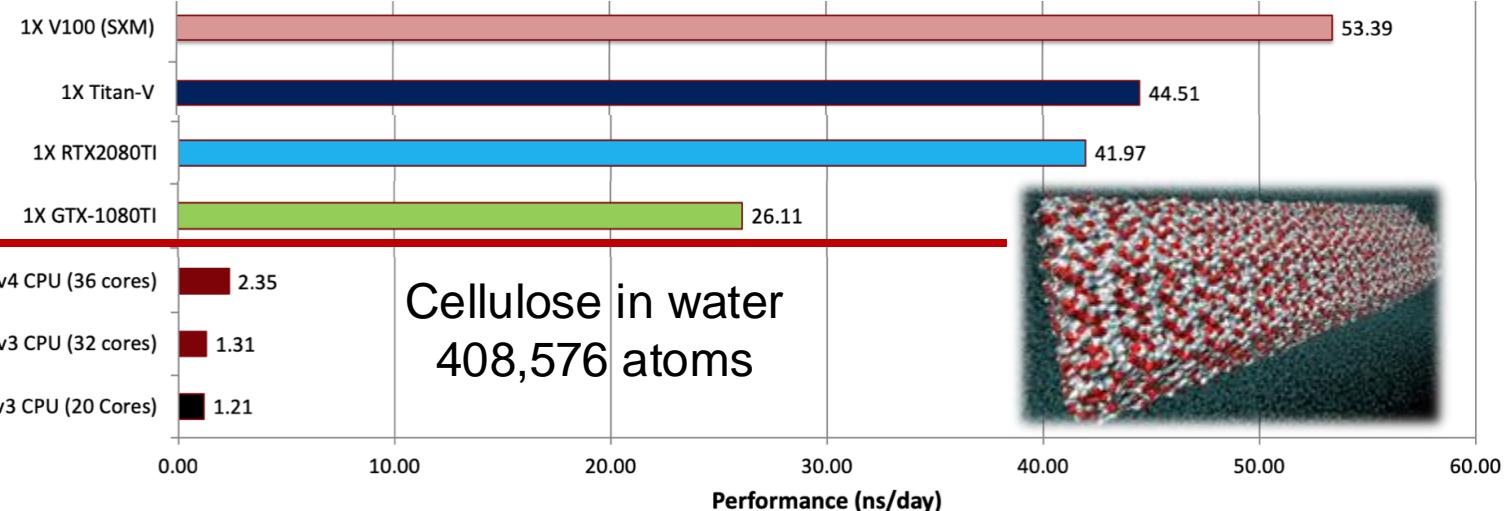


## Amber 18 molecular dynamics software

Götz, Williamson, Xu, Poole, Le Grand, Walker, *J Chem Theory Comput* 2012 (8) 1542.

Le Grand, Götz, Walker, *Comput Phys Comm* 2013 (184) 374.

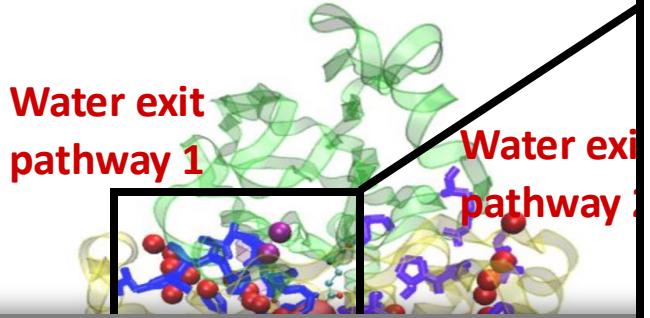
Salomon-Ferrer, Götz, Poole, Le Grand, Walker, *J Chem Theory Comput* 2012 (8) 1542.



# Molecular Dynamics

## Classical Molecular dynamics

- Amber code: Atomistic



### Amber 18 molecular dynamics

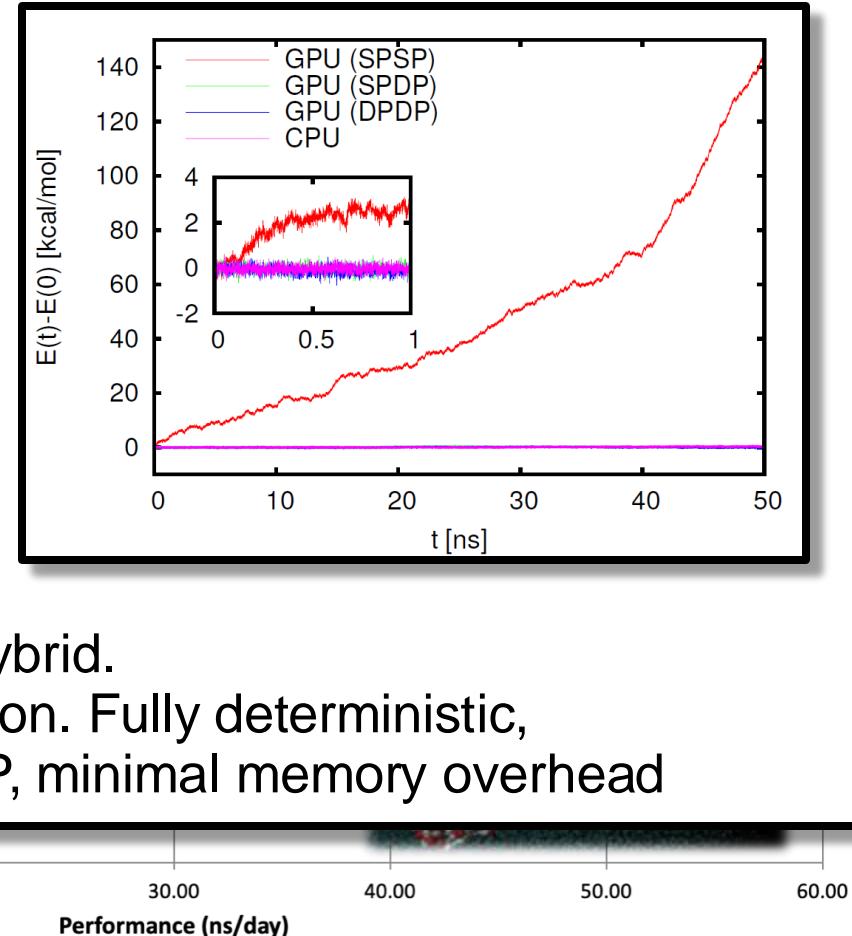
Götz, Williamson, Xu, Poole, Le Grand, Walker, *J Chem Theory Comput* 2012 (8) 1542.

Le Grand, Götz, Walker, *Comput Phys Comm* 2013 (184) 374.

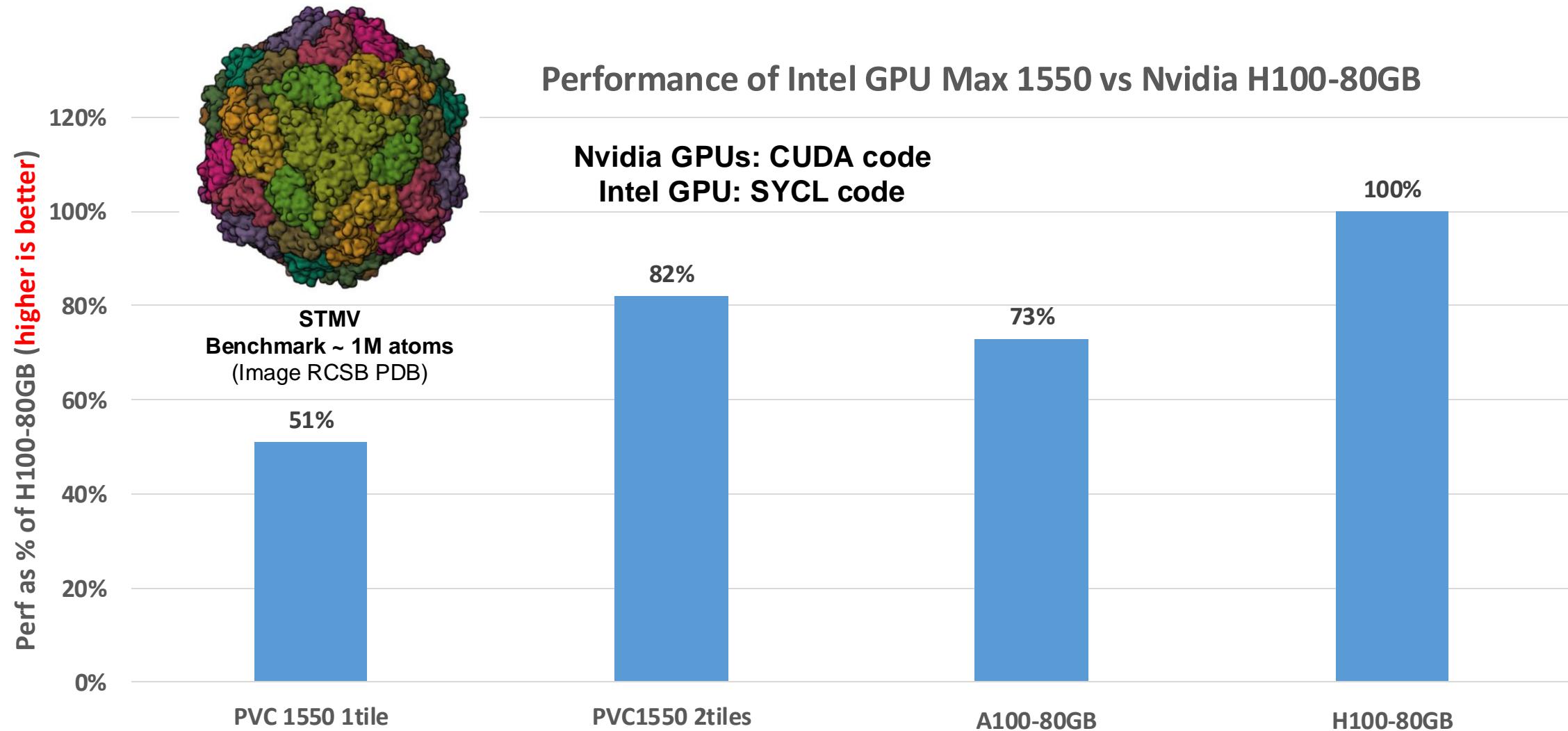
Salomon-Ferrer, Götz, Poole, Le Grand, Walker, *J Chem Theory Comput* 2012 (8) 1542.

### Precision matters

- SPSP**  
Only single precision
- SPDP**  
Single precision for calculation  
Double precision for accumulation
- DPDP**  
Full double precision
- SPFP**  
Single / Double / Fixed precision hybrid.  
Uses atomic ops for FP accumulation. Fully deterministic, faster and more precise than SPDP, minimal memory overhead



# Molecular dynamics powered by GPUs – Nvidia and Intel



# Training overview

## We will cover the following topics

- GPU hardware overview
- Accessing SDSC Expanse GPU nodes
- GPU accelerated software examples
- **Programming GPUs – Overview for Nvidia GPUs**
- GPU enabled libraries
  - 15 minutes break –
- CUDA C programming basics
- Hands-on exercises on SDSC Expanse – CUDA
- OpenACC introduction
- Hands-on exercises on SDSC Expanse – OpenACC

# GPU programming languages

## CUDA

- Proprietary, works only for Nvidia GPUs
- De-facto standard for high-performance code

## HIP

- Open-source C++ runtime API and kernel language developed by AMD
- Similar to CUDA, works with Nvidia (via CUDA) and AMD (via ROCm)

## OpenCL

- Industry standard, works for Nvidia, AMD and Intel GPUs (and other devices)

## SYCL

- Single-source, high-level, standard C++ programming model
- Can target a range of heterogeneous platforms (CPUs, GPUs, FPGAs)

## OpenACC

- Accelerator directives for Nvidia and AMD
- Works with C/C++ and Fortran

## OpenMP

- Version 4.x includes accelerator and vectorization directives
- Not mature for GPUs

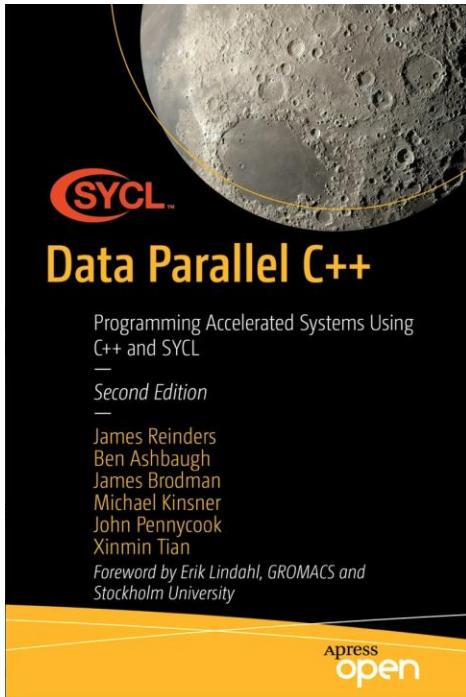
# Recommended Reading

SYCL reference: [https://github.khronos.org/SYCL\\_Reference/](https://github.khronos.org/SYCL_Reference/)

SYCL training material:

<https://www.intel.com/content/www/us/en/developer/tools/oneapi/training/overview.html>

SYCL free ebook: <https://link.springer.com/book/10.1007/978-1-4842-9691-2>



# Recommended Reading

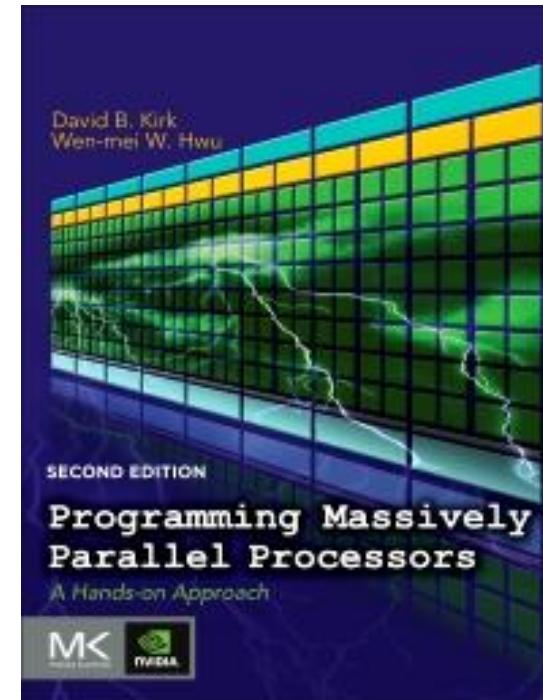
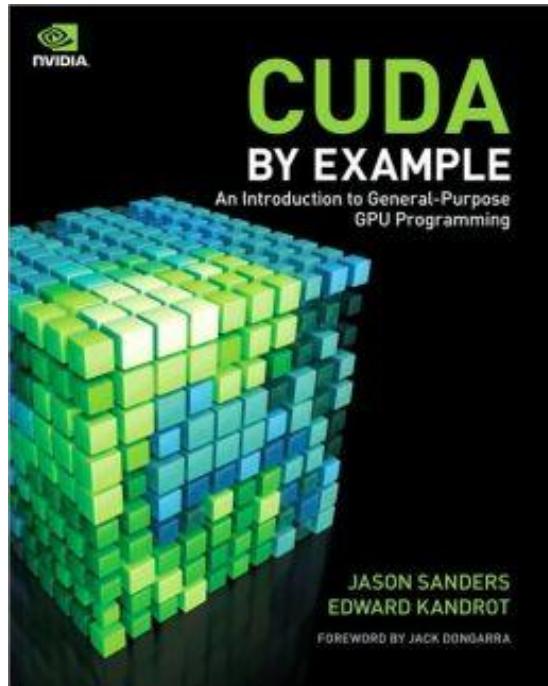
NVIDIA HPC SDK: <https://docs.nvidia.com/hpc-sdk/index.html>

CUDA C: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>

CUDA Fortran: <https://docs.nvidia.com/hpc-sdk/compilers/cuda-fortran-prog-guide/>

Many resources here: <https://www.gpuhackathons.org/technical-resources>

Good books to get started



# Nvidia GPU computing universe

GPU Computing Applications						
Libraries and Middleware						
cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)	
CUDA-Enabled NVIDIA GPUs						
NVIDIA Ampere Architecture (compute capabilities 8.x)					Tesla A Series	
NVIDIA Turing Architecture (compute capabilities 7.x)			GeForce 2000 Series	Quadro RTX Series	Tesla T Series	
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier			Quadro GV Series	Tesla V Series	
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2		GeForce 1000 Series	Quadro P Series	Tesla P Series	
	 Embedded	 Consumer Desktop/Laptop	 Professional Workstation	 Data Center		

Source: CUDA C programming guide <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

# Nvidia CUDA development tools

## CUDA Toolkit/SDK (free)

- CUDA C compiler (nvcc)
- Libraries (cuBLAS, cuFFT, cuDNN, cuRAND, cuSPARSE, cuSOLVER, Thrust, CUDA Math lib)
- Debugging tools (CUDA-gdb, CUDA-memcheck)
- Profiling tools (nvprof, nvvp, Nsight Systems/Compute)
- Code samples (now available at <https://github.com/NVIDIA/cuda-samples>)
- <https://developer.nvidia.com/cuda-zone>
- <https://nvidia.com/getcuda>

## Activate on Expanse GPU nodes

```
$> module reset  
$> module load cuda11.7/toolkit
```

# Nvidia CUDA development tools

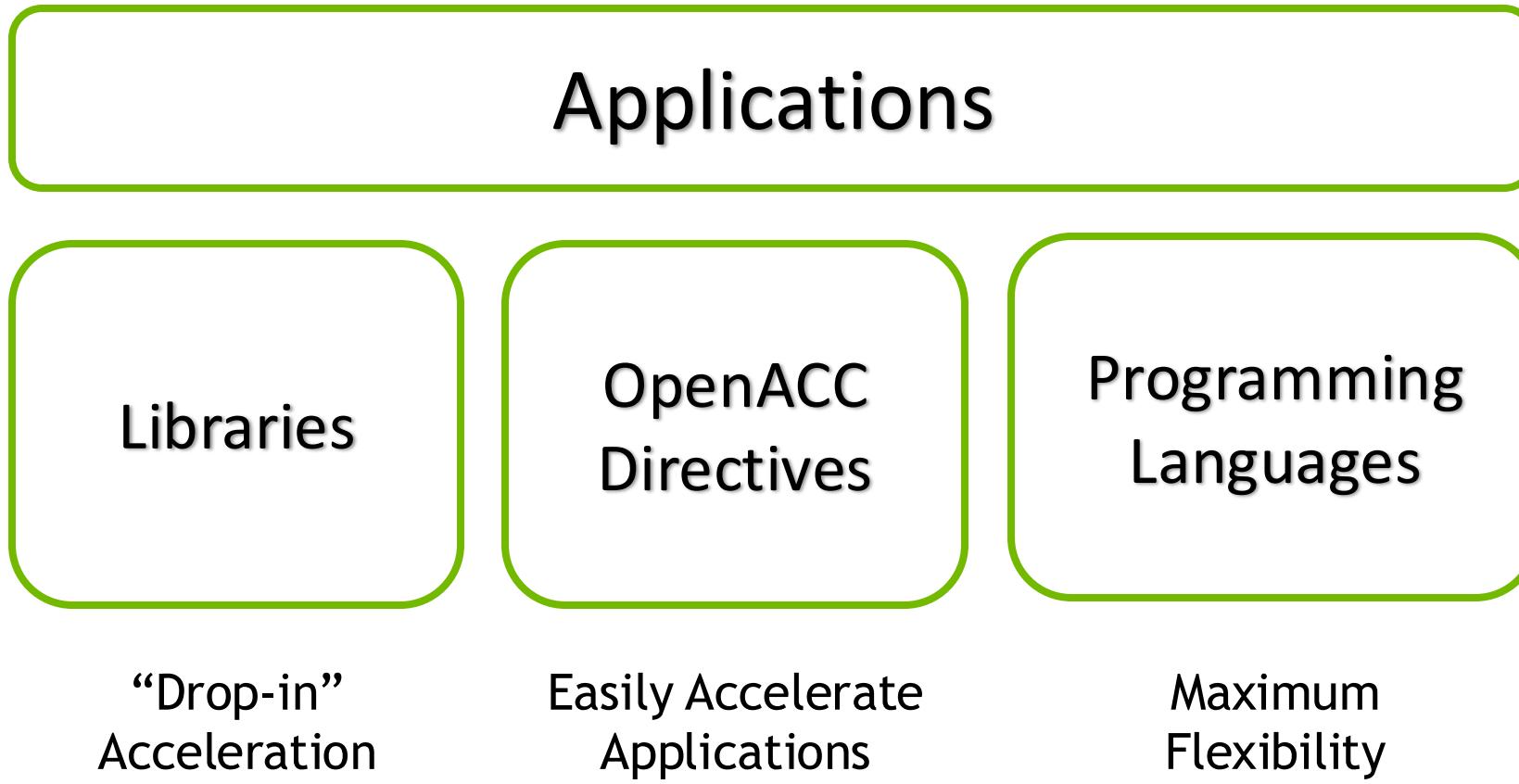
## Nvidia HPC SDK (free)

- Replaces the CUDA Toolkit
- Contains most of CUDA Toolkit including CUDA compiler nvcc, libraries, debuggers, profiler
- Nvidia C/C++, Fortran compiler (nvfortran, nvc, nvc++) (formerly PGI compilers)
- <https://developer.nvidia.com/hpc-sdk>

## Activate on Expanse GPU nodes

```
$> module reset  
$> module load nvhpc/21.9
```

# 3 ways to program GPUs



**Note: Deep Learning frameworks like Tensorflow and PyTorch come with built-in GPU support**

- This means you do not have to install or use the CUDA toolkit separately

# PROGRAMMING THE NVIDIA PLATFORM

CPU, GPU, and Network

## ACCELERATED STANDARD LANGUAGES

ISO C++, ISO Fortran

```
std::transform(par, x, x+n, y, y,
    [=](float x, float y){ return y +  
        a*x; })  
;
```

```
do concurrent (i = 1:n)  
    y(i) = y(i) + a*x(i)  
enddo
```

```
import cunumeric as np  
...  
def saxpy(a, x, y):  
    y[:] += a*x
```

## INCREMENTAL PORTABLE OPTIMIZATION

OpenACC, OpenMP

```
#pragma acc data copy(x,y) {  
...  
std::transform(par, x, x+n, y, y,  
    [=](float x, float y){  
        return y + a*x;  
});  
...  
}  
  
#pragma omp target data map(x,y) {  
...  
std::transform(par, x, x+n, y, y,  
    [=](float x, float y){  
        return y + a*x;  
});  
...  
}
```

## PLATFORM SPECIALIZATION

CUDA

```
__global__  
void saxpy(int n, float a,  
           float *x, float *y) {  
    int i = blockIdx.x*blockDim.x +  
            threadIdx.x;  
    if (i < n) y[i] += a*x[i];  
}  
  
int main(void) {  
    ...  
    cudaMemcpy(d_x, x, ...);  
    cudaMemcpy(d_y, y, ...);  
  
    saxpy<<<(N+255)/256,256>>>(...);  
  
    cudaMemcpy(y, d_y, ...);
```

## ACCELERATION LIBRARIES

Core

Math

Communication

Data Analytics

AI

Quantum

# Training overview

## We will cover the following topics

- GPU hardware overview
- Accessing SDSC Expanse GPU nodes
- GPU accelerated software examples
- Programming GPUs – Overview for Nvidia GPUs
- **GPU enabled libraries**
  - 15 minutes break –
- CUDA C programming basics
- Hands-on exercises on SDSC Expanse – CUDA
- OpenACC introduction
- Hands-on exercises on SDSC Expanse – OpenACC

# GPU accelerated libraries

## Ease of use

- GPU acceleration without in-depth knowledge of GPU programming

## “Drop-in”

- Many GPU accelerated libraries follow standard APIs
- Minimal code changes required

## Quality

- High-quality implementations of functions encountered in a broad range of applications

## Performance

- Libraries are tuned by experts

=> Use if you can – (do not write your own matrix multiplication)

# GPU accelerated libraries

See <https://developer.nvidia.com/gpu-accelerated-libraries>

## Deep Learning Libraries



GPU-accelerated library of primitives for deep neural networks



GPU-accelerated neural network inference library for building deep learning applications



Advanced GPU-accelerated video inference library

## Signal, Image and Video Libraries



cuFFT

GPU-accelerated library for Fast Fourier Transforms



NVIDIA Performance Primitives

GPU-accelerated library for image and signal processing



NVIDIA Codec SDK

High-performance APIs and tools for hardware accelerated video encode and decode

## Linear Algebra and Math Libraries



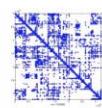
cuBLAS

GPU-accelerated standard BLAS library



CUDA Math Library

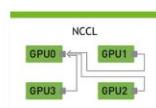
GPU-accelerated standard mathematical function library



cuSPARSE

GPU-accelerated BLAS for sparse matrices

## Parallel Algorithm Libraries



NCCL

Collective Communications Library for scaling apps across multiple GPUs and nodes



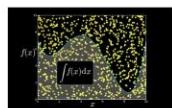
nvGRAPH

GPU-accelerated library for graph analytics



Thrust

GPU-accelerated library of parallel algorithms and data structures



cuRAND

GPU-accelerated random number generation (RNG)



cuSOLVER

Dense and sparse direct solvers for Computer Vision, CFD, Computational Chemistry, and Linear Optimization applications



AmgX

GPU accelerated linear solvers for simulations and implicit unstructured methods

## Partner Libraries



OpenCV



FFmpeg



ARRAYFIRE

... and several others

# GPU accelerated libraries

## 3 steps to using libraries

- Step 1: Substitute library calls with equivalent CUDA library calls

saxpy ( ... )            cublasSaxpy ( ... )

- Step 2: Manage data locality

- with CUDA:    cudaMalloc(), cudaMemcpy(), etc.
- with CUBLAS:    cublasSetVector(), cublasGetVector()  
etc.

- Step 3: Rebuild and link the CUDA-accelerated library

nvcc myobj.o -l cublas

# CUBLAS library example

```
int N = 1 << 20;
```

```
// Perform SAXPY on 1M elements: y[] = a*x[] + y[]
saxpy(N, 2.0, d_x, 1, d_y, 1);
```

saxpy =  
single precision  
**a** times **x** plus **y**

$$y = a * x + y$$

# CUBLAS library example

```
int N = 1 << 20;  
  
// Perform SAXPY on 1M elements: d_y[] = a*d_x[] + d_y[]  
cublasSaxpy(handle, N, 2.0, d_x, 1, d_y, 1);
```

Add “cublas” prefix  
and use device  
variables

# CUBLAS library example

```
int N = 1 << 20;  
cublasCreate(&handle);
```

Initialize CUBLAS

```
// Perform SAXPY on 1M elements: d_y[] = a*d_x[] + d_y[]  
cublasSaxpy(handle, N, 2.0, d_x, 1, d_y, 1);
```

```
cublasDestroy(handle);
```

Shut down CUBLAS

# CUBLAS library example

```
int N = 1 << 20;  
cublasCreate(&handle);  
cudaMalloc((void**)&d_x, N*sizeof(float));  
cudaMalloc((void**)&d_y, N*sizeof(float));
```

Allocate device  
vectors

```
// Perform SAXPY on 1M elements: d_y[] = a*d_x[] + d_y[]  
cublasSaxpy(handle, N, 2.0, d_x, 1, d_y, 1);
```

```
cudaFree(d_x);  
cudaFree(d_y);  
cublasDestroy(handle);
```

Deallocate device  
vectors

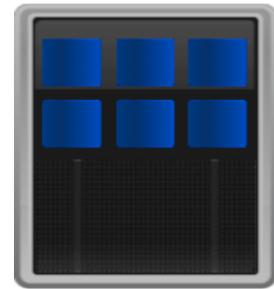
# CUBLAS library example

```
int N = 1 << 20;  
cublasCreate(&handle);  
cudaMalloc((void**)&d_x, N*sizeof(float));  
cudaMalloc((void**)&d_y, N*sizeof(float));  
  
cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1); ◀ Transfer data to GPU  
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);  
  
// Perform SAXPY on 1M elements: d_y[] = a * d_x[] + d_y[]  
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);  
  
cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1); ◀ Read data back from GPU  
  
cublasFree(d_x);  
cublasFree(d_y);  
cublasDestroy(handle);
```

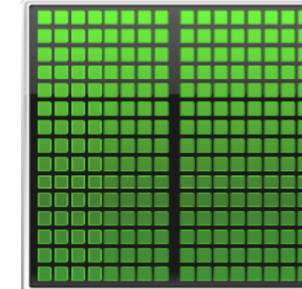
# CUBLAS library example

```
int N = 1 << 20;  
cublasCreate(&handle);  
cudaMalloc((void**)&d_x, N*sizeof(float));  
cudaMalloc((void**)&d_y, N*sizeof(float));  
  
cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);  
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);  
  
// Perform SAXPY on 1M elements: d_y[] = a * d_x[] + d_y[]  
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);  
  
cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);  
  
cublasFree(d_x);  
cublasFree(d_y);  
cublasDestroy(handle);
```

# NUMERICAL COMPUTING IN PYTHON



- Mathematical focus
- Operates on arrays of data
  - *ndarray*, holds data of same type
- Many years of development
- Highly tuned for CPUs



- NumPy like interface
- Trivially port code to GPU
- Copy data to GPU
  - CuPy *ndarray*
- Data interoperability with DL frameworks, RAPIDS, and Numba
- Uses high tuned NVIDIA libraries
- Can write custom CUDA functions

# CUPY

A NumPy like interface to GPU-acceleration ND-Array operations

**BEFORE**

```
import numpy as np  
  
size = 4096  
A = np.random.randn(size,size)  
  
Q, R = np.linalg.qr(A)
```

**AFTER**

```
import cupy as np  
  
size = 4096  
A = np.random.randn(size,size)  
  
Q, R = np.linalg.qr(A)
```



*52x Speedup!*



# Training overview

## We will cover the following topics

- GPU hardware overview
- Accessing SDSC Expanse GPU nodes
- GPU accelerated software examples
- Programming GPUs – Overview for Nvidia GPUs
- GPU enabled libraries
  - **15 minutes break** –
- CUDA C programming basics
- Hands-on exercises on SDSC Expanse – CUDA
- OpenACC introduction
- Hands-on exercises on SDSC Expanse – OpenACC

# Training overview

## We will cover the following topics

- GPU hardware overview
- Accessing SDSC Expanse GPU nodes
- GPU accelerated software examples
- Programming GPUs – Overview for Nvidia GPUs
- GPU enabled libraries
  - 15 minutes break –
- **CUDA C programming basics**
- Hands-on exercises on SDSC Expanse – CUDA
- OpenACC introduction
- Hands-on exercises on SDSC Expanse – OpenACC

# Nvidia CUDA

See <https://developer.nvidia.com/cuda-zone>

## CUDA C

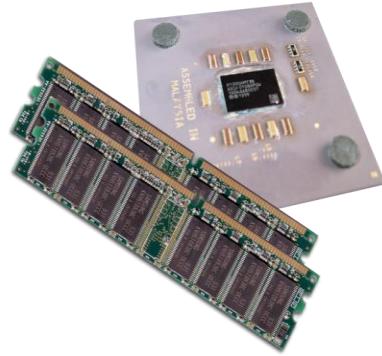
- Solution to run C seamlessly on GPUs (Nvidia only)
- De-facto standard for high-performance code on Nvidia GPUs
- Nvidia proprietary
- Modest extensions but major rewriting of code

## CUDA Fortran

- Supports CUDA extensions in Fortran, developed by Portland Group Inc (PGI)
- Available in the Nvidia Fortran compiler (formerly PGI Fortran Compiler)
- PGI is now part of Nvidia

# Heterogeneous Computing

- **Host** The CPU and its memory
- **Device** The GPU and its memory
- Device code is launched from Host code

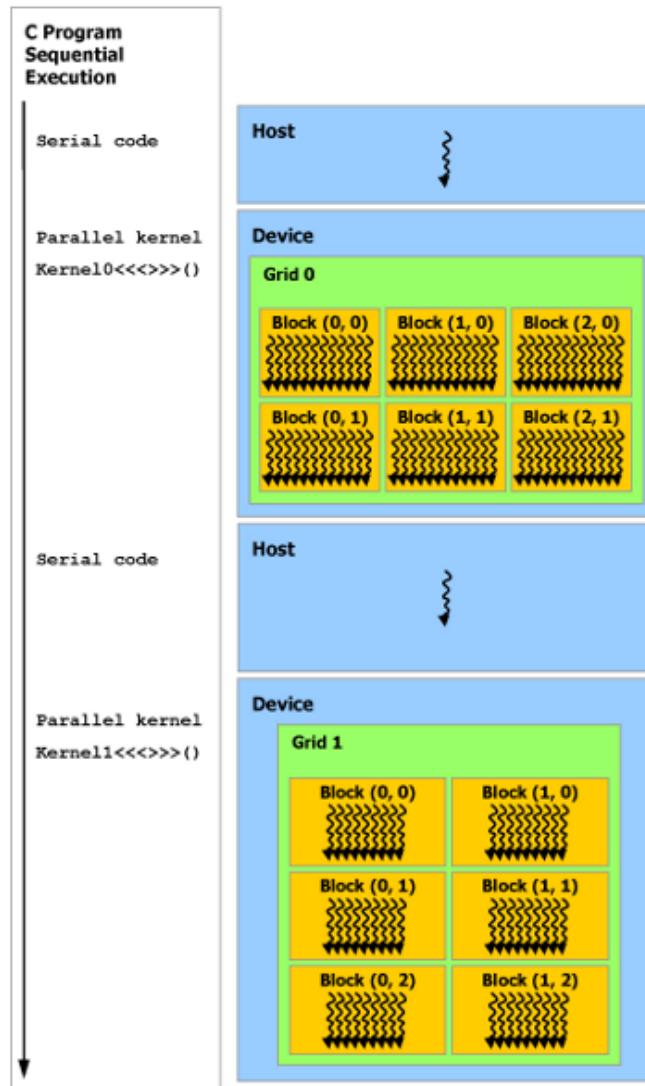


*Host*

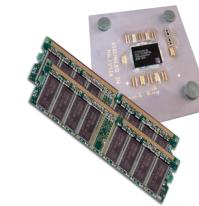


*Device*

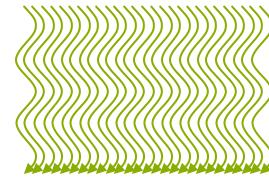
# Heterogeneous Computing



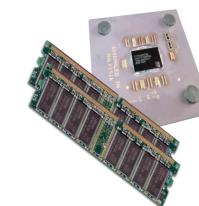
*serial code*



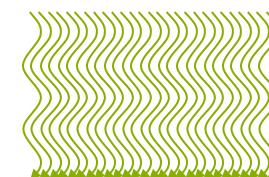
*parallel code*



*serial code*

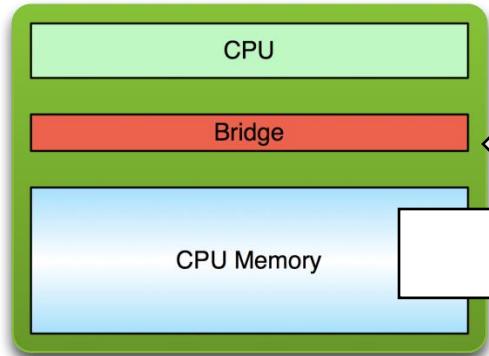


*parallel code*

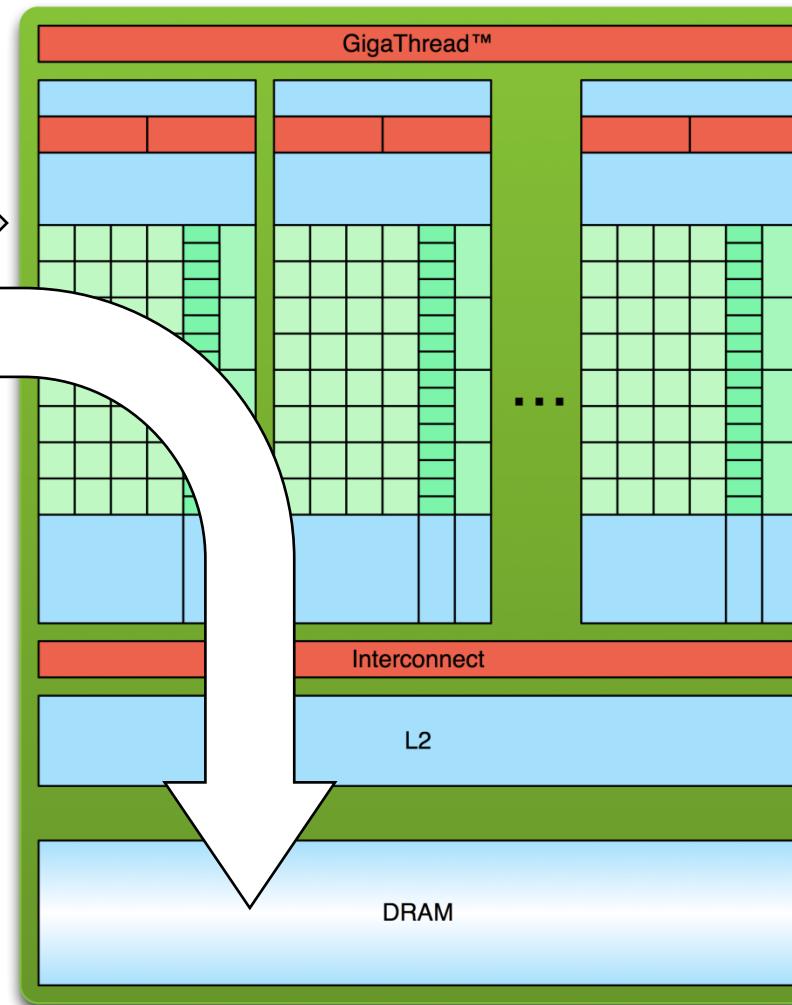


# Processing Flow

Host



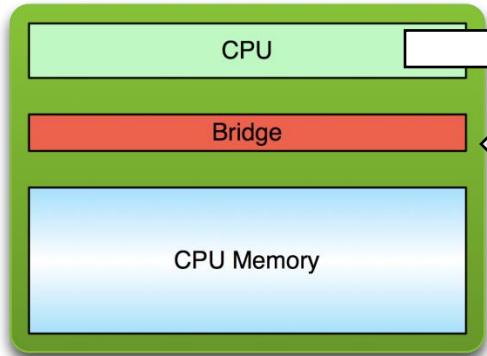
Device



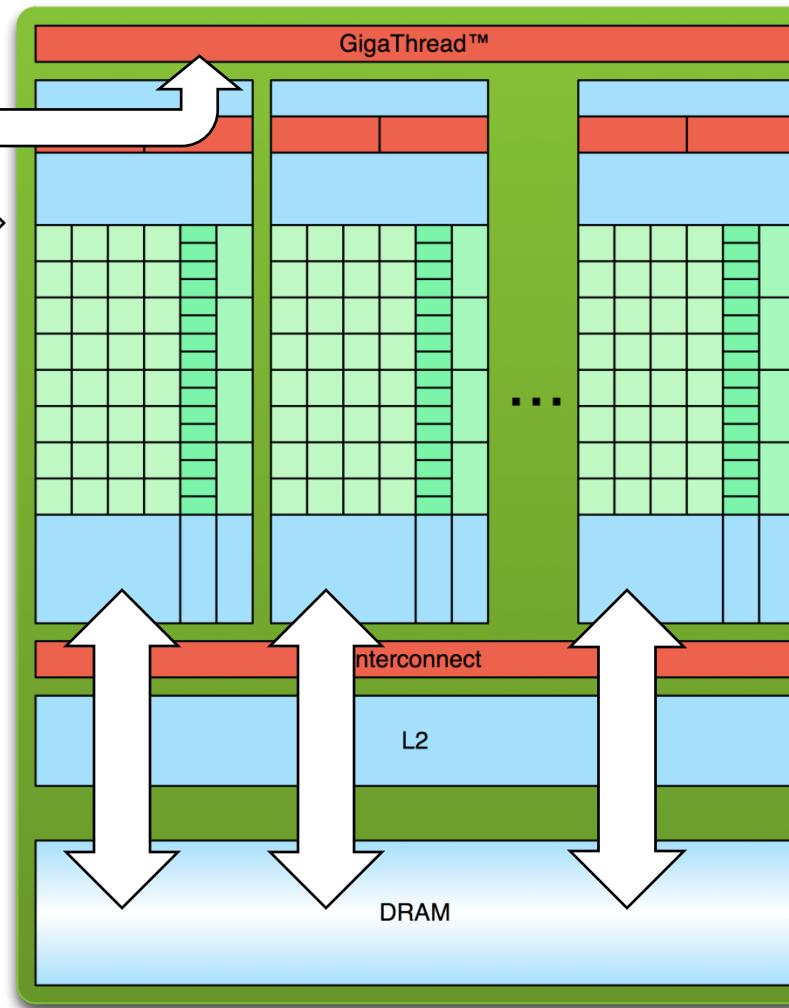
1. Copy input data from CPU memory to GPU memory

# Processing Flow

Host



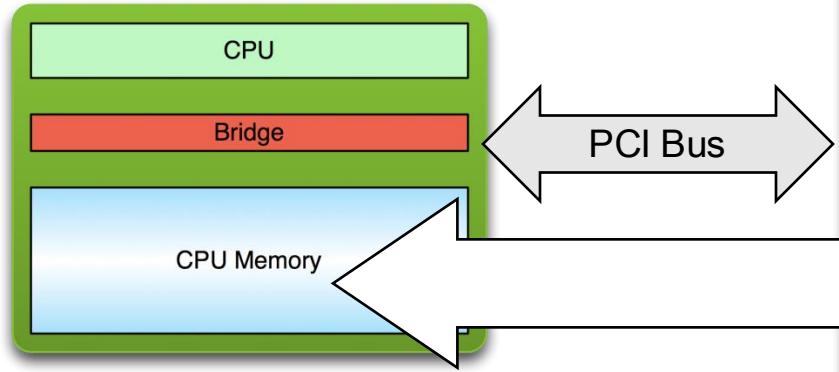
Device



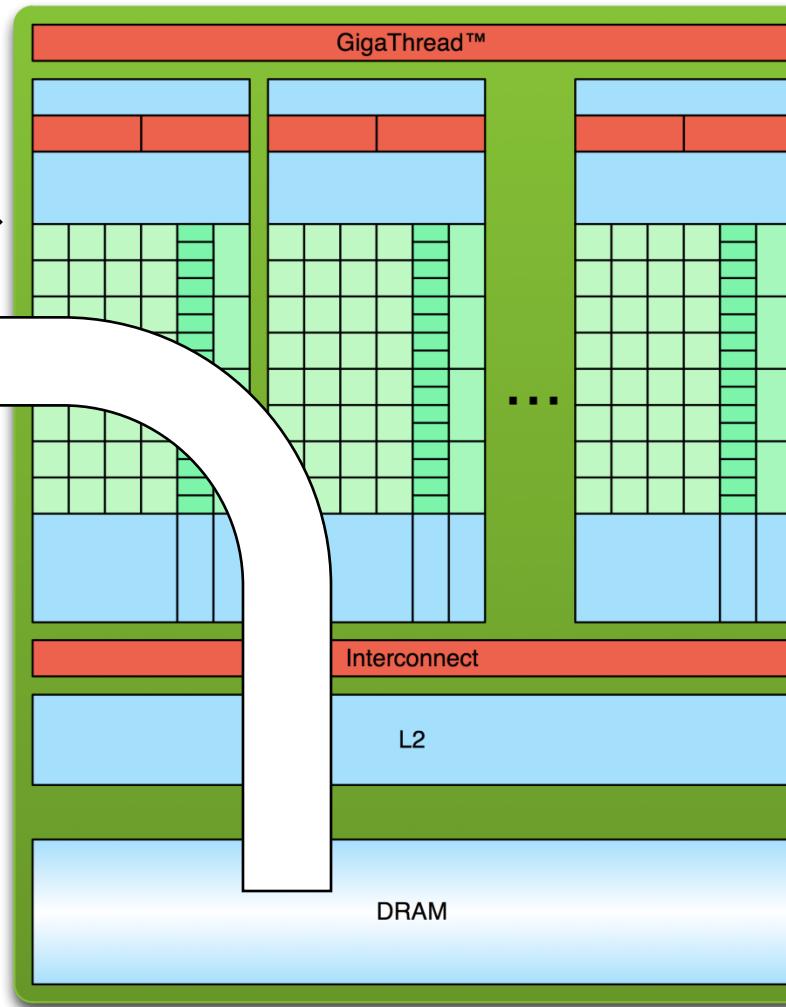
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

# Processing Flow

Host



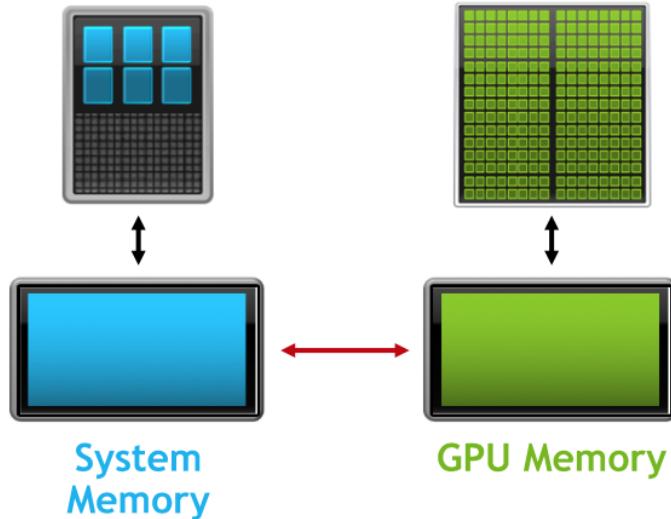
Device



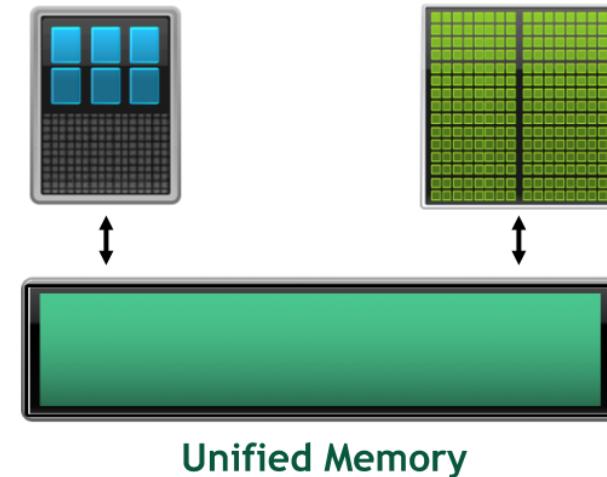
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

# Unified memory

Developer view so far



Developer view with CUDA Unified Memory



- Pool of managed memory that is shared between host and device
- Primarily productivity feature
- Memory copies still happen under the hood

# Hello World!

## Standard C code

```
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- Standard C code that runs on the host
- Compile with NVIDIA CUDA compiler (nvcc)
  - nvcc separates into host and device code
  - Host code is compiled by host compiler

```
$> nvcc hello_world_cpu.cu
$> ./a.out
Hello World!
```

## Standard C code with CUDA extensions

```
__global__ void my_kernel(void) {
}

int main(void) {
    my_kernel<<<16,32>>>();
    printf("Hello World!\n");
    return 0;
}
```

- Contains code that is executed on the device (though doing nothing)
- Two new syntactic elements...

# Hello World!

- CUDA C keyword `__global__` indicates a function that
  - runs on the device (and must return `void`)
  - can be called from the host code

```
__global__ void my_kernel(void) {  
}
```

- `nvcc` separates source code into host and device components
  - device functions processed by `nvcc`
  - host functions processed by standard C compiler, e.g. `gcc`

# Hello World!

- Triple angle brackets mark a call from host code to device code
  - Called kernel launch
  - Parameters in brackets are kernel launch configuration (explained later)

```
__global__ void my_kernel(void) {  
}  
  
int main(void) {  
    my_kernel<<<16,32>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- The kernel `my_kernel()` does not compute anything ...
- Let's look at writing code to be executed on the GPU

# Addition on the device

```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

- Remember: `__global__` is a CUDA C keyword, thus
  - `add()` will execute on the device
  - `add()` will be called from the host
- Thus `a`, `b` and `c` must point to device memory

# Memory management

- Host and device memory are separate
  - Host pointers point to CPU memory
    - Can be passed to/from device code
    - Cannot be dereferenced in device code!
  - Device pointers point to GPU memory
    - Can be passed to/from host code
    - Cannot be dereferenced in host code!
- CUDA API handles device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - equivalent to C `malloc()`, `free()`, `memcpy()`



# Addition on the device

```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

- How do we reserve memory on the device?
- How do we transfer data from the host to the device?
- This happens in the host C code that launches the kernel. Let's see how this works...

# Addition on the device

```
int main(void){  
    int h_a, h_b, h_c;      // host copies  
    int *d_a, *d_b, *d_c; // device copies  
    int size = sizeof(int);  
  
    // Allocate memory on device  
    cudaMalloc((void **) &d_a, size);  
    cudaMalloc((void **) &d_b, size);  
    cudaMalloc((void **) &d_c, size);  
  
    // Setup input values  
    h_a = 5;  
    h_b = 7;  
  
    // Copy input data to device  
    cudaMemcpy(d_a, &h_a, size,  
              cudaMemcpyHostToDevice);  
    cudaMemcpy(d_b, &h_b, size,  
              cudaMemcpyHostToDevice);  
  
    // Launch add() kernel  
    add<<<1,1>>>(d_a, d_b, d_c);  
  
    // Copy results back to host  
    cudaMemcpy(&h_c, d_c, size,  
              cudaMemcpyDeviceToHost);  
  
    // Deallocate memory  
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);  
  
    printf("%d + %d = %d", h_a, h_b, h_c);  
    return 0;  
}
```

# Basic CUDA workflow

- Allocate memory on the device
- Copy input data to the device
- Launch CUDA kernel on the device
- Copy results back to the host
- Deallocate memory on the device

Let's move on to parallel computing on the GPU using CUDA

# Parallelization with CUDA

- GPU computing is about massive parallelism
  - How do we run code in parallel using CUDA?
- Instead of executing the kernel add() once, we execute it N times in parallel
- The **central idea defining GPU computing**:
  - Kernels look like serial programs
  - Write programs as if they run on a single thread
  - The GPU will run that program on many threads

# Parallelization with CUDA

- Executing `add()`  $N$  times

```
add<<<1,1>>>(); // launch 1 copy  
↓  
add<<<N,1>>>(); // launch N copies
```

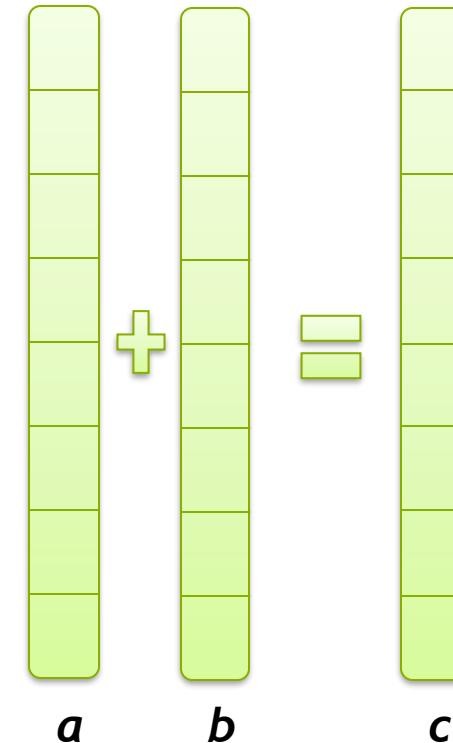
- The GPU is good at
  - efficiently launching lots of threads
  - running lots of threads in parallel  
(many more than processors on the device)
- But why launch  $N$  identical copies?

# Vector addition

- CPU code (serial) uses a loop

```
#define N 512
void vadd_cpu(int *a, int *b, int *c){
    int i;
    for (i=0; i<N; i++) {
        c[i] = a[i] + b[i];
    }
}
```

- The GPU does this in parallel by running N copies of the `add()` kernel, each copy working on a different vector element



# Parallel vector addition (1)

- Parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c){  
    int tid = blockIdx.x;  
    c[tid] = a[tid] + b[tid];  
}
```

- Each parallel invocation of `add()` is called a **block**. The set of blocks is called a **grid**
- Each kernel instance knows its block index
- By using its index, each block operates on different data

# Parallel vector addition (1)

- We launch N blocks of the `add()` kernel

```
// launch N copies
add<<<N,1>>>(d_a, d_b, d_c);
```

- Kernel call needs to be consistent with kernel implementation
- On the device, each block can execute in parallel,  
depending on the number and type of available multiprocessors

Block 0

Block 1

Block 2

Block 3

```
c[0] = a[0] + b[0];
```

```
c[1] = a[1] + b[1];
```

```
c[2] = a[2] + b[2];
```

```
c[3] = a[3] + b[3];
```

# Parallel vector addition (1)

```
// CUDA kernel for vector addition
__global__ void add(int *a, int *b, int *c) {
    int tid = blockIdx.x;
    c[tid] = a[tid] + b[tid];
}

#define N 512
int main(void) {
    int h_a[N], h_b[N], h_c[N]; // host copies
    int *d_a, *d_b, *d_c;      // device copies
    int size = N * sizeof(int);

    // Allocate memory on device
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);
```

# Parallel vector addition (1)

```
// Setup input values
get_input_vectors(h_a, h_b);

// Copy input data to device
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

// Launch N blocks of the add() kernel
add<<<N,1>>>(d_a, d_b, d_c);

// Copy results back to host
cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

// Deallocate memory
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);

return 0;
}
```

# CUDA blocks and threads

- Blocks contain **threads** executing in parallel
- Parallelized **add()** kernel **using threads**

```
__global__ void add(int *a, int *b, int *c){  
    int tid = threadIdx.x;  
    c[tid] = a[tid] + b[tid];  
}
```

- Each kernel instance knows its index
- Parallel kernel call

```
// launch N copies  
add<<<1,N>>>(d_a, d_b, d_c);
```

# CUDA blocks and threads

- Blocks contain **threads** executing in parallel
- Parallelized **add()** kernel **using threads**

```
__global__ void add(int *a, int *b, int *c) {
    int tid = threadIdx.x;
    c[tid] = a[tid] + b[tid];
}
```

- Each kernel instance knows its index
- Parallel kernel call

```
// launch N copies
add<<<1,N>>>(d_a, d_b, d_c);
```

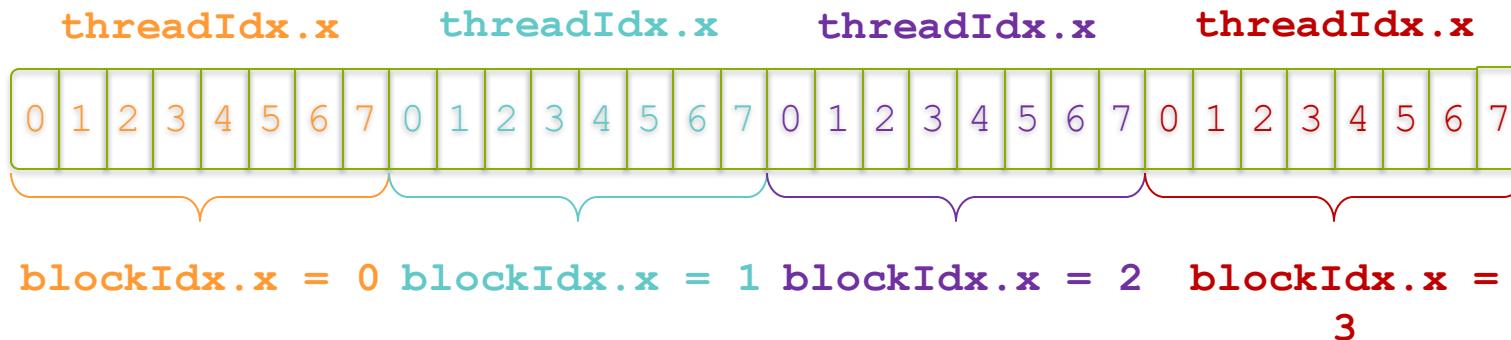
- We can combine blocks and threads
  - need to take care with indexing

- CUDA supports 3D blocks and threads (more later)
- Number of allowed threads per block and concurrent blocks is limited by hardware (up to 2048 threads per block on current GPUs)
- Threads and blocks map to the underlying hardware
- All threads in a block are guaranteed to execute on the same streaming multiprocessor (SM), thus sharing resources
- Threads within a block can communicate and synchronize
- **Note:** Thread block size should be a multiple of warp size (32) for performance reasons since kernels issue instructions in warps

# Indexing with blocks and threads

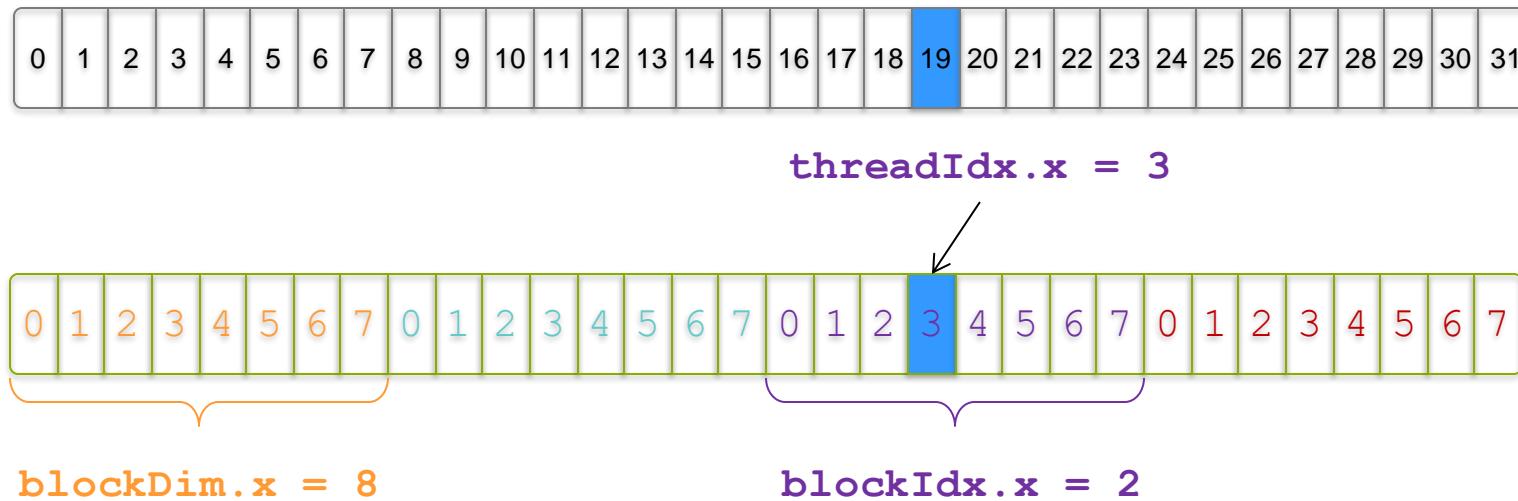
## Example to compute offset into an array

- 1 element per thread
- 8 threads per block



- Built-in variable `blockDim.x` contains the number of threads per block
  - this makes it possible to calculate a unique index (e.g. offset into an array) for each kernel

# Indexing with blocks and threads



- Calculate a unique index (Example: blue field above has index 19)

```
int tid = threadIdx.x + blockIdx.x * blockDim.x  
= 3           + 2           * 8  
= 19
```

# Parallel vector addition (2)

- Parallelized `add()` using blocks and threads

```
__global__ void add(int *a, int *b, int *c){  
    int tid = threadIdx.x + blockDim.x * blockIdx.x;  
    c[tid] = a[tid] + b[tid];  
}
```

- Corresponding kernel call

```
// launch N/TPB copies with  
// TPB threads per block  
add<<<N/TPB,TPB>>>(d_a, d_b, d_c);
```

# Handling arbitrary vector sizes (1)

- Problem size **N** is typically not a multiple of our chosen **blockDim.x**
- Launch a sufficient number of kernels

```
// launch at least N/TBP copies with
// TPB threads per block
add<<< (N+TPB-1)/TPB,TPB>>>(d_a, d_b, d_c, N);
```

- Ensure to stay within array boundaries

```
__global__ void add(int *a, int *b, int *c, int n){
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    if (tid < n)
        c[tid] = a[tid] + b[tid];
}
```

- If **n** is not a multiple of the number of threads per block **TBP**, a few threads will do no-ops

# Review

- We write a kernel that looks like it runs on one thread
- We can launch that kernel on any number of threads
  - Use `kernel<<<(N+TPB-1) / TPB, TPB>>>()`
- Each thread knows its index in the block and grid
  - use `blockIdx.x` to get the block index
  - use `blockDim.x` to get the block size
  - use `threadIdx.x` to get the thread index

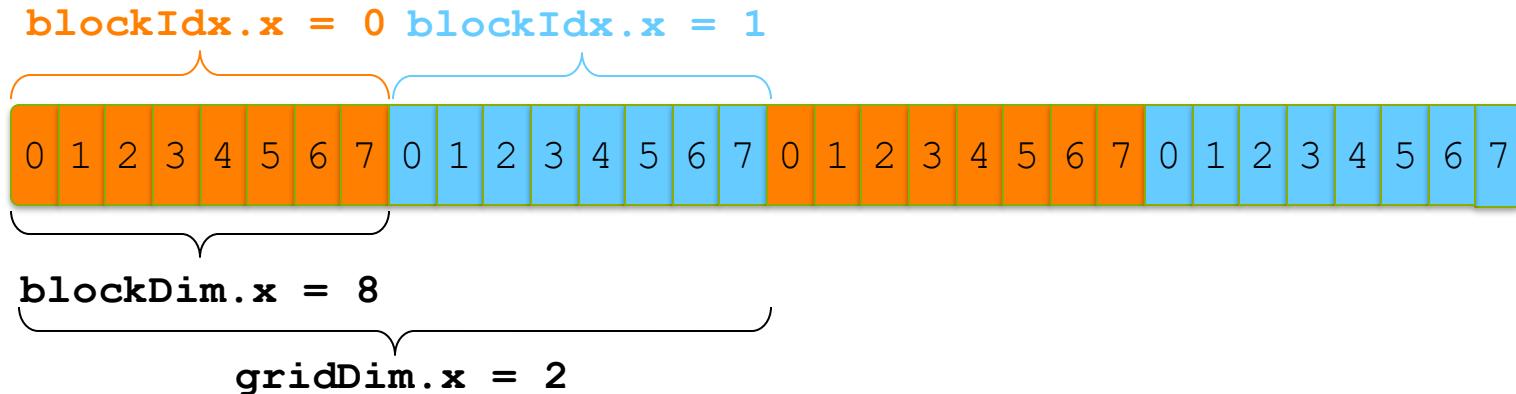
```
tid = threadIdx.x + blockIdx.x * blockDim.x
```

# Handle arbitrary vector sizes (2)

- Maximum grid and block size is limited by hardware
- We thus need to
  - launch a fixed number of blocks and threads
  - rewrite our kernel for fixed grid and block size
- Built-in variable `gridDim.x` contains number of blocks in grid
- We can use this to compute strides
- For many kernels, performance will be optimal for a GPU-hardware dependent combination of grid / block size
  - Query hardware with CUDA calls to determine optimal kernel launch configuration at runtime

# Handle arbitrary vector sizes (2)

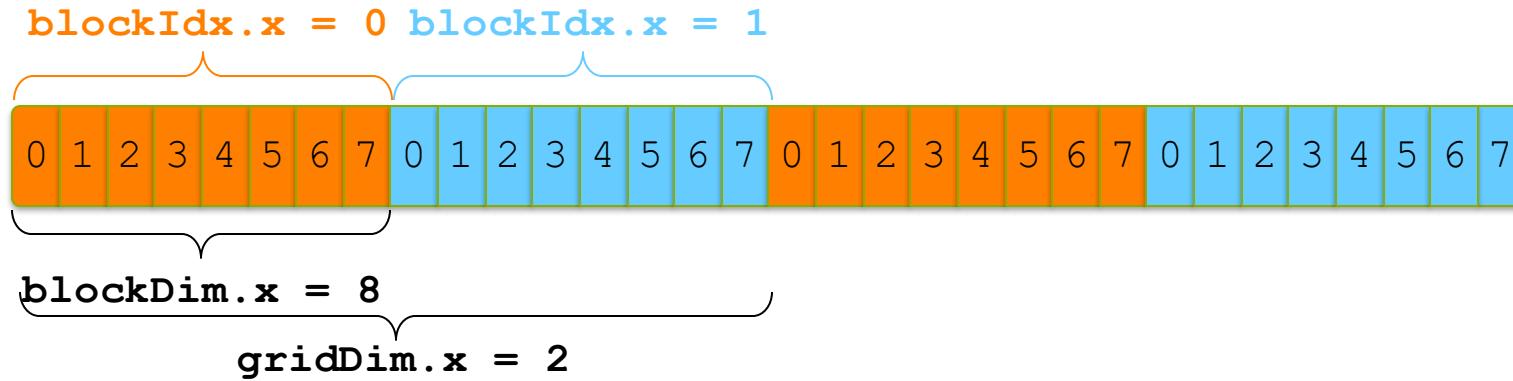
- Example: If we launch 2 blocks with 8 threads each, then kernels with
  - `blockIdx.x=0` need to work on blocks 0 and 2
  - `blockIdx.x=1` need to work on blocks 1 and 3



- Each kernel needs to know the stride required for accessing its data elements

```
int stride = blockDim.x * gridDim.x
```

## Handle arbitrary vector sizes (2)



```
__global__ void add(int *a, int *b, int *c, int n){  
    int tid = threadIdx.x + blockDim.x * blockIdx.x;  
    int stride = blockDim.x * gridDim.x;  
    while (tid < n) {  
        c[tid] = a[tid] + b[tid];  
        tid += stride  
    }  
}
```

- We can now launch a fixed number of kernels:

```
add<<<NBL,TPB>>>(d_a, d_b, d_c, N);
```

# Multi-dimensional indexing

- CUDA supports
  - 3D grids of blocks and
  - 3D blocks of threads
- Convenient for mapping multi-dimensional problems (bitmaps, matrix operations etc)
- **grid3** data type, dimensions default to 1
- Example: launch grid of **(bx\*by\*bz)** blocks of **(tx\*ty\*tz)** threads with

```
kernel<<<dim3(bx,by,bz),dim3(tx,ty,tz)>>>();
```

# Multi-dimensional indexing

- Get grid dimension from  
`gridDim.x, gridDim.y, gridDim.z`
- Get block index in grid from  
`blockIdx.x, blockIdx.y, blockIdx.z`
- Get block dimension from  
`blockDim.x, blockDim.y, blockDim.z`
- Get thread index in block from  
`threadIdx.x, threadIdx.y, threadIdx.z`
- Use these to determine data access offsets and strides

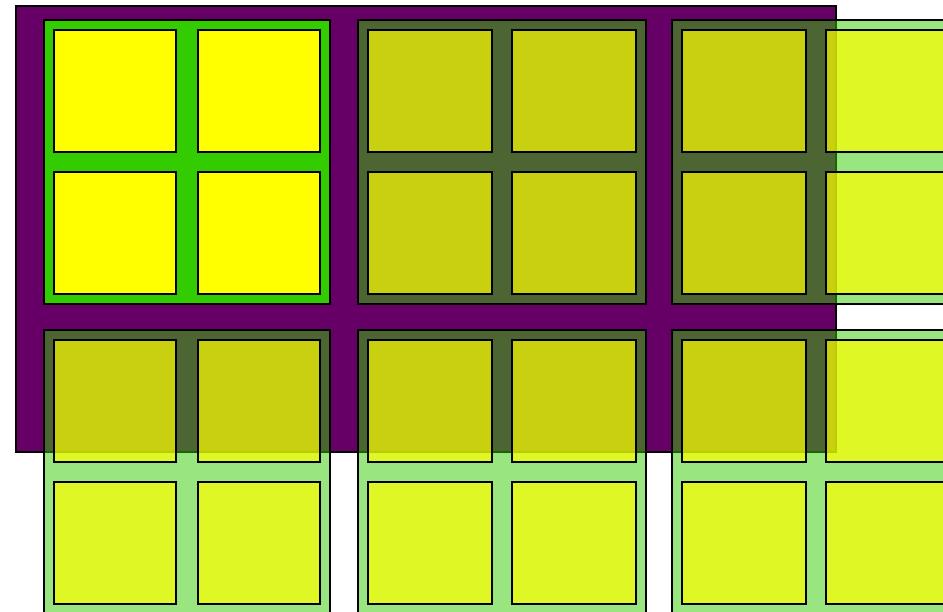
# Example: 2D array

Due to time constraints we will skip over

- 2D array example
- 1D stencil example
- Histogram calculation

I will leave these in the slides for you to look at / practice.

- Example: Kernel that squares a 2D array using a 2D grid of 2D blocks:



- Matrix (purple)
- 2D grid (green, 2x2)
- 2D blocks (yellow)
- (threads not shown)

# Example: 2D array

Skipped

- Example: Kernel that squares a 2D array using a 2D grid of 2D blocks, assuming linear storage in memory:

```
__global__ void square(int *arr, int maxrow, int maxcol){  
    // indices and strides  
    int rowinit = threadIdx.x + blockDim.x * blockIdx.x;  
    int colinit = threadIdx.y + blockDim.y * blockIdx.y;  
    int rowstride = gridDim.x * blockDim.x;  
    int colstride = gridDim.y * blockDim.y;  
  
    // operate on all 2D "submatrices"  
    for (int row = rowinit; row < maxrow; row += rowstride) {  
        for (int col = colinit; col < maxcol; col += colstride) {  
            pos = row * maxcol + col  
            arr[pos] *= arr[pos]  
        }  
    }  
}
```

# Example: 2D array

Skipped

- We can launch the kernel for example with a grid of (16\*16) blocks of (16\*16) threads, i.e. a total of  $256 \times 256 = 65,536$  concurrent threads

```
#define NROW 2048
#define NCOL 512
int main(void){
    int h_a[NROW][NCOL];           // host copy
    int *d_a;                      // device copy
    int size = NROW * NCOL * sizeof(int);

    // Allocate memory on device
    cudaMalloc((void **) &d_a, size);

    // Setup input values
    get_input_array(h_a);

    // Copy input data to device
    cudaMemcpy(d_a, h_a, size,
              cudaMemcpyHostToDevice);
}
```

```
// Launch square() kernel
dim3 gridSize(16,16);
dim3 blockSize(16,16);
square<<<gridSize,blockSize>>>(d_a, NROW, NCOL);

// Copy results back to host
cudaMemcpy(h_a, d_a, size,
           cudaMemcpyDeviceToHost);

// Deallocate memory
cudaFree(d_a);

return 0;
```

# Unified memory

Skipped

- Great to get started, simplifies programming

```
cudaMallocManaged(...);
```

- CUDA keeps track of memory location and migrates data from device to host and vice versa as required
- Developer needs to ensure that no race conditions are caused by simultaneous access to host/device memory
  - Pre-Pascal architecture will segfault; Pascal give wrong results
  - Therefore synchronize CPU/GPU (wait for kernel to finish):

```
cudaDeviceSynchronize();
```

# Example: 2D array with unified memory

Skipped

```
#define NROW 2048
#define NCOL 512
int main(void) {

    int size = NROW * NCOL * sizeof(int);
    int *array

    // Allocate managed memory, get data
    cudaMallocManaged(&array, size);
    get_input_array(array);

    // Launch square() kernel
    dim3 gridSize(16,16); dim3 blockSize(16,16);
    square<<<gridSize,blockSize>>>(array, NROW, NCOL);

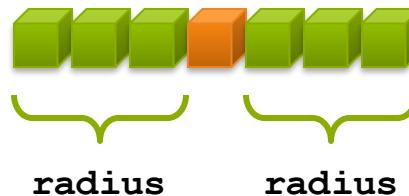
    // Wait for kernel to finish before accessing data
    cudaDeviceSynchronize();
    print_results(array);
    cudaFree(array)
}
```

# Communication among threads – shared memory

Skipped

## Example: 1D stencil

- 1D stencil for 1D array:
  - Each output element is the sum of input elements within a given radius
- If the radius is 3, then each output element is the sum of 7 input elements:



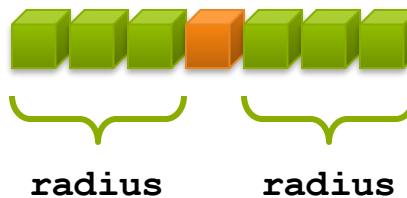
$$y'_i = y_i + \sum_{j=1}^3 y_{i-j} + \sum_{j=1}^3 y_{i+j}$$

# Communication among threads – shared memory

Skipped

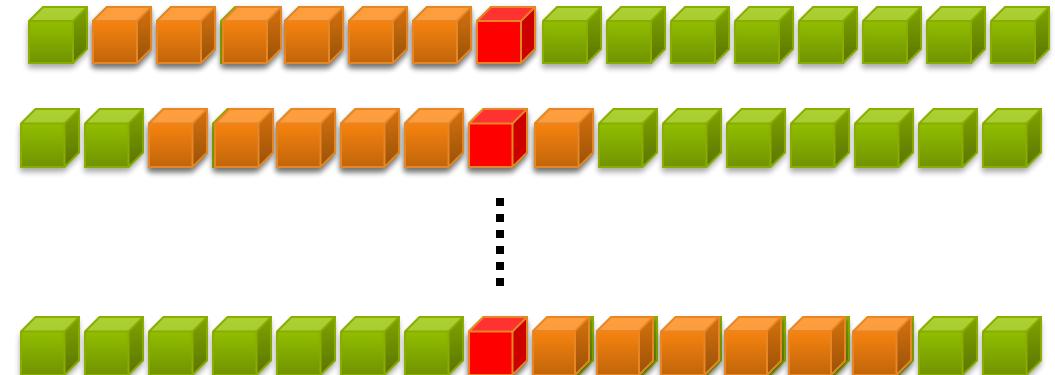
## Example: 1D stencil

- 1D stencil for 1D array:
  - Each output element is the sum of input elements within a given radius
- If the radius is 3, then each output element is the sum of 7 input elements:



$$y'_i = y_i + \sum_{j=1}^3 y_{i-j} + \sum_{j=1}^3 y_{i+j}$$

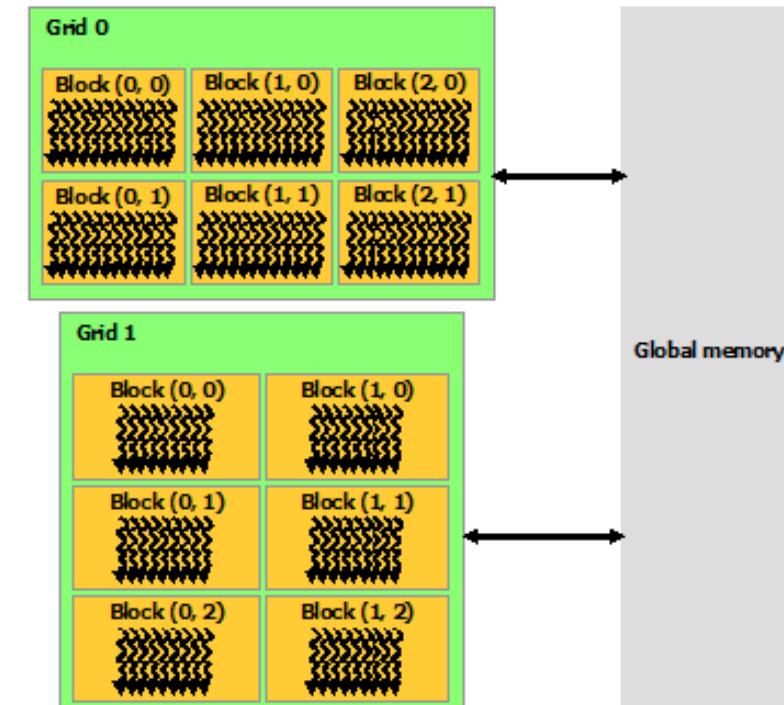
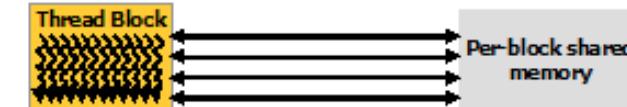
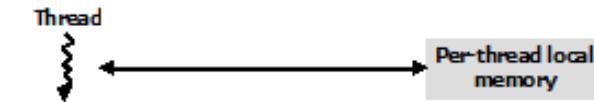
- Each thread processes one output element
  - `blockDim.x` elements are processed per block
- As a consequence, input elements have to be read several times from slow global memory
  - with radius 3, each input element is read 7 times!



# Communication among threads – shared memory

Skipped

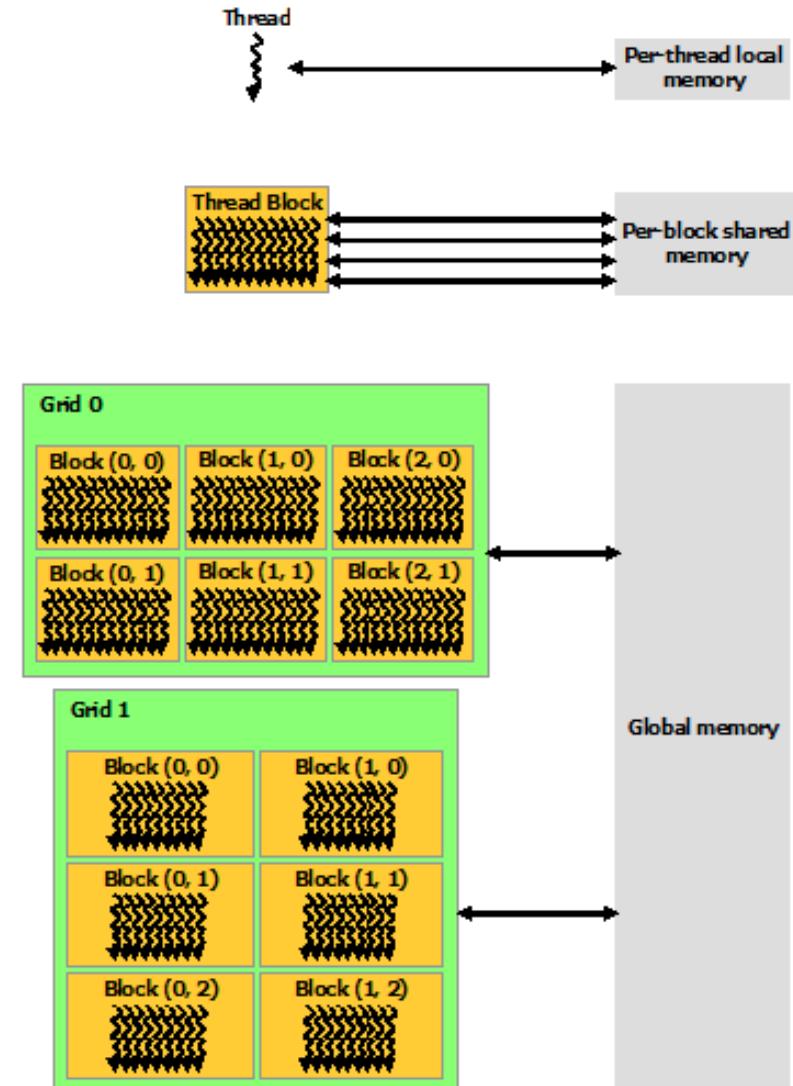
- Within a block, threads can share data via **shared memory**
- This is very fast on-chip memory
- Shared memory is user-managed
- Declare as **\_\_shared\_\_**, will be allocated per block
- Data in shared memory is not visible to other blocks



# CUDA memory hierarchy

Skipped

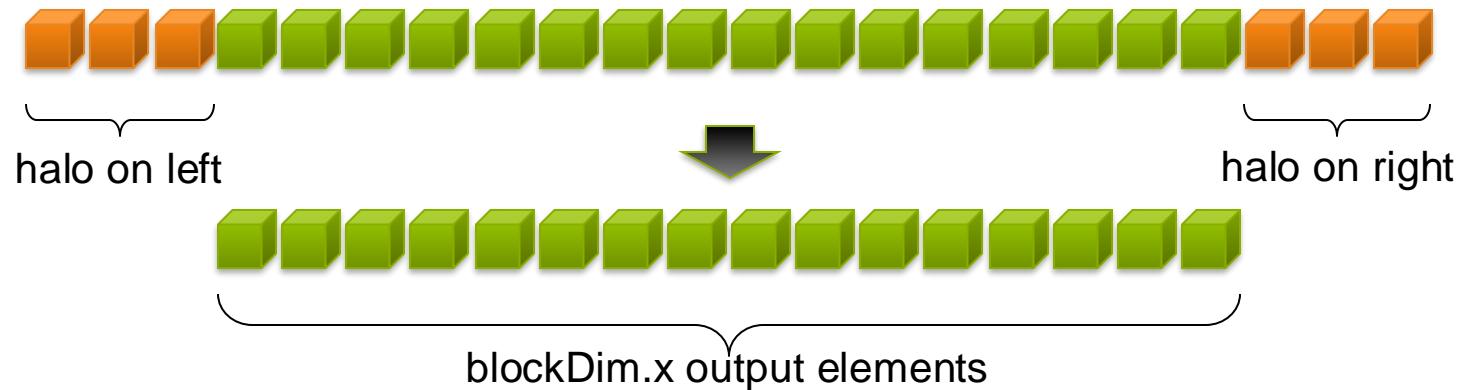
Memory	Latency (cycles)	Cached	Privacy
Global	100s	Yes	Application
Local	100s	Yes	Thread
Constant	1s-100s	Yes	Application
Texture	1s-100s	Yes	Application
Shared	1	–	Block
Register	1	–	Thread



# 1D stencil using shared memory

Skipped

- Cache data for use by different threads in shared memory
  - Read (**blockDim.x** + 2 \* **radius**) input elements from global to shared memory
  - Compute **blockDim.x** output elements
  - Write **blockDim.x** output elements to global memory
  - Each block needs a “halo” of **radius** elements at each boundary



# 1D stencil using shared memory

Skipped

Data in shared memory



```
__global__ void stencil_1D(int *in, int *out) {  
  
    __shared__ int temp[BLOCK_SIZE + 2*RADIUS];  
  
    int gindex = threadIdx.x + blockDim.x * blockIdx.x;  
    int lindex = threadIdx.x + RADIUS  
    int tid = threadIdx.x  
  
}  
}
```

# 1D stencil using shared memory

Skipped

Data in shared memory



```
__global__ void stencil_1D(int *in, int *out) {  
  
    __shared__ int temp[BLOCK_SIZE + 2*RADIUS];  
  
    int gindex = threadIdx.x + blockDim.x * blockIdx.x;  
    int lindex = threadIdx.x + RADIUS  
    int tid = threadIdx.x  
  
    // Read input elements into shared memory  
    temp[lindex] = in[gindex];  
    if (tid < RADIUS) {  
        temp[lindex - RADIUS] = in[gindex - RADIUS];  
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
    }  
  
    ...  
}
```

# 1D stencil using shared memory

Skipped

Data in shared memory



```
__global__ void stencil_1D(int *in, int *out) {  
  
    __shared__ int temp[BLOCK_SIZE + 2*RADIUS];  
  
    int gindex = threadIdx.x + blockDim.x * blockIdx.x;  
    int lindex = threadIdx.x + RADIUS  
    int tid = threadIdx.x  
  
    // Read input elements into shared memory  
    temp[lindex] = in[gindex];  
    if (tid < RADIUS) {  
        temp[lindex - RADIUS] = in[gindex - RADIUS];  
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
    }  
    // Apply the stencil  
    int result = 0;  
    for (int offset = -RADIUS; offset <= RADIUS; offset++) {  
        result += temp[lindex + offset];  
    }  
    // Store the result  
    out[gindex] = result;  
}
```

# 1D stencil using shared memory

```
__global__ void stencil_1D(int *in, int *out) {  
  
    __shared__ int temp[BLOCK_SIZE + 2*RADIUS];  
  
    int gindex = threadIdx.x + blockDim.x * blockIdx.x;  
    int lindex = threadIdx.x + RADIUS  
    int tid = threadIdx.x  
  
    // Read input elements into shared memory  
    temp[lindex] = in[gindex];  
    if (tid < RADIUS) {  
        temp[lindex - RADIUS] = in[gindex - RADIUS];  
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
    }  
    // Apply the stencil  
    int result = 0;  
    for (int offset = -RADIUS; offset <= RADIUS; offset++) {  
        result += temp[lindex + offset];  
    }  
    // Store the result  
    out[gindex] = result;  
}
```

Skipped

Data in shared memory



Unfortunately the code as it is will not work – Why ?

# 1D stencil using shared memory

Skipped

- We have a data race!
- Suppose thread 15 reads the halo before thread 0 has fetched it:

```
// Read input elements into shared memory
temp[lindex] = in[gindex]; // store at temp[18]
```



```
if (tid < RADIUS) {           // skipped by thread 15 (tid > RADIUS)
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
```

```
int result = 0;
result += temp[lindex+1]; // load from temp[19]
```



# Thread synchronization in a block

Skipped

- To avoid race conditions we need to synchronize our threads in the block
  - used to prevent RAW / WAR / WAW hazards
- `void __syncthreads();`
- All threads in the block must reach the barrier
  - Similar to `MPI_Barrier()`
  - In conditional code, the condition must be uniform across the block to avoid deadlocks

# 1D stencil kernel using shared memory

Skipped

```
__global__ void stencil_1D(int *in, int *out) {  
  
    __shared__ int temp[BLOCK_SIZE + 2*RADIUS];  
  
    int gindex = threadIdx.x + blockDim.x * blockIdx.x;  
    int lindex = threadIdx.x + RADIUS  
    int tid = threadIdx.x  
  
    // Read input elements into shared memory  
    temp[lindex] = in[gindex];  
    if (tid < RADIUS) {  
        temp[lindex - RADIUS] = in[gindex - RADIUS];  
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
    }  
  
    // Synchronize to ensure that all data is available  
    __syncthreads();
```

# 1D stencil kernel using shared memory

Skipped

```
// Synchronize to ensure that all data is available
__syncthreads();

// Apply the stencil
int result = 0;
for (int offset = -RADIUS; offset <= RADIUS; offset++) {
    result += temp[lindex + offset];
}
// Store the result
out[gindex] = result;
}
```

This kernel

- improves performance by using shared memory
- avoids race conditions by synchronizing the threads within a block

# 1D stencil kernel using shared memory

Skipped

```
// Synchronize to ensure that all data is available
__syncthreads();

// Apply the stencil
int result = 0;
for (int offset = -RADIUS; offset <= RADIUS; offset++) {
    result += temp[lindex + offset];
}
// Store the result
out[gindex] = result;
}
```

This kernel

- improves performance by using shared memory
- avoids race conditions by synchronizing the threads within a block

- Use `__shared__` to declare a variable / array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks
- Use `__syncthreads()` as a barrier
  - Required to prevent data hazards / race conditions

# Constant memory

Skipped

In addition to shared memory, there is constant memory

- Read-only during kernel execution
- Located off-chip in global memory but accessed via dedicated hardware
  - broadcasts to all threads in a half-warp (16 threads), saving bandwidth
  - cached
- Define constant memory
  - `__constant__ int c_a[dimension];`
- Copy data into constant memory
  - `cudaMemcpyToSymbol(c_a, h_a, size);`

# CUDA basics summary

## Kernel

- In CUDA, a kernel is code (typically a function), that can be executed on the GPU.
- The kernel code operates in lock-step on the multiprocessors of the GPU.  
(In so-called warps, currently consisting of 32 threads)
- SIMD – single instruction multiple threads

## Thread

- A thread is an execution of a kernel with a given index.
- Each thread uses its index to access a subset of data (e.g. array) to operate on.

## Block

- Threads are grouped into blocks, which are guaranteed to execute on the same multiprocessor.
- Threads within a thread block can synchronize and share data

## Grid

- Thread blocks are arranged into a grid of blocks.
- The number of threads per block times the number of blocks gives the total number of running threads.

# CUDA basics summary

## Threads, blocks, grids, warps

### Grids

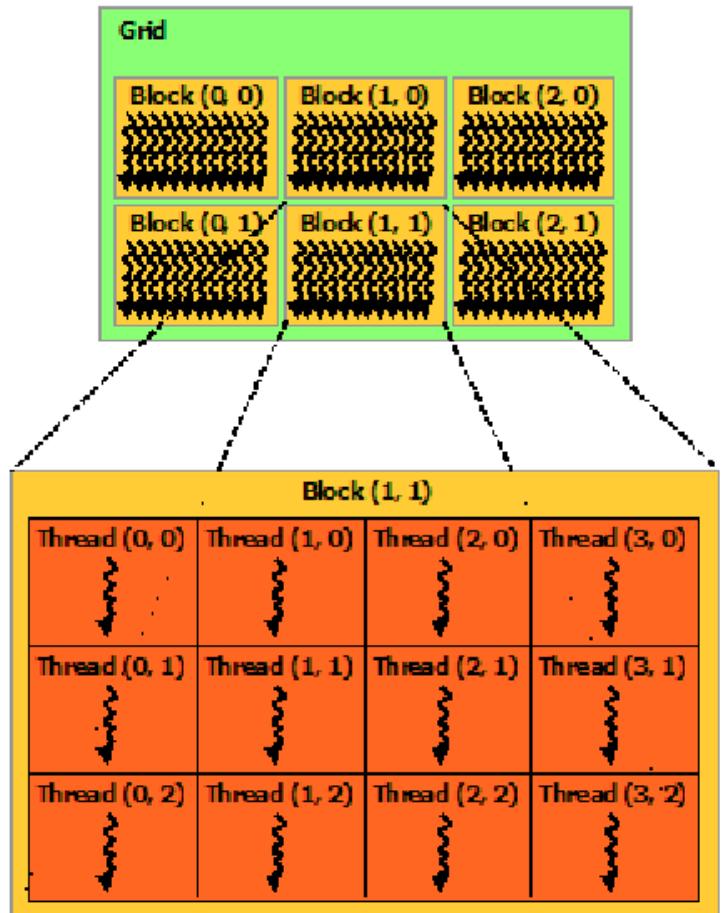
- Grids map to GPUs

### Blocks

- Blocks map to the multiprocessors (MP)
- Blocks are never split across MPs
- Multiple blocks can execute simultaneously on an MP

### Threads

- Threads are executed on stream processors (GPU cores)
- Warps are groups of threads that execute simultaneously, in lock-step (currently 32, not guaranteed to remain fixed).



# CUDA basics summary

## CUDA built-in variables

- Following variables allow to compute the ID of each individual thread that is executing in a grid block.

## Block indexes

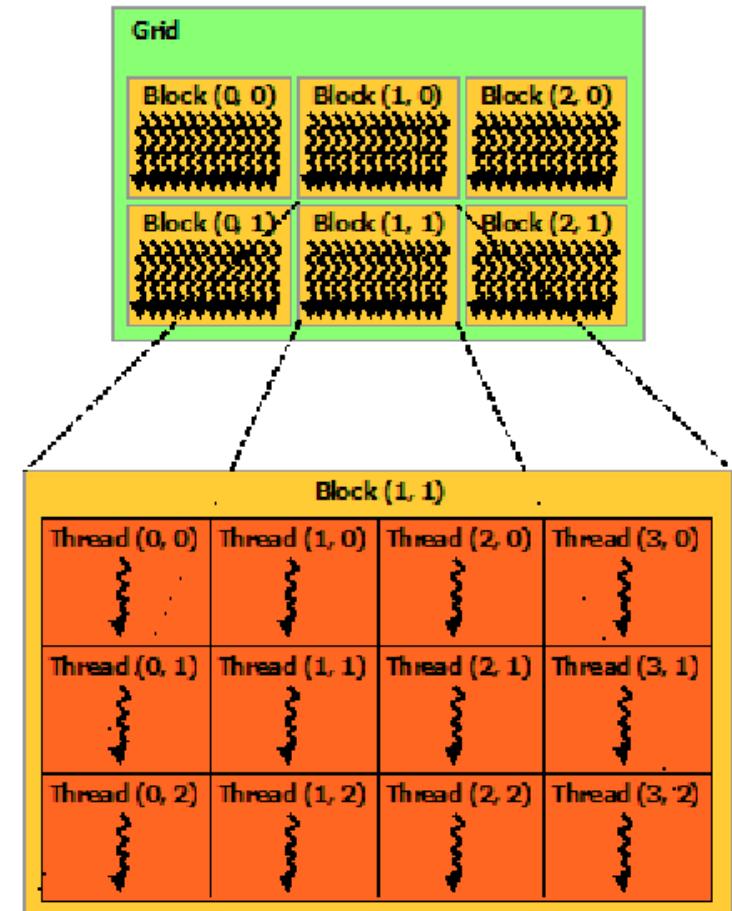
- `gridDim.x`, `gridDim.y`, `gridDim.z` (unused)
- `blockIdx.x`, `blockIdx.y`, `blockIdx.z`
- Variables that return the grid dimension (number of blocks) and block ID in the x-, y-, and z-axis.

## Thread indexes

- `blockDim.x`, `blockDim.y`, `blockDim.z`
- `threadIdx.x`, `threadIdx.y`, `threadIdx.z`
- Variables that return the block dimension (number of threads per block) and thread ID in the x-, y-, and z-axis.

Example in the figure is executing 72 threads

- $(3 \times 2)$  blocks = 6 blocks
- $(4 \times 3)$  threads per block = 12 threads per block



# CUDA basics summary

## **\_\_global\_\_ keyword**

- Function that executes on the device (GPU), must return `void`, and is called from host code.

```
__global__ vector_add_kernel(int *a, int *b, int *c, int n) {
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    int stride = blockDim.x * gridDim.x;
    while (tid < n) {
        c[tid] = a[tid] + b[tid];
        tid += stride;
    }
}
```

## **CUDA API handles device memory**

- `cudaMalloc()` , `cudaFree()` , `cudaMemcpy()`
- Equivalent to C `malloc()` , `free()` , `memcpy()`
- `cudaMemcpy()` is used to transfer data between CPU and GPU memory.

## **CUDA kernel launch specification**

- Triple angle bracket determines grid and block size (i.e. total number of threads) for kernel launch:

```
vector_add_kernel<<<dim3(bx,by,bz), dim3(tx,ty,tz)>>>(d_a, d_b, d_c, N);
```

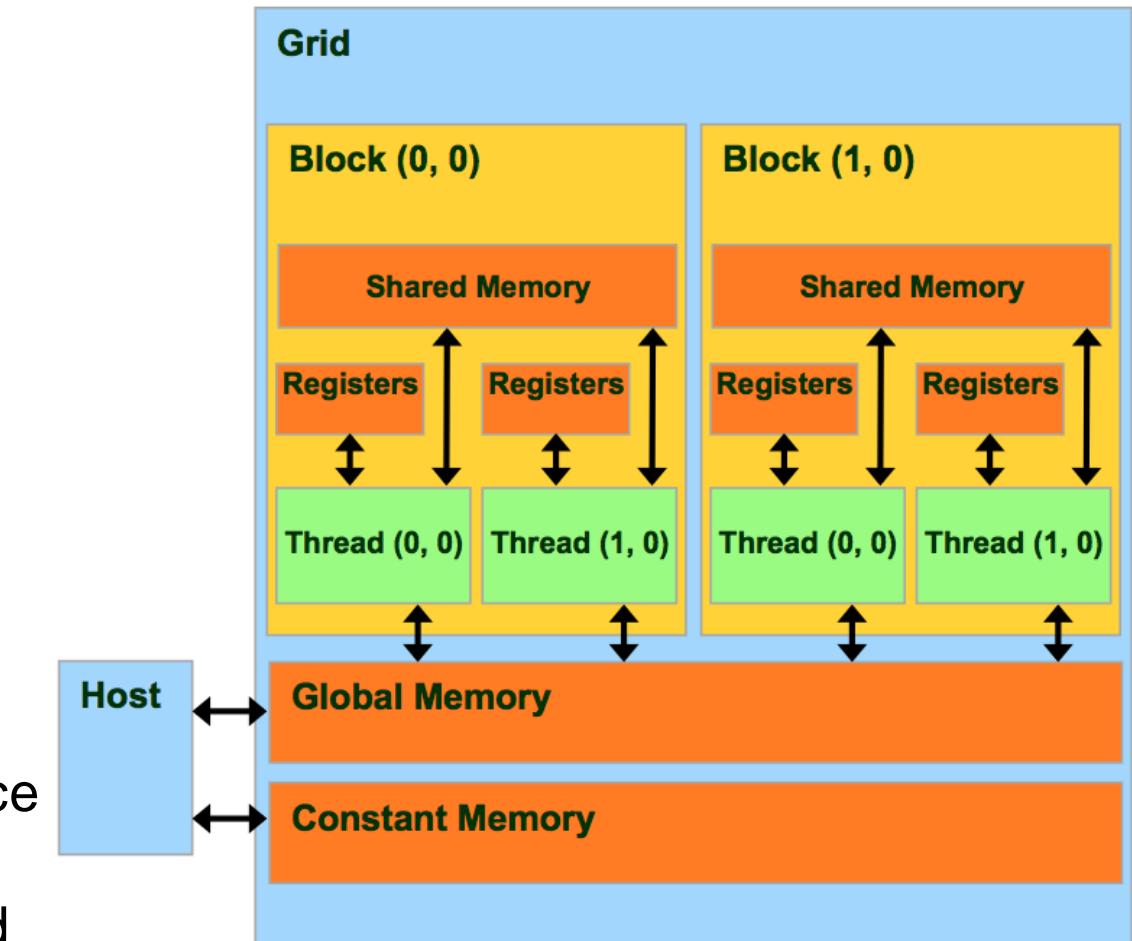
# CUDA basics: Memory overview

## CUDA memory hierarchy

- Host memory (x86 server)
- Device memory (GPU)

## Device memory

- **Global memory**  
visible to all threads, slow
- **Shared memory**  
visible to all threads in a block, fast on-chip
- **Registers**  
per-thread memory, fast on-chip
- **Local memory**  
per-thread, slow, stored in Global Memory space
- **Constant memory**  
visible to all threads, read only, off-chip, cached  
broadcast to all threads in a half-warp (16 threads)



# General CUDA programming strategy

## Avoid data transfers between CPU and GPU

- These are slow due to low PCI express bus bandwidth

## Minimize access to global memory

- Hide memory access latency by launching many threads

## Take advantage of fast shared memory by tiling data

- Partition data into subsets that fit into shared memory
- Handle each data subset with one thread block
- Load the subset from global to shared memory using multiple threads to exploit parallelism in memory access
- Perform computation on data subset in shared memory (each thread in thread block can access data multiple times)
- Copy results from shared memory to global memory

# General CUDA programming strategy

## Use of constant memory for data that is constant during run time

- Data that is accessed by all threads within a block (half-warp, actually) at the same time
  - this reduces memory bandwidth requirements
- Do not use constant memory if threads access different elements of the data
  - half-warps can place only a single read-request at a time
  - if threads need different data from constant memory, these reads get serialized

# CUDA Example: Matrix-matrix multiply

```
float* host_A, host_B, host_C;  
float* device_A, device_B, device_C;  
  
// Allocate host memory  
host_A = (float*) malloc(mem_size_A);  
host_B = (float*) malloc(mem_size_B);  
host_C = (float*) malloc(mem_size_C);  
  
// Allocate device memory  
cudaMalloc((void**) &device_A, mem_size_A);  
cudaMalloc((void**) &device_B, mem_size_B);  
cudamalloc((void**) &device_C, mem_size_C);  
  
// Set up the initial values of A and B here.  
...
```

Explain only  
concepts without  
following details

Skip if there is  
not sufficient time

# CUDA Example: Matrix-matrix multiply - 2

Concepts only / skip

```
// copy host memory to device
cudaMemcpy(device_A, host_A, mem_size_A, cudaMemcpyHostToDevice);
cudaMemcpy(device_B, host_B, mem_size_B, cudaMemcpyHostToDevice);

// setup execution parameters
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(WC / threads.x, HC / threads.y);

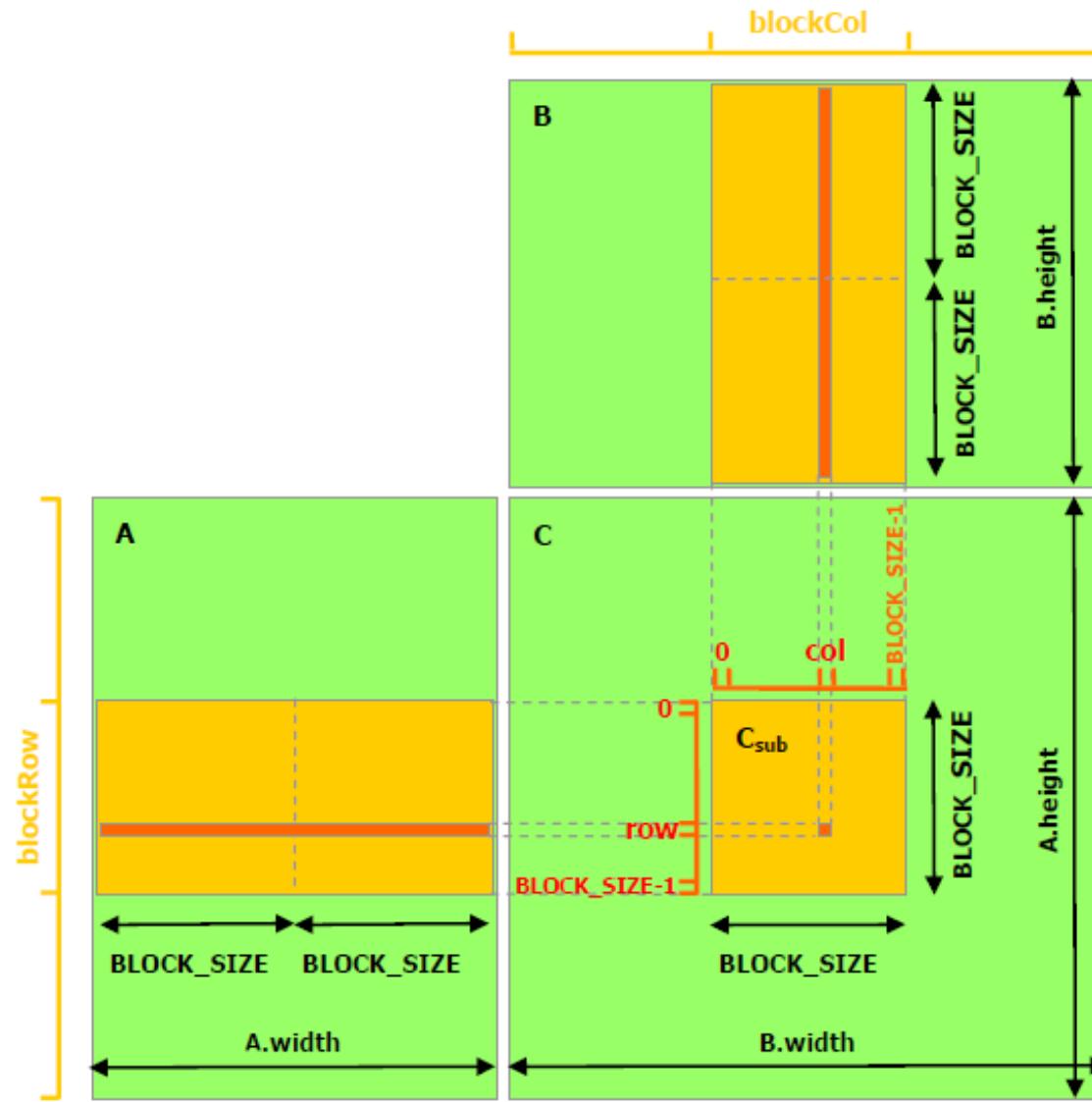
// execute the kernel
matrixMul<<< grid, threads >>>(device_C, device_A, device_B, WA, WB);

// copy result from device to host
cudaMemcpy(host_C, device_C, mem_size_C, cudaMemcpyDeviceToHost);

// Free host and device memory
...
```

# CUDA Example: Matrix-matrix multiply kernel

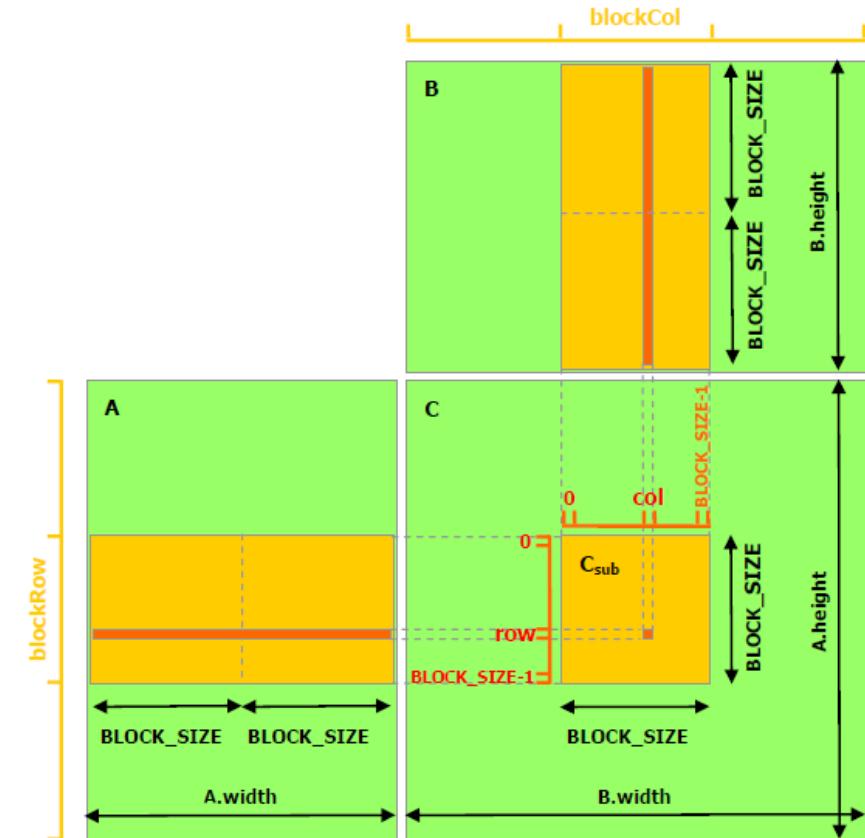
Concepts only / skip



# CUDA Example: Matrix-matrix multiply kernel

Concepts only / skip

```
__global__ void matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;
    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;
    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;
    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;
    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    float Csub = 0;
```

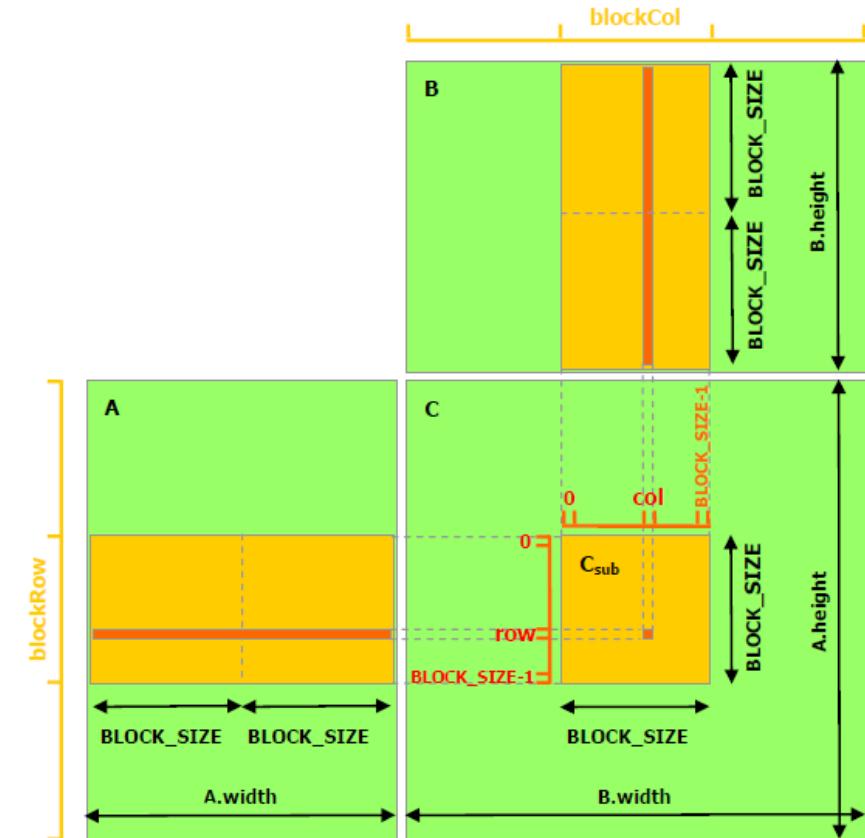


# CUDA Example: Matrix-matrix multiply kernel – 2

Concepts only / skip

```
// Loop over all the sub-matrices of A and B
// required to compute the block sub-matrix

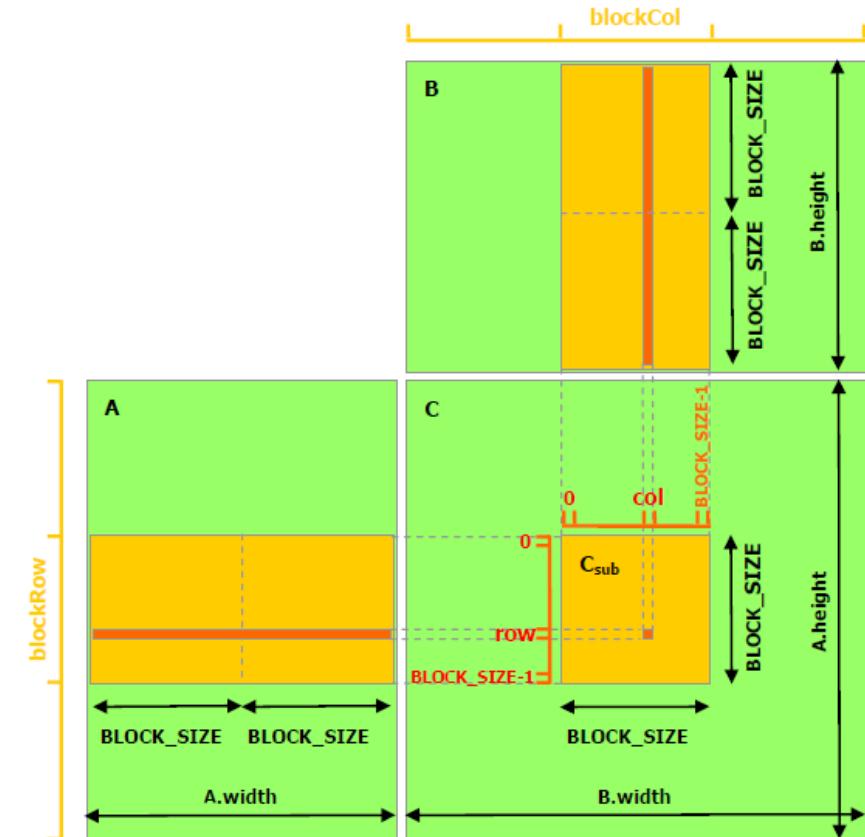
for (int a = aBegin, b = bBegin;
     a <= aEnd;
     a += aStep, b += bStep) {
    // Declaration of the shared memory array As
    // store the sub-matrix of A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    // Declaration of the shared memory array Bs
    // store the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    // Load the matrices from device memory
    // to shared memory; each thread loads
    // one element of each matrix
    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    // Synchronize to make sure the matrices are loaded
    __syncthreads();
```



# CUDA Example: Matrix-matrix multiply kernel – 3

Concepts only / skip

```
// Multiply the two matrices together;  
// each thread computes one element of the block sub-matrix  
  
for (int k = 0; k < BLOCK_SIZE; ++k)  
    Csub += AS(ty, k) * BS(k, tx);  
  
// Synchronize to make sure that the preceding  
// computation is done before loading two new  
// sub-matrices of A and B in the next iteration  
  
__syncthreads();  
  
}  
  
// Write the block sub-matrix to device memory;  
// each thread writes one element  
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;  
C[c + wB * ty + tx] = Csub;  
}
```



# CUDA Example: Matrix-matrix multiply summary

Concepts only / skip

## Summary

- We made use of a variety of CUDA features including
- 2D grids and blocks
- Shared memory
- Thread synchronization

## Note

- In reality we would not write a matrix-matrix multiplication function
- The CUDA implementation of BLAS is highly optimized for GPUs

# Avoiding race conditions with atomic operations

## Atomic operations

- An atomic operation cannot be divided into several operations
- What happens when we increment a counter?  
`i++;`
- This consists of three steps
  - 1) Read the value stored at the address of `i`
  - 2) Add 1 to the value read in step 1)
  - 3) Write the result back to the address of `i`
- What happens if multiple threads increment the counter?

# Avoiding race conditions with atomic operations

## Atomic operations

- An atomic operation cannot be divided into several operations
- What happens when we increment a counter?  
`i++;`
- This consists of three steps
  - 1) Read the value stored at the address of `i`
  - 2) Add 1 to the value read in step 1)
  - 3) Write the result back to the address of `i`
- What happens if multiple threads increment the counter?
- If multiple threads modify an address, the result is unpredictable
- Atomic operations are performed without interference from other threads
- Atomic operations are available on newer GPUs (efficient since Kepler chips) and can operate on global or shared memory
  - `atomicAdd()` , `atomicSub()` , `atomicMin()` , ...
  - etc. see programming guide for full list
- Use wisely – code execution will be serialized

# Example: Histogram

Skipped

- Assume data set of 8-bit (1 byte) values
- Compute occurrence of each of the 256 possible values
- Serial CPU code:

```
#define SIZE (100*1024*1024)
int main(void){
    unsigned char buffer[SIZE];
    unsigned int histo[256];

    get_data(buffer,SIZE);           //read data

    for (int i=0; i<256; i++)      //initialize to zero
        histo[i] = 0;

    for (int i=0; i<SIZE; i++)     //compute histogram
        histo[buffer[i]]++;

    return 0;
}
```

# Histogram CUDA code

Skipped

- Using atomic add operation to avoid data races

```
#define SIZE (100*1024*1024)

// histogram kernel
__global__ histo_kernel(unsigned char *buffer,
                        int size, unsigned int *histo){

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    while (tid < size) {
        atomicAdd( &(histo[buffer[tid]]), 1 );
        tid += stride;
    }
}
```

- This will work – but with terrible performance – why?

# Histogram CUDA code

Skipped

- Using atomic add operation to avoid data races

```
#define SIZE (100*1024*1024)

// histogram kernel
__global__ histo_kernel(unsigned char *buffer,
                        int size, unsigned int *histo) {

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    while (tid < size) {
        atomicAdd( &(histo[buffer[tid]]), 1 );
        tid += stride;
    }
}
```

- This will work – but with terrible performance – why?

## Bad performance because

- the kernel does very little work
- thousands of threads access few (256) memory locations with atomic add operations

## Improve performance by using shared memory

- write operations to shared memory are fast
- fewer threads will try to access the same memory locations with the atomic add operation

# Histogram CUDA code

Skipped

## Histogram kernel with shared memory

```
#define SIZE (100*1024*1024)

// histogram kernel
__global__ histo_kernel(unsigned char *buffer,
                        int size, unsigned int *histo){

    // shared memory,
    // assume 256 threads per block
    __shared__ unsigned int temp[256];
    temp[threadIdx.x] = 0;
    __syncthreads();

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
```

```
        // compute histogram in shared memory
        // for this block
    while (tid < size) {
        atomicAdd( &(temp[buffer[tid]]), 1 );
        tid += stride;
    }
    __syncthreads();

    // now merge each block's histogram
    // into global memory
    // again assuming 256 threads per block
    atomicAdd( &(histo[threadIdx.x]),
                temp[threadIdx.x] );
}
```

# Histogram CUDA code

Skipped

## Histogram host code

```
int main(void) {  
  
    // host and device memory / pointers  
    unsigned char h_buffer[SIZE], *d_buffer;  
    unsigned int h_hist[256], *d_hist;  
  
    // get our input data  
    get_data(buffer, SIZE);  
  
    // allocate device memory  
    cudaMalloc( (void**)&d_buffer, SIZE);  
    cudaMalloc( (void**)&d_hist,  
                256 * sizeof(int) );  
  
    // copy data to device  
    cudaMemcpy( d_buffer, h_buffer, SIZE,  
               cudaMemcpyHostToDevice);
```

```
// initialize histogram to zero  
cudaMemset(d_hist, 0, 256);  
  
// launch kernel  
histo_kernel<<<32,256>>>(d_buffer,  
                                SIZE, d_hist);  
  
// copy results back  
cudaMemcpy( h_hist, d_hist,  
            256*sizeof(int), cudaMemcpyDeviceToHost);  
  
// free memory  
cudaFree(d_buffer); cudaFree(d_hist);  
  
// print our histogram  
print_histogram(h_hist);  
  
return 0;  
}
```

# Training overview

## We will cover the following topics

- GPU hardware overview
- Accessing SDSC Expanse GPU nodes
- GPU accelerated software examples
- Programming GPUs – Overview for Nvidia GPUs
- GPU enabled libraries
  - 15 minutes break –
- CUDA C programming basics
- **Hands-on exercises on SDSC Expanse – CUDA**
- OpenACC introduction
- Hands-on exercises on SDSC Expanse – OpenACC

# CUDA Toolkit Samples

- Available from Nvidia Github.

```
[agoetz@exp-8-59 ~]$ git clone https://github.com/NVIDIA/cuda-samples
```

- A copy is also staged on Expanse. Move into the scratch directory and unpack there (much faster than working in your NFS mounted home)

```
[agoetz@exp-8-59 ~]$ cd $SLURM_TMPDIR
```

```
[agoetz@exp-8-59 ~]$ tar xvf ~/data/cuda-samples-v12.2.tar.gz ./
```

- Explore CUDA Toolkit samples – great resource!

```
[agoetz@exp-8-59 ~]$ cd cuda-samples-v12.2/Samples
```

```
[agoetz@exp-8-59 CUDA_samples]$ ls
```

0_Introduction	2_Concepts_and_Techniques	4_CUDA_Libraries	6_Performance
1_Utils	3_CUDA_Features	5_Domain_Specific	7_libNVM

# CUDA Toolkit Samples

- Compile CUDA Toolkit samples
- You can compile in parallel, we requested 10 CPU cores
- Executables will reside in sub directory `bin/x86_64/linux/release/`

```
[agoetz@exp-8-59 cuda-samples]$ make -j 10
make[1]: Entering directory `/home/agoetz/CUDA_samples/0_Simple/simpleMultiCopy'
/usr/local/cuda-10.2/bin/nvcc -ccbin g++ -I../../common/inc -m64 -gencode
arch=compute_30,code=sm_30 -gencode arch=compute_35,code=sm_35 -gencode
...
arch=compute_75,code=sm_75 -gencode arch=compute_75,code=compute_75 -o simpleMultiCopy.o -c
simpleMultiCopy.cu
```

# CUDA Toolkit Samples

- Compilation takes a while, you can also compile individual examples, e.g. `deviceQuery`, which prints information on available GPUs

```
[agoetz@exp-1-57 cuda-samples]$ cd Samples/1_Utilsilities/deviceQuery
[agoetz@exp-1-57 cuda-samples]$ make SMS=70
/cm/shared/apps/cuda12.2/toolkit/12.2.2/bin/nvcc -ccbin g++ -I../../Common -m64 ...
...
[agoetz@exp-1-57 deviceQuery]$ ./deviceQuery
./deviceQuery Starting...

CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Tesla V100-SXM2-32GB"
      CUDA Driver Version / Runtime Version      11.0 / 10.2
      CUDA Capability Major/Minor version number:    7.0
      Total amount of global memory:             32510 MBytes (34089730048 bytes)
      (80) Multiprocessors, ( 64) CUDA Cores/MP:   5120 CUDA Cores
```

# SDSC Expanse GPU nodes

## CUDA Toolkit

- Matrix multiplication example

```
[agoetz@exp-3-58 ~]$ cd ~/cuda-samples/Samples/0_Introduction/
[agoetz@exp-3-58 0_Simple]$ ./matrixMul/matrixMul
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "Volta" with compute capability 7.0

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 3274.22 GFlop/s, Time= 0.040 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS
```

- Matrix multiplication example with CUBLAS

```
[agoetz@exp-3-58 ~]$ cd ~/cuda-samples/Samples/4_CUDA_Libraries/
[agoetz@exp-3-58 0_Simple]$ ./matrixMulCUBLAS/matrixMulCUBLAS
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Volta" with compute capability 7.0

MatrixA(640,480), MatrixB(480,320), MatrixC(640,320)
Computing result using CUBLAS...done.
Performance= 7588.93 GFlop/s, Time= 0.026 msec, Size= 196608000 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS
```

# Code samples from this course

- In SI 2024 Github repository  
<https://github.com/sdsc/sdsc-summer-institute-2024>  
directory 5 . 3a \_gpu \_computing \_and \_programming
- Directory 5 . 3a \_gpu \_computing \_and \_programming / cuda - samples
  - “Hello world”
  - Addition, vector addition
  - Squaring matrix elements (not discussed)
  - 1D stencil (not discussed)

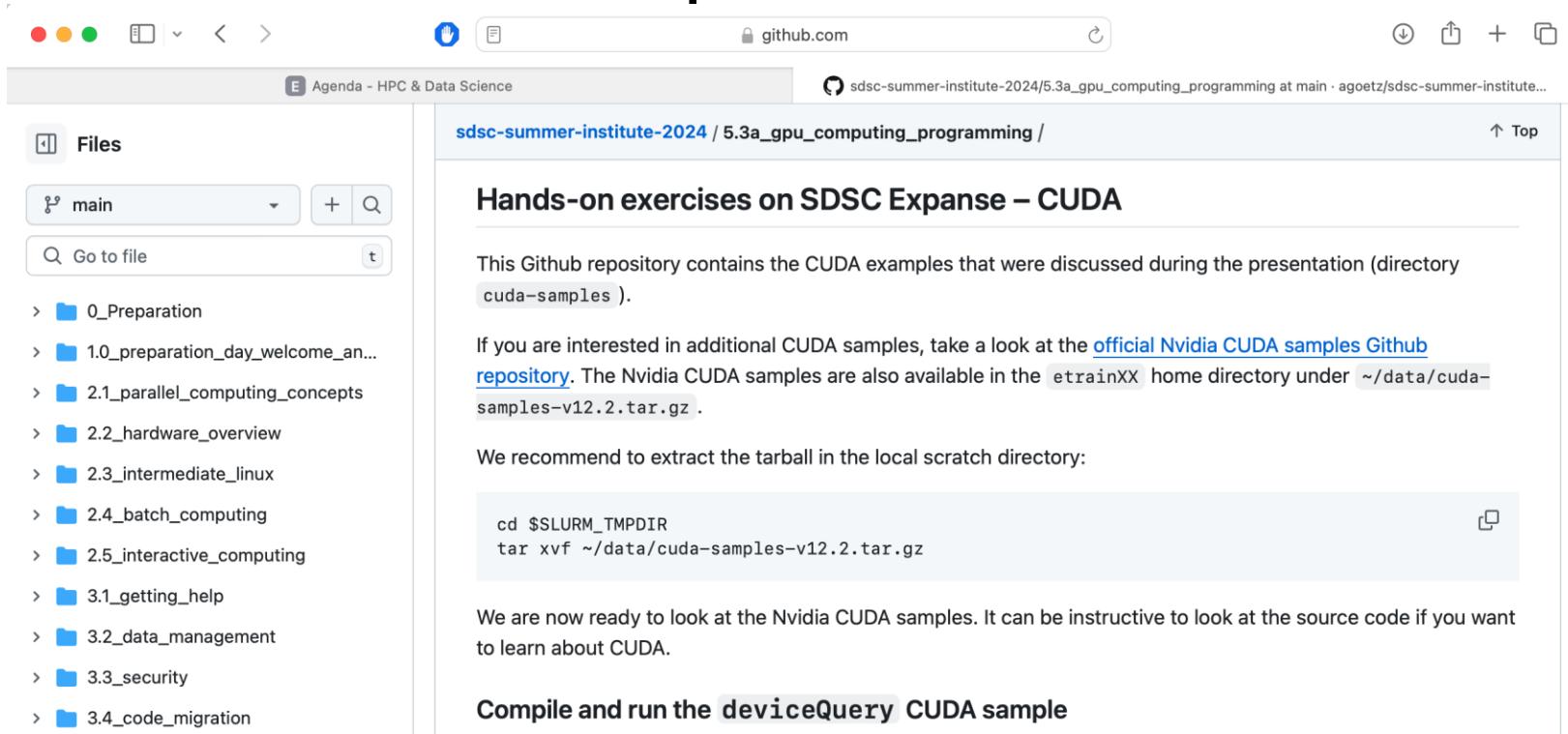
**If you have not done so, please clone the repository now**

- Check the README files
- Compile and run CUDA examples
- Try the exercises (look for **FIXME** comments and try to replace with correct code)

# Hands-on session

- In SI 2024 Github repository  
<https://github.com/sdsc/sdsc-summer-institute-2024>  
directory 5 . 3a \_gpu \_computing \_and \_programming

Follow the instructions in the Github repo README.md



The screenshot shows a GitHub repository page for 'sdsc-summer-institute-2024 / 5.3a\_gpu\_computing\_programming'. The left sidebar shows a tree view of files and sub-directories under 'main'. The main content area displays the README.md file's content:

```
Hands-on exercises on SDSC Expanse – CUDA

This Github repository contains the CUDA examples that were discussed during the presentation (directory cuda-samples).

If you are interested in additional CUDA samples, take a look at the official Nvidia CUDA samples Github repository. The Nvidia CUDA samples are also available in the etrainXX home directory under ~/data/cuda-samples-v12.2.tar.gz.

We recommend to extract the tarball in the local scratch directory:

cd $SLURM_TMPDIR
tar xvf ~/data/cuda-samples-v12.2.tar.gz

We are now ready to look at the Nvidia CUDA samples. It can be instructive to look at the source code if you want to learn about CUDA.

Compile and run the deviceQuery CUDA sample
```

# Training overview

## We will cover the following topics

- GPU hardware overview
- Accessing SDSC Expanse GPU nodes
- GPU accelerated software examples
- Programming GPUs – Overview for Nvidia GPUs
- GPU enabled libraries
  - 15 minutes break –
- CUDA C programming basics
- Hands-on exercises on SDSC Expanse – CUDA
- **OpenACC introduction**
- Hands-on exercises on SDSC Expanse – OpenACC

# Directive based programming

## OpenACC

- See <https://www.openacc.org>
- Open standard for expressing accelerator parallelism
- Designed to make porting to GPUs easy, quick, and portable
- OpenMP-like compiler directives language
  - If the compiler does not understand the directives, it will ignore them.
  - Same code can work with or without accelerators.
- Fortran and C
- Full support by PGI compilers and Cray compilers on Crays
- Partial support by GNU compilers (experimental since version 5.1)
- Also some less commonly used and experimental compilers

## OpenMP

- See <https://www.openmp.org>
- Is becoming mature for GPUs, but will not discuss here

# Directive based programming

## Nvidia HPC SDK

- See <https://developer.nvidia.com/hpc-sdk>
- Available on SDSC Expanse
- Nvidia Fortran / C / C++ / CUDA compilers (nvfortran, nvc, nvc++, nvcc)
  - compiler support OpenACC, CUDA Fortran, CUDA C/C++, OpenMP (for multicore CPUs)
- nvprof performance profiler
- cuda-gdb debugger
- GPU-enabled libraries
- OpenACC code samples

## Activate on Expanse GPU nodes

```
$> module load nvhpc/21.9
```

# A simple OpenACC exercise: SAXPY

SAXPY in C

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

SAXPY in Fortran

```
subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
!$acc kernels
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
!$acc end kernels
end subroutine saxpy

...
! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

# Complete SAXPY code

## Trivial first example

- Apply a loop directive
- Learn compiler commands

```
#include <stdlib.h>

void saxpy(int n,
           float a,
           float *x,
           float * restrict y)
{
#pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
```

```
int main(int argc, char **argv)
{
    int N = 1<<20; // 1 million floats

    float *x = (float*)malloc(N *
sizeof(float));
    float *y = (float*)malloc(N *
sizeof(float));

    for (int i = 0; i < N; ++i)
    {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }

    saxpy(N, 3.0f, x, y);

    return 0;
}
```

# Compile and run SAXPY OpenACC code

- C:

```
pgcc -acc -ta=tesla -Minfo=accel -o saxpy_acc saxpy.c
```

- Fortran:

```
pgf90 -acc -ta=tesla -Minfo=accel -o saxpy_acc saxpy.f90
```

- Compiler output:

```
pgcc saxpy.c -acc -ta=tesla -Minfo=accel -o saxpy-gpu.x
saxpy:
  8, Generating copyin(x[:n])
      Generating copy(y[:n])
  9, Loop is parallelizable
      Accelerator kernel generated
      Generating Tesla code
      9, #pragma acc loop gang, vector(128) /* blockIdx.x
threadIdx.x */
```

# OpenACC directives syntax

## Fortran

```
!$acc directive [clause [,] clause] ...]
```

Often paired with a matching end directive  
surrounding a structured code block

```
!$acc end directive
```

## constructs

```
!$acc kernels [clause ...]
  structured code block
!$acc end kernels
```

## Clauses

```
if( condition )
async( expression )
or data clauses
```

## C

```
#pragma acc directive [clause [,] clause] ...]
```

Often followed by a structured code block

## constructs

```
#pragma acc kernels [clause ...]
{ structured code block }
```

# OpenACC directives syntax

## Data clauses

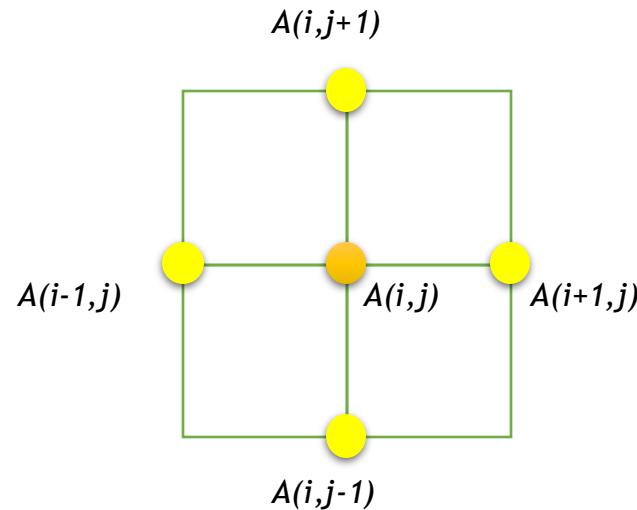
- `copy ( list )` Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
- `copyin ( list )` Allocates memory on GPU and copies data from host to GPU when entering region.
- `copyout ( list )` Allocates memory on GPU and copies data to the host when exiting region.
- `create ( list )` Allocates memory on GPU but does not copy.
- `present ( list )` Data is already present on GPU from another containing data region.

and `present_or_copy[in|out]`, `present_or_create`, `deviceptr`.

# OpenACC example: Jacobi iteration

Iteratively converges to correct value (e.g. Temperature),  
by computing new values at each point from the average of neighboring points.

- Common, useful algorithm
- Example: Solve Laplace equation in 2D:  $\nabla^2 u(x, y) = 0$



$$A_{k+1}(i, j) = \frac{A_k(i - 1, j) + A_k(i + 1, j) + A_k(i, j - 1) + A_k(i, j + 1)}{4}$$

# OpenACC example: Jacobi iteration

```
while ( error > tol && iter < iter_max )  
{  
    error=0.0;  
  
    for( int j = 1; j < n-1; j++) {  
        for(int i = 1; i < m-1; i++) {  
  
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +  
                                  A[j-1][i] + A[j+1][i]);  
  
            error = max(error, abs(Anew[j][i] - A[j][i]));  
        }  
    }  
  
    for( int j = 1; j < n-1; j++) {  
        for( int i = 1; i < m-1; i++ ) {  
            A[j][i] = Anew[j][i];  
        }  
    }  
  
    iter++;  
}
```



Iterate until converged



Iterate across matrix elements



Calculate new value from neighbors



Compute max error for convergence



Swap input/output arrays

# OpenACC example: Jacobi iteration – first attempt

```
while ( error > tol && iter < iter_max )
{
    error=0.0;

    #pragma acc kernels
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {

            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                  A[j-1][i] + A[j+1][i]);

            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

    #pragma acc kernels
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
```



Execute GPU kernel for  
loop nest



Execute GPU kernel for  
loop nest

# OpenACC example: Jacobi iteration – first attempt

## Compiler output

```
pgf90 -acc -ta=tesla -Minfo=accel -o jacobi-pgf90-acc-v1.x jacobi-acc-v1.f90
laplace:
 44, Generating copyout(anew(1:4094,1:4094))
    Generating copyin(a(0:4095,0:4095))
 45, Loop is parallelizable
 46, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
 45, !$acc loop gang ! blockidx%y
 46, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
 49, Max reduction generated for error
 57, Generating copyin(anew(1:4094,1:4094))
    Generating copyout(a(1:4094,1:4094))
 58, Loop is parallelizable
 59, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
 58, !$acc loop gang ! blockidx%y
 59, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
```

# OpenACC example: Jacobi iteration – first attempt

**SDSC Comet**

CPU: Intel Xeon E5-2680 v3

GPU: NVIDIA Tesla K80  
(using single GPU)

Execution	Time (s)	Speedup
CPU 1 OpenMP thread	69	--
CPU 2 OpenMP threads	36	1.92x
CPU 4 OpenMP threads	22	3.14x
CPU 6 OpenMP threads	17	4.06x
OpenACC GPU	501	0.03x FAIL

Compiler:  
pgcc 17.5-0

CPU flags:  
-fastsse -O3 -mp [-Minfo=mp]

GPU flags:  
-acc [-Minfo=accel]

**Speedup vs.  
1 CPU core**

**Speedup vs.  
6 CPU cores**

# OpenACC example: Jacobi iteration – first attempt

```
export PGI_ACC_TIME=1      ! Activate profiling, then run again
```

Accelerator Kernel Timing data

/server-home1/agoetz/UCSD\_Phys244/2017/openacc-samples/laplace-2d/jacobi-acc-v1.f90

laplace NVIDIA devicenum=0

time(us): 89,612,134

..... <snip – some lines cut>

44: **data region** reached 2000 times

  44: **data copyin transfers**: 8000

    device time(us): total=**22,587,486** max=2,898 min=2,799 avg=2,823

  52: **data copyout transfers**: 8000

    device time(us): total=**20,278,262** max=2,612 min=2,497 avg=2,534

57: **compute region** reached 1000 times

  59: **kernel launched** 1000 times

    grid: [128x1024] block: [32x4]

      device time(us): total=**1,456,273** max=1,465 min=1,452 avg=1,456

      elapsed time(us): total=1,498,877 max=1,524 min=1,492 avg=1,498

57: **data region** reached 2000 times

  57: **data copyin transfers**: 8000

    device time(us): total=22,664,227 max=2,902 min=2,802 avg=2,833

  63: **data copyout transfers**: 8000

    device time(us): total=20,278,000 max=2,618 min=2,498 avg=2,534

22.5 seconds

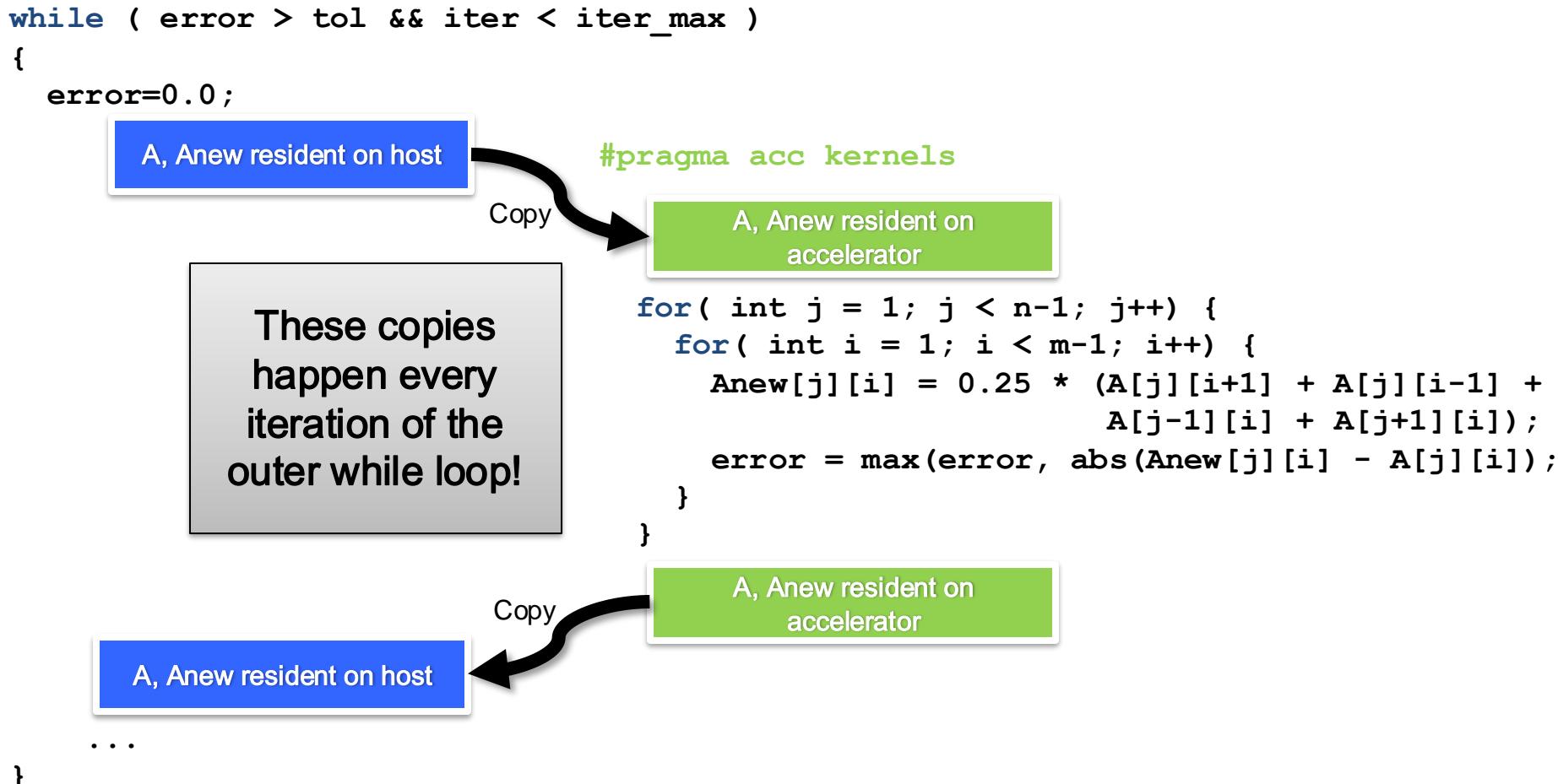
1.5 seconds

## What went wrong?

- We spent all the time with data transfers between host and device

# OpenACC example: Jacobi iteration – first attempt

## Excessive data transfers



# OpenACC example: Jacobi iteration – second attempt

```
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
    error=0.0;

#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++ ) {

            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                  A[j-1][i] + A[j+1][i]);

            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
```

Copy A in at beginning of loop, out at end. Allocate Anew on accelerator

# OpenACC example: Jacobi iteration – second attempt

SDSC Comet

CPU: Intel Xeon E5-2680 v3

GPU: NVIDIA Tesla K80  
(using single GPU)

Execution	Time (s)	Speedup
CPU 1 OpenMP thread	69	--
CPU 2 OpenMP threads	36	1.92x
CPU 4 OpenMP threads	23	3.14x
CPU 6 OpenMP threads	17	4.06x
OpenACC GPU	5	3.4x

Compiler:  
pgcc 17.5-0

CPU flags:  
-fastsse -O3 -mp [-Minfo=mp]

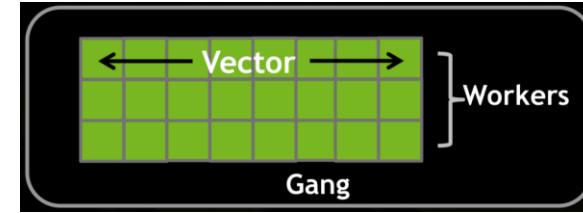
GPU flags:  
-acc [-Minfo=accel]

**CPU Speedup**  
vs.  
**1 CPU core**

**GPU Speedup**  
vs.  
**6 CPU cores**

# More OpenACC

- OpenACC gives us more detailed control over parallelization
  - Via **gang**, **worker**, and **vector** clauses
  - Gang corresponds to block, shares resources such as cache, streaming multiprocessor etc)
  - Vector threads work in lockstep (warp)
  - Workers compute a vector, correspond to threads
- By understanding more about OpenACC execution model and GPU hardware organization, we can get higher speedups on this code
- By understanding bottlenecks in the code via profiling, we can reorganize the code for higher performance



# More OpenACC

## Finding and exploiting parallelism in your code

- (Nested) for loops are best for parallelization
- Large loop counts needed to offset GPU/memcpy overhead
- Iterations of loops must be independent of each other
  - To help compiler: `restrict` keyword (C), `independent` clause
- Compiler must be able to figure out sizes of data regions
  - Can use directives to explicitly control sizes
- Pointer arithmetic should be avoided if possible
  - Use subscripted arrays, rather than pointer-indexed arrays.
- Function calls within accelerated region must be inlineable.

# More OpenACC

## Tips and Tricks

- (PGI) Use time option to learn where time is being spent via compiler flag  
`-ta=nvidia,time`  
or use environment variable at run time  
`export PGI_ACC_TIME=1`
- Eliminate pointer arithmetic
- Inline function calls in directives regions  
(PGI): `-Minline` or `-Minline=levels:N`
- Use contiguous memory for multi-dimensional arrays
- Use data regions to avoid excessive memory transfers
- Conditional compilation with `_OPENACC` macro

# Training overview

## We will cover the following topics

- GPU hardware overview
- Accessing SDSC Expanse GPU nodes
- GPU accelerated software examples
- Programming GPUs – Overview for Nvidia GPUs
- GPU enabled libraries
  - 15 minutes break –
- CUDA C programming basics
- Hands-on exercises on SDSC Expanse – CUDA
- OpenACC introduction
- **Hands-on exercises on SDSC Expanse – OpenACC**

# Code samples for this course

- In SI 2024 Github repository  
<https://github.com/sdsc/sdsc-summer-institute-2024>  
directory `5.3a_gpu_computing_and_programming`
- Directory `5.3a_gpu_computing_and_programming/openacc-samples`
  - `saxpy`
  - `laplace-2D`

**If you have not done so, please clone the repository now**

- Check the `README` files
- Compile and run OpenACC examples
- Check timings of Jacobi iterations on CPU in serial, with OpenMP, and on GPU with OpenACC

# Hands-on session OpenACC

- In SI 2024 Github repository

<https://github.com/sdsc/sdsc-summer-institute-2024>

directory 5 . 3a \_gpu \_computing \_and \_programming

The screenshot shows a GitHub repository page for 'sdsc-summer-institute-2024 / 5.3a\_gpu\_computing\_programming'. The left sidebar shows a file tree under 'main' with sections like '0\_Preparation', '1.0\_preparation\_day\_welcome\_an...', '2.1\_parallel\_computing\_concepts', etc. The main content area has a title 'Hands-on exercises on SDSC Expanse – OpenACC'. It contains text about the repository containing OpenACC examples and instructions to load the NVHPC SDK. It includes code snippets for module loading:

```
module reset
module load nvhpc/21.9
```

and a command to check the compiler version:

```
pgcc --version
```

It also mentions replacing pgcc with nvc and pgf90 with nvfortran.

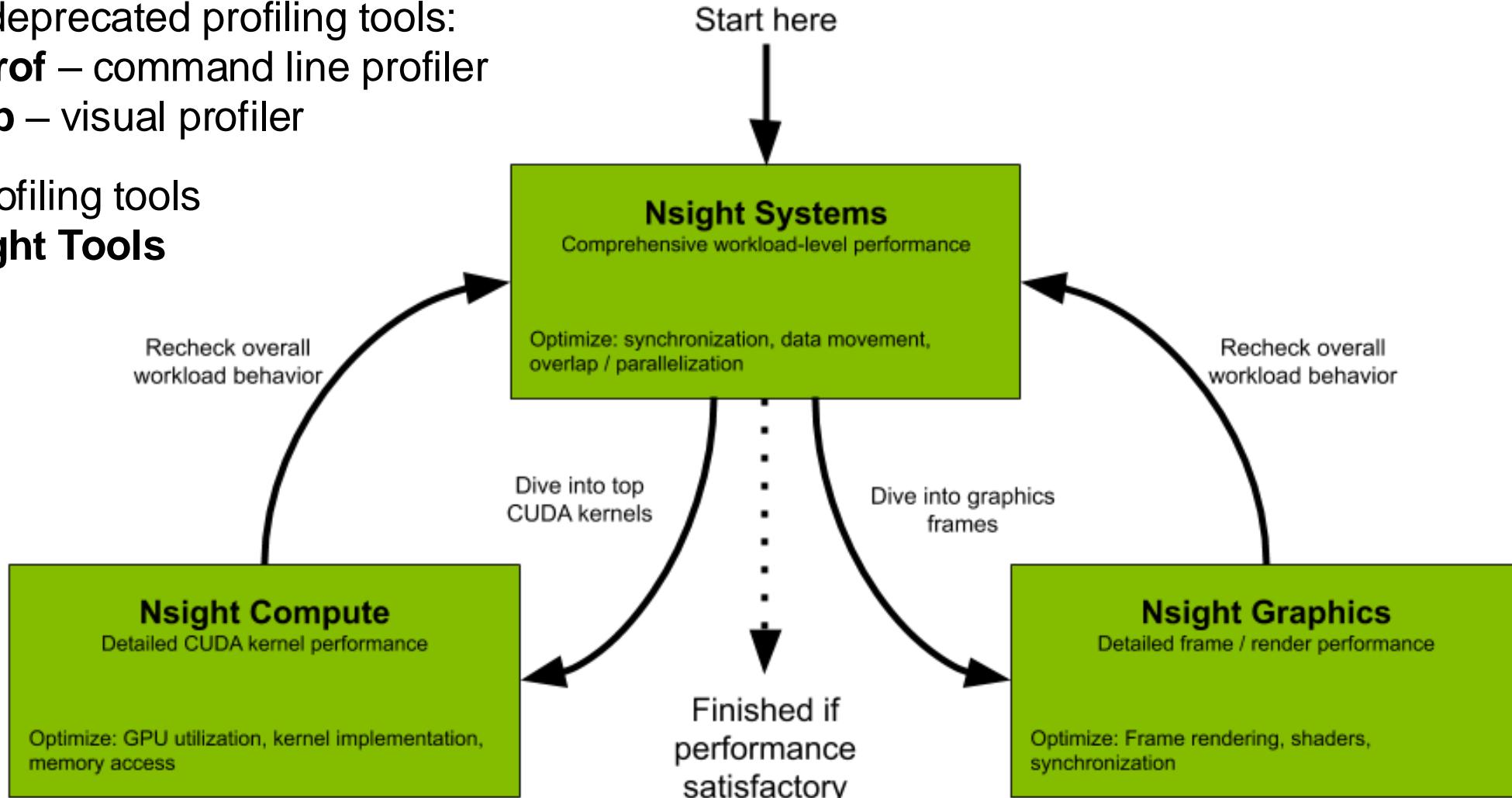
# Nvidia profiling tools

Older, deprecated profiling tools:

- **nvprof** – command line profiler
- **nvvvp** – visual profiler

New profiling tools

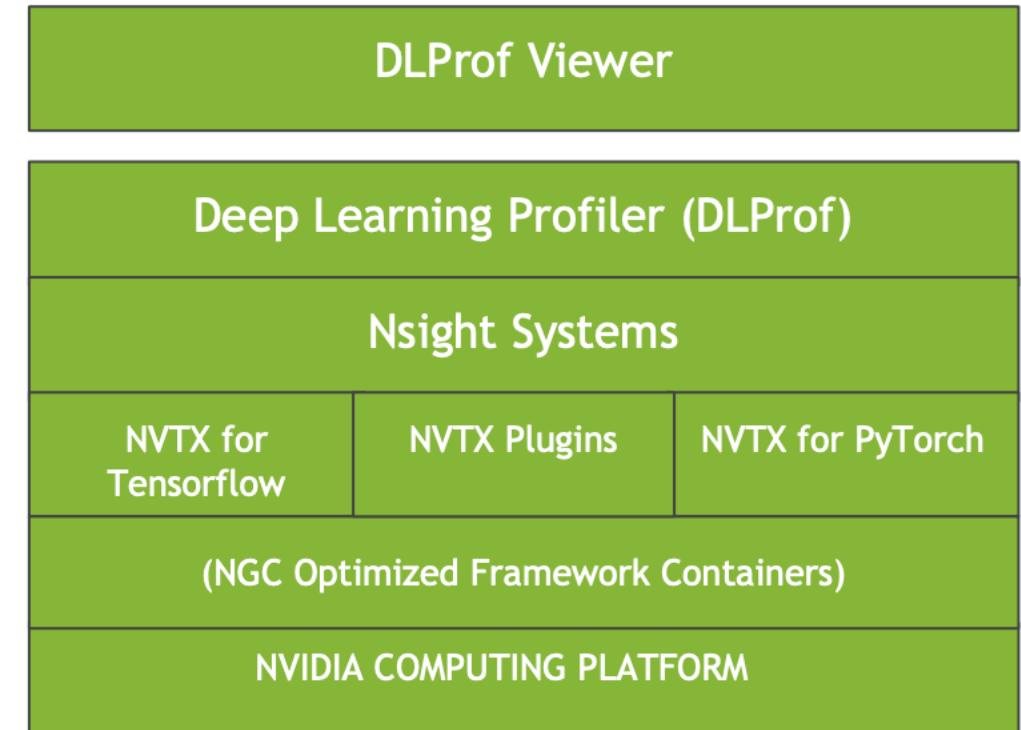
- **Nsight Tools**



# Nvidia profiling tools

## DLProf: Profiler for Deep Learning applications

- Nsight Systems and Nsight Compute have been built using CUDA Profiling Tools Interface (CUPTI)
- NVTX Nvidia Tools Extension Library is a way to annotate source code with markers
- NVTX markers are used to annotate and focus on sections of code important to the user
- TensorFlow optimized by Nvidia (nvidia-tensorflow) contains support for NVTX markers
- NVTX plugins are Python bindings for users to add markers easily
- DLProf calls Nsight systems to collect profile data and correlate with the DL model



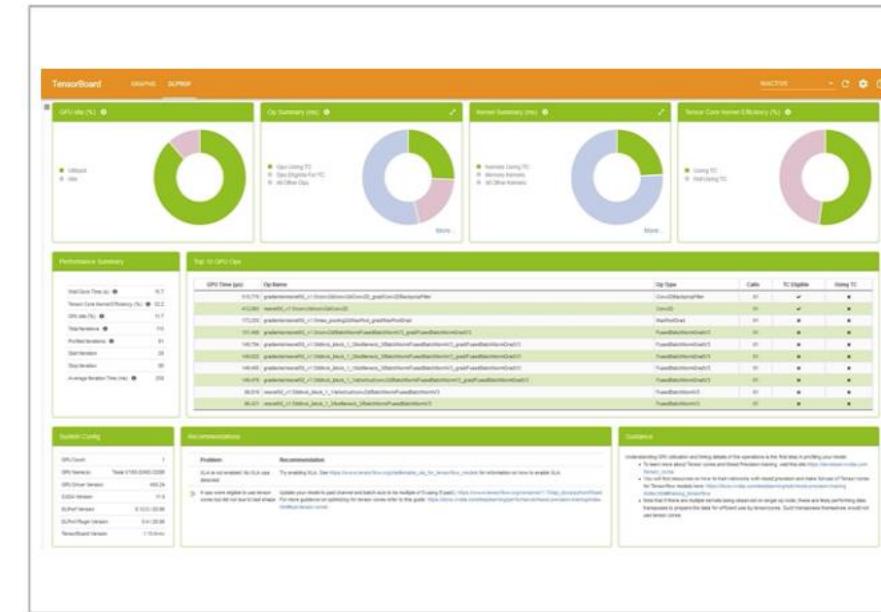
# Nvidia profiling tools

**DLProf:** Profiler for Deep Learning applications

- Are my GPUs being utilized?
- Am I using Tensor Cores?
- How can I improve performance?



FW Support: TF1, TF2, PyT, and TRT  
Lib Support: DALI, NCCL



Visualize Analysis and Recommendations

# Nvidia profiling tools

## DLProf: Profiler for Deep Learning applications

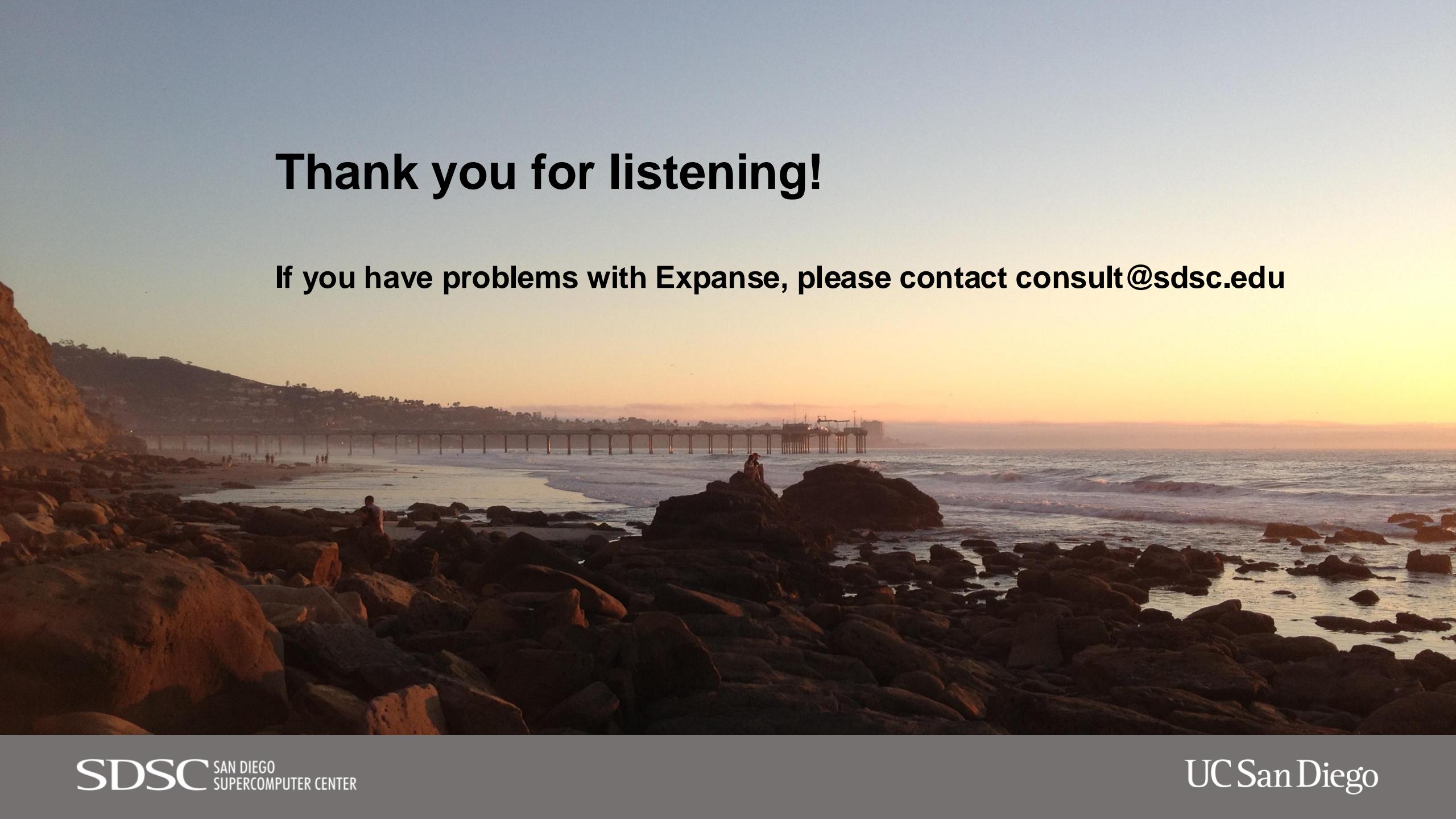


1. TensorFlow and TRT require no additional code modification
2. Profile using DLProf CLI - prepend with ***dlprof***
3. Visualize results with DLProf Viewer



1. Add few lines of code to your training script to enable **nvidia\_dlprof\_pytorch\_nvtx** module
2. Profile using DLProf CLI - prepend with ***dlprof***
3. Visualize with DLProf Viewer



A wide-angle photograph of a coastal scene at sunset. In the foreground, a rocky shoreline is visible with several people sitting on the rocks. In the middle ground, a long wooden pier extends from a small town built on a hillside down to the ocean. The sky is a gradient of warm colors, transitioning from blue at the top to orange and yellow near the horizon. The ocean waves are calm, reflecting the sunset light.

**Thank you for listening!**

**If you have problems with Expanse, please contact [consult@sdsc.edu](mailto:consult@sdsc.edu)**

# ***COMPLETE OPENACC API***

# Kernels Construct

Fortran

```
!$acc kernels [clause ...]
  structured block
 !$acc end kernels
```

Clauses

```
if( condition )
async( expression )
```

Also any data clause

C

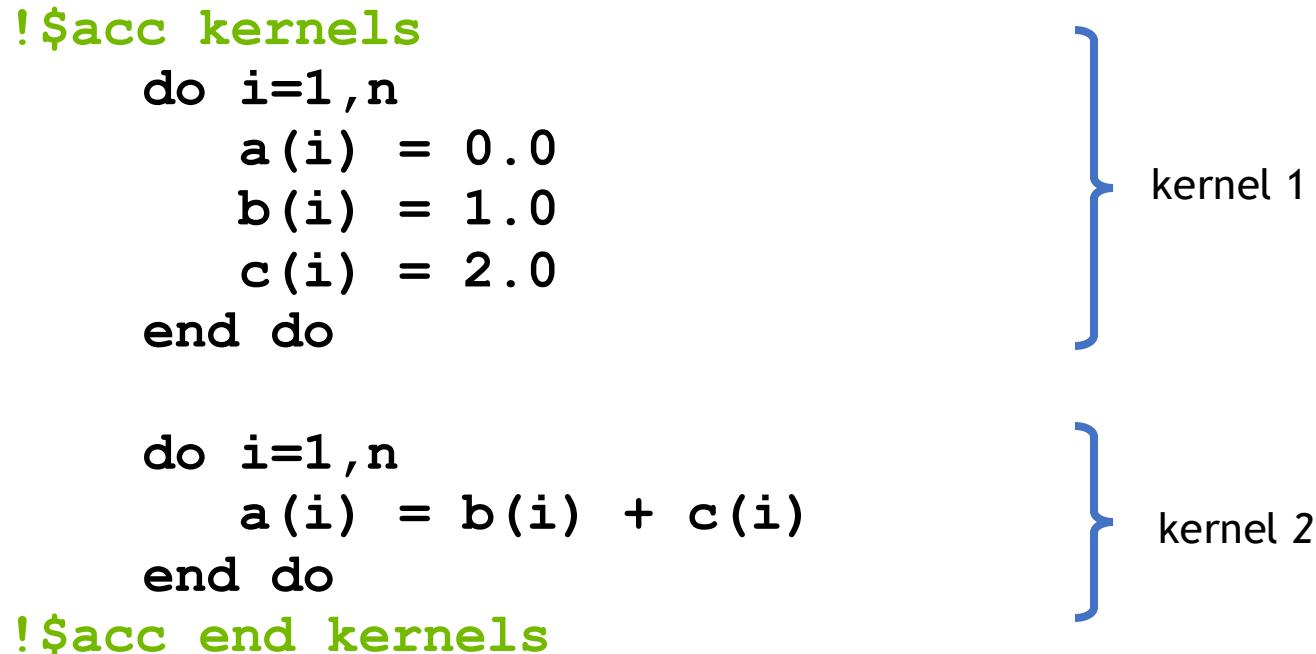
```
#pragma acc kernels [clause ...]
{ structured block }
```

# Kernels Construct

Each loop executed as a separate kernel on the GPU.

```
!$acc kernels
  do i=1,n
    a(i) = 0.0
    b(i) = 1.0
    c(i) = 2.0
  end do

  do i=1,n
    a(i) = b(i) + c(i)
  end do
 !$acc end kernels
```



The diagram illustrates the mapping of loops to kernels. The innermost loop, which initializes arrays a, b, and c, is grouped by a blue brace and labeled 'kernel 1'. The outermost loop, which performs the summation operation, is grouped by another blue brace and labeled 'kernel 2'.

# Parallel Construct

Fortran

```
!$acc parallel [clause ...]  
    structured block  
 !$acc end parallel
```

Clauses

```
if( condition )  
async( expression )  
num_gangs( expression )  
num_workers( expression )  
vector_length( expression )
```

C

```
#pragma acc parallel [clause ...]  
{ structured block }
```

```
private( list )  
firstprivate( list )  
reduction( operator:list )
```

Also any data clause

# Parallel Clauses

`num_gangs ( expression )`

Controls how many parallel gangs are created (CUDA `gridDim`).

`num_workers ( expression )`

Controls how many workers are created in each gang (CUDA `blockDim`).

`vector_length ( list )`

Controls vector length of each worker (SIMD execution)

`private( list )`

A copy of each variable in list is allocated to each gang

`firstprivate ( list )`

private variables initialized from host

`reduction( operator:list )`

private variables combined across gangs

# Loop Construct

Fortran

```
!$acc loop [clause ...]  
    loop  
!$acc end loop
```

Combined directives

```
!$acc parallel loop [clause ...]  
!$acc kernels loop [clause ...]
```

C

```
#pragma acc loop [clause ...]  
    { loop }
```

```
!$acc parallel loop [clause ...]  
!$acc kernels loop [clause ...]
```

Detailed control of the parallel execution of the following loop.

# Loop Clauses

`collapse( n )`

Applies directive to the following `n` nested loops.

`seq`

Executes the loop sequentially on the GPU.

`private( list )`

A copy of each variable in list is created for each iteration of the loop.

`reduction( operator:list )`

`private` variables combined across iterations.

# Loop Clauses Inside parallel Region

**gang**

Shares iterations across the gangs of the parallel region.

**worker**

Shares iterations across the workers of the gang.

**vector**

Execute the iterations in SIMD mode.

# Loop Clauses Inside kernels Region

`gang [ ( num_gangs ) ]`

Shares iterations across across at most *num\_gangs* gangs.

`worker [ ( num_workers ) ]`

Shares iterations across at most *num\_workers* of a single gang.

`vector [ ( vector_length ) ]`

Execute the iterations in SIMD mode with maximum *vector\_length*.

`independent`

Specify that the loop iterations are independent.

## **OTHER SYNTAX**

# Other Directives

**cache** construct

Cache data in software managed data cache (CUDA shared memory).

**host\_data** construct

Makes the address of device data available on the host.

**wait** directive

Waits for asynchronous GPU activity to complete.

**declare** directive

Specify that data is to be allocated in device memory for the duration of an implicit data region created during the execution of a subprogram.

# Runtime Library Routines

Fortran

```
use openacc  
#include "openacc_lib.h"
```

```
acc_get_num_devices  
acc_set_device_type  
acc_get_device_type  
acc_set_device_num  
acc_get_device_num  
acc_async_test  
acc_async_test_all
```

C

```
#include "openacc.h"  
  
acc_async_wait  
acc_async_wait_all  
acc_shutdown  
acc_on_device  
acc_malloc  
acc_free
```

# Environment and Conditional Compilation

`ACC_DEVICE device`

Specifies which device type to connect to.

`ACC_DEVICE_NUM num`

Specifies which device number to connect to.

`_OPENACC`

Preprocessor directive for conditional compilation.  
Set to OpenACC version