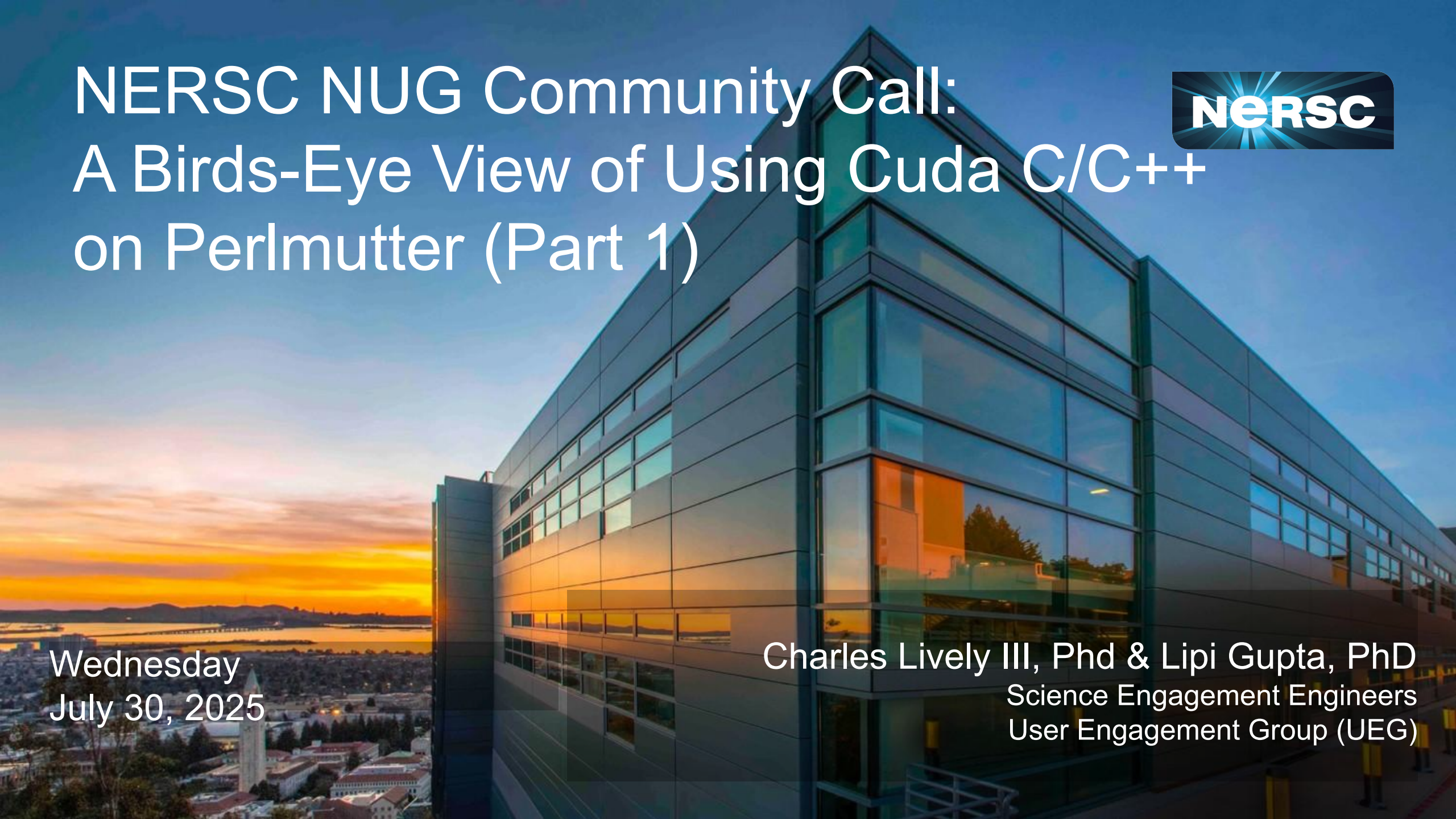
NERSC NUG Community Call: A Birds-Eye View of Using Cuda C/C++ on Perlmutter (Part 1)



Wednesday
July 30, 2025

Charles Lively III, Phd & Lipi Gupta, PhD
Science Engagement Engineers
User Engagement Group (UEG)

Agenda

- Welcome
- NERSC Updates/Housekeeping
- Cuda C/C++ (Part 1)
  - Introduction to Heterogeneous Parallel Computing
  - Portability and Scalability in Heterogeneous Parallel Computing
  - CUDA C vs. CUDA Libs vs. OpenACC
  - Memory Allocation and Data Movement API Functions
- Questions & Open Discussion

User Engagement Group - Our People

Rebecca Hartman-Baker
UEG Group Lead

Kevin Gott

Lipi Gupta

Lisa Claus

Liz Ball

Margie Wylie

Helen He

Charles Lively

Kelly Rowland

Annette Greiner

NERSC User Group (NUG)

- Community of NERSC users
- Source of advice and feedback for NERSC (we listen!)
- Regular teleconferences hosted by NERSC
- Join the NUG Slack: https://www.nersc.gov/users/user-news/join-the-nersc-users-slack-sponsored-by-nug-today

NERSC User Training

- NERSC provides a robust training program for users of all skill levels, interests, and personas
  - All trainings are recorded, professionally captioned, & posted to NERSC YouTube channel
  - Slides posted to training event webpage
- For more information on upcoming and past events, see https://www.nersc.gov/users/training/events/
- Training Events Archive
  - includes collection of previous archived training events

Some Logistics

- In-person attendees please also join Zoom for full participation
- Please change your name in Zoom session
  - to: first_name last_name
  - Click “Participants”, then “More” next to your name to rename
- Click the CC button to toggle captions and View Full Transcript
- Session is being recorded
- Users are muted upon joining Zoom
  - Feel free to unmute and ask questions or ask in GDoc below
- GDoc is used for Q&A (instead of Zoom chat)
  - https://tinyurl.com/2m3vn85u
- Please answering a short survey afterward
  - https://forms.gle/bdpRddkZiDxtMMYr8

Some Logistics

- Slides and videos will be available on NERSC Training Event page
  - https://www.nersc.gov/nug-community-calls/nug-community-call-a-birds-eye-view-of-using-cuda-with-cc-on-perlmutter-part-1
- Encourage to attend Perlmutter Office Hours, Jun-Aug 2025
  - https://www.nersc.gov/perlmutter-office-hours-2025
  - Weekly, on different weekdays, 1.5 hrs

Hands-on Exercises on Perlmutter

`ssh <user>@perlmutter.nersc.gov`, land on login node:

- `% cd $SCRATCH`
- `% git clone https://github.com/NERSC/NUG-CUDA-C-2025`
- References
  - Running Jobs: https://docs.nersc.gov/jobs/
  - Interactive Jobs: https://docs.nersc.gov/jobs/interactive/

Using Perlmutter Compute Node Reservations

- Existing NERSC users (at time of registration) have been added to "**`trn017`**" project
- Account in trn012 is valid through June 30
- Perlmutter node reservations: 10:30 am - 4:30 pm PDT today
  - **`--reservation=nug_cuda_c -A trn017 -C gpu`** for sbatch or salloc sessions
  - No need to use **`--reservation`** or **`-A`** when outside of the reservation hours

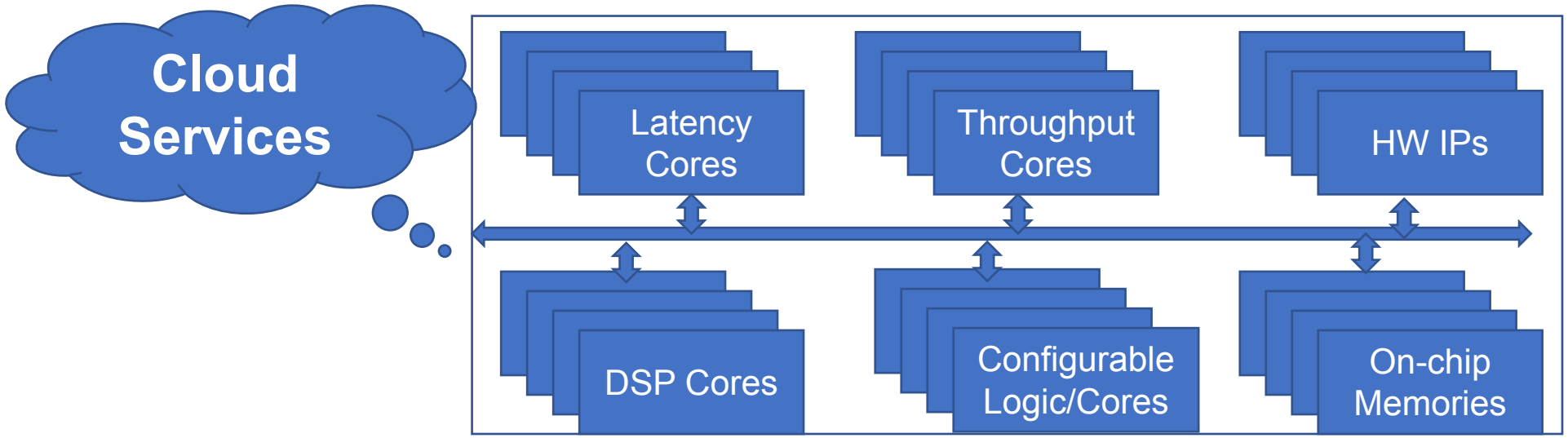
Heterogeneous Computing

Objectives

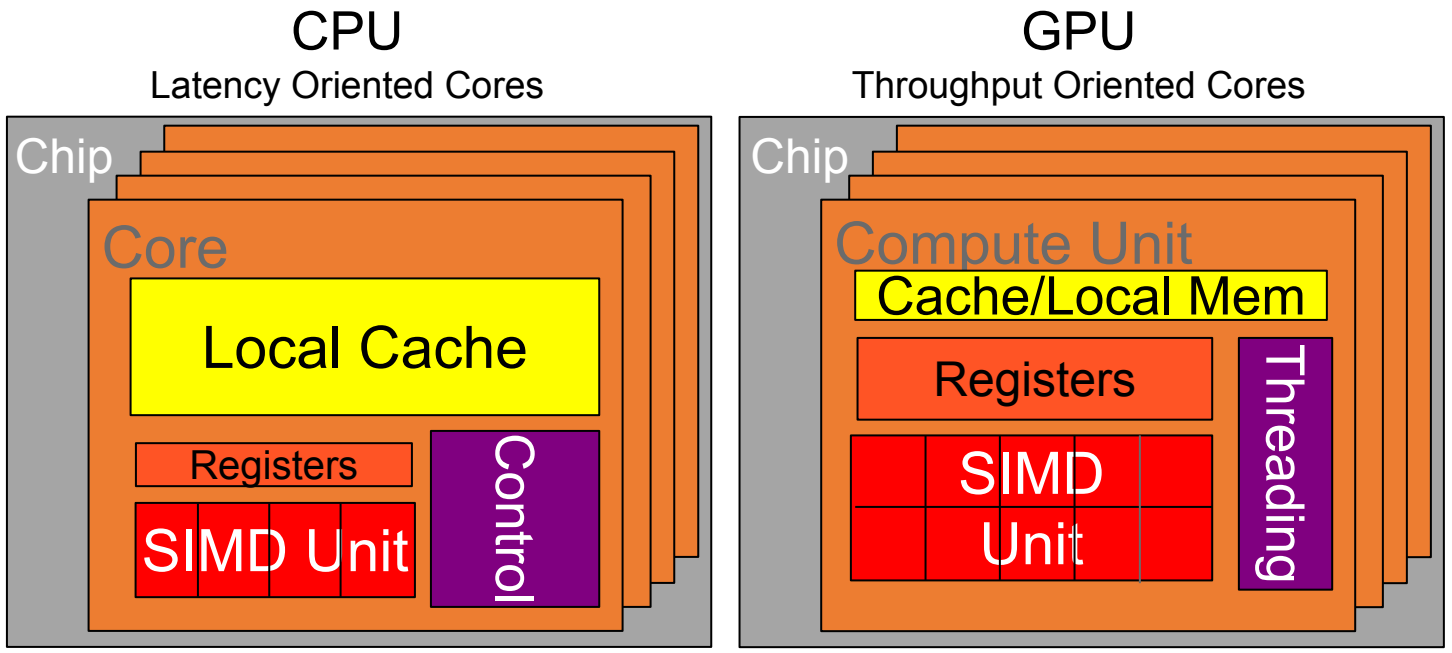
- To learn the major differences between latency devices (CPU cores) and throughput devices (GPU cores)
- To understand why winning applications increasingly use both types of devices

Heterogeneous Parallel Computing

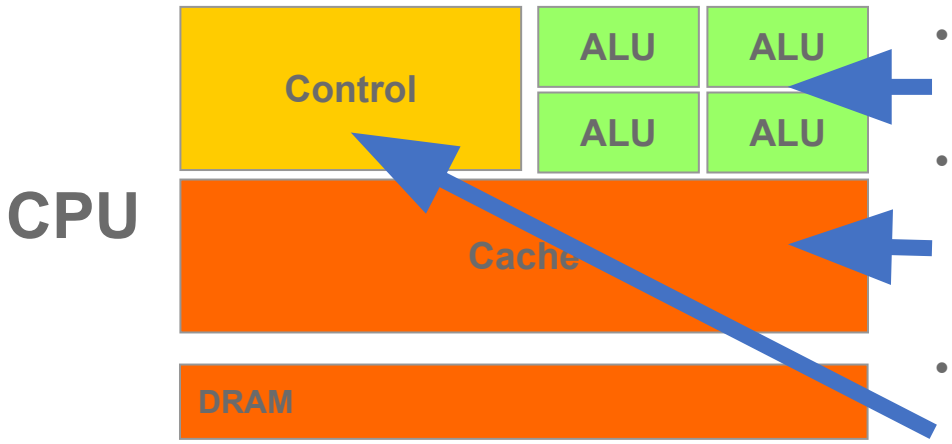
– Use the best match for the job (heterogeneity in mobile SOC)



CPU and GPU are designed very differently

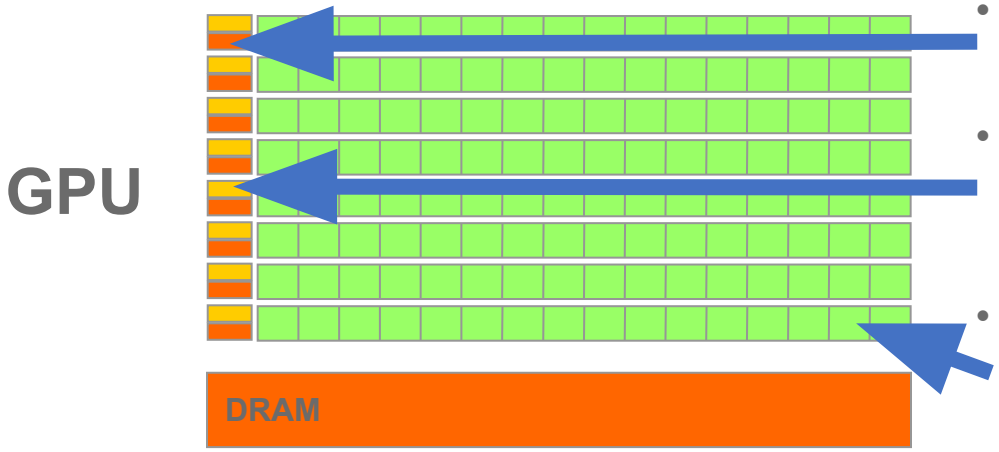


CPUs: Latency Oriented Design



- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency

GPUs: Throughput Oriented Design

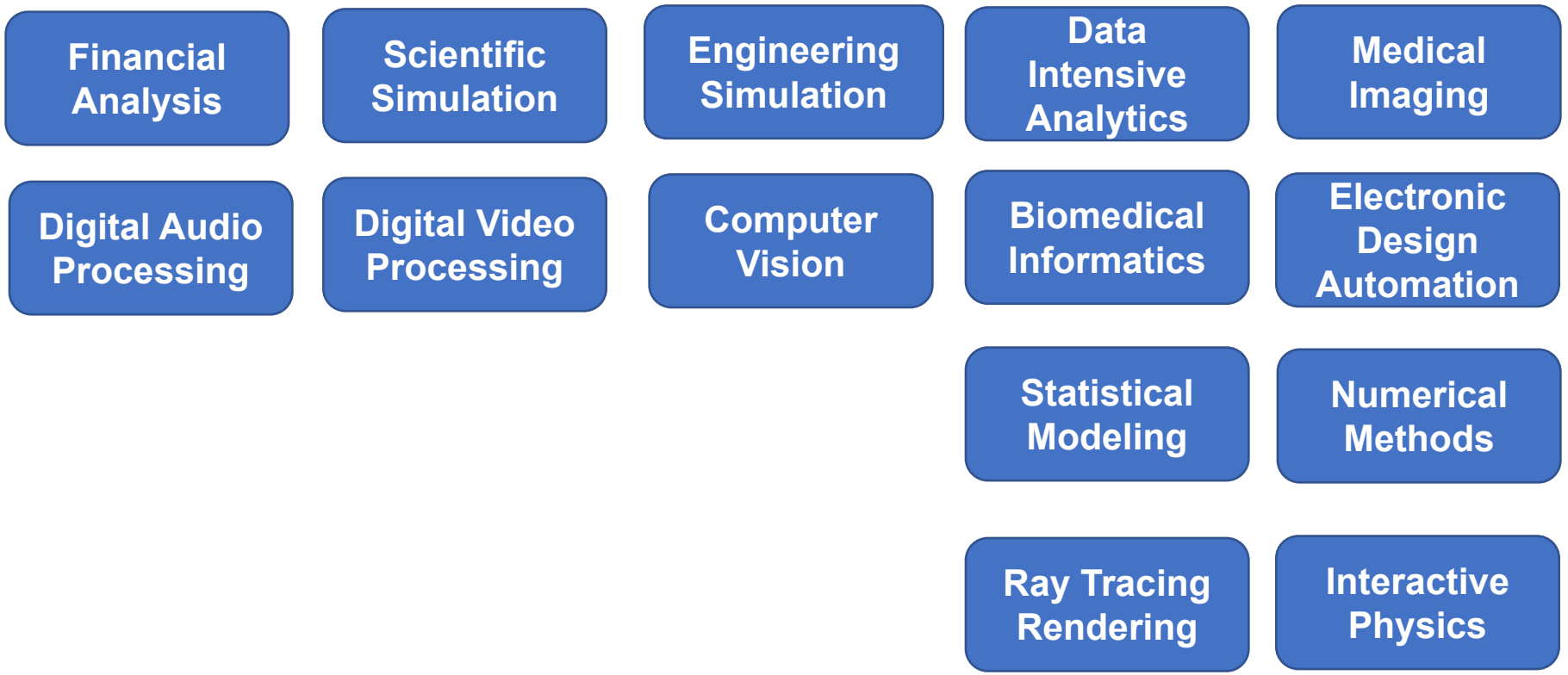


- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies
  - Threading logic
  - Thread state

Winning Applications Use Both CPU and GPU

- CPUs for sequential parts where latency matters
  - CPUs can be 10X+ faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
  - GPUs can be 10X+ faster than CPUs for parallel code

Heterogeneous Parallel Computing in Many Disciplines



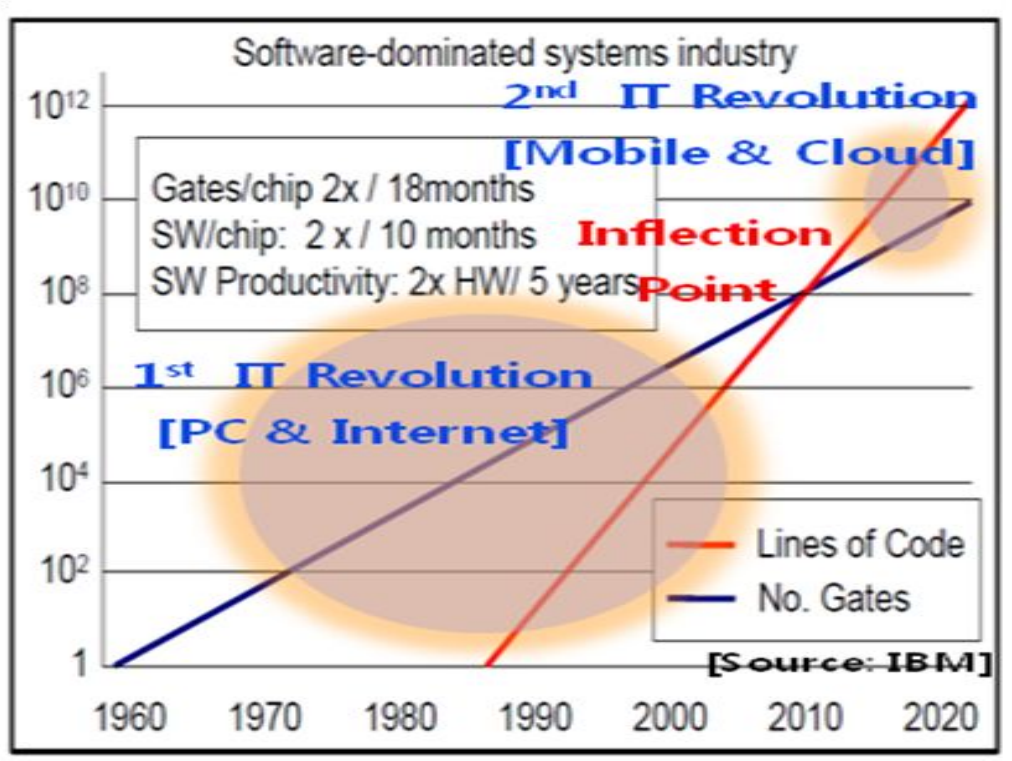
Scalability and Portability

Objectives

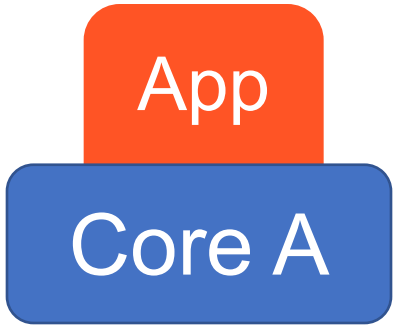
- To understand the importance and nature of scalability and portability in parallel programming

Software Dominates System Cost

- SW lines per chip increases at 2x/10 months
- HW gates per chip increases at 2x/18 months
- Future systems must minimize software redevelopment



Keys to Software Cost Control

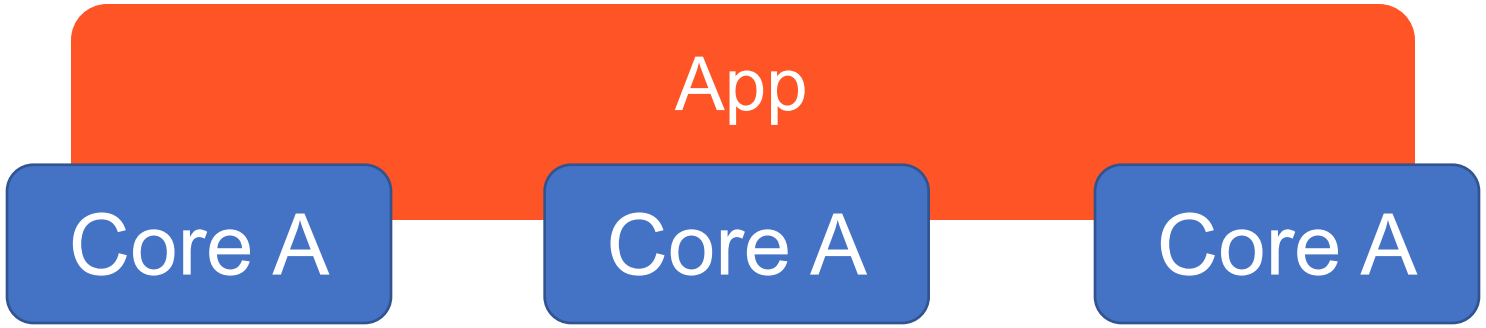


- Scalability

Keys to Software Cost Control

- Scalability
  - **The same application runs efficiently on new generations of cores**

Keys to Software Cost Control



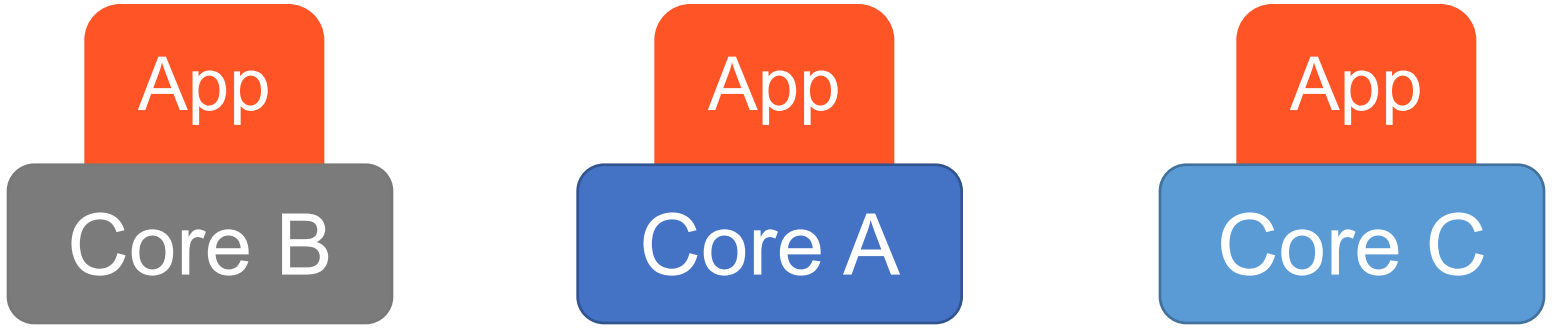
- Scalability
  - The same application runs efficiently on new generations of cores
  - **The same application runs efficiently on more of the same cores**

More on Scalability

- Performance growth with HW generations
  - Increasing number of compute units (cores)
  - Increasing number of threads
  - Increasing vector length
  - Increasing pipeline depth
  - Increasing DRAM burst size
  - Increasing number of DRAM channels
  - Increasing data movement latency

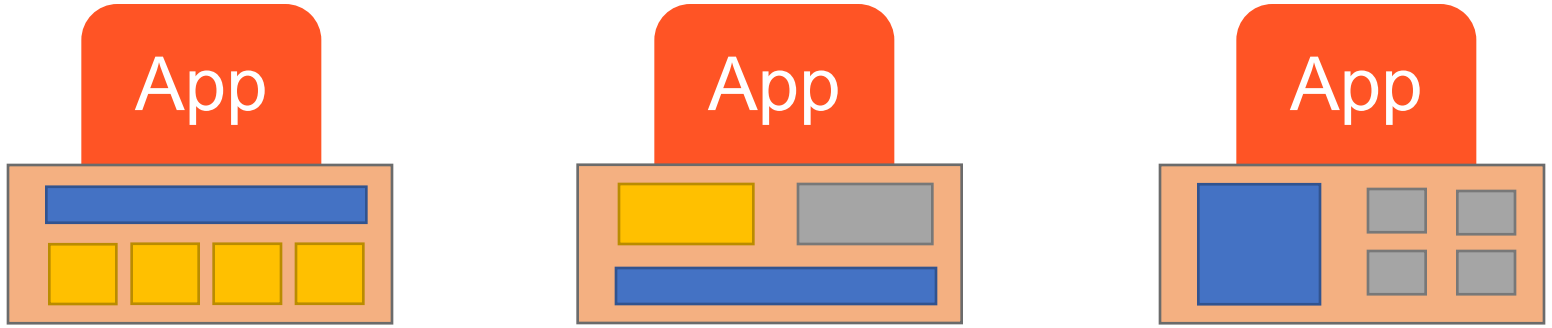
The programming style we use in this course supports scalability through fine-grained problem decomposition and dynamic thread scheduling

Keys to Software Cost Control



- Scalability
- **Portability**
  - The same application runs efficiently on different types of cores

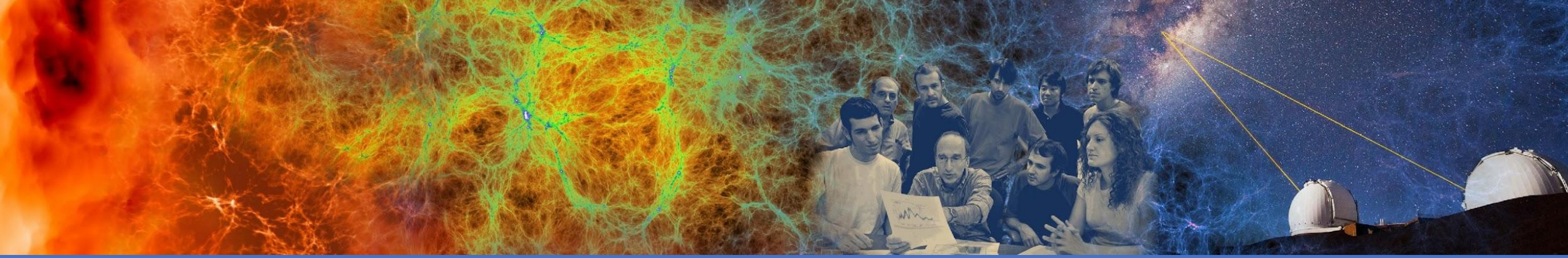
Keys to Software Cost Control



- Scalability
- Portability
  - The same application runs efficiently on different types of cores
  - The same application runs efficiently on systems with different organizations and interfaces

More on Portability

- Portability across many different HW types
  - Across ISAs (Instruction Set Architectures) - X86 vs. ARM, etc.
  - Latency oriented CPUs vs. throughput oriented GPUs
  - Across parallelism models - VLIW vs. SIMD vs. threading
  - Across memory models - Shared memory vs. distributed memory

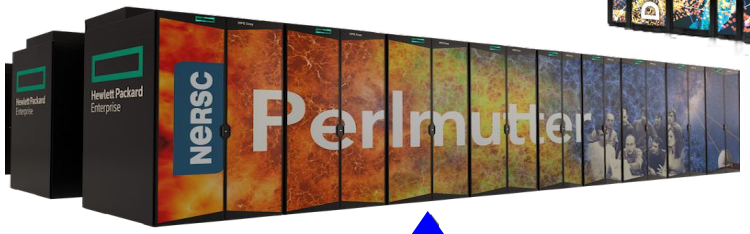


NERSC Hardware

NERSC system roadmap

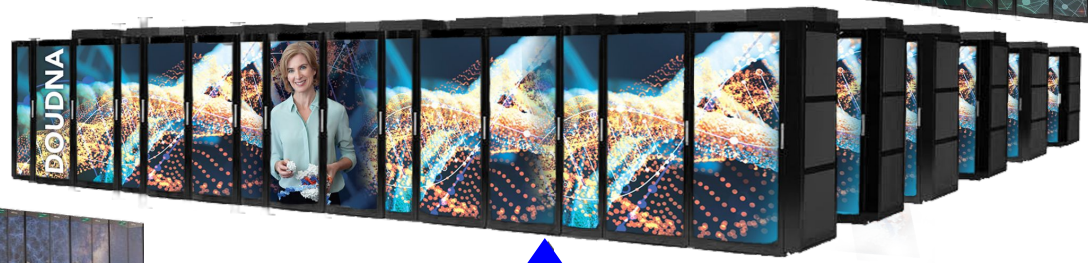
NERSC-8: Cori
Manycore CPU architectures

2016



NERSC-9: Perlmutter
CPU and GPU nodes
Expanded Simulation, Learning & Data

2020



NERSC-10: Doudna
GPU nodes
Accelerating end-to-end workflows

2026

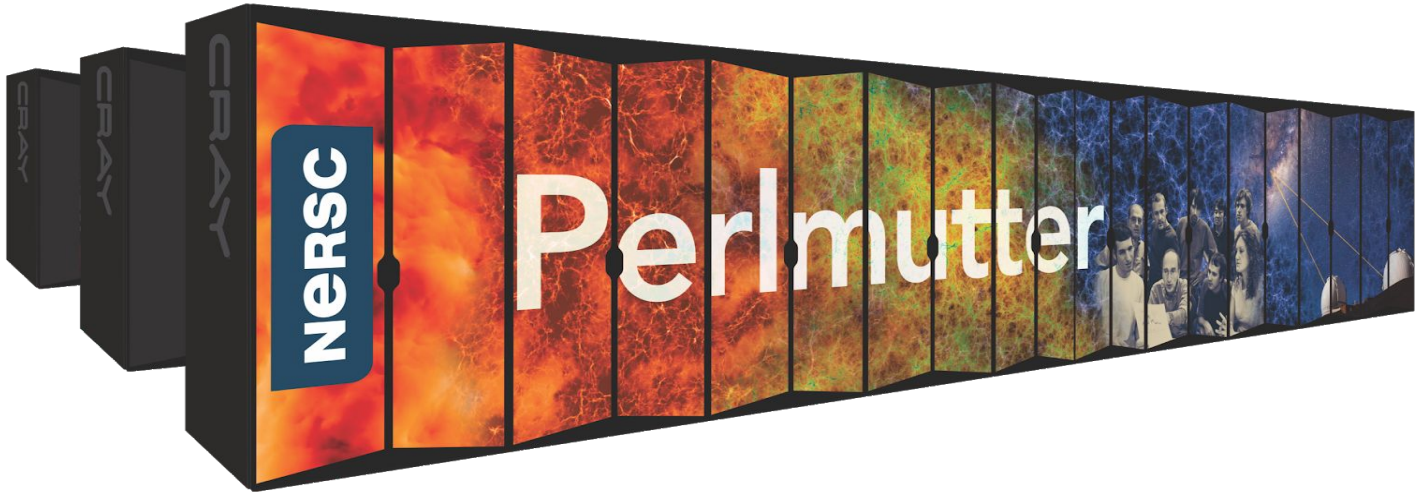
NERSC-11:
Beyond Moore

2030+

Increasingly energy-efficient architectures

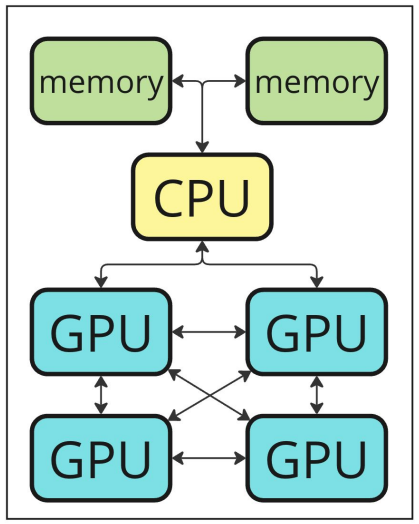
We name our systems after scientists

- Saul Perlmutter shared the 2011 Nobel Prize in Physics for discovery of the accelerating expansion of the universe.
- Supernova Cosmology Project, lead by Perlmutter, was a pioneer in using NERSC supercomputers combine large scale simulations with experimental data analysis
- Login “saul.nersc.gov”

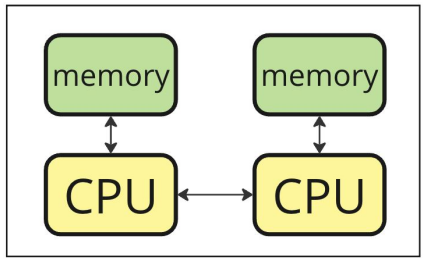


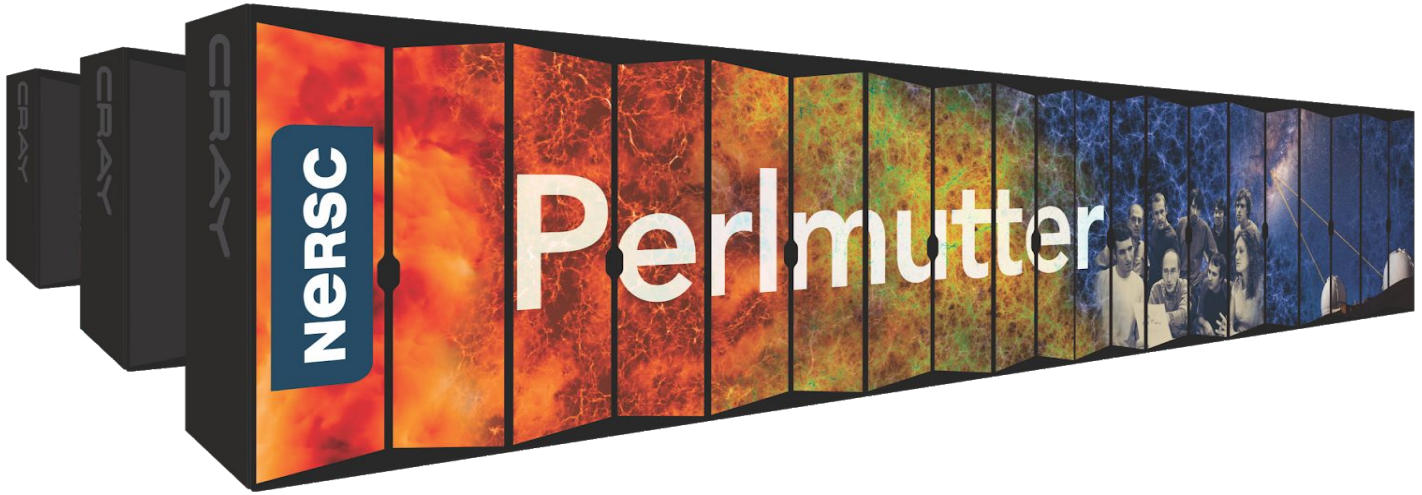
2 types of nodes

GPU nodes



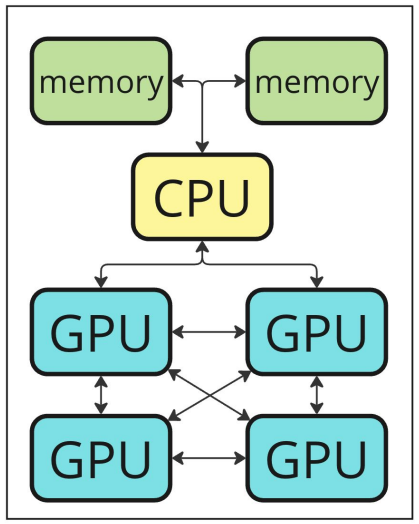
CPU nodes



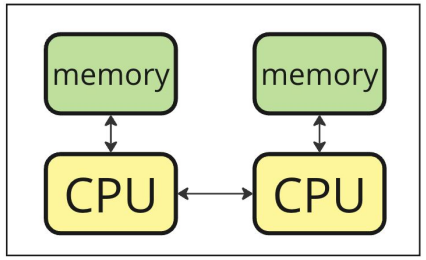


2 types of nodes

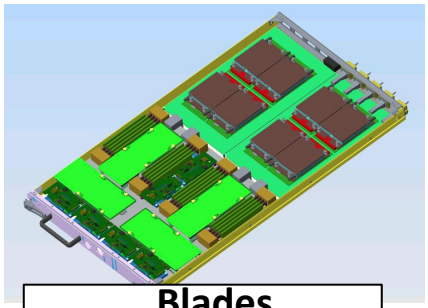
GPU nodes



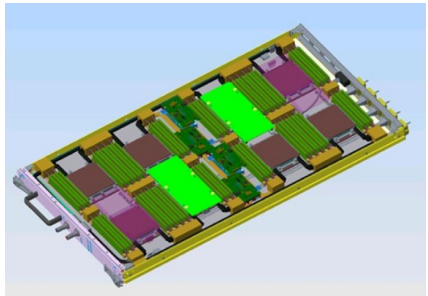
CPU nodes

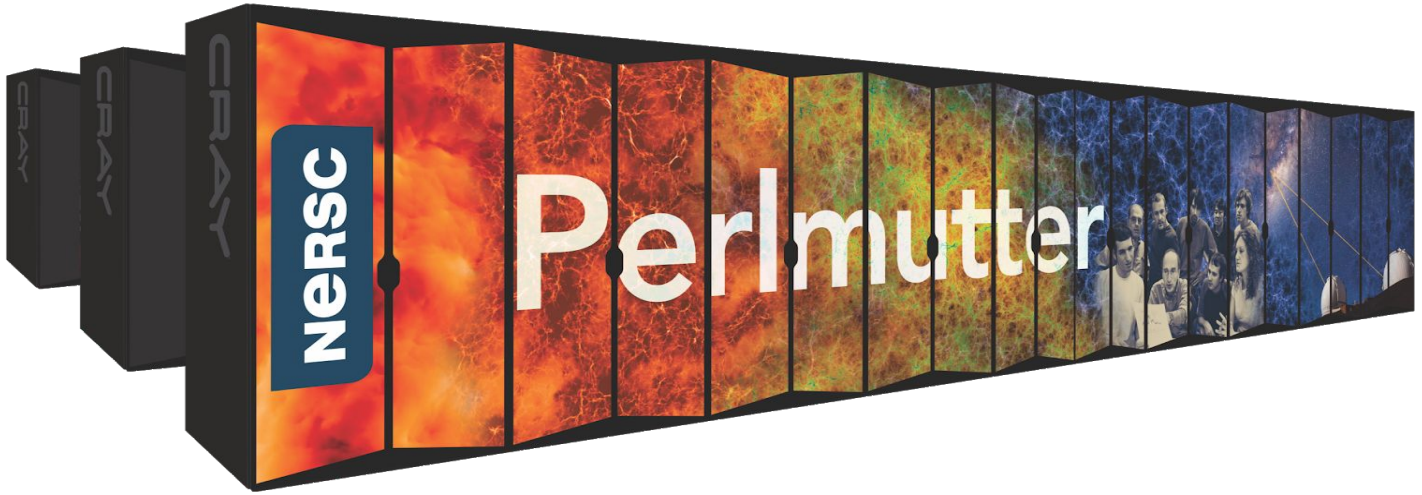


Nodes are combined into “blades”



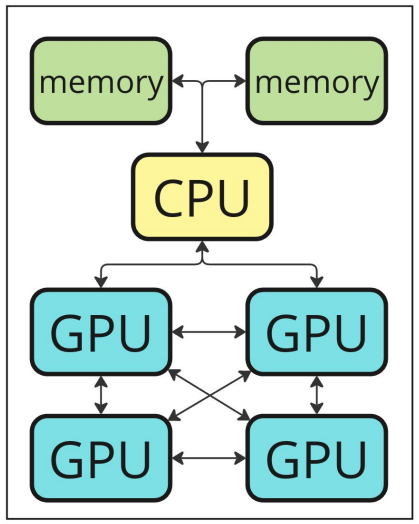
Blades
2x GPU nodes or
4x CPU nodes



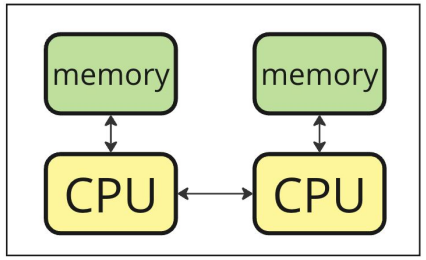


2 types of nodes

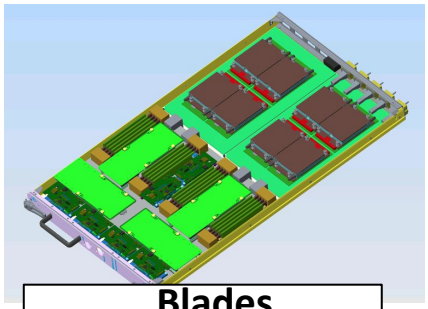
GPU nodes



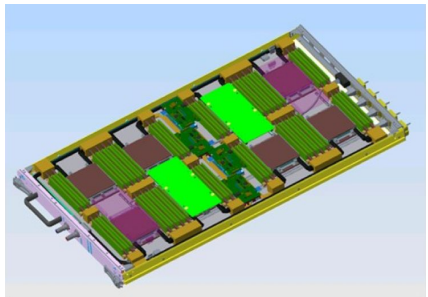
CPU nodes



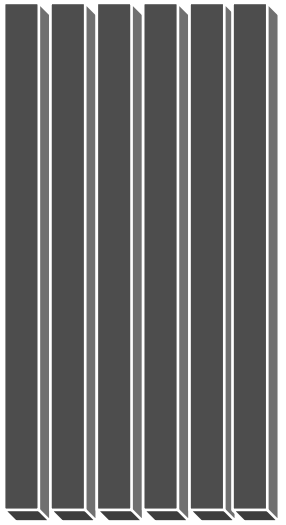
Nodes are combined into “blades”

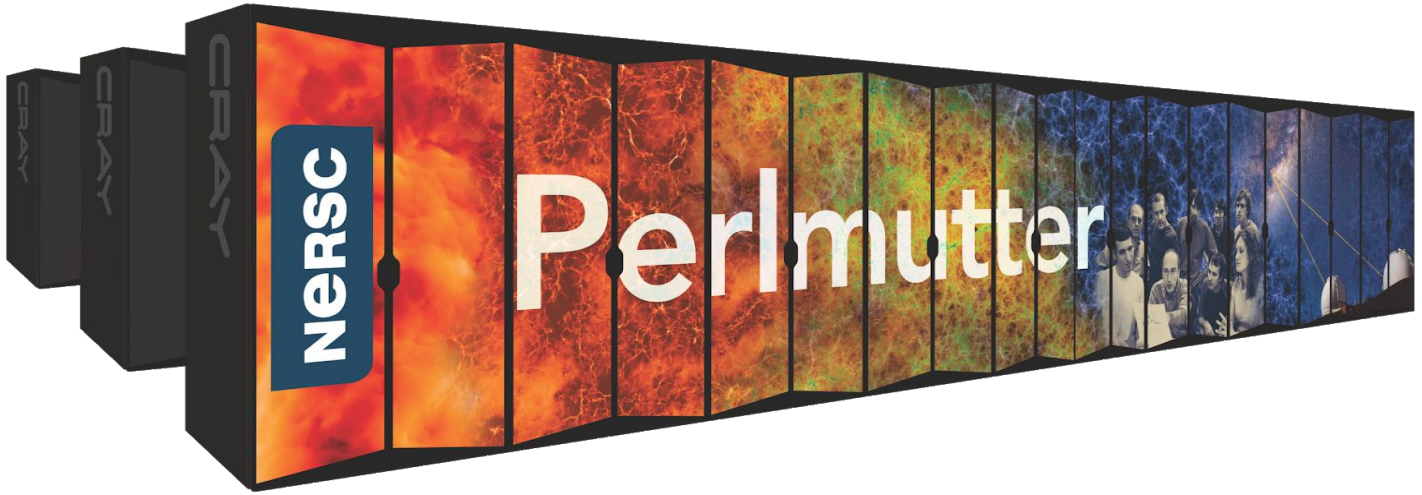


Blades
2x GPU nodes or
4x CPU nodes



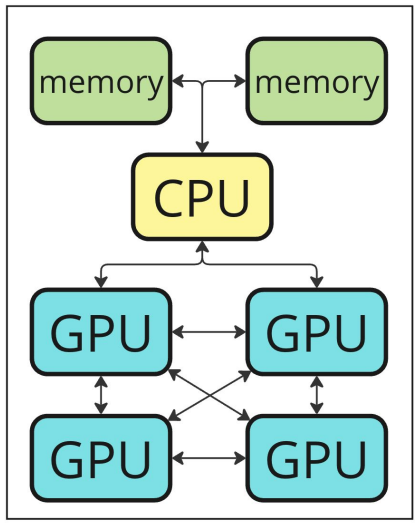
Stack blades sideways into server racks



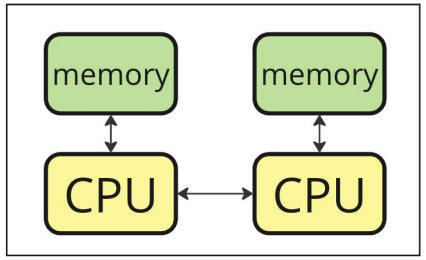


2 types of nodes

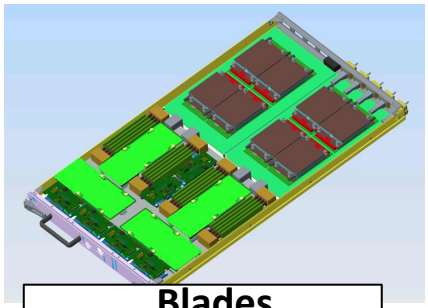
GPU nodes



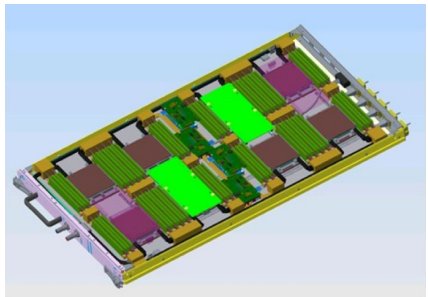
CPU nodes

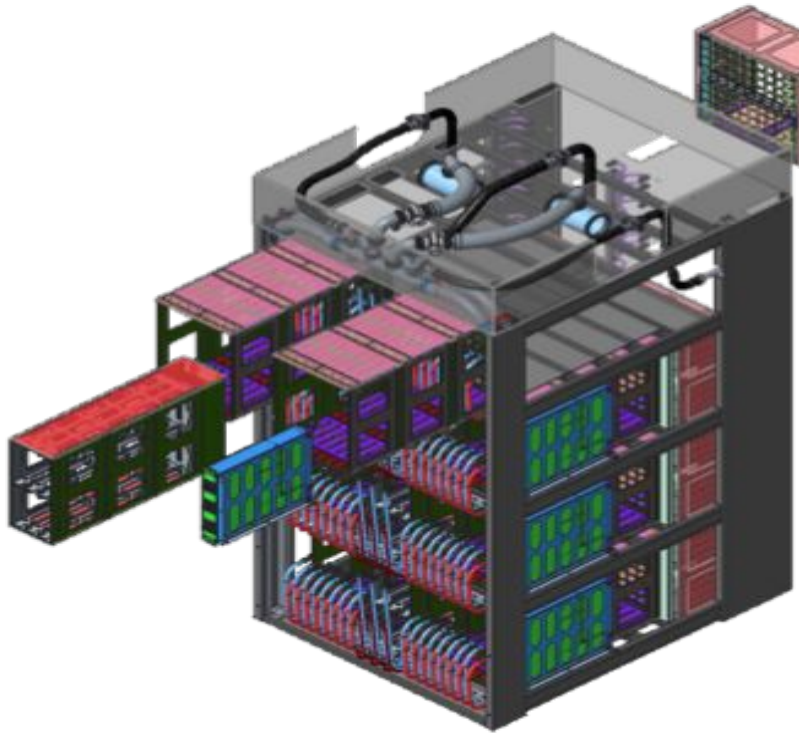


Nodes are combined into “blades”

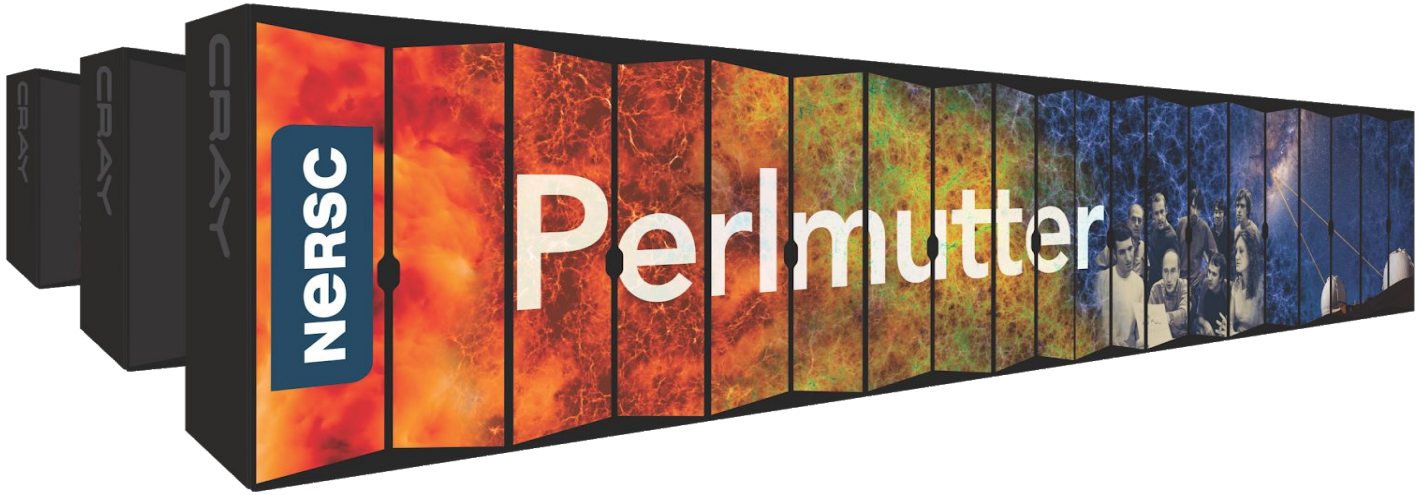


Blades
2x GPU nodes or
4x CPU nodes



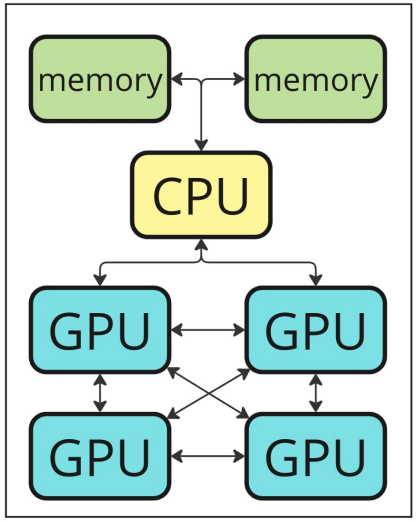


Compute racks
64 blades

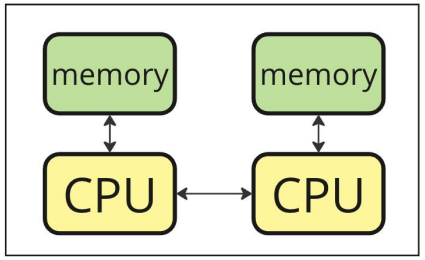


2 types of nodes

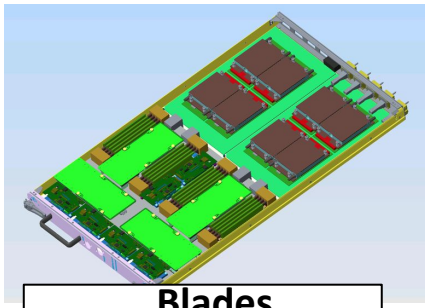
GPU nodes



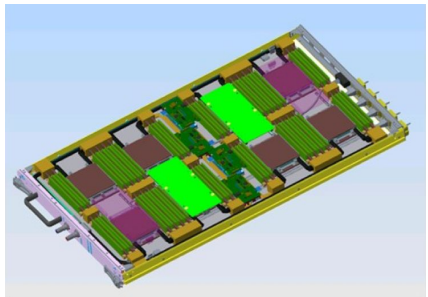
CPU nodes



Nodes are combined into “blades”



Blades
2x GPU nodes or
4x CPU nodes



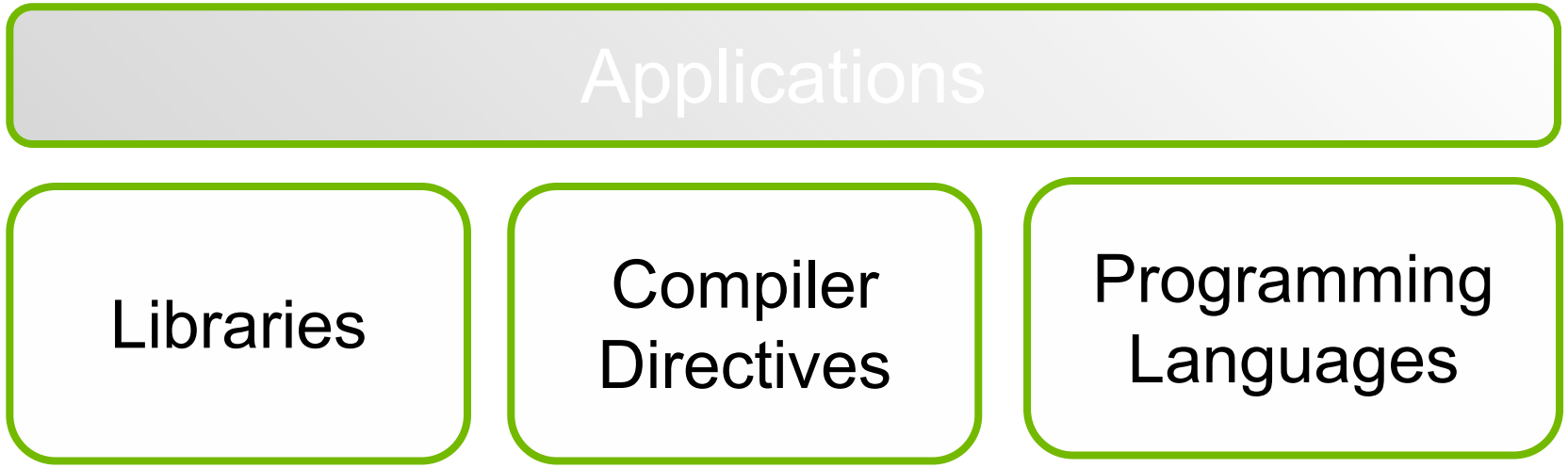
Compute racks
64 blades

Introduction to CUDA C

Objectives

- To learn the main venues and developer resources for GPU computing
  - Where CUDA C fits in the big picture

3 Ways to Accelerate Applications

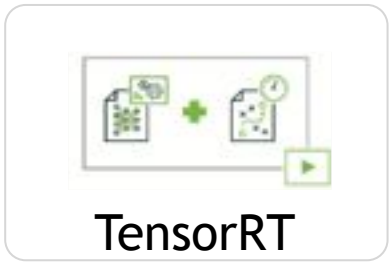


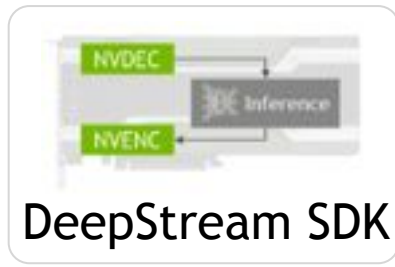
Libraries: Easy, High-Quality Acceleration

- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **"Drop-in":** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications

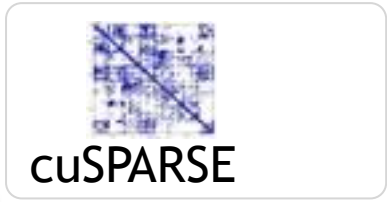
NVIDIA GPU Accelerated Libraries

DEEP LEARNING

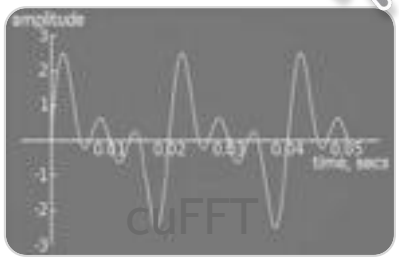




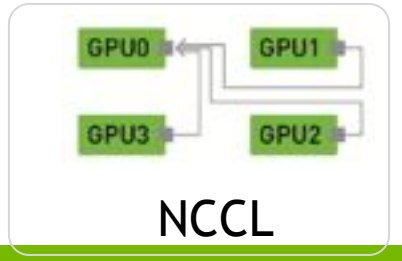
LINEAR ALGEBRA



SIGNAL, IMAGE, VIDEO



PARALLEL ALGORITHMS



Vector Addition in Thrust

```
#include <thrust/device_vector.h>
#include <thrust/copy.h>

int main(void) {
  size_t inputLength = 500;
  thrust::host_vector<float> hostInput1(inputLength);
  thrust::host_vector<float> hostInput2(inputLength);
  thrust::device_vector<float> deviceInput1(inputLength);
  thrust::device_vector<float> deviceInput2(inputLength);
  thrust::device_vector<float> deviceOutput(inputLength);

  thrust::copy(hostInput1.begin(), hostInput1.end(), deviceInput1.begin());
  thrust::copy(hostInput2.begin(), hostInput2.end(), deviceInput2.begin());

  thrust::transform(deviceInput1.begin(), deviceInput1.end(),
                    deviceInput2.begin(), deviceOutput.begin(),
                    thrust::plus<float>());
}
```

Compiler Directives: Easy, Portable Acceleration

- **Ease of use:** Compiler takes care of details of parallelism management and data movement
- **Portable:** The code is generic, not specific to any type of hardware and can be deployed into multiple languages
- **Uncertain:** Performance of code can vary across compiler versions

OpenACC

– Compiler directives for C, C++, and FORTRAN

```
#pragma acc parallel loop
copyin(input1[0:inputLength],input2[0:inputLength]),
copyout(output[0:inputLength])
    for(i = 0; i < inputLength; ++i) {
        output[i] = input1[i] + input2[i];
    }
```

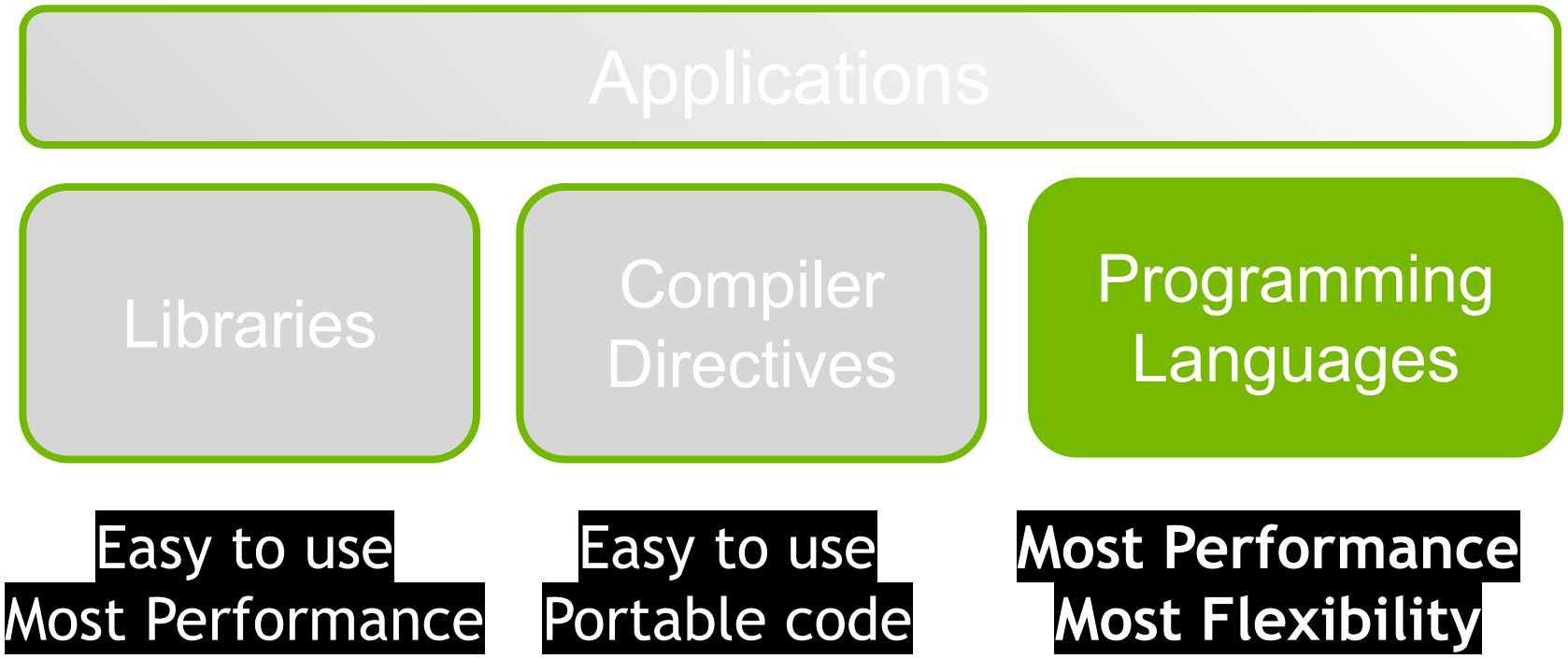
Programming Languages: Most Performance and Flexible Acceleration

- **Performance:** Programmer has best control of parallelism and data movement
- **Flexible:** The computation does not need to fit into a limited set of library patterns or directive types
- **Verbose:** The programmer often needs to express more details

GPU Programming Languages

Numerical analytics	MATLAB,, Mathematica, LabVIEW
Python	PyCUDA, Numba
Fortran	CUDA Fortran, OpenACC
C	CUDA C, OpenACC
C++	CUDA C++, Thrust
C#	Hybridizer

CUDA - C

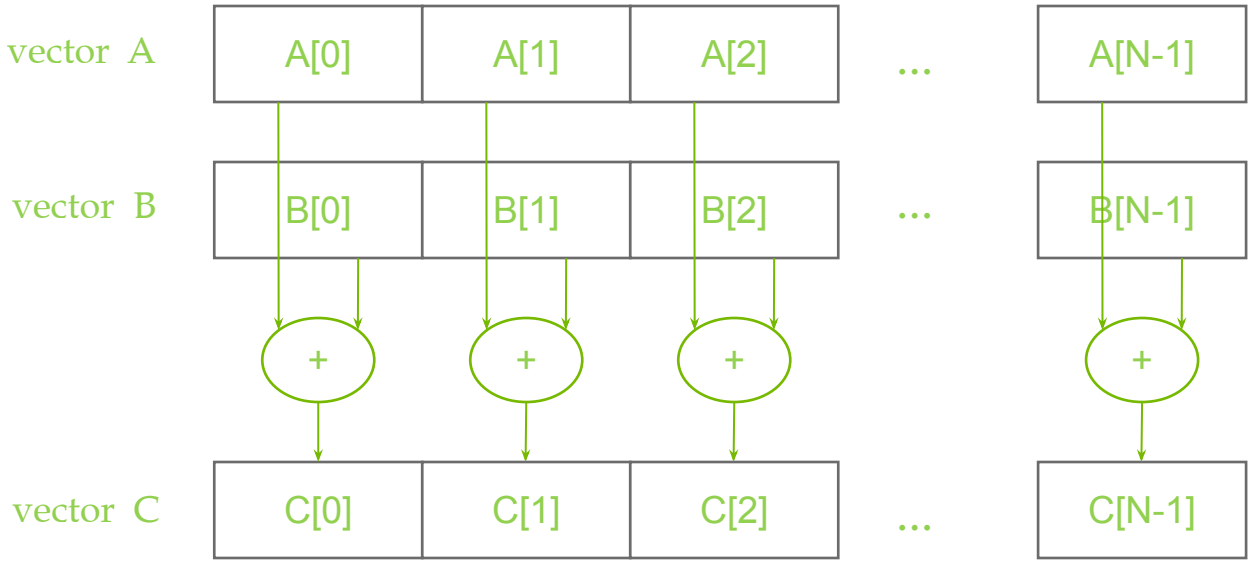

Applications
Libraries
Compiler Directives
Programming Languages
Easy to use
Most Performance
Easy to use
Portable code
Most Performance
Most Flexibility

CUDA C API Functions

Objectives

- To learn the basic API functions in CUDA host code
  - Device Memory Allocation
  - Host-Device Data Transfer

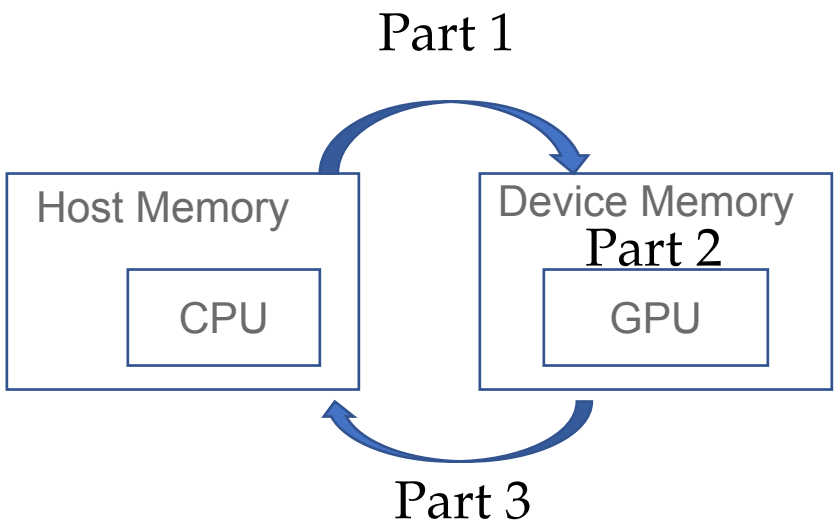
Data Parallelism - Vector Addition Example



Vector Addition – Traditional C Code

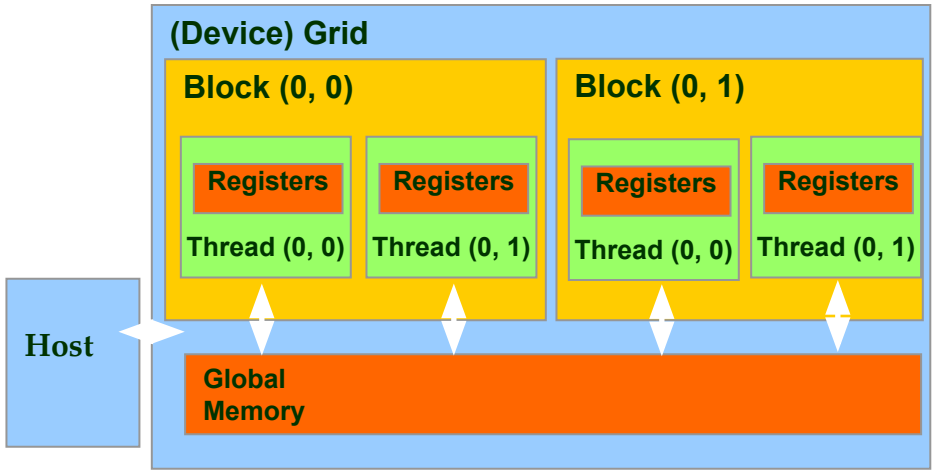
```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

Heterogeneous Computing vecAdd CUDA Host Code




```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n* sizeof(float);
    float *d_A, *d_B, *d_C;
    // Part 1
    // Allocate device memory for A, B, and C
    // copy A and B to device memory

    // Part 2
    // Kernel launch code – the device performs the actual vector addition

    // Part 3
        // copy C from the device memory

}
```

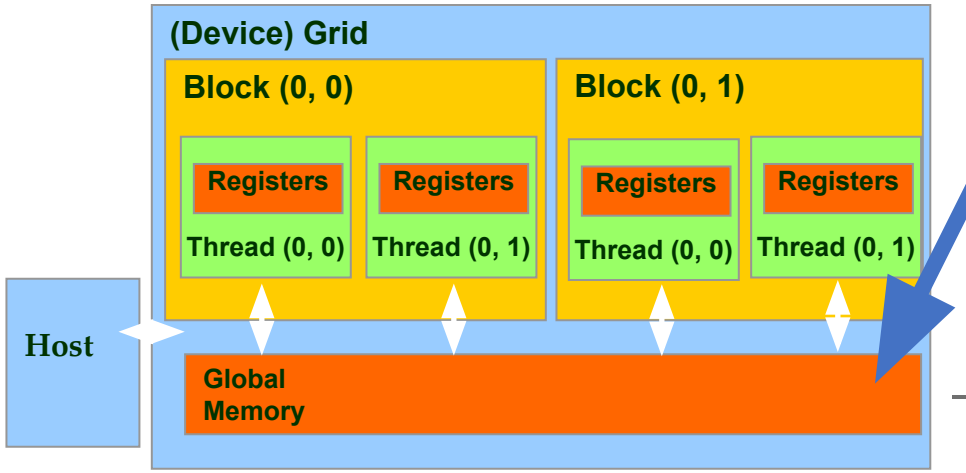
Partial Overview of CUDA Memories



- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
- Host code can
  - Transfer data to/from per grid global memory

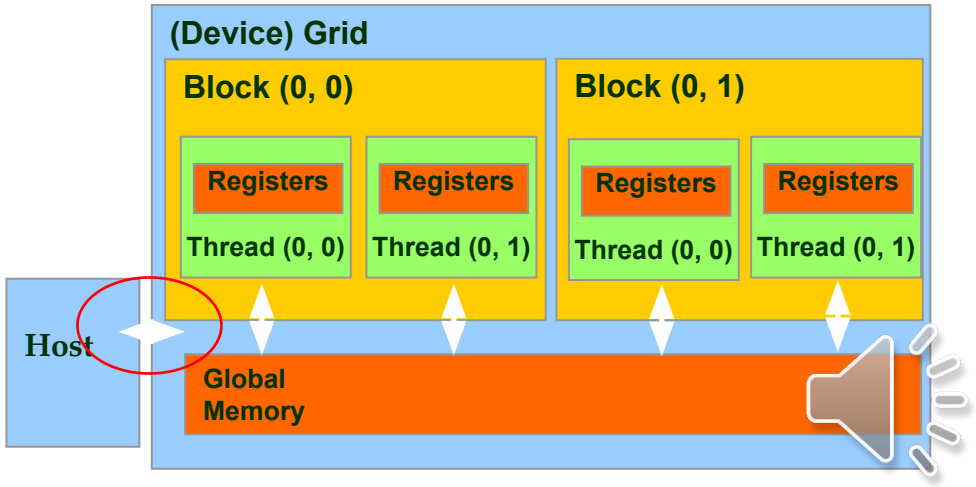
We will cover more memory types and more sophisticated memory models later.

CUDA Device Memory Management API functions



- cudaMalloc()
  - Allocates an object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
  - One parameter
    - **Pointer** to freed object

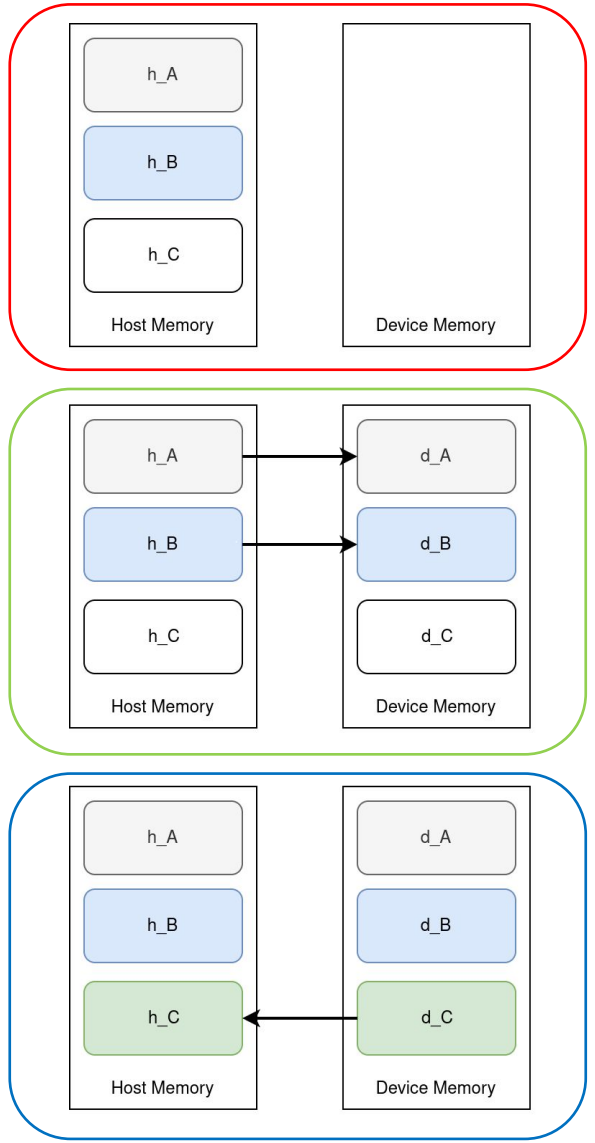
Host-Device Data Transfer API functions



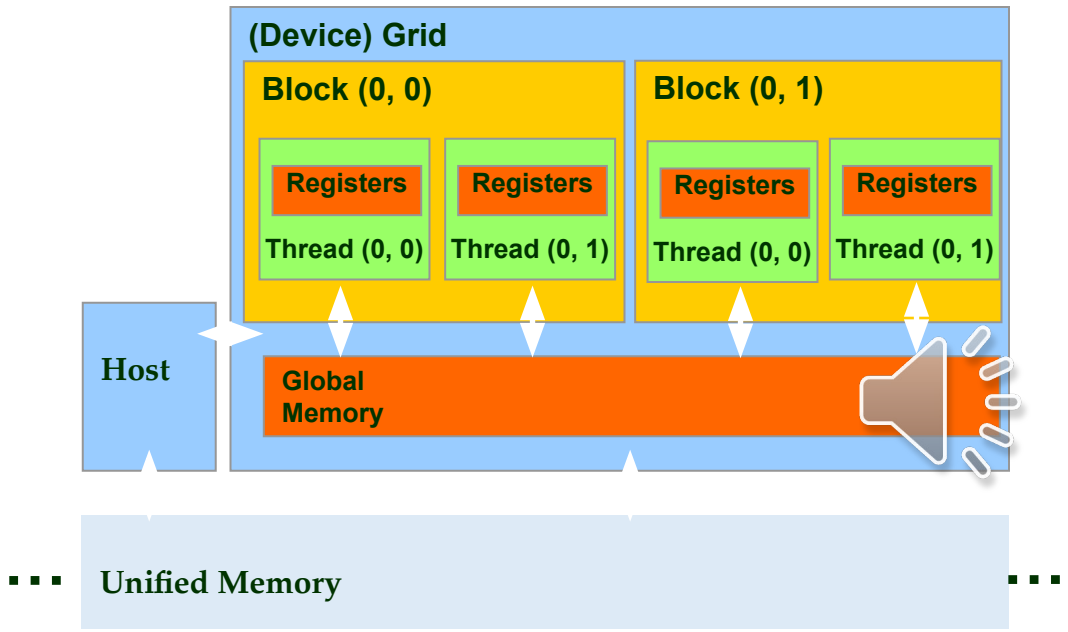
- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is synchronous with respect to the host

Vector Addition, Explicit Memory Management

```
… Allocate h_A, h_B, h_C …

void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Kernel invocation code – to be shown later

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}

… Free h_A, h_B, h_C …
```



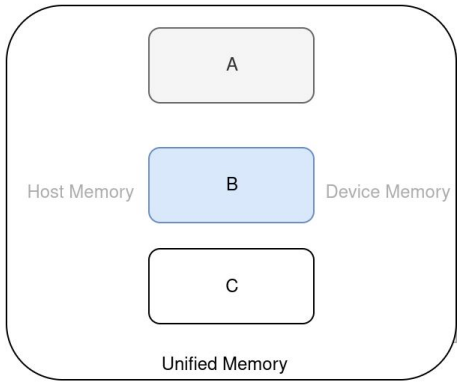
Unified Memory



- cudaMallocManaged(void** ptr, size_t size)
  - Single memory space for all CPUs/GPUs
    - Maintain single copy of data
  - CUDA-managed data
    - On-demand page migration
  - Compatible with cudaMalloc(), cudaFree()
  - Can be optimized
    - cudaMemAdvise(), cudaMemPrefetchAsync(),
    - cudaMemcpyAsync()

Vector Addition, Unified Memory

```
float *A, *B, *C
cudaMallocManaged(&A, n * sizeof(float));
cudaMallocManaged(&B, n * sizeof(float));
cudaMallocManaged(&C, n * sizeof(float));

// Initialize A, B

void vecAdd(float *A, float *B, float *C, int n)
{
 // Kernel invocation code – to be shown later
}

cudaFree(A);
cudaFree(B);
cudaFree(C);
```



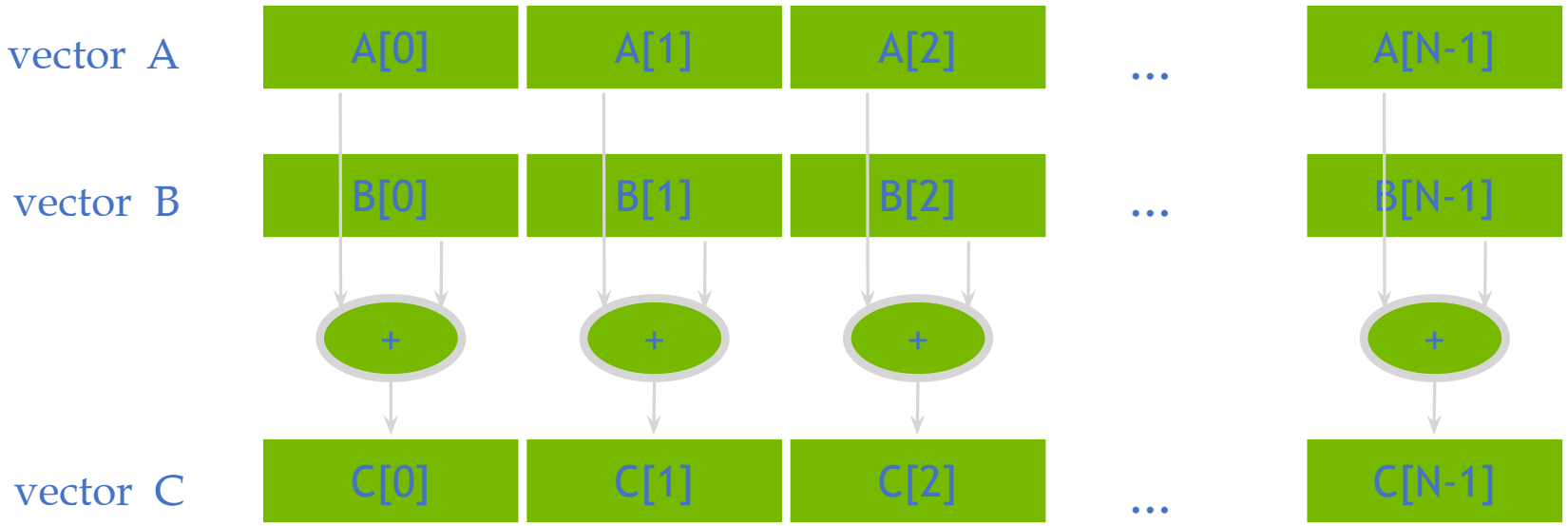
In Practice, Check for API Errors in Host Code

```
cudaError_t  err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess)  {
   printf("%s in %s at line %d\n",   cudaGetErrorString(err), __FILE__,
   __LINE__);
   exit(EXIT_FAILURE);
}
```

Objective

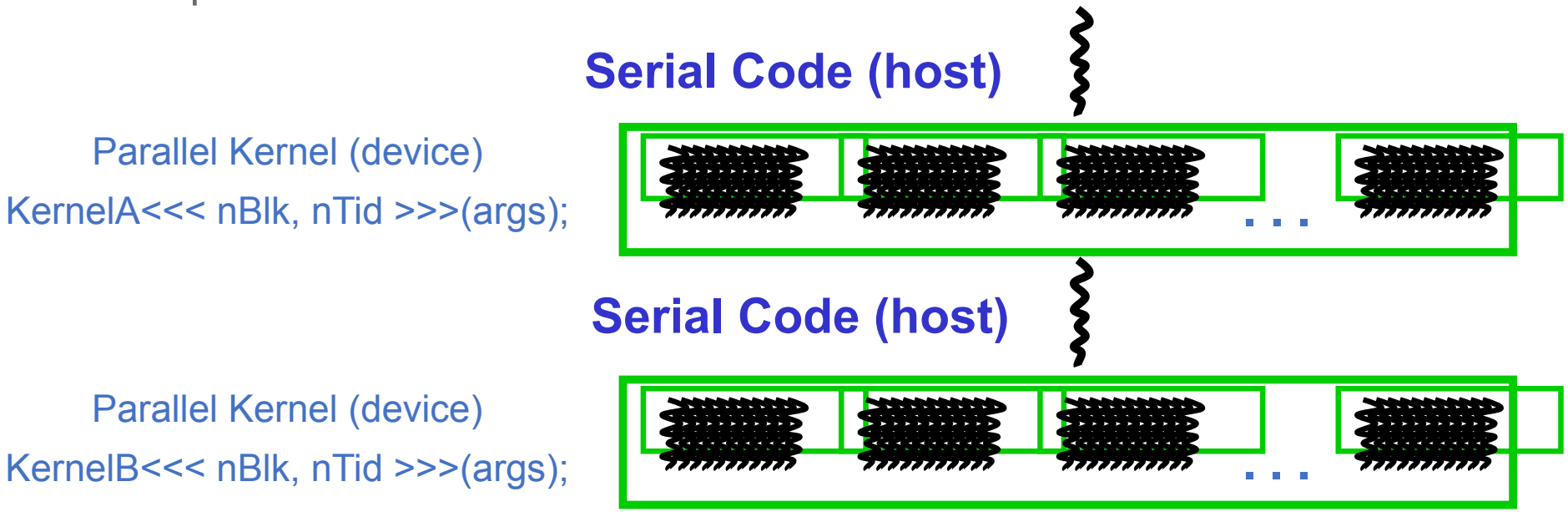
- To learn about CUDA threads, the main mechanism for exploiting of data parallelism
  - Hierarchical thread organization
  - Launching parallel execution
  - Thread index to data index mapping

Data Parallelism - Vector Addition Example

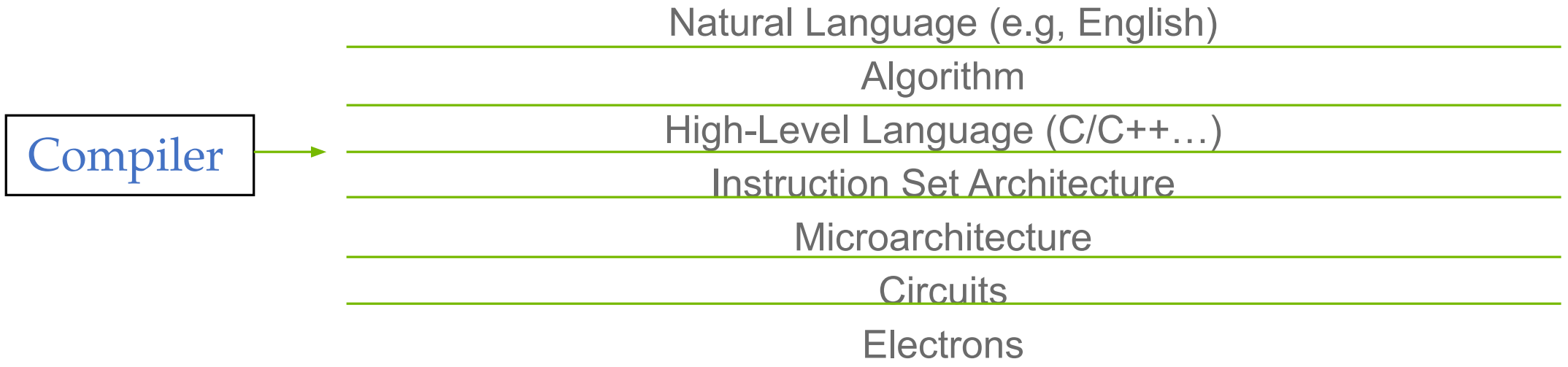


CUDA Execution Model

- Heterogeneous host (CPU) + device (GPU) application C program
  - Serial parts in **host** C code
  - Parallel parts in **device** SPMD kernel code



From Natural Language to Electrons



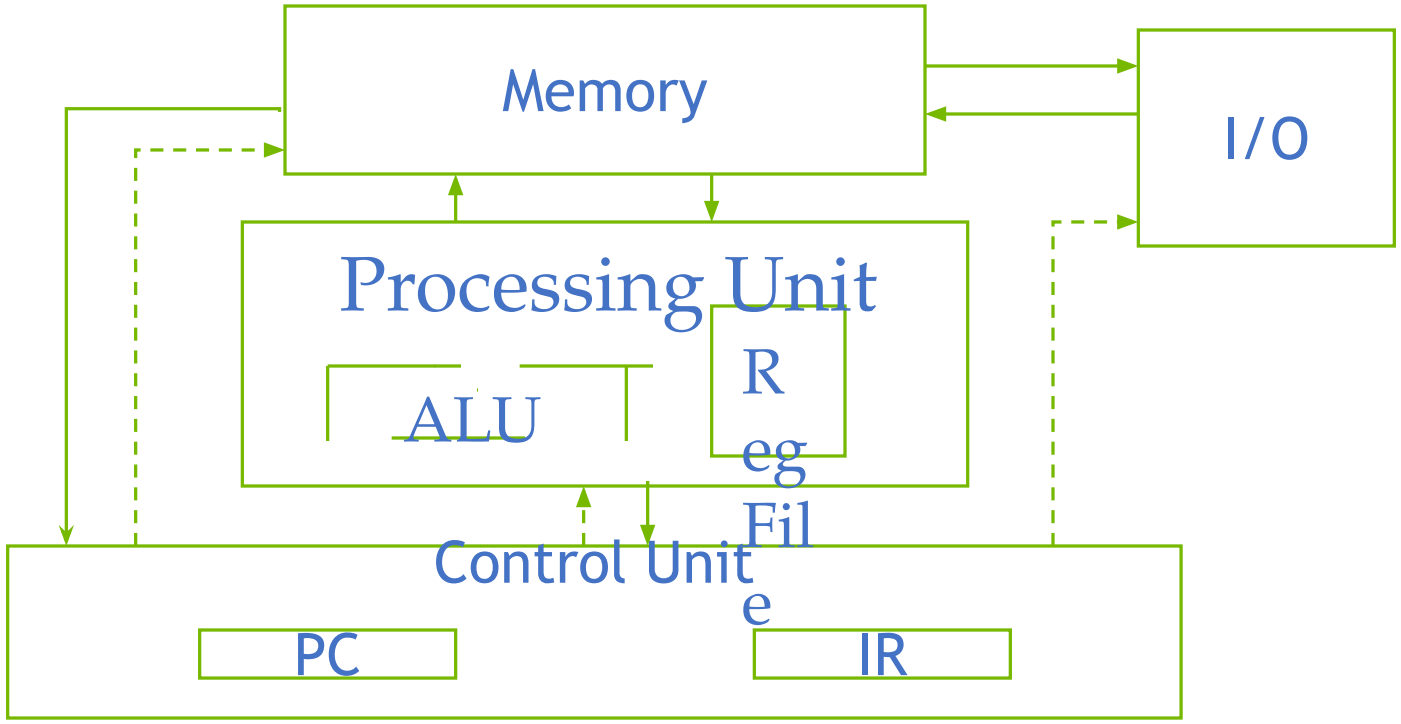
A program at the ISA level

- A program is a set of instructions stored in memory that can be read, interpreted, and executed
  - Both CPUs and GPUs are designed based on (different) instruction sets

- Program instructions operate on data stored in memory and/or registers.

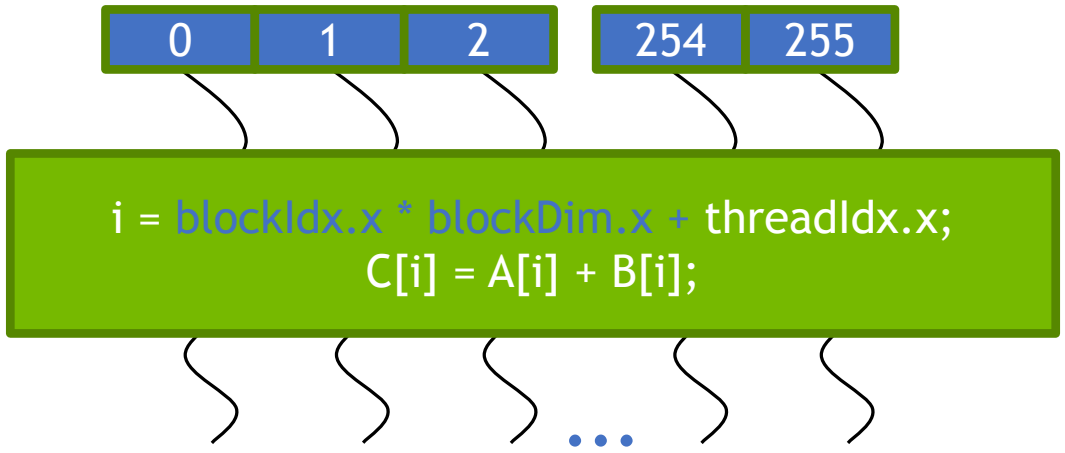
A Thread as a Von-Neumann Processor

A thread is a “virtualized” or “abstracted” Von-Neumann Processor

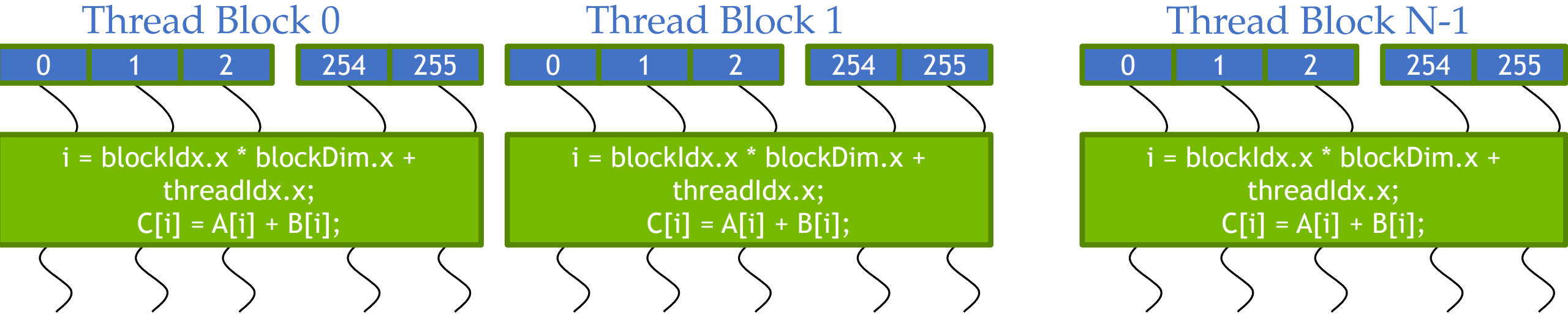


Arrays of Parallel Threads

- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (Single Program Multiple Data)
  - Each thread has indexes that it uses to compute memory addresses and make control decisions



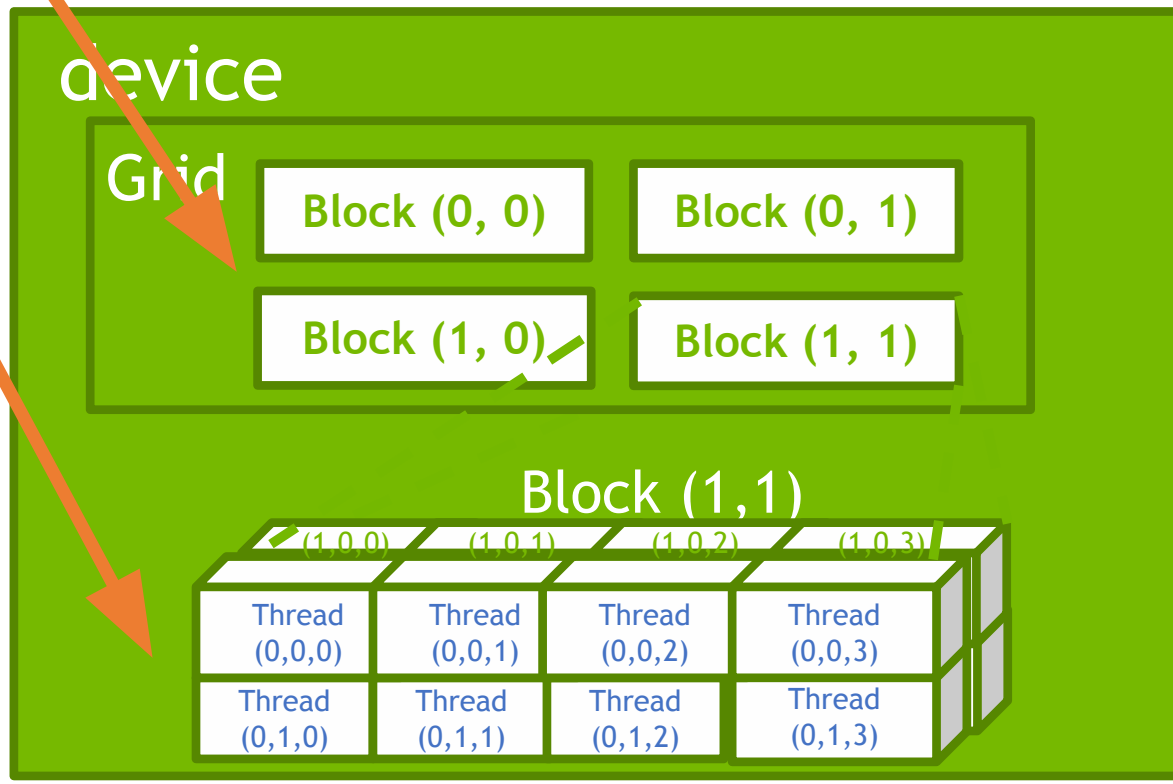
Thread Blocks: Scalable Cooperation



- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks do not interact

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



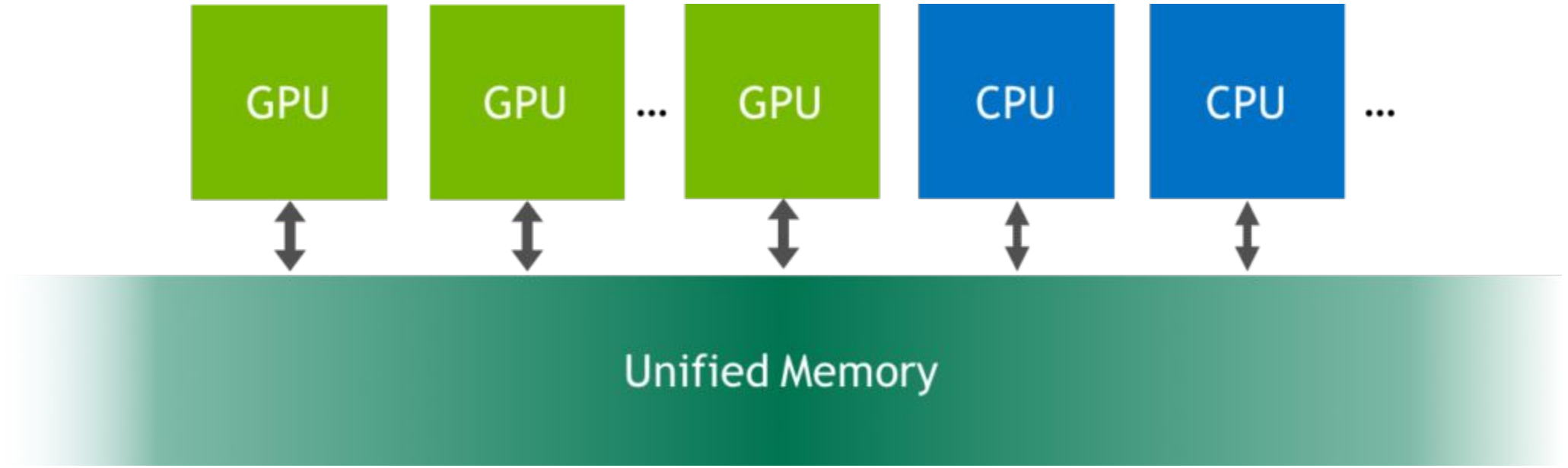
Objectives

-To learn the basic API functions in CUDA host code for CUDA Unified Memory

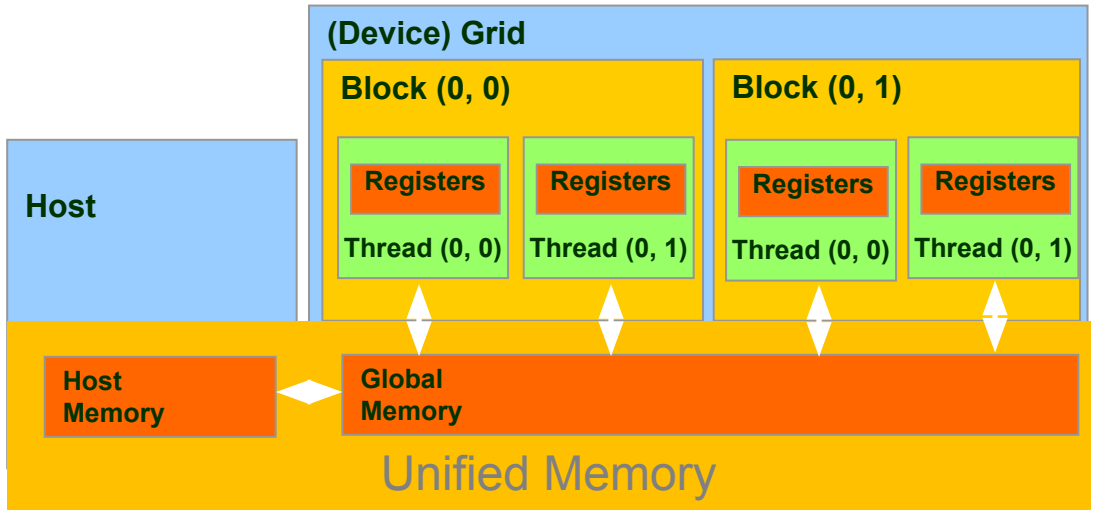
- Unified Memory Allocation
- Data Transfer in Unified Memory

CUDA Unified Memory (UM)

- Is a single memory address space accessible both from the host and from the device.
- The hardware/software handles automatically the data migration between the host and the device maintaining consistency between them.

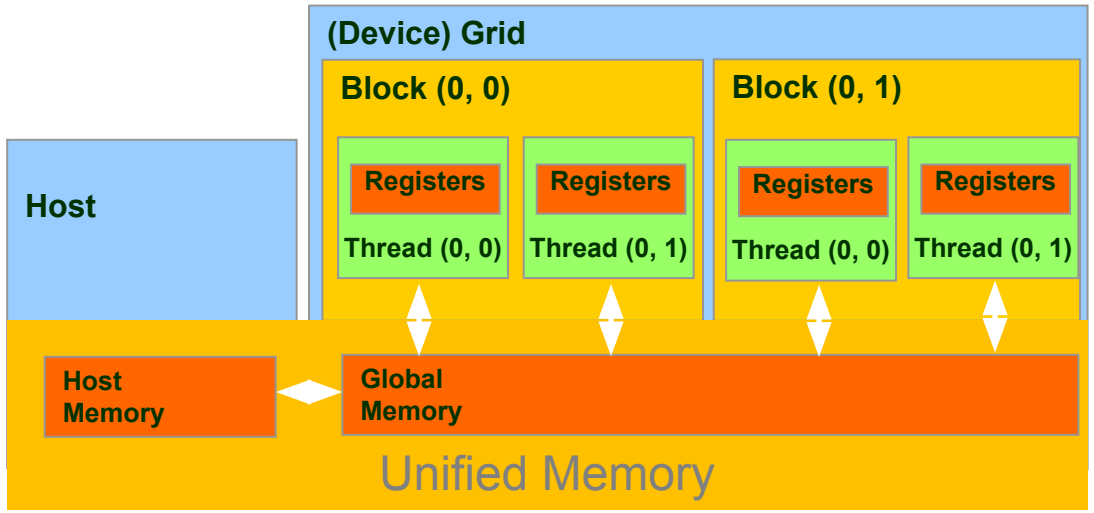


Partial Overview of CUDA Memories



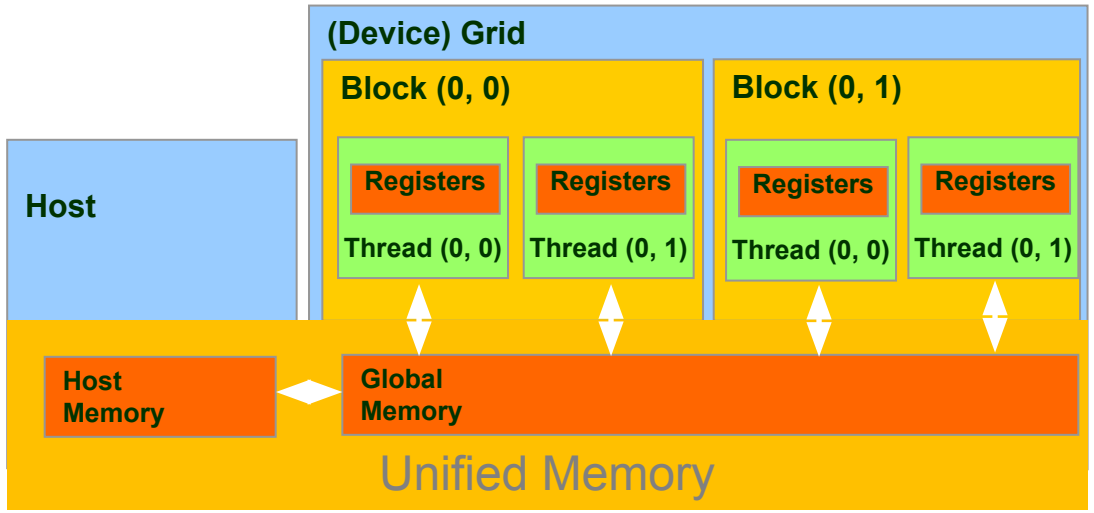
- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
  - R/W managed memory (Unified Memory)
- Host code can
  - Transfer data to/from per grid global memory
  - R/W managed memory

Partial Overview of CUDA Memories



- cudaMallocManaged()
  - Allocates an object in the Unified Memory address space.
  - Two parameters, with an optional third parameter.
    - Address of a pointer to the allocated object
    - Size of the allocated object in terms of bytes
    - [Optional] Flag indicating if memory can be accessed from any device or stream
- cudaFree()
  - Frees object from unified memory.
  - One parameter
    - Pointer to freed object

Partial Overview of CUDA Memories



- cudaMemcpy()
  - Memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Depending on the transfer type, the driver may decide to use the memory on the host or the device.
  - In Unified Memory this function is utilized to copy data between different arrays, regardless of position.

Putting it all together, vecAdd CUDA host code using Unified Memory

```
int main() {

    float *m_A, float *m_B, float *m_C, int n;

    int size = n * sizeof(float);

    cudaMallocManaged((void**) &m_A, size);
    cudaMallocManaged((void**) &m_B, size);
    cudaMallocManaged((void**) &m_C, size);

    // Memory initialization on the Host

    // Kernel invocation code - to be shown later

    cudaFree(m_A); cudaFree(m_B); cudaFree(m_C);
}
```

Allocation of Managed Memory

m_A, m_B gets initialized on the host

The device performs the actual vector addition

CUDA Unified Memory for different architectures

Prior to compute capability 6.x

– There is no specialized hardware units to improve UM efficiency.

– For data migration the full memory block needs to be copied synchronically by the driver.

– No memory oversubscription.

Compute capability 6.x onwards

– There are specialized hardware units managing page faulting.

– Data is migrated on demand, meaning that data gets copied only on page fault.

– Possibility to oversubscribe memory, enabling larger arrays than the device memory size.