NERSC User's Group (NUG) Community Call:

Using CUDA with C/C++ on Perlmutter@NERSC

18 July, 2024

U.S. DEPARTMENT OF ENERGY | Office of Science

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Bringing Science Solutions to the World

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory

Today's Pipeline

- Logistics and Introduction
- Essentials of CUDA Programming with C++
- CUDA Constructs and Program Structure
- CUDA Memory Hierarchy

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)
- Please answer a short survey afterward

Some Logistics

- Slides and videos will be available on Grads@NERSC page and NERSC Training Event page
- Crash Course in Supercomputing
  - https://www.nersc.gov/hpc-crash-course-jun2024/
    - HPC concepts, MPI, OpenMP
- Introduction to CUDA Programming Training
  - https://www.nersc.gov/users/training/training-materials/
    - previous training materials available
    - 13-Part Detailed CUDA Training

Hands-on Exercises on Perlmutter

```
ssh <user>@perlmutter.nersc.gov
```

- `% cd $SCRATCH`
- `% git clone`
    - Downloads all exercises (and answers!)
- References
    - https://docs.nersc.gov/jobs/
    - https://docs.nersc.gov/jobs/examples/#interactive

NERSC
U.S. DEPARTMENT OF ENERGY | Office of Science
BERKELEY LAB Bringing Science Solutions to the World
U.S. DEPARTMENT OF ENERGY | Office of Science

Using Perlmutter Compute Node Reservations

- Existing NERSC users (at time of registration) have been added to "**`ntrain3`**" project
- Non-NERSC users have received email instructions on apply for a training account
  - Please let us know if you need one
- Perlmutter node reservations: 10:10 am - 1:10 pm PDT today
  - **`--reservation=birds_eye_cudaC`**

    `-A ntrain3 -C gpu`

    (add `-q shared -c 32 -G 1` for shared)

    for sbatch or salloc sessions
  - No need to use **`--reservation`** or **`-A`** when outside of the reservation hours

NERSC Code of Conduct

Team Science
Service
Trust
Innovation
Respect

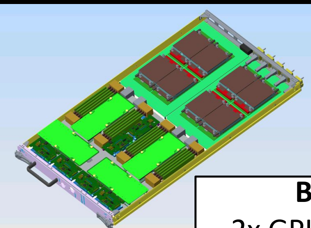
- We agree to **work together professionally and productively** towards our shared goals while respecting each other's differences and ideas.

  https://www.nersc.gov/nersc-code-of-conduct

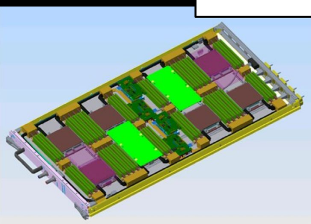
- We should all feel free to speak up to maintain this environment and remember there are resources available to **report violations** to foster an inclusive, collaborative environment.
  Email nersc-training@lbl.gov for any concerns

Perlmutter system configuration

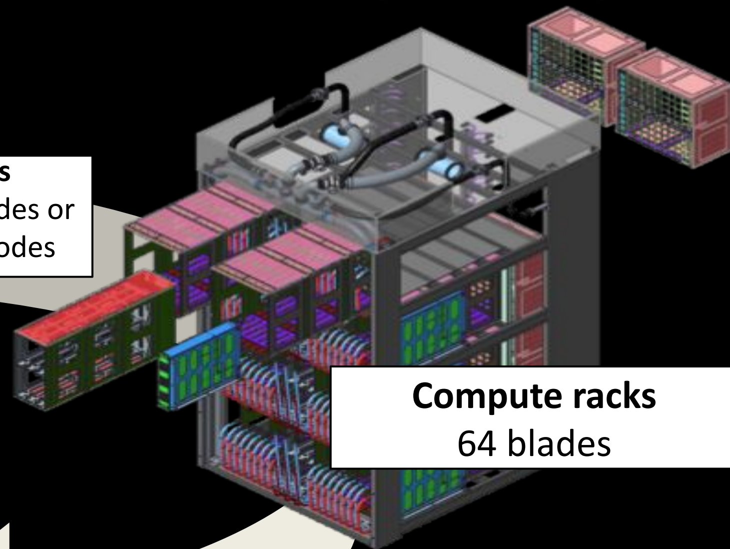
NVIDIA "Ampere" GPU Nodes
4x GPU + 1x CPU (>75 TF)
>160 GiB HBM + DDR
4x 200G "Slingshot" NICs



AMD "Milan" CPU Node
2x CPUs
> 256 GiB DDR4
1x 200G "Slingshot" NIC



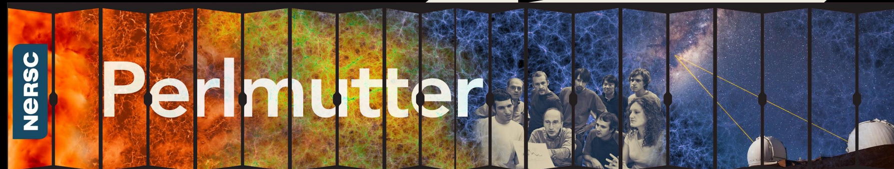
Blades
2x GPU nodes or
4x CPU nodes



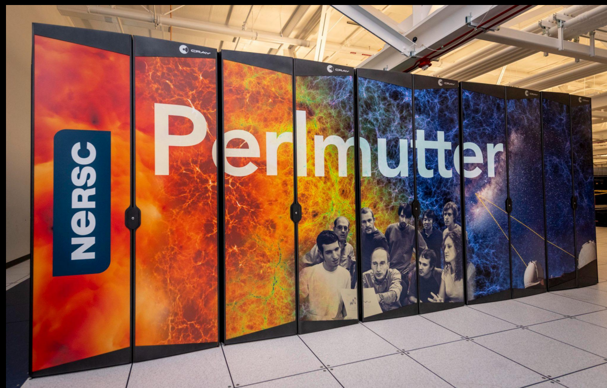
Compute racks
64 blades

Centers of Excellence
Network
Storage
App. Readiness
System SW

Perlmutter system
GPU racks
CPU racks
~6 MW



The System



System Specifications

Partition	# of nodes	CPU	GPU	NIC
GPU	1536	1x AMD EPYC 7763	4x NVIDIA A100 (40GB)	4x HPE Slingshot 11
	256	1x AMD EPYC 7763	4x NVIDIA A100 (80GB)	4x HPE Slingshot 11
CPU	3072	2x AMD EPYC 7763	-	1x HPE Slingshot 11
Login	40	1x AMD EPYC 7713	1x NVIDIA A100 (40GB)	-
Large Memory	4	1x AMD EPYC 7713	1x NVIDIA A100 (40GB)	1x HPE Slingshot 11

System Performance

Partition	Type	Aggregate Peak FP64 (PFLOPS)	Aggregate Memory (TB)
GPU	CPU	3.9	384
GPU	GPU	59.9 tensor: 119.8	240
CPU	CPU	7.7	1536

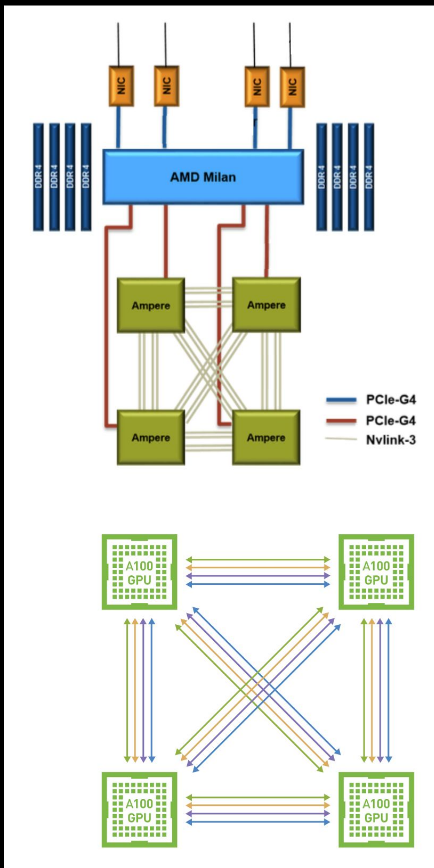
ore Details:

The System

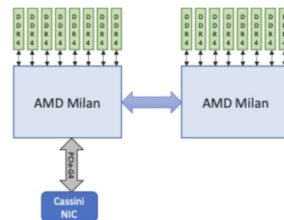
GPU Nodes:

- Single AMD EPYC 7763 (Milan) CPU
- 64 cores per CPU
- Four NVIDIA A100 (Ampere) GPUs
- PCIe 4.0 GPU-CPU connection
- PCIe 4.0 NIC-CPU connection
- 4 HPE Slingshot 11 NICs
- 256 GB of DDR4 DRAM
- 40 GB of HBM per GPU with
- 1555.2 GB/s GPU memory bandwidth
- 204.8 GB/s CPU memory bandwidth
- 12 third generation NVLink links between each pair of gpus
- 25 GB/s/direction for each link

Data type	GPU TFLOPS
FP32	19.5
FP64	9.7
TF32 (tensor)	155.9
FP16 (tensor)	311.9
FP64 (tensor)	19.5



CPU Nodes:



- 2x AMD EPYC 7763 (Milan) CPUs
- 64 cores per CPU
- AVX2 instruction set
- 512 GB of DDR4 memory total
- 204.8 GB/s memory bandwidth per CPU
- 1x HPE Slingshot 11 NIC
- PCIe 4.0 NIC-CPU connection
- 39.2 GFlops per core
- 2.51 TFlops per socket
- 4 NUMA domains per socket (NPS=4)

The System

All Flash Filesystem:

- 35 PB of disk space
- an aggregate bandwidth of >5 TB/sec
- 4 million IOPS (4 KiB random)
- It has 16 MDS (metadata servers)
- 274 I/O servers called OSSs
- 3,792 dual-ported NVMe SSDs.

Our Common Challenge

Enable a diverse community of scientific users and codes to run efficiently on advanced architectures like Cori, Perlmutter and beyond

What is CUDA?

- CUDA Architecture
  - Expose GPU parallelism for general-purpose
  - computing Retain performance
- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous
  - programming Straightforward APIs to manage devices, memory etc.
- This session introduces CUDA C/C++

Introduction to CUDA C/C++

- What will you learn in this session?
  - Start from "Hello World!"
  - Write and launch CUDA C/C++
  - kernels Manage GPU memory
  - Manage communication and synchronization

NERSC
U.S. DEPARTMENT OF ENERGY | Office of Science
BERKELEY LAB Bringing Science Solutions to the World
U.S. DEPARTMENT OF ENERGY | Office of Science

3 Ways to Accelerate Applications

Applications

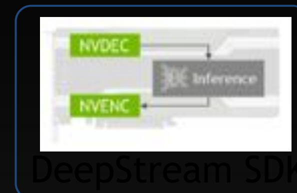
Libraries	Compiler Directives	Programming Languages
Easy to use Most Performance	Easy to use Portable code	Most Performance Most Flexibility

Libraries: Easy, High-Quality Acceleration

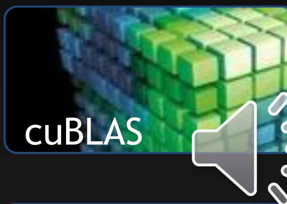
- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **"Drop-in":** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions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

NVIDIA GPU Accelerated Libraries

DEEP LEARNING

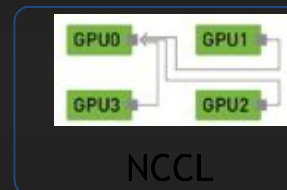


LINEAR ALGEBRA



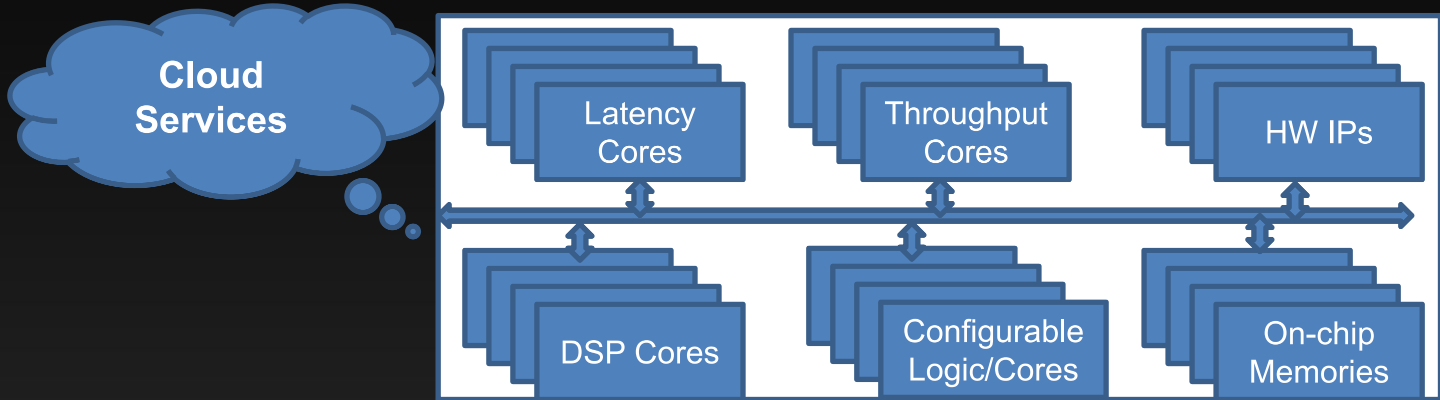
SIGNAL, IMAGE, VIDEO

PARALLEL ALGORITHMS



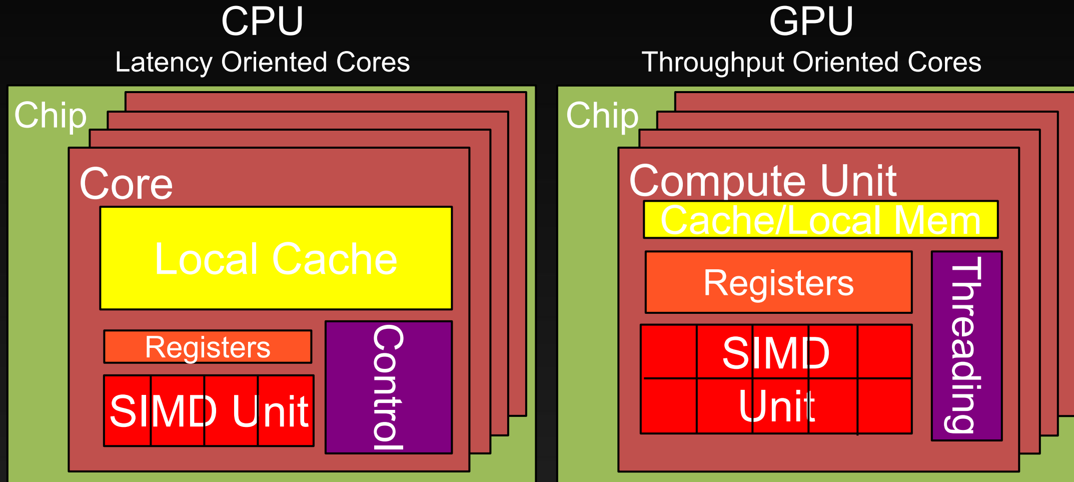
Heterogeneous Parallel Computing

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

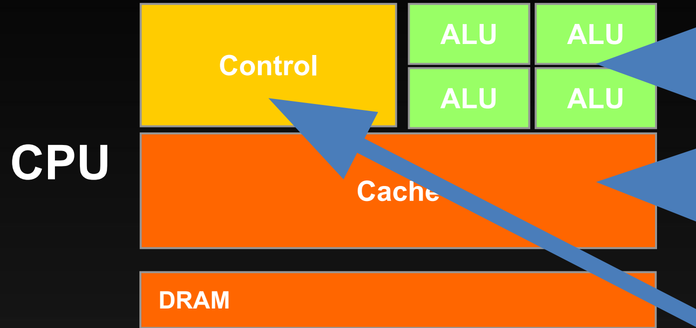


NVIDIA

CPU and GPU are designed very differently



CPUs: Latency Oriented Design



Powerful ALU

Reduced operation latency

Large caches

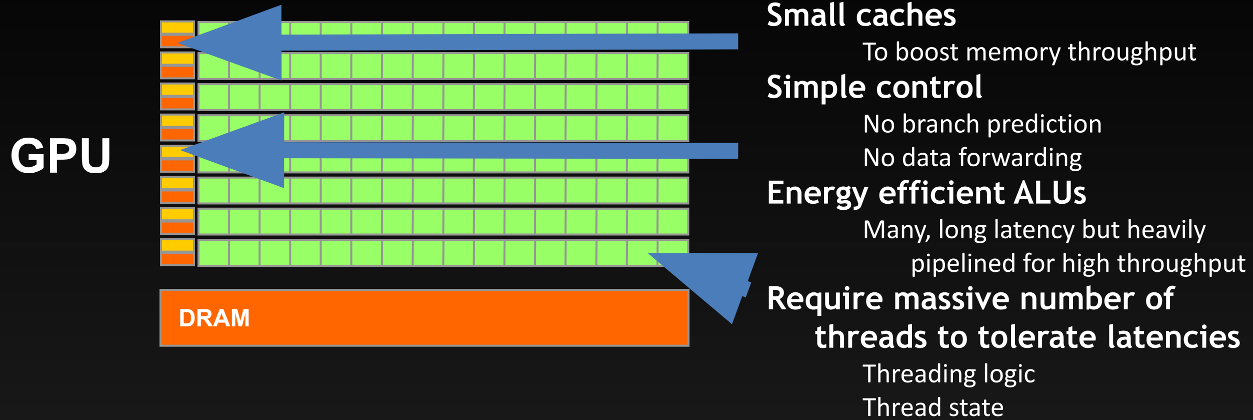
Convert long latency memory accesses to short latency cache accesses

Sophisticated control

Branch prediction for reduced branch latency

Data forwarding for reduced data latency

GPUs: Throughput Oriented Design



NVIDIA

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

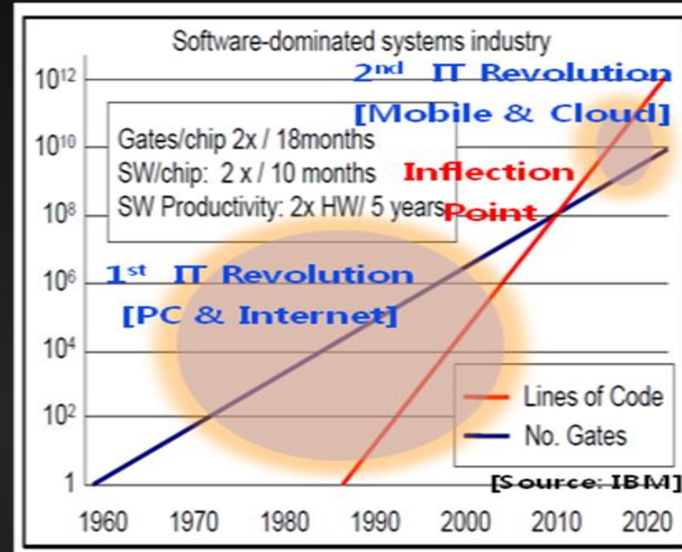
- Financial Analysis
- Scientific Simulation
- Engineering Simulation
- Data Intensive Analytics
- Medical Imaging
- Digital Audio Processing
- Digital Video Processing
- Computer Vision
- Biomedical Informatics
- Electronic Design Automation
- Statistical Modeling
- Numerical Methods
- Ray Tracing Rendering
- Interactive Physics

Software Dominates System Cost

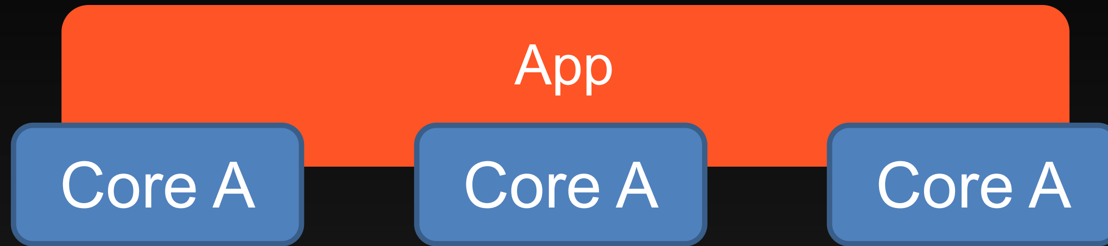
SW lines per chip increases at 2x/10 months

HW gates per chip increases at 2x/18 months

Future systems must
minimize software redevelopment



Keys to Software Cost Control



Scalability

The same application runs efficiently on new generations of cores

The same application runs efficiently on more of the same cores

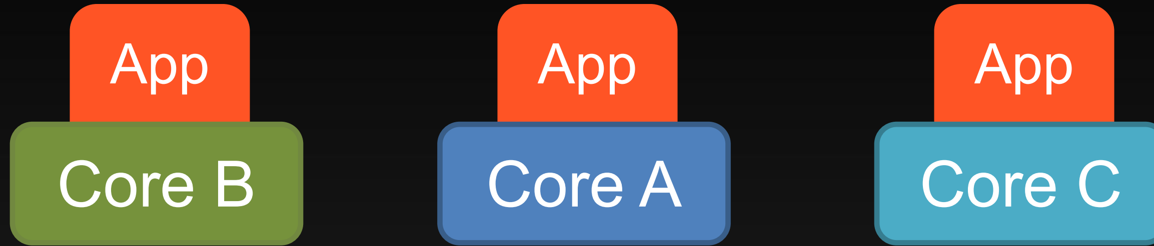
More on Scalability

- **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

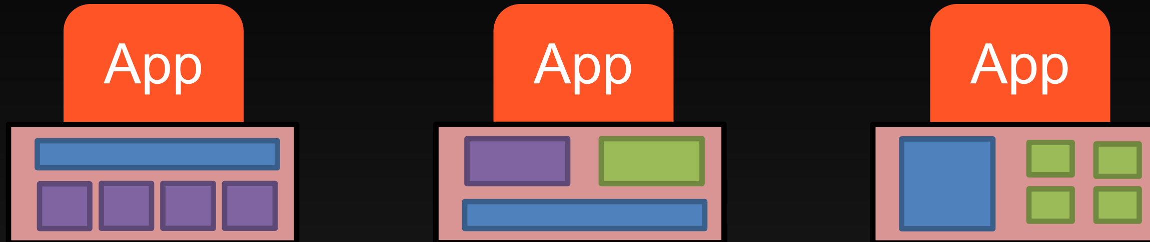
NVIDIA

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

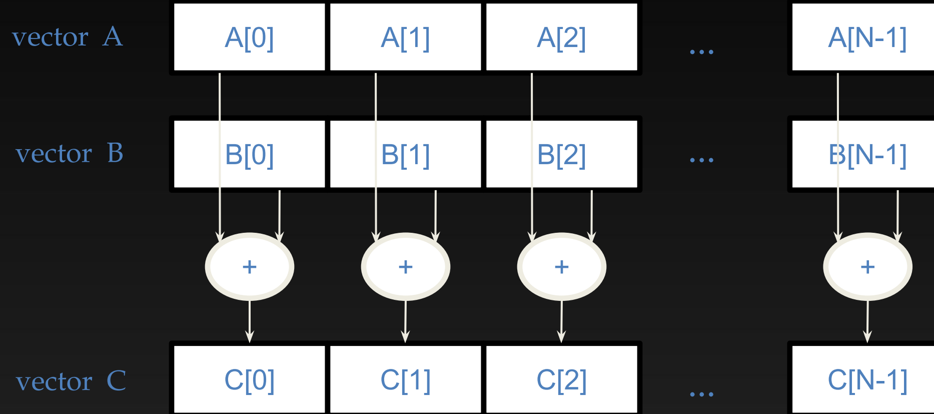
More on Portability

- **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

Objective

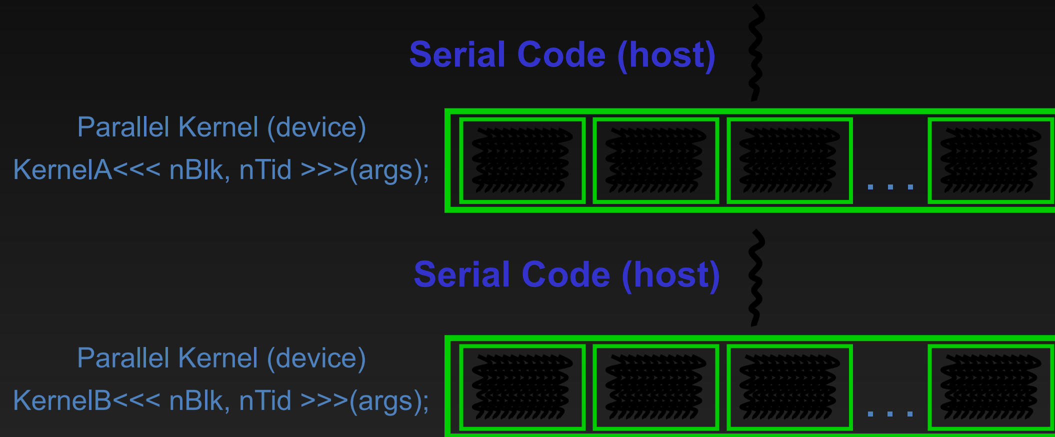
- **To learn about CUDA threads, the main mechanism for exploiting of data parallelism**
  - Hierarchical thread organization
  - Launching parallel execution
  - Thread index to data index mapping

Data Parallelism - Vector Addition Example

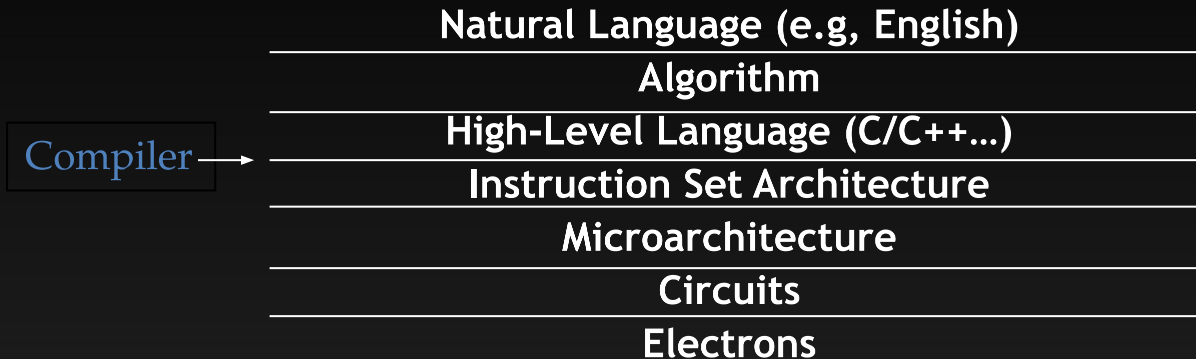


CUDA Execution Model

- **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



©Yale Patt and Sanjay Patel, *From bits and bytes to gates and beyond*

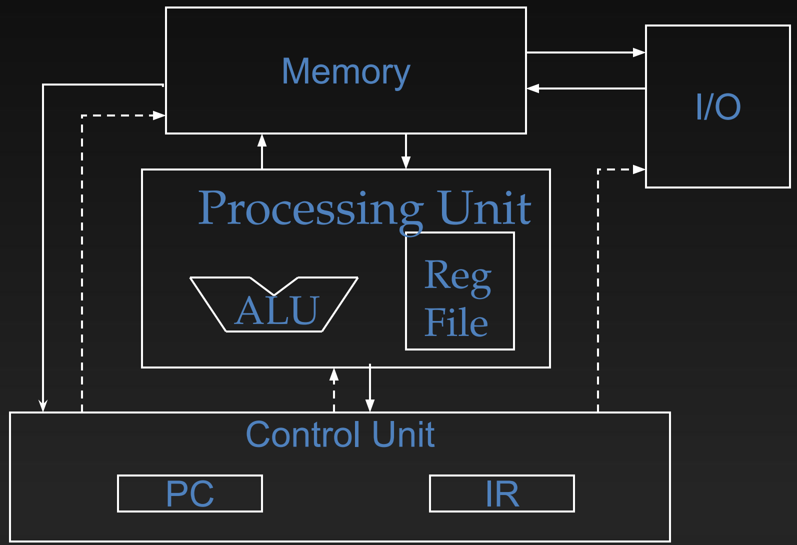
A program at the ISA level

- A program is a set of instructions stored in memory that can be read, interpreted, and executed by the hardware.
  - Both CPUs and GPUs are designed based on (different) instruction sets
- Program instructions operate on data stored in memory and/or registers.

NVIDIA

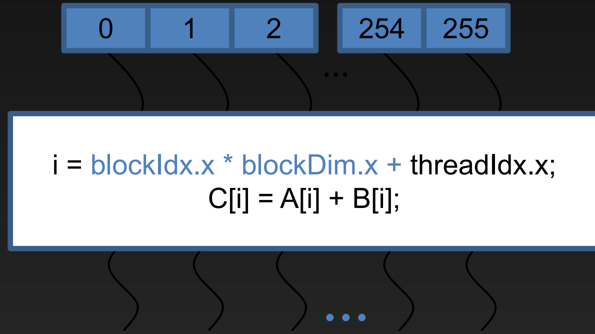
A Thread as a Von-Neumann Processor

A thread is a "virtualized" or "abstracted" Von-Neumann Processor

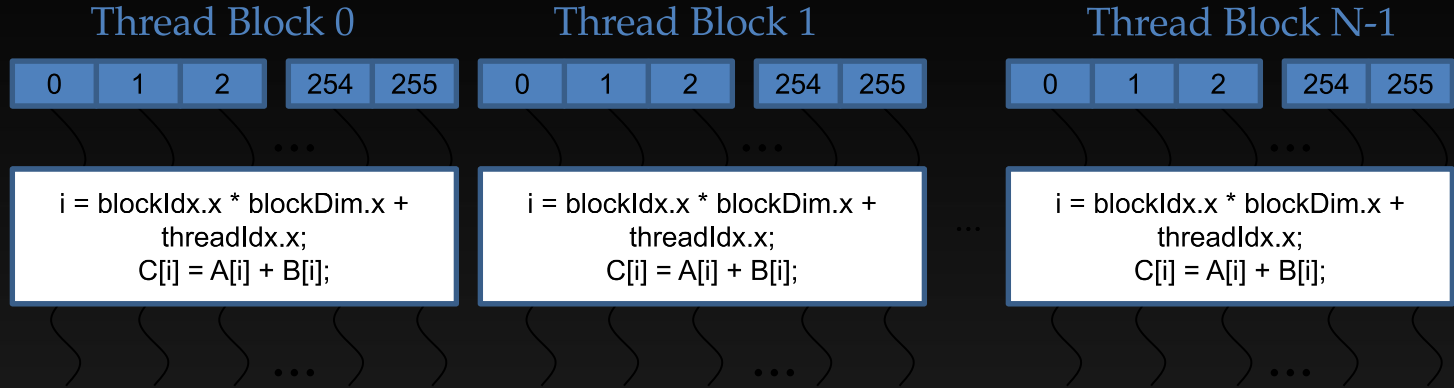


Arrays of Parallel Threads

- **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

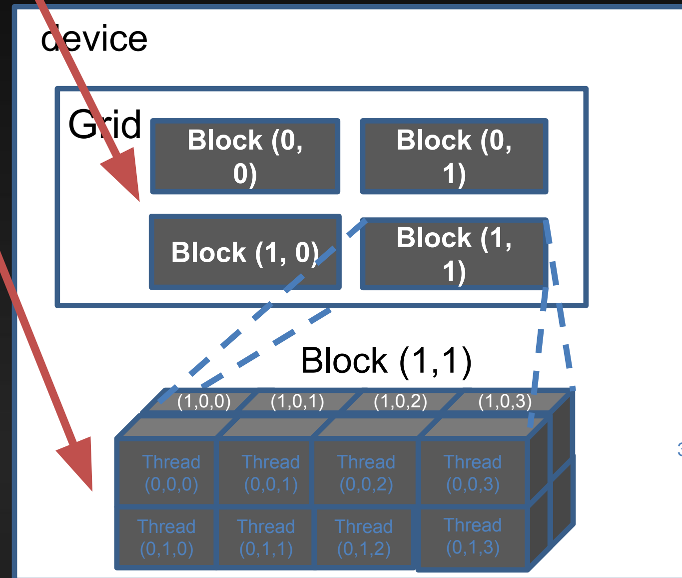


- **Divide thread array into multiple blocks**
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks do not interact

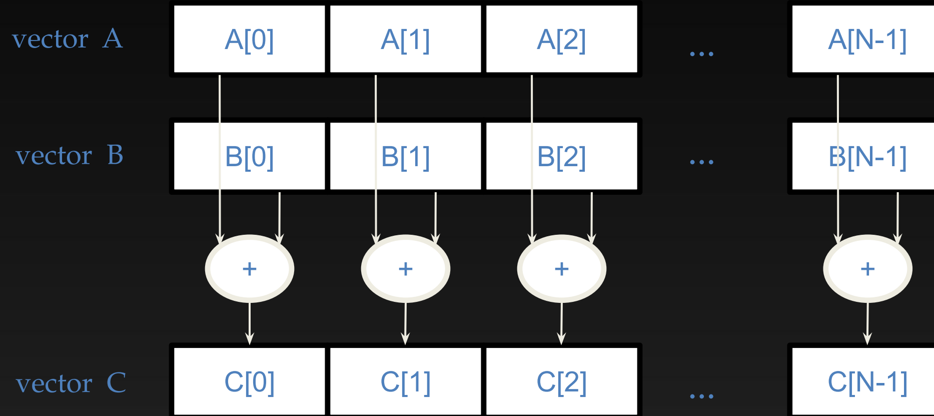
blockIdx and threadIdx

- **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
  - …

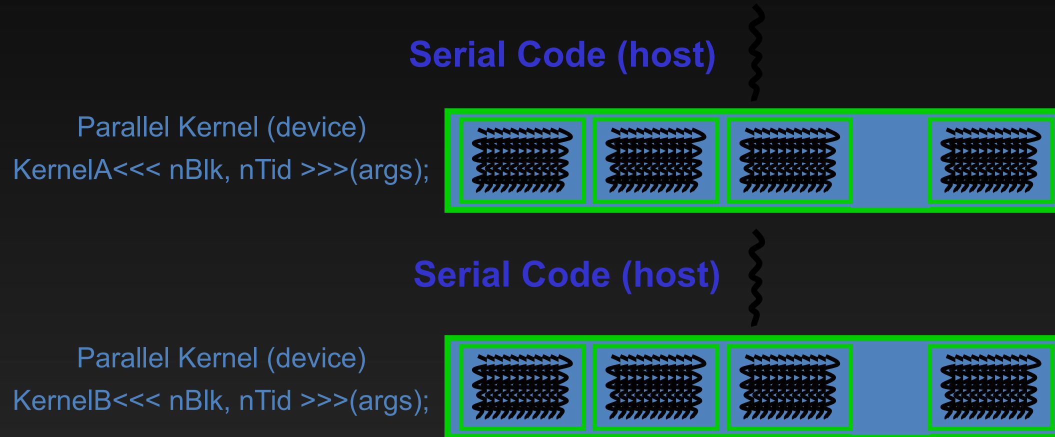


Data Parallelism - Vector Addition Example

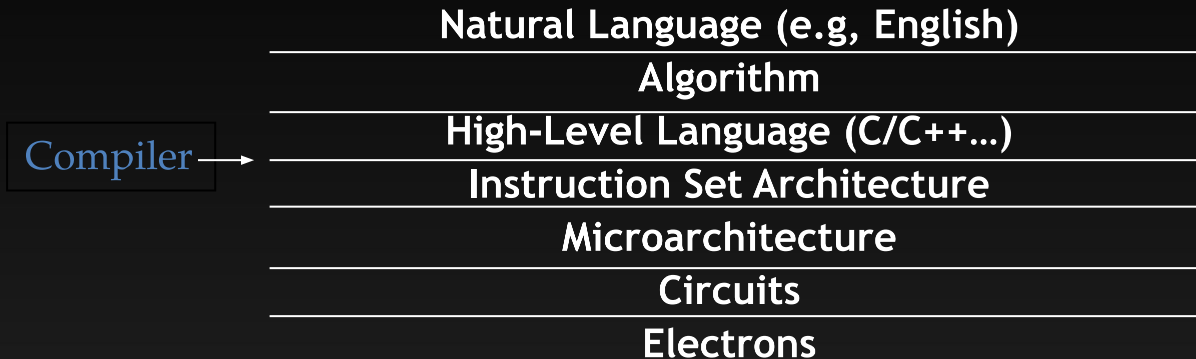


CUDA Execution Model

- **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



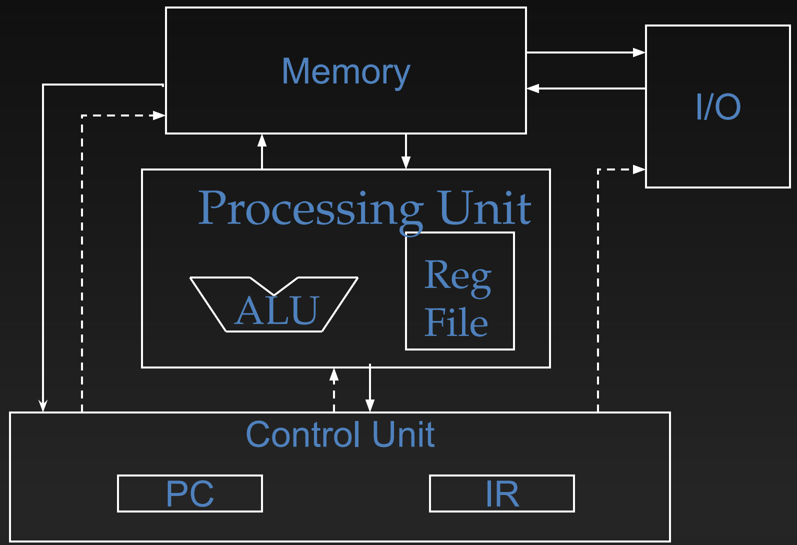
©Yale Patt and Sanjay Patel, *From bits and bytes to gates and beyond*

A program at the ISA level

- **A program is a set of instructions stored in memory that can be read, interpreted, and executed by the hardware.**
  - Both CPUs and GPUs are designed based on (different) instruction sets
- **Program instructions operate on data stored in memory and/or registers.**

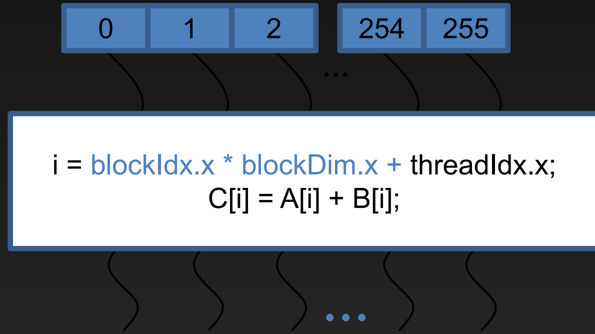
A Thread as a Von-Neumann Processor

A thread is a "virtualized" or "abstracted" Von-Neumann Processor

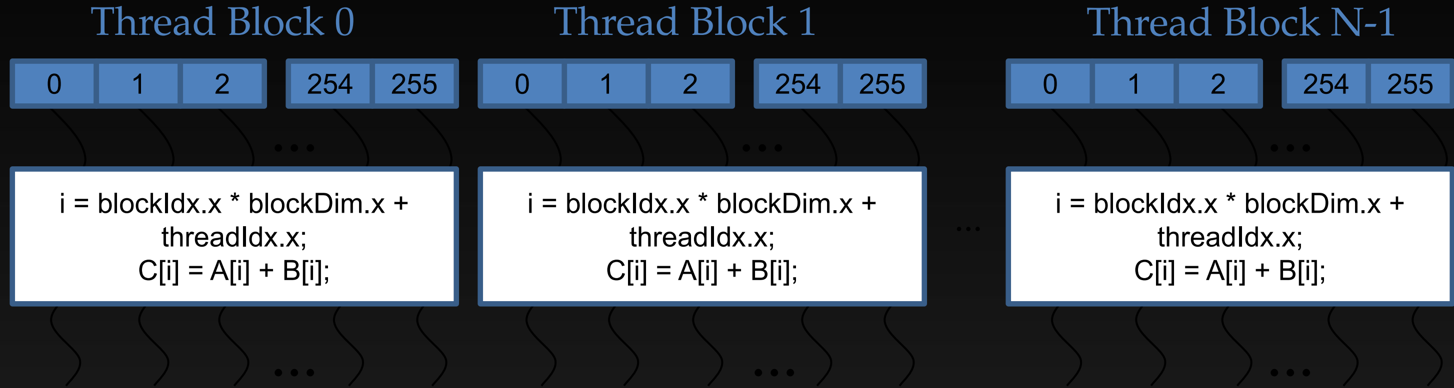


Arrays of Parallel Threads

- **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



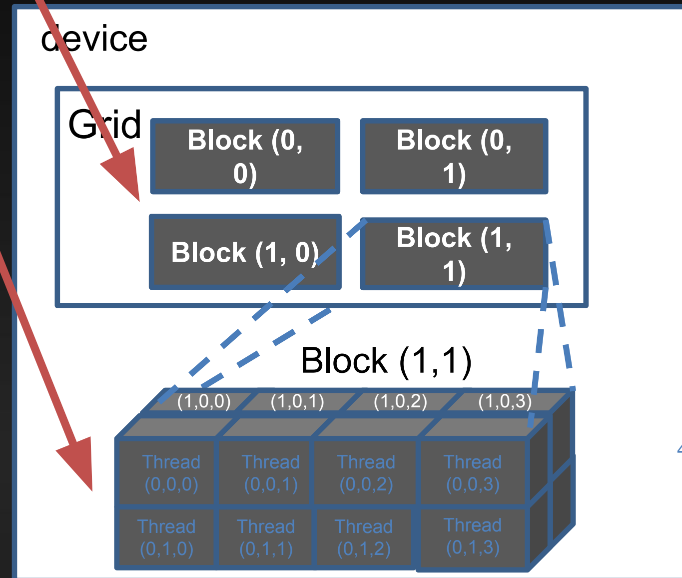
Thread Blocks: Scalable Cooperation



- **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
  - …



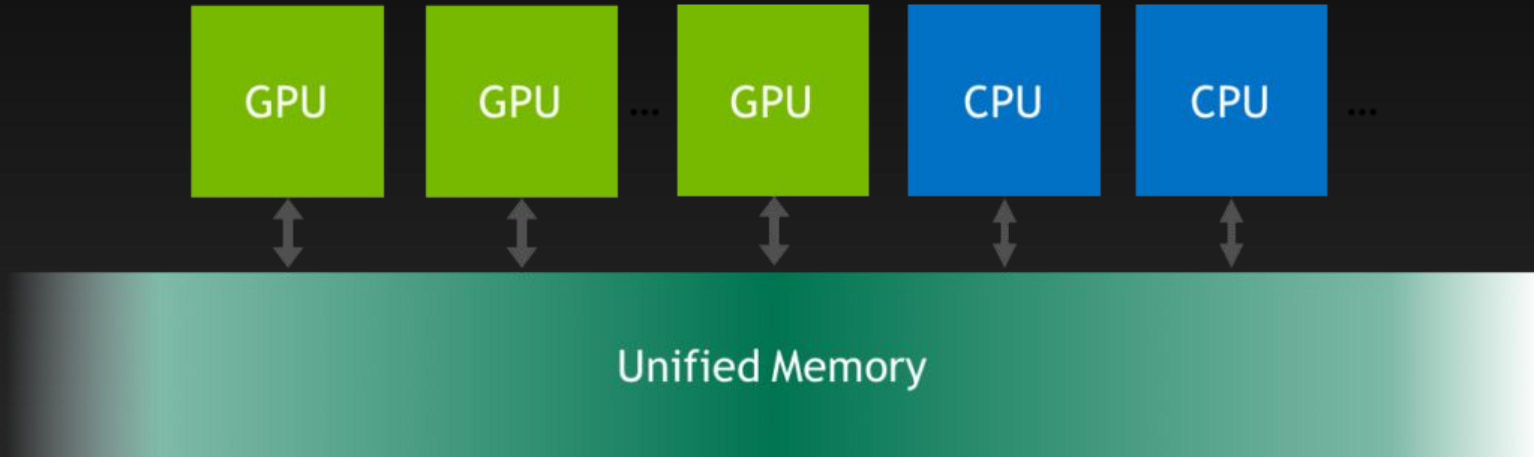
Objective

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

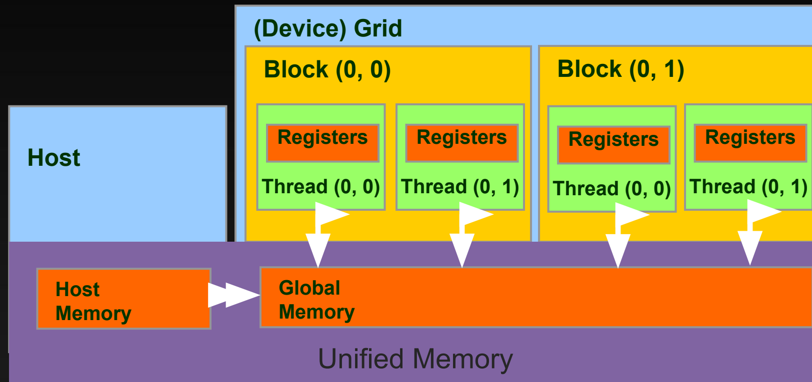
- Unified Memory Allocation
- Data Transfer in Unified Memory

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.

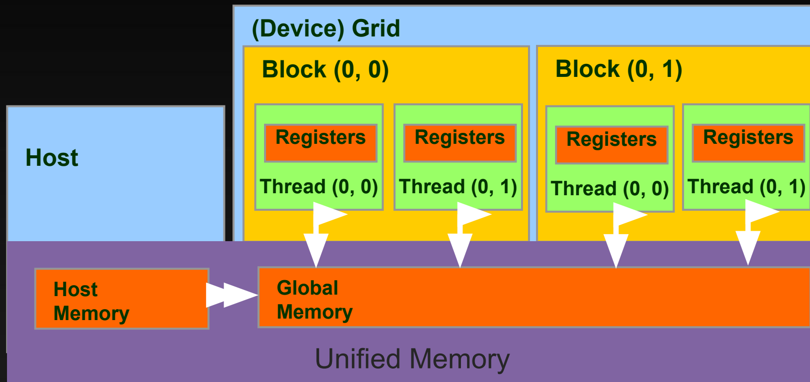


Partial Overview of CUDA Memories



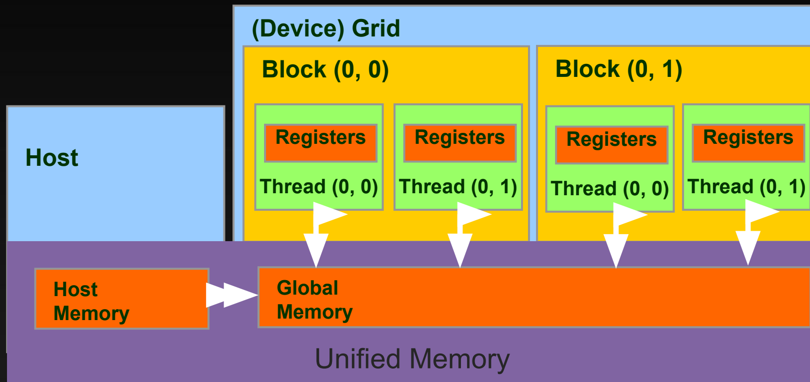
- **Device code can:**
  - R/W per-thread registers
  - R/W all-shared global memory
  - R/W managed memory (Unified Memory)
- **Host code can**
  - Transfer data to/from per grid global memory
  - R/W managed memory

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

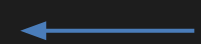


- **cudaMemcpy()**
  - Memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Depending on the transfer type, the driver may decide to use the memory on the host or the device.
  - In Unified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

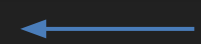
Putting it all together, vecAdd CUDA host code using Unified Memory

```
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.

Example: Vector Addition Kernel

NVIDIA

Device Code

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition

__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
  int i = threadIdx.x+blockDim.x*blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

Example: Vector Addition Kernel Launch (Host Code)

NVIDIA

Host Code

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
 // d_A, d_B, d_C allocations and copies omitted
 // Run ceil(n/256.0) blocks of 256 threads each
  vecAddKernel<<<ceil(n/256.0),256>>>(d_A, d_B, d_C, n);
}
```

The ceiling function makes sure that there are enough threads to cover all elements.

More on Kernel Launch (Host Code)

Host Code

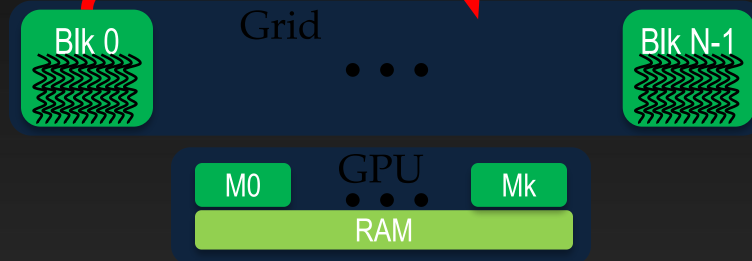
```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  dim3 DimGrid((n-1)/256 + 1, 1, 1);
  dim3 DimBlock(256, 1, 1);
  vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

This is an equivalent way to express the ceiling function.

Kernel execution in a nutshell

```
__host__
void vecAdd(…)
{
  dim3 DimGrid(ceil(n/256.0),1,1);
  dim3 DimBlock(256,1,1);
vecAddKernel<<<DimGrid,DimBlock>>>(d_A,d_B,d_C,n);
}
```

```
__global__
void vecAddKernel(float *A,
     float *B, float *C, int n)
{
   int i = blockIdx.x * blockDim.x
           + threadIdx.x;

   if( i<n ) C[i] = A[i]+B[i];
}
```



More on CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Each “__” consists of two underscore characters
  - A kernel function must return `void`
- `__device__` and `__host__` can be used together
- `__host__` is optional if used alone

Compiling A CUDA Program