GPU programming for DL

Julie Bernauer and Ryan Olson

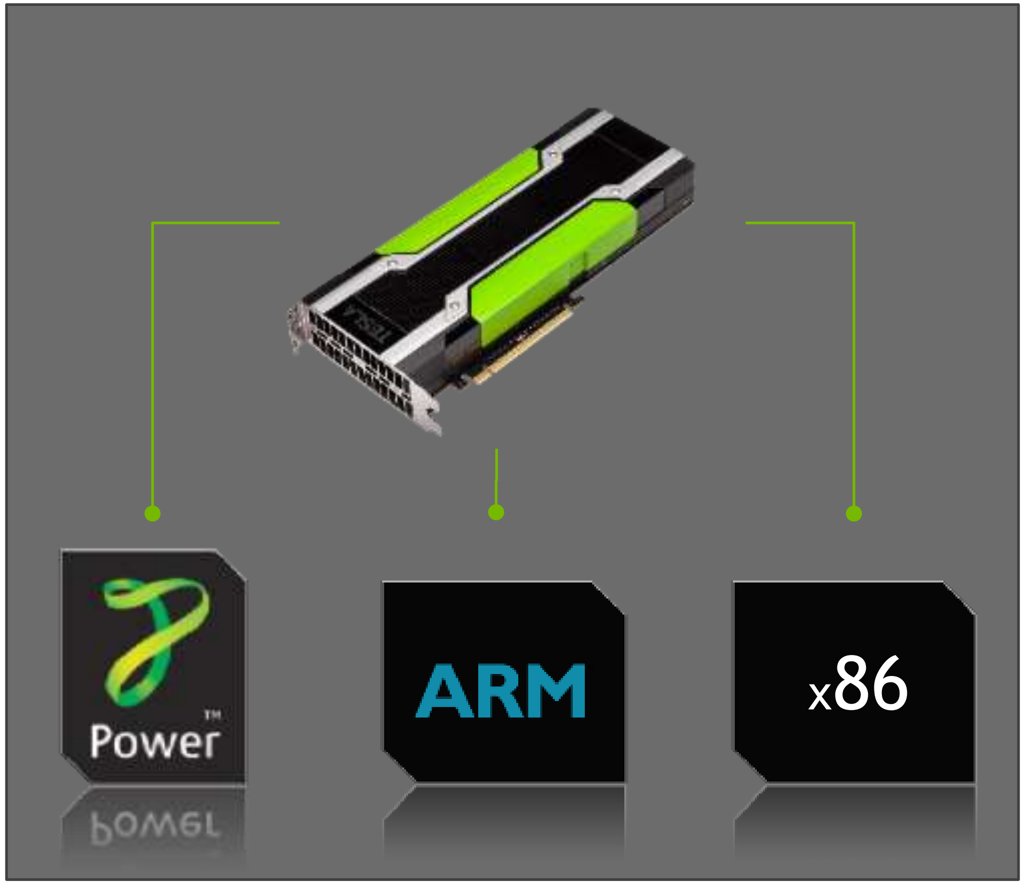
Outline

Presentation & Hands-on session

- Intro to GPU computing / Libraries for DL /Platform
- Intro to CUDA
- Hands-on labs
    - Accelerating Applications with CUDA C/C++
    - (optional) Accelerating Applications with GPU-Accelerated Libraries in C/C++
    - (optional) GPU Memory Optimizations (C/C++)

GPU Computing

GPU Computing



CUDA

Framework to Program NVIDIA GPUs

A simple sum of two vectors (arrays) in C

```c
void vector_add(int n, const float *a, const float *b, float *c)
{
  for( int idx = 0 ; idx < n ; ++idx )
    c[idx] = a[idx] + b[idx];
}
```

GPU friendly version in CUDA

```cuda
__global__ void vector_add(int n, const float *a, const float *b, float *c)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if( idx < n )
    c[idx] = a[idx] + b[idx];
}
```

GPU accelerated libraries

“Drop-in” Acceleration for Your Applications

Linear Algebra
FFT, BLAS,
SPARSE, Matrix, cuSolver

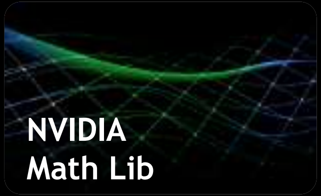
NVIDIA
cuFFT,
cuBLAS,
cuSPARSE

CULA | tools

MAGMA

CUSP

Numerical & Math
RAND, Statistics

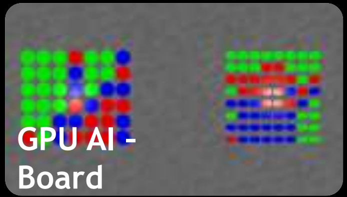

NVIDIA
Math Lib

ArrayFire

NVIDIA cuRAND

Data Struct. & AI
Sort, Scan, Zero Sum

Thrust


GPU AI -
Board
Games

GPU AI -
Path Finding

Visual Processing
Image & Video

NVIDIA
NPP

NVIDIA
Video
Encode

Sundog
Software

Deep Neural Networks and GPUs

ACCELERATING INSIGHTS

“Now You Can Build Google’s $1M Artificial Brain on the Cheap”

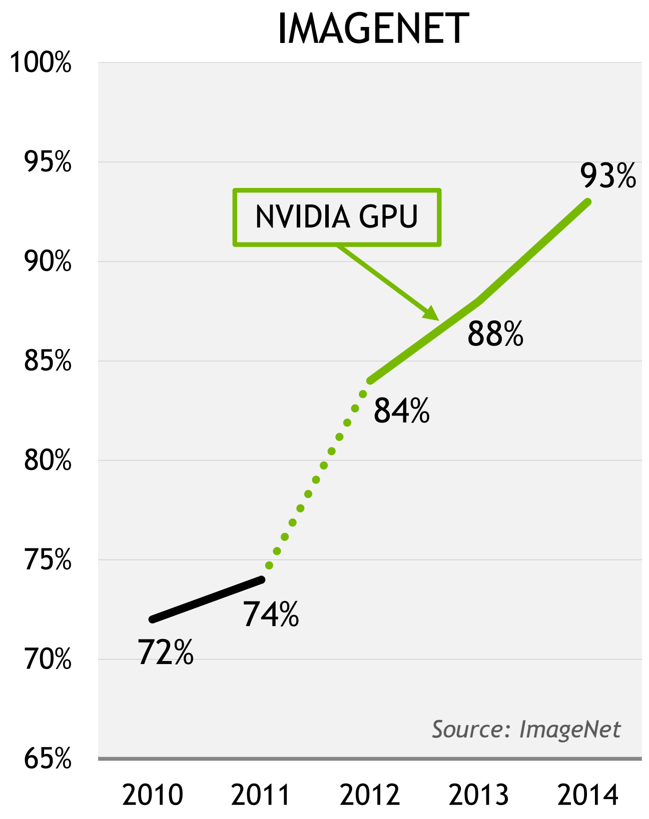
WIRED

GOOGLE DATACENTER	
1,000 CPU Servers 2,000 CPUs • 16,000 cores	600 kWatts $5,000,000

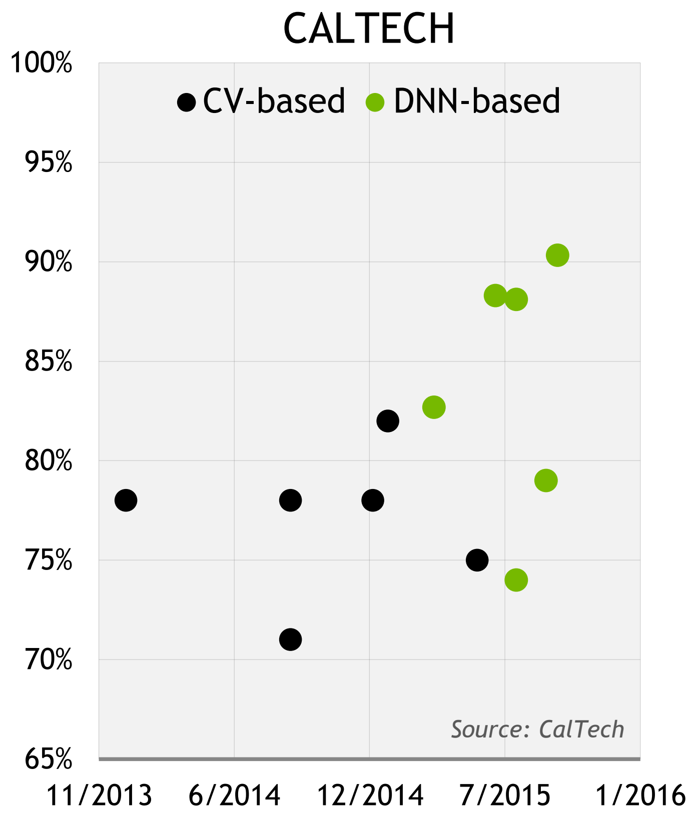
STANFORD AI LAB	
3 GPU-Accelerated Servers 12 GPUs • 18,432 cores	4 kWatts $33,000

Recent improvements

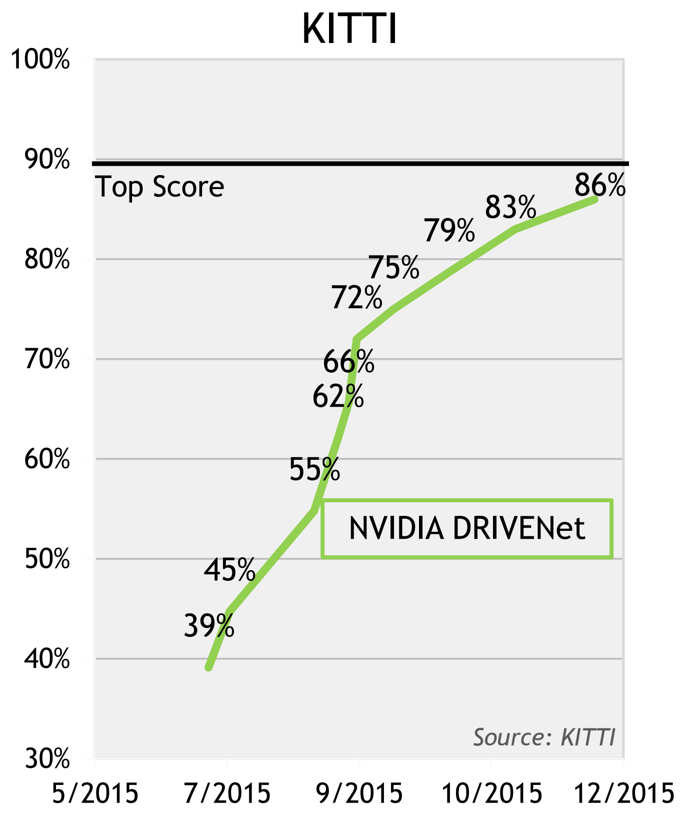
Image Recognition



Pedestrian Detection



Object Detection

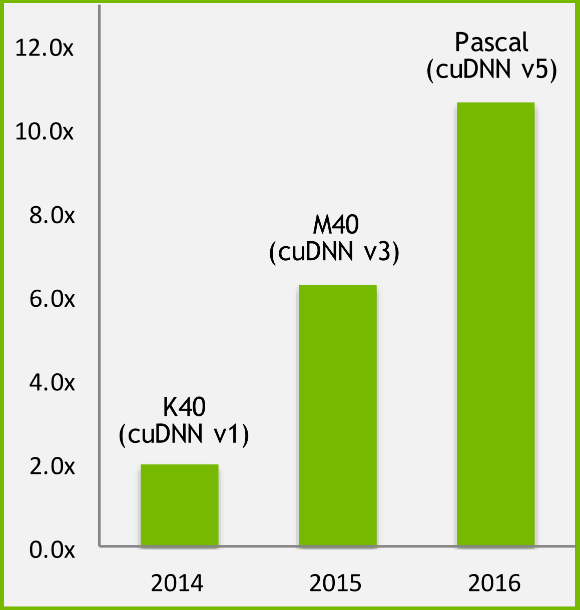


NVIDIA cuDNN

Building blocks for accelerating deep neural networks on GPUs

- High performance deep neural network training
- Accelerates Deep Learning: Caffe, CNTK, Tensorflow, Theano, Torch
- Performance continues to improve over time

developer.nvidia.com/*cudnn*



AlexNet training throughput based on 20 iterations, CPU: 1x E5-2680v3 12 Core 2.5GHz.

"NVIDIA has improved the speed of cuDNN with each release while extending the interface to more operations and devices at the same time."

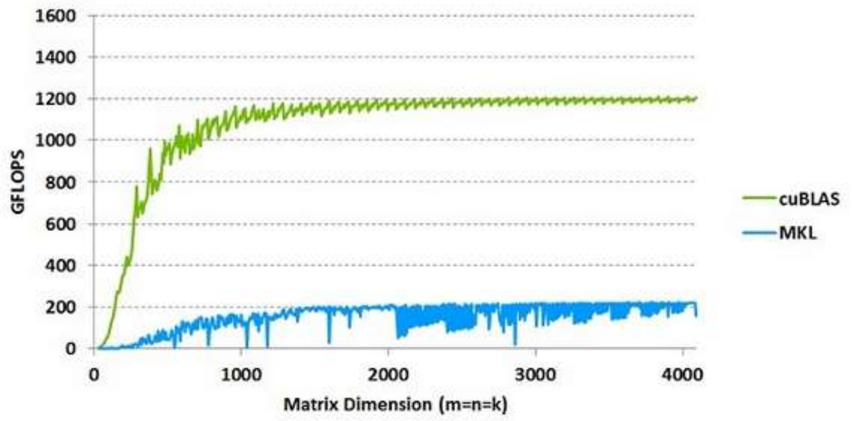
— Evan Shelhamer, Lead Caffe Developer, UC Berkeley

Accelerating linear algebra: cuBLAS

Accelerated Level 3 BLAS

- GEMM, SYMM, TRSM, SYRK
- >3 TFlops Single Precision on a single K40

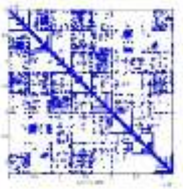
Multi-GPU BLAS support available in cuBLAS-XT



cuBLAS on K40m, ECC ON, input and output data on device. MKL 11.0.4 on Intel IvyBridge single socket 12 -core E5-2697 v2 @ 2.70GHz

developer.nvidia.com/cublas

Accelerating sparse operations: cuSPARSE



The (Dense matrix) X (sparse vector) example

cusparse<T>gemvi()

$y = \alpha * op(A) * x + \beta * y$

A = dense matrix

x = sparse vector

y = dense vector

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \alpha \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} \\ A_{21} & A_{22} & A_{23} & A_{24} & A_{25} \\ A_{31} & A_{32} & A_{33} & A_{34} & A_{35} \end{bmatrix} \begin{bmatrix} - \\ 2 \\ - \\ - \\ 1 \end{bmatrix} + \beta \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

$\uparrow$

Sparse vector could be frequencies of words in a text sample

cuSPARSE provides a full suite of accelerated sparse matrix functions

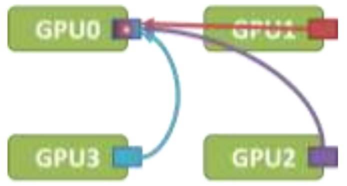
developer.nvidia.com/cusparse

Multi-GPU communication: NCCL

Collective library

- Research library of accelerated collectives that is easily integrated and topology-aware so as to improve the scalability of multi-GPU applications

- Pattern the library after MPI's collectives
- Handle the intra-node communication in an optimal way
- Provide the necessary functionality for MPI to build on top to handle inter-node



github.com/NVIDIA/nccl

NCCL Example

All-reduce

```
#include <nccl.h>
ncclComm_t comm[4];
ncclCommInitAll(comm, 4, {0, 1, 2, 3});
foreach g in (GPUs) { // or foreach thread
  cudaSetDevice(g);
  double *d_send, *d_recv;
  // allocate d_send, d_recv; fill d_send with data
  ncclAllReduce(d_send, d_recv, N, ncclDouble, ncclSum, comm[g], stream[g]);
  // consume d_recv
}
```

Platform

Developer workstation

Titan X Pascal

- 11 TFLOPS FP32
- INT8
- 3,584 CUDA
- 12 GB DDR5X

DGX-1

World’s First Deep Learning Supercomputer

Engineered for deep learning

170 TF FP16

8x Tesla P100 in hybrid cube mesh

+ SDD RAID, 4x IB

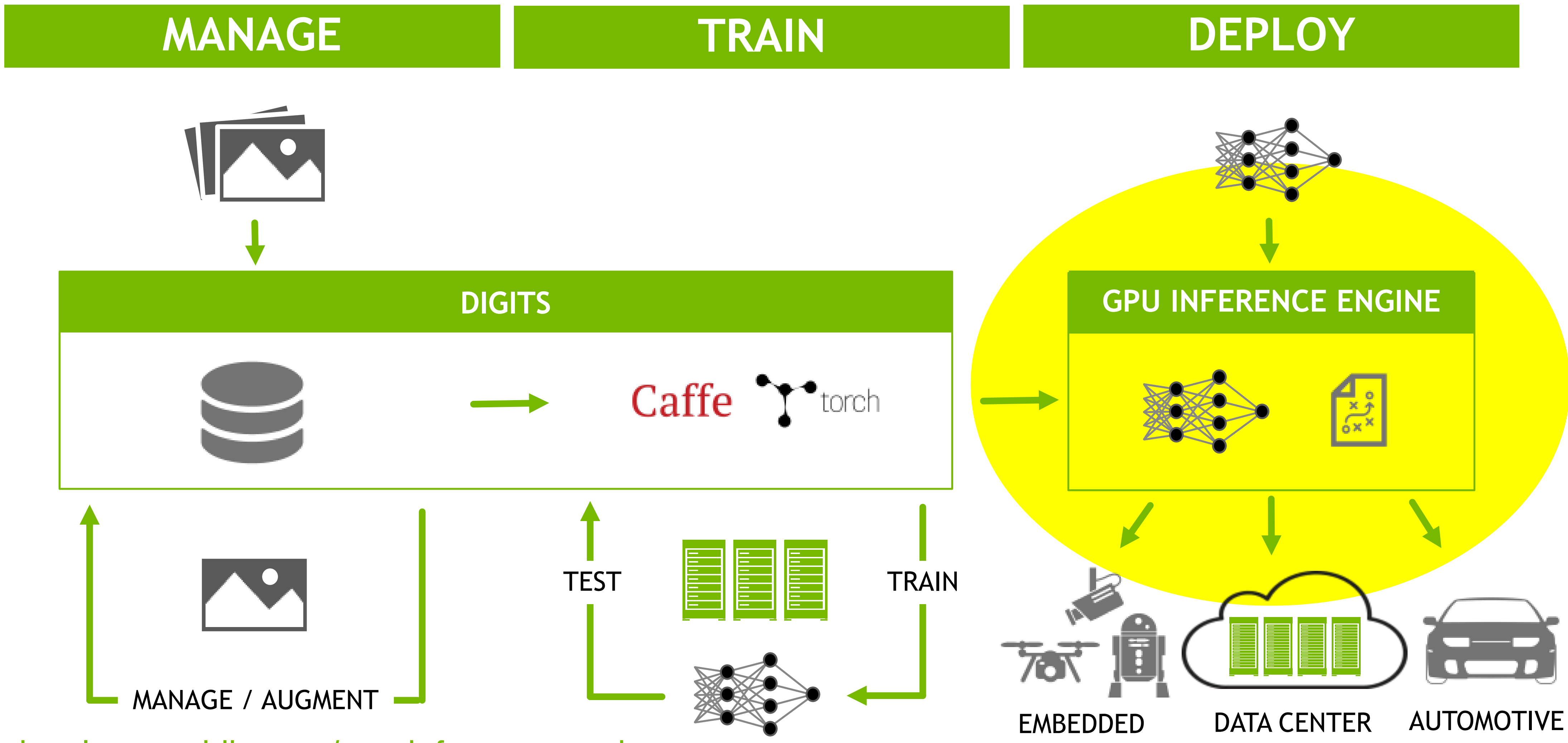
Accelerates major AI frameworks

nvidia.com/dgx1

Tesla p100 accelerator

Compute	5.3 TF DP · 10.6 TF SP · 21.2 TF HP
Memory	HBM2: 720 GB/s · 16 GB
Interconnect	NVLink (up to 8 way) + PCIe Gen3
Programmability	Page Migration Engine Unified Memory

GIE (GPU Inference Engine)



developer.nvidia.com/gpu-inference-engine

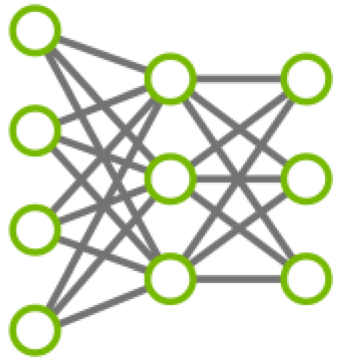
EXAMPLE: DL EMBEDDED DEPLOYMENT

Jetson TX1 devkit

- Jetson TX1
    - Inference at 258 img/s
    - No need to change code
- Simply compile Caffe and copy a trained `.caffemodel` to TX1

GPU INFERENCE ENGINE

Optimizations



TRAINED
NEURAL NETWORK

- Fuse network layers
- Eliminate concatenation layers
- Kernel specialization
- Auto-tuning for target platform
- Select optimal tensor layout
- Batch size tuning

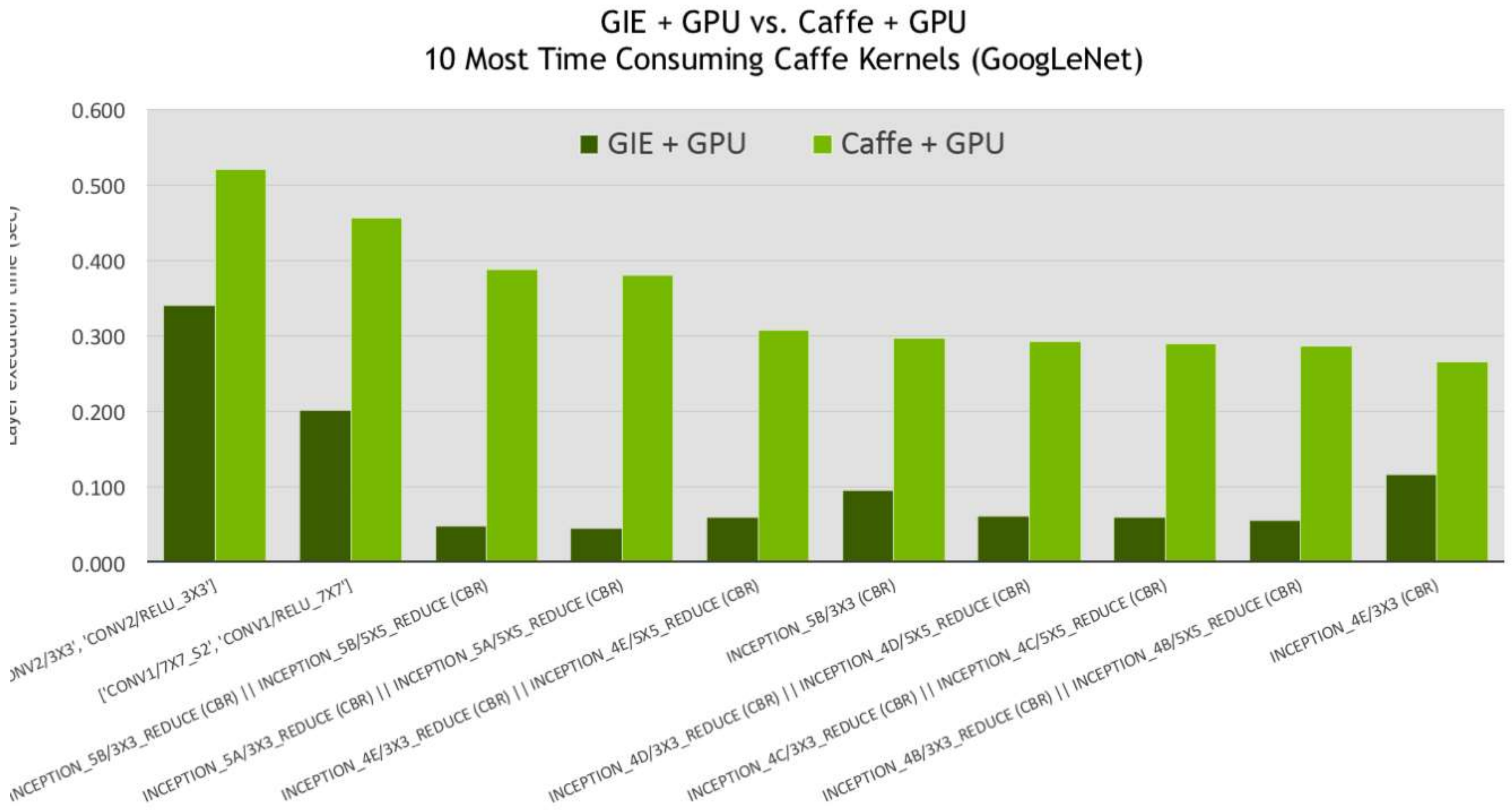
OPTIMIZED
INFERENCE
RUNTIME

See the parallel for all blog post for GIE:
https://devblogs.nvidia.com/parallelforall/production-deep-learning-nvidia-gpu-inference-engine/

developer.nvidia.com/gpu-inference-engine

GPU INFERENCE ENGINE

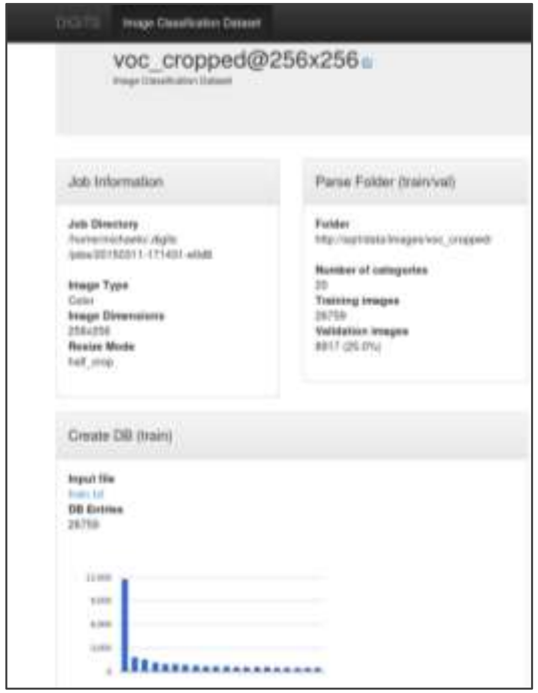
Performance



NVIDIA DIGITS

Interactive Deep Learning GPU Training System

Process Data



voc_cropped@256x256
Image Classification Dataset

Job Information

Job Directory
/home/michaelo/.digits/jobs/20150311-171401-e0d8

Image Type
Color
Image Dimensions
256x256
Resize Mode
half_crop

Parse Folder (train/val)

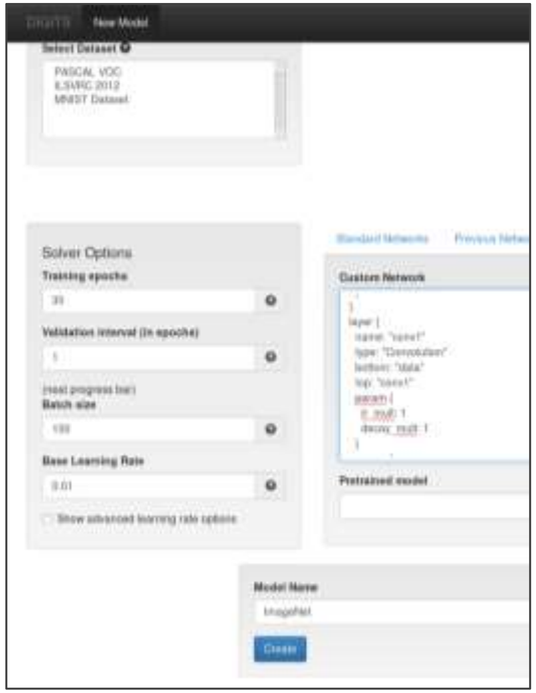
Folder
http://opt/data/images/voc_cropped

Number of categories
20
Training images
26759
Validation images
8917 (25.0%)

Create DB (train)

Input file
train.txt
DB Entries
26759

Configure DNN



Select Dataset
PASCAL VOC
ILSVRC 2012
MNIST Dataset

Solver Options

Training epochs
30

Validation interval (in epochs)
1

(next progress bar)
Batch size
100

Base Learning Rate
0.01

Show advanced learning rate options

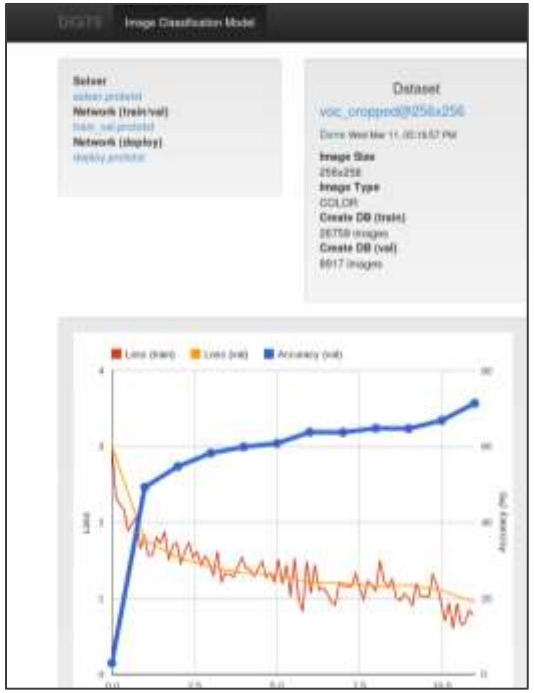
Custom Network

```
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
```

Pretrained model

Model Name
ImageNet

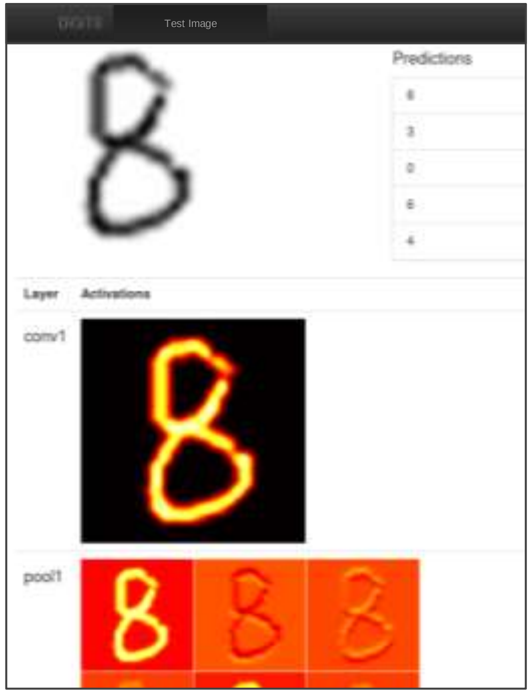
Monitor Progress



Solver
solver.prototxt
Network (train/val)
train_val.prototxt
Network (deploy)
deploy.prototxt

Dataset
voc_cropped@256x256
Done Wed Mar 11, 05:19:57 PM
Image Size
256x256
Image Type
COLOR
Create DB (train)
26759 images
Create DB (val)
8917 images

Visualize Layers



developer.nvidia.com/digits
github.com/NVIDIA/DIGITS

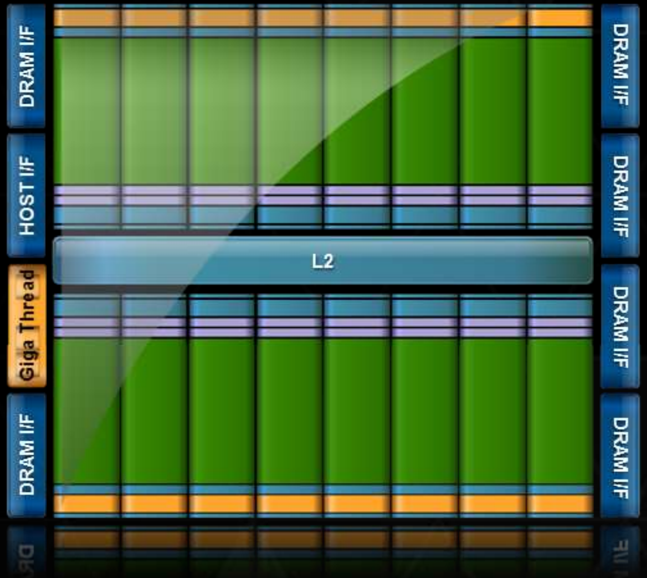
CUDA

GPU architecture

GPU ARCHITECTURE

Two Main Components

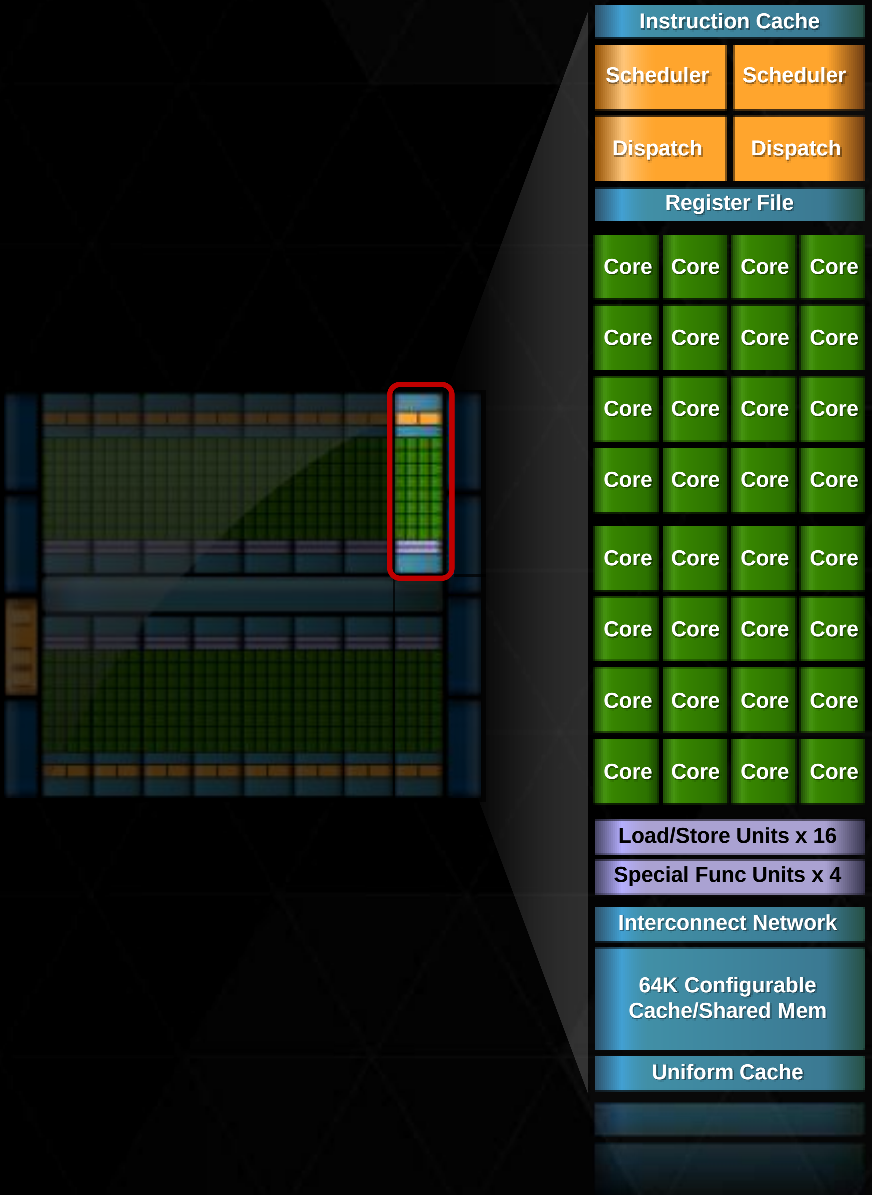
- Global memory
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 24 GB
  - ECC on/off options for Quadro and Tesla products
- Streaming Multiprocessors (SM)
  - Perform the actual computation
  - Each SM has its own: Control units, registers, execution pipelines, caches



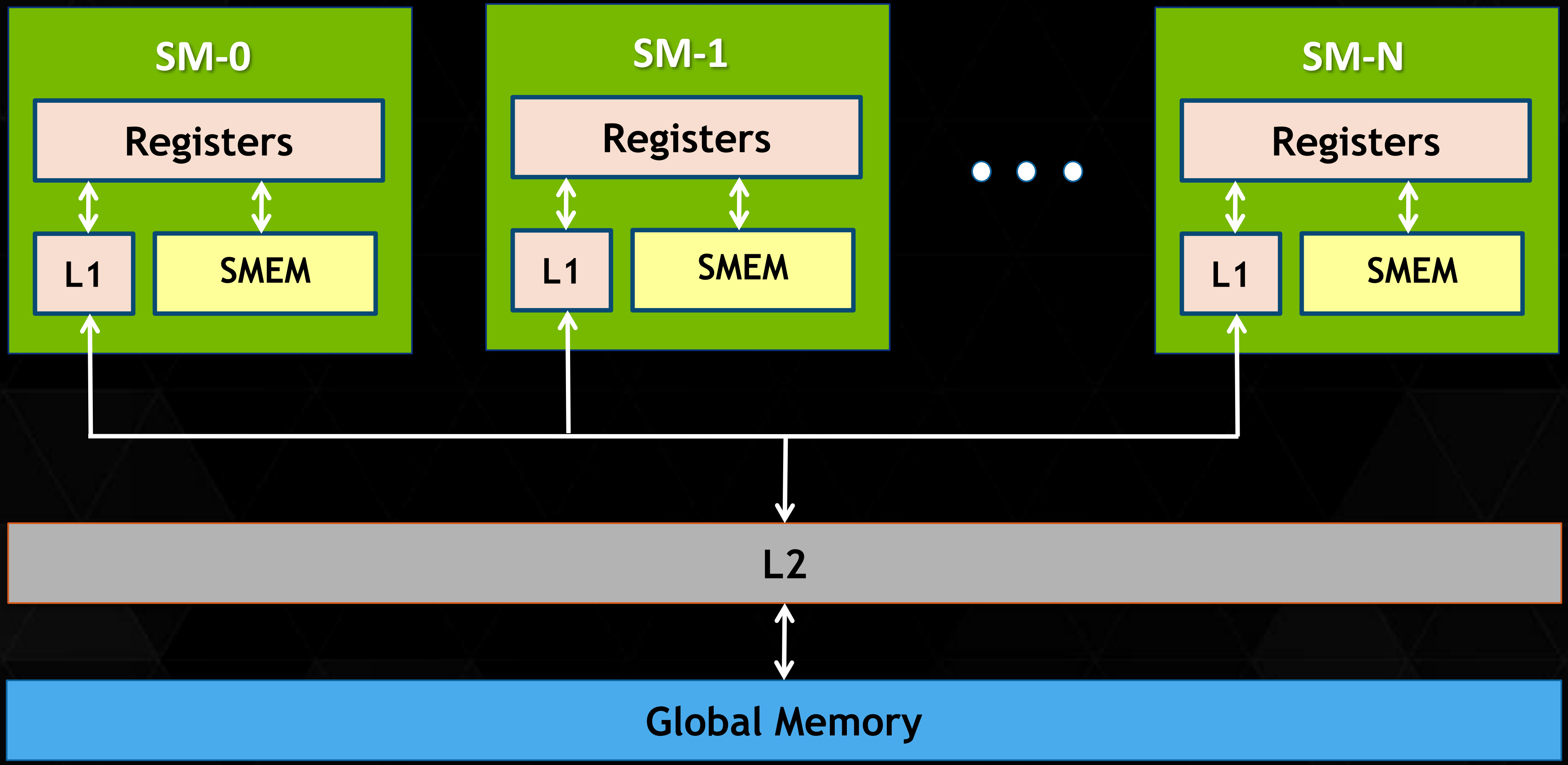
GPU ARCHITECTURE

Streaming Multiprocessor (SM)

- Many CUDA Cores per SM
  - Architecture dependent
- Special-function units
  - cos/sin/tan, etc.
- Shared mem + L1 cache
- Thousands of 32-bit registers



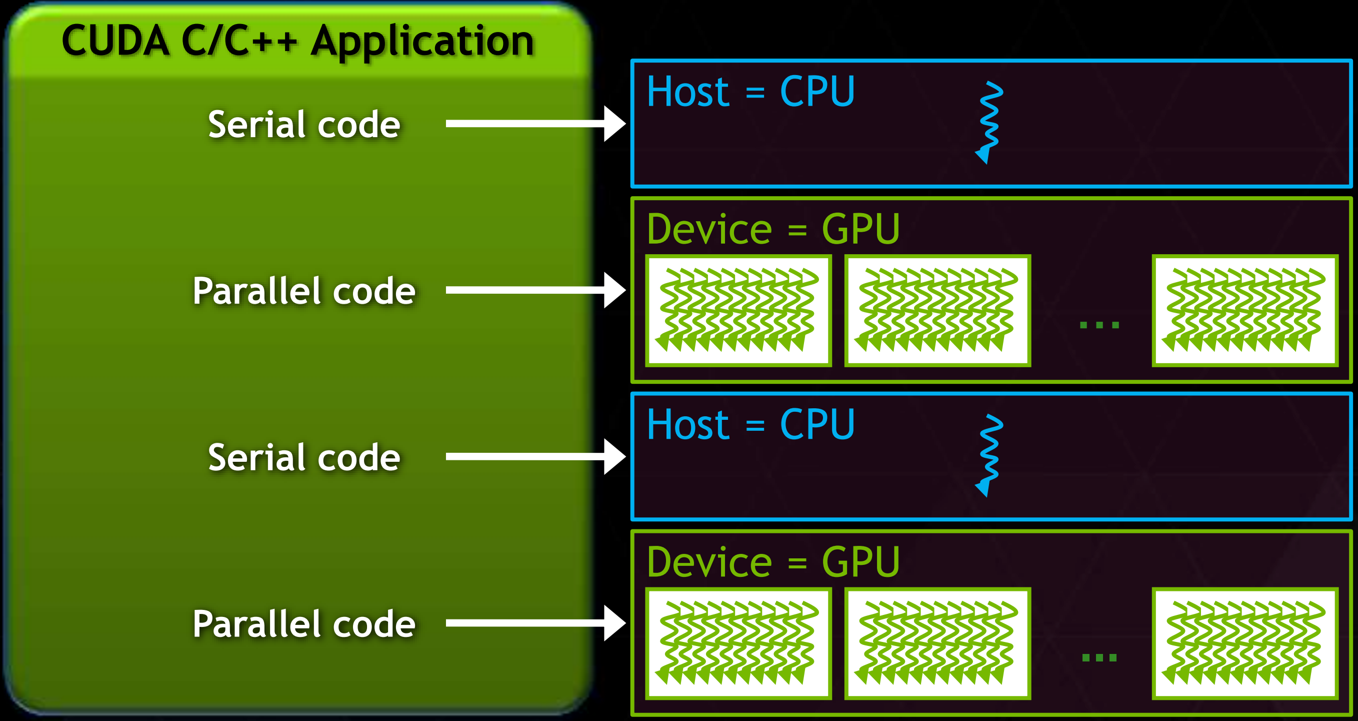
GPU MEMORY HIERARCHY REVIEW



CUDA Programming model

ANATOMY OF A CUDA C/C++ APPLICATION

- Serial code executes in a Host (CPU) thread
- Parallel code executes in many Device (GPU) threads across multiple processing elements



CUDA C : C WITH A FEW KEYWORDS

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Standard C Code

```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Parallel C Code

CUDA KERNELS

- Parallel portion of application: execute as a kernel
  - Entire GPU executes kernel, many threads
- CUDA threads:
  - Lightweight
  - Fast switching
  - 1000s execute simultaneously

CPU	Host	Executes functions
GPU	Device	Executes kernels

CUDA KERNELS: PARALLEL THREADS

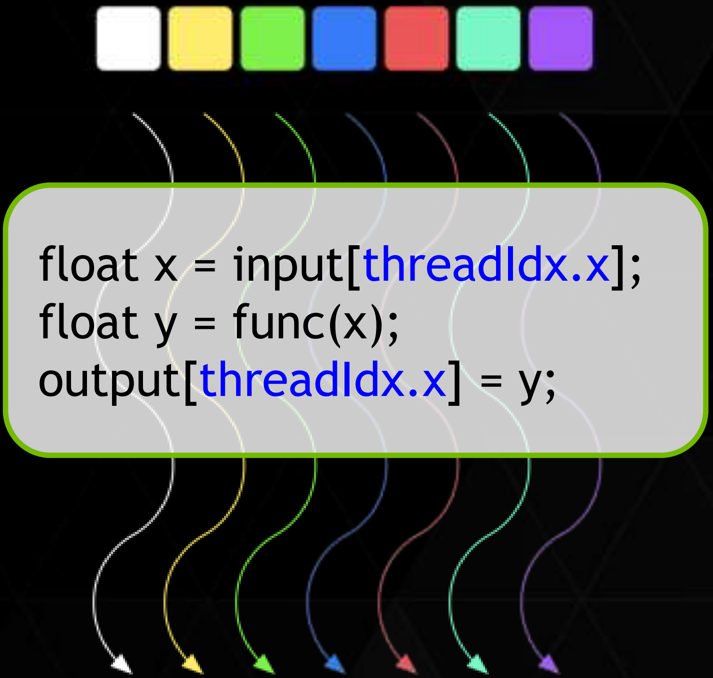
- A kernel is a function executed on the GPU as an array of threads in parallel

- All threads execute the same code, can take different paths

- Each thread has an ID

  - Select input/output data

  - Control decisions



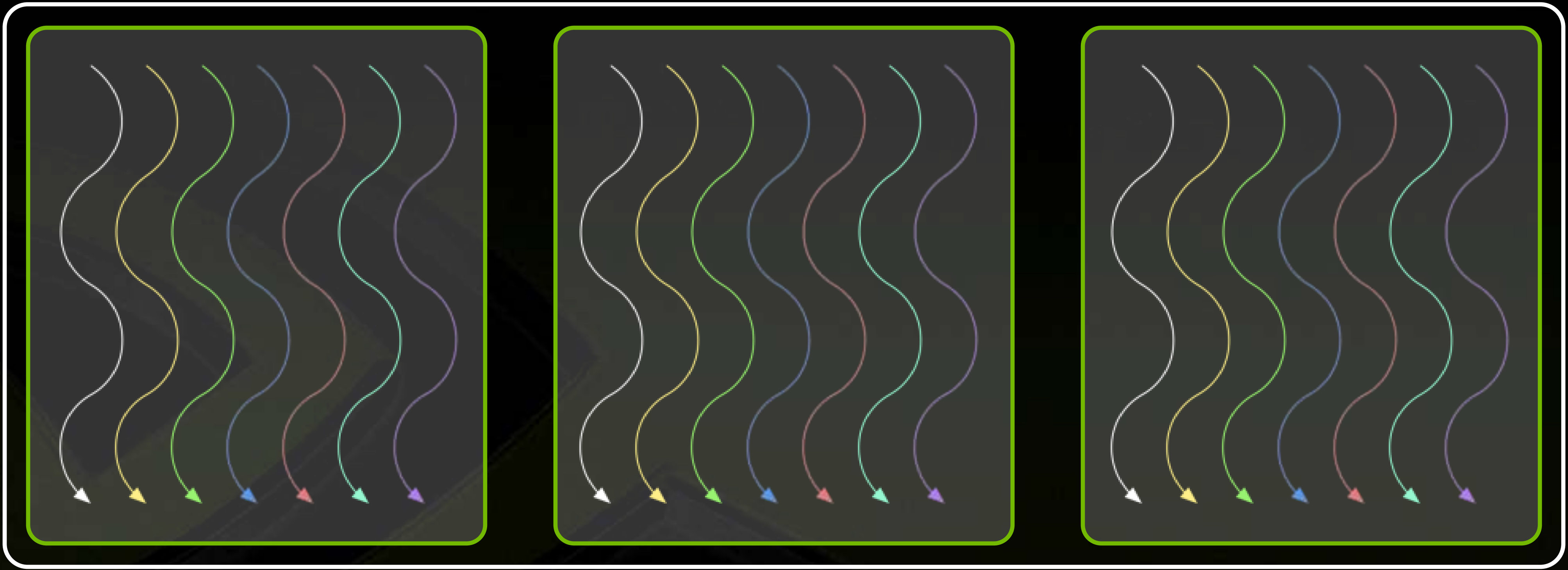
```
float x = input[threadIdx.x];
float y = func(x);
output[threadIdx.x] = y;
```

CUDA Kernels: Subdivide into Blocks

CUDA Kernels: Subdivide into Blocks

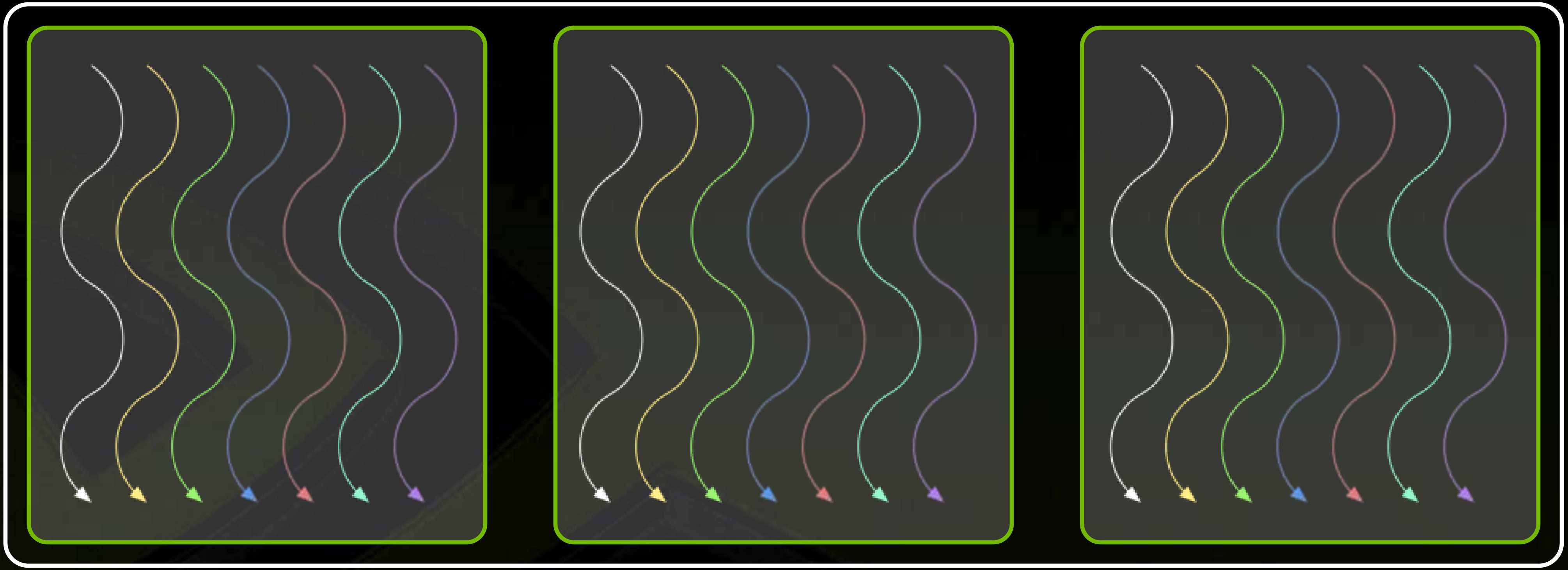
- Threads are grouped into **blocks**

CUDA Kernels: Subdivide into Blocks



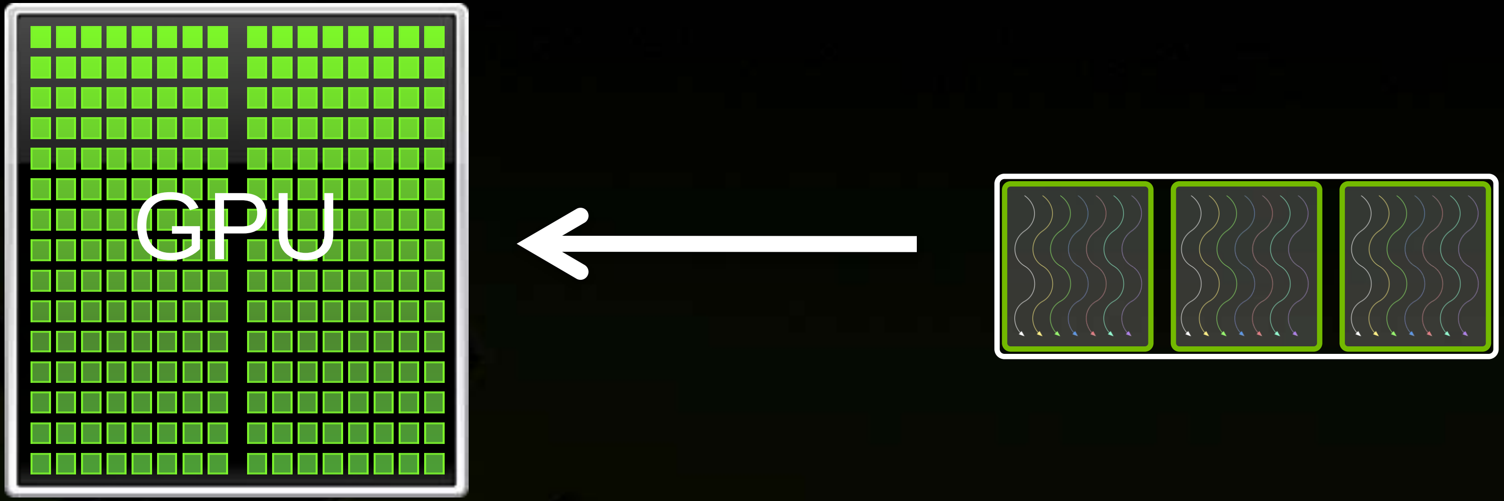
- Threads are grouped into **blocks**
- **Blocks** are grouped into **a grid**

CUDA Kernels: Subdivide into Blocks



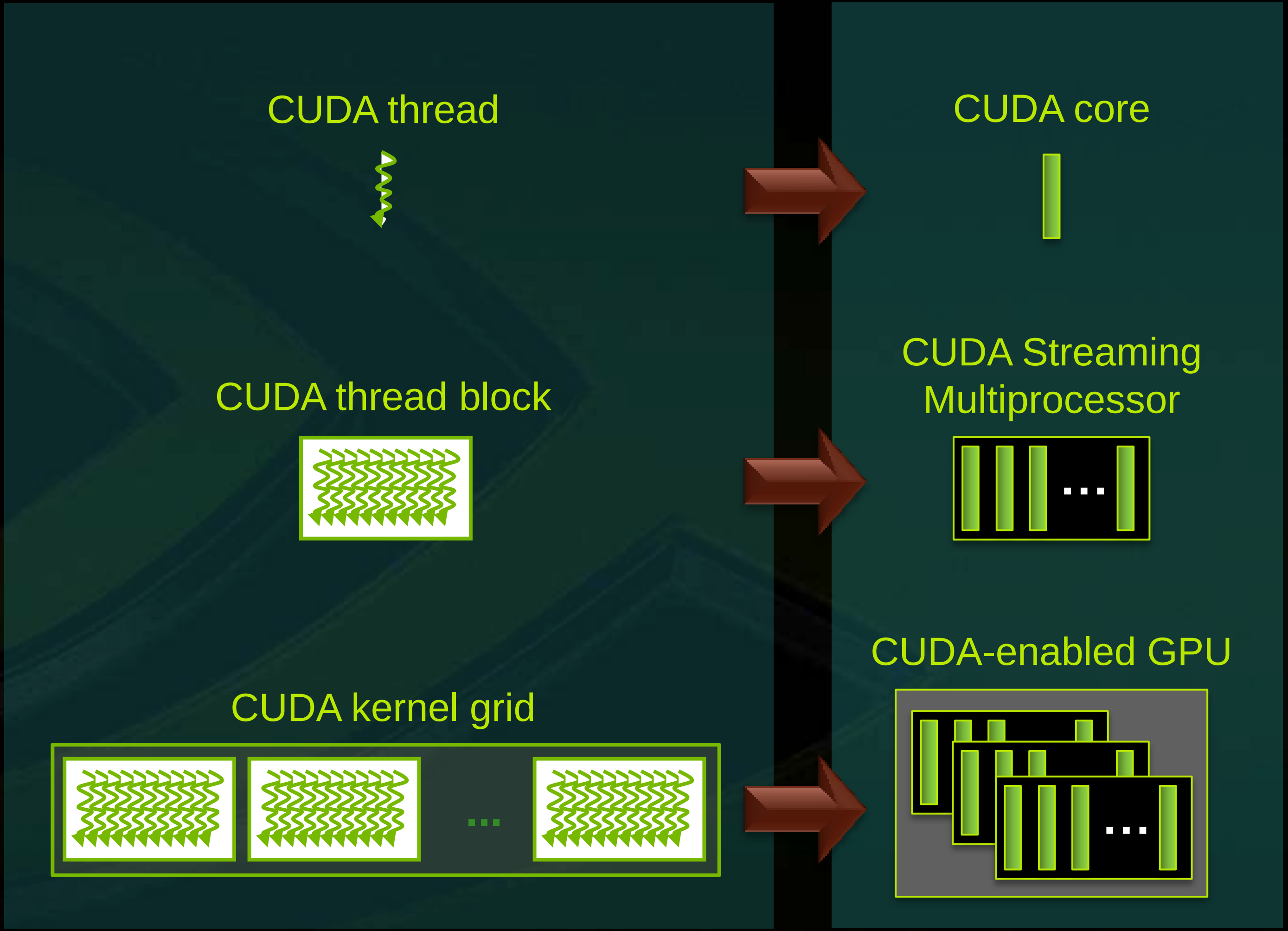
- Threads are grouped into **blocks**
- **Blocks** are grouped into **a grid**
- A **kernel** is executed as a **grid** of **blocks** of **threads**

CUDA Kernels: Subdivide into Blocks



- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

Kernel Execution



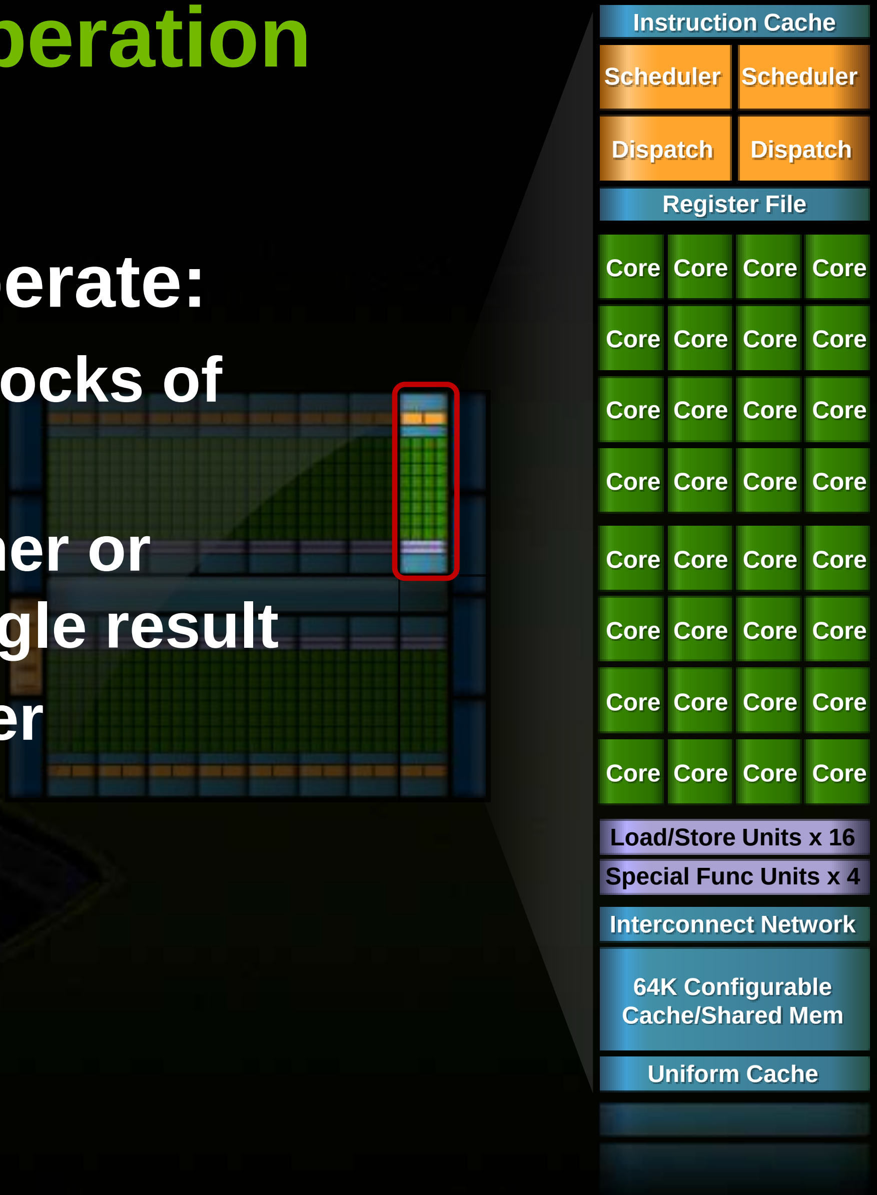
- Each thread is executed by a core

- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources

- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

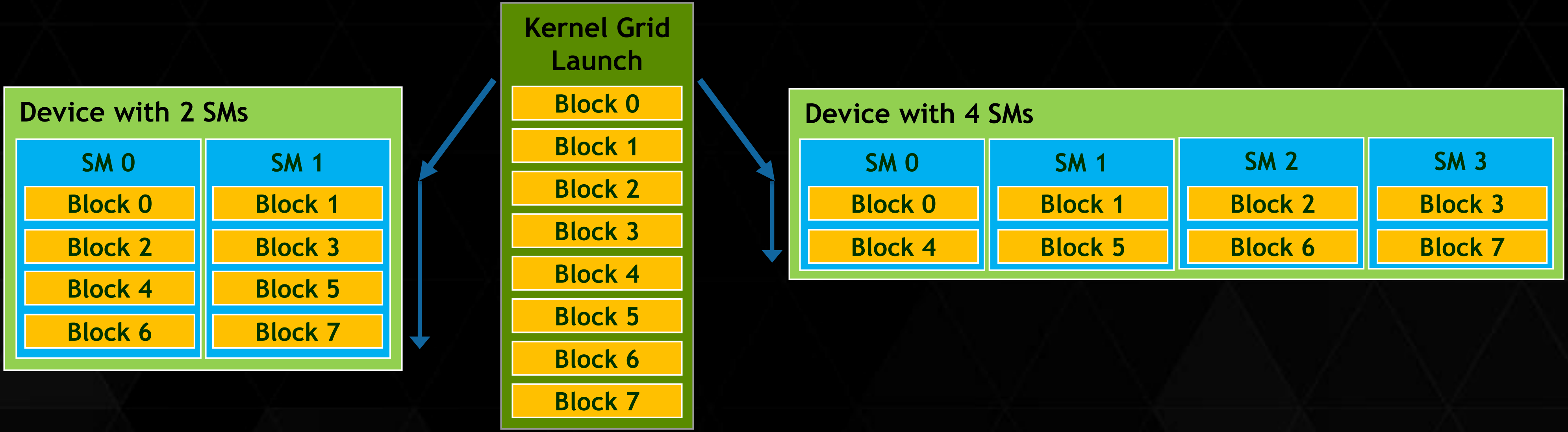
Thread blocks allow cooperation

- Threads may need to cooperate:
  - Cooperatively load/store blocks of memory all will use
  - Share results with each other or cooperate to produce a single result
  - Synchronize with each other



THREAD BLOCKS ALLOW SCALABILITY

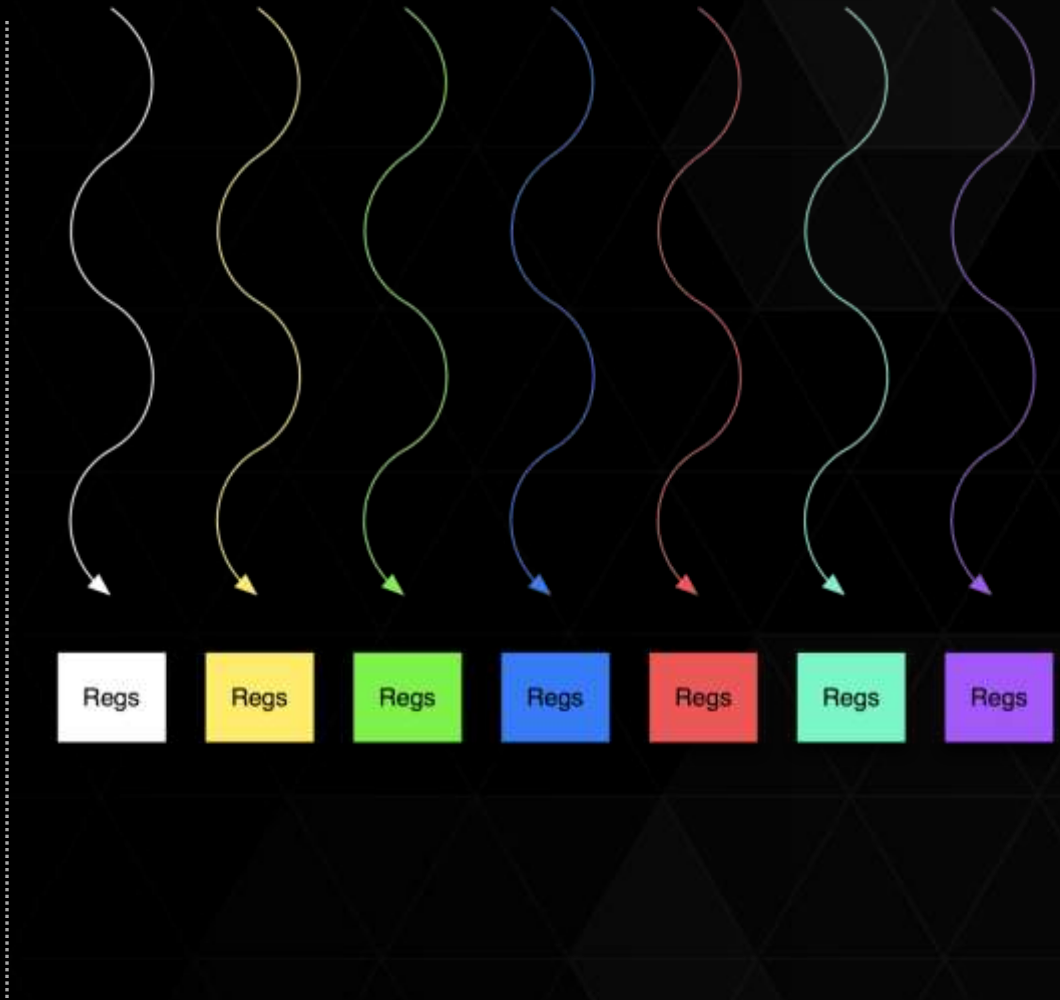
- Blocks can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
    - A kernel scales across any number of SMs



Memory System Hierarchy

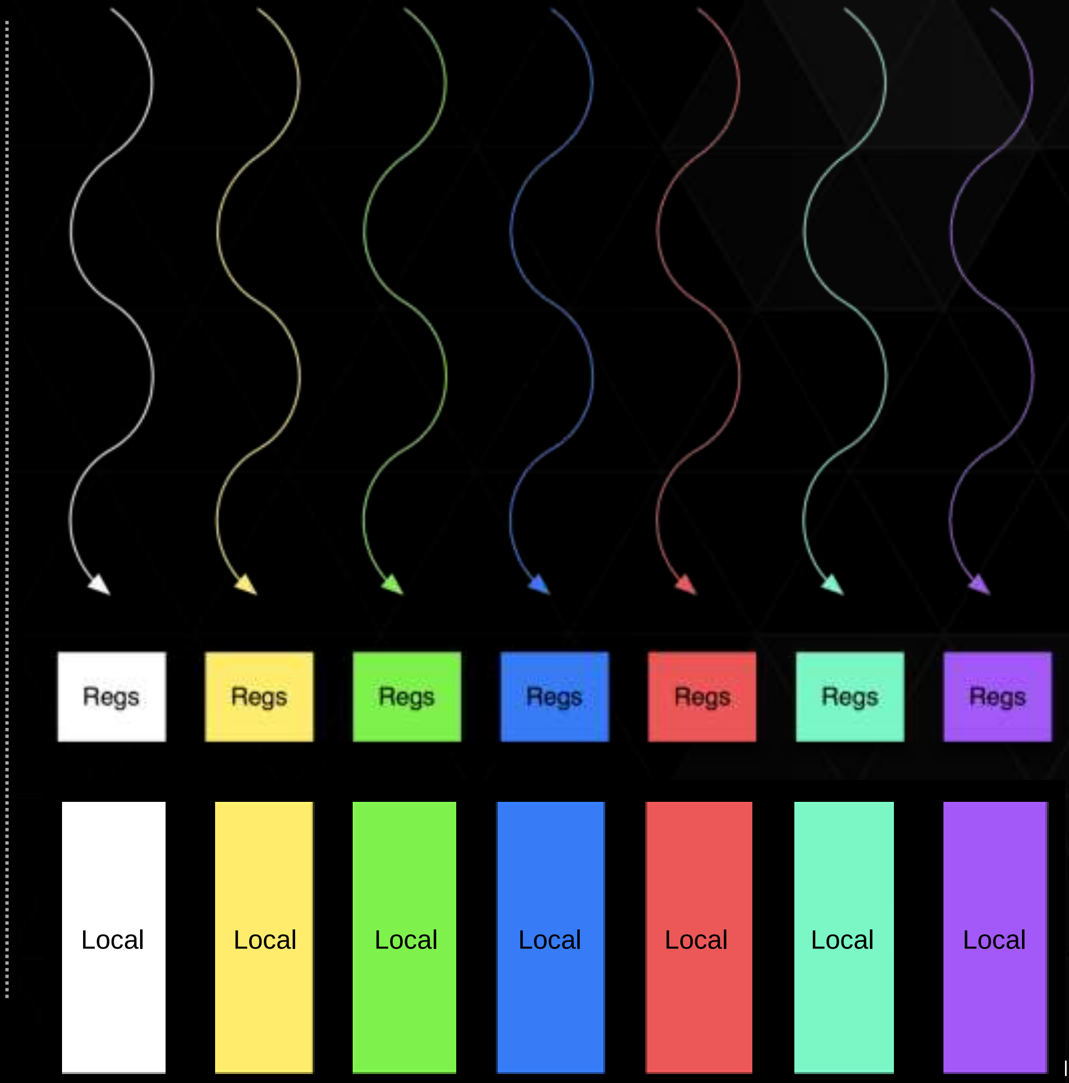
MEMORY HIERARCHY

- Thread:
  - Registers



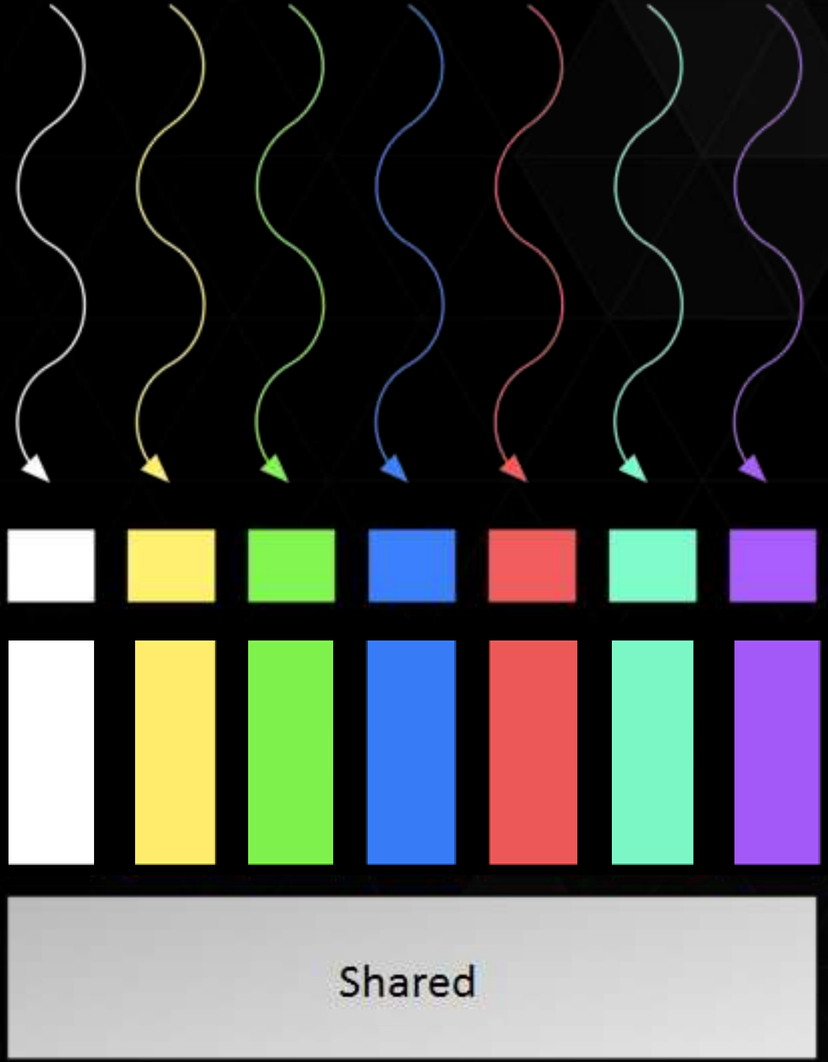
MEMORY HIERARCHY

- Thread:
  - Registers
  - Local memory



MEMORY HIERARCHY

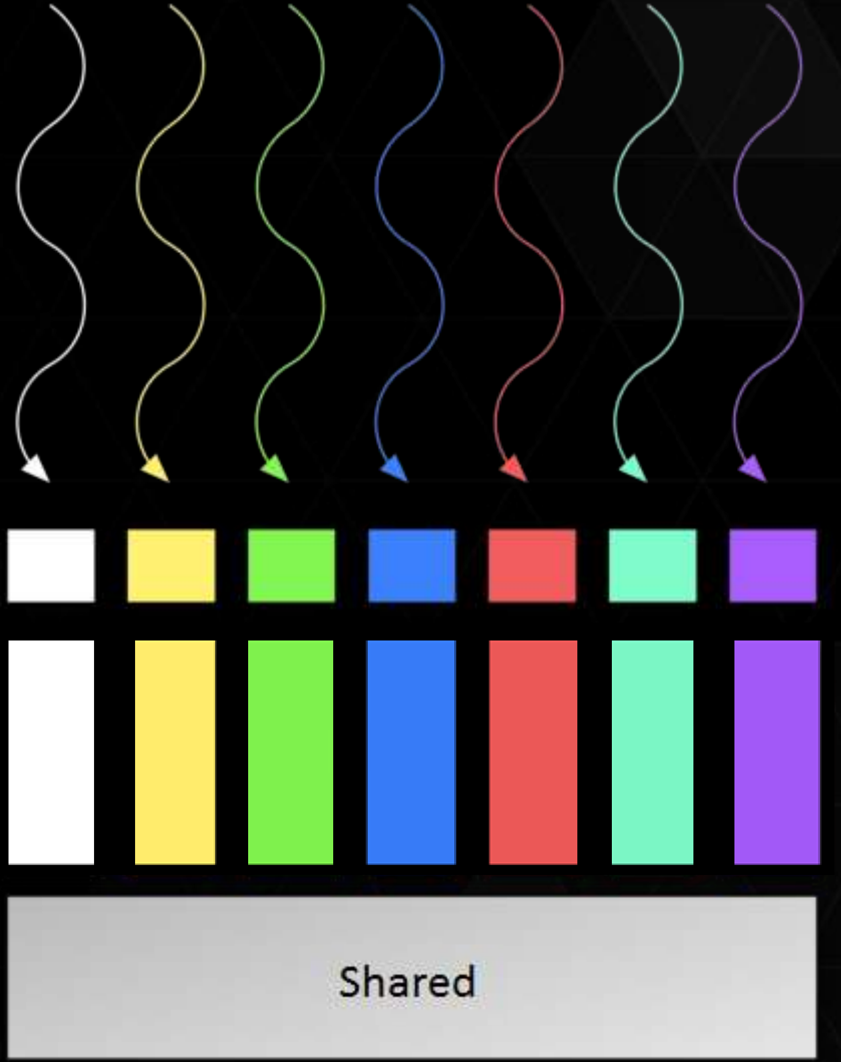
- Thread:
  - Registers
  - Local memory
- Block of threads:
  - Shared memory



MEMORY HIERARCHY : SHARED MEMORY

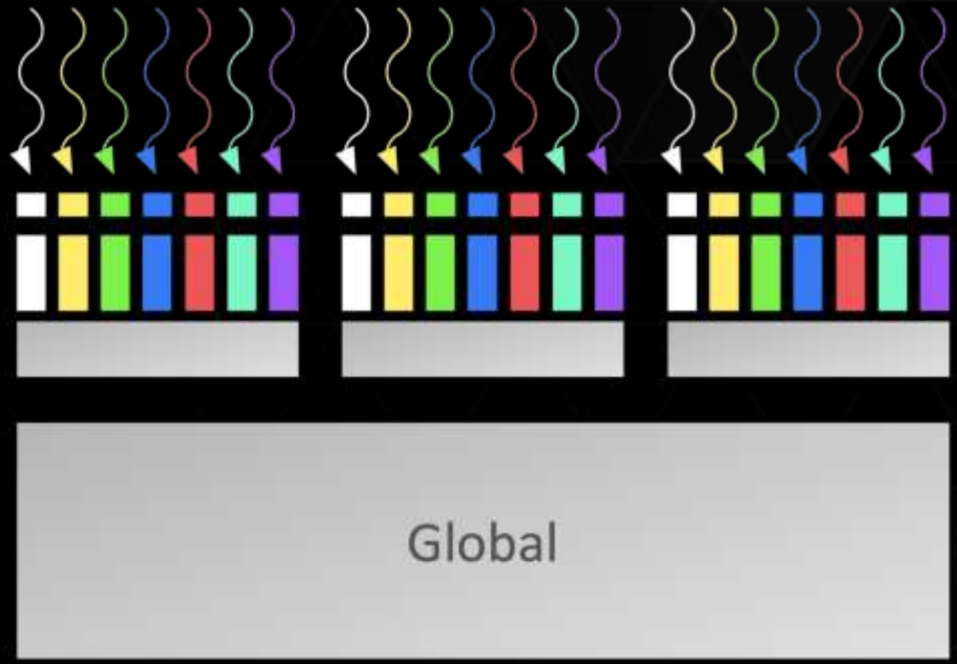
```
__shared__ int a[SIZE];
```

- Allocated per thread block, same lifetime as the block
- Accessible by any thread in the block
- Several uses:
  - Sharing data among threads in a block
  - User-managed cache (reducing gmem accesses)



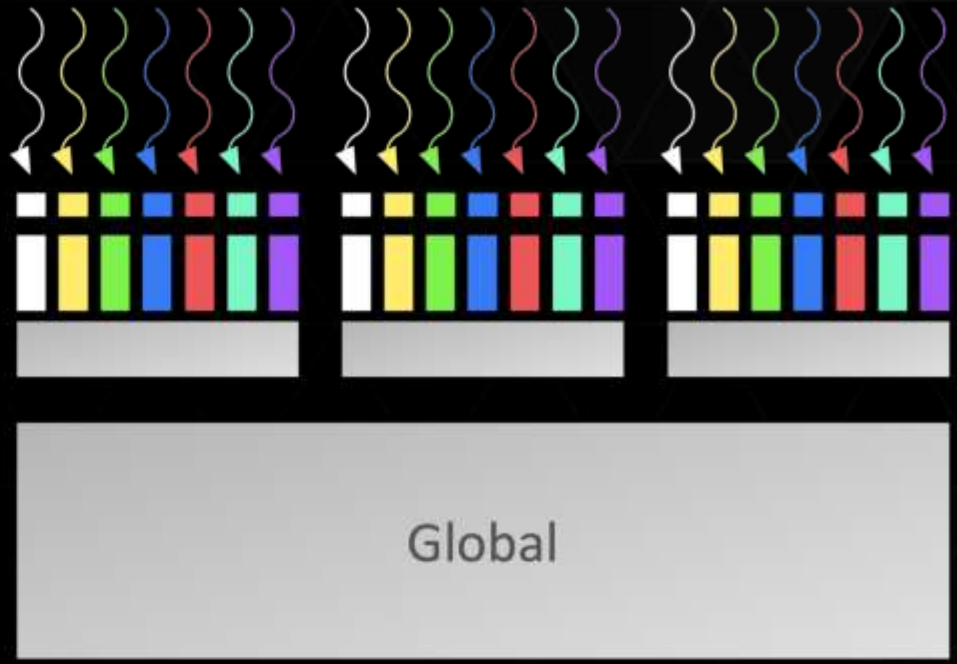
MEMORY HIERARCHY

- Thread:
  - Registers
  - Local memory
- Block of threads:
  - Shared memory
- All blocks:
  - Global memory



MEMORY HIERARCHY : GLOBAL MEMORY

- Accessible by all threads of any kernel
- Data lifetime: from allocation to deallocation by host code
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaMemset (void * pointer, int value, size_t count)
  - cudaFree (void* pointer)



CUDA memory management

MEMORY SPACES

CPU and GPU have separate memory spaces

- Data is moved across PCIe bus
- Use functions to allocate/set/copy memory on GPU just like standard C

Pointers are just addresses

- Can’t tell from the pointer value whether the address is on CPU or GPU
    - Must use cudaPointerGetAttributes(…)
- Must exercise care when dereferencing:
    - Dereferencing CPU pointer on GPU will likely crash
    - Dereferencing GPU pointer on CPU will likely crash

GPU MEMORY ALLOCATION / RELEASE

Host (CPU) manages device (GPU) memory

- cudaMalloc (void ** pointer, size_t nbytes)
- cudaMemset (void * pointer, int value, size_t count)
- cudaFree (void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**)&d_a,  nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Note: Device memory from GPU point of view is also referred to as global memory.

DATA COPIES

```
cudaMemcpy( void *dst,   void *src,   size_t nbytes,
  enum cudaMemcpyKind direction);
```

- returns after the copy is complete
- blocks CPU thread until all bytes have been copied
- doesn’t start copying until previous CUDA calls complete

enum cudaMemcpyKind

- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

Non-blocking memcopies are provided

Basic kernels and execution

CUDA PROGRAMMING MODEL REVISITED

- Parallel code (kernel) is launched and executed on a device by many threads
- Threads are grouped into thread blocks
- Parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in thread and block ID variables

THREAD HIERARCHY

- Threads launched for a parallel section are partitioned into thread blocks
  - Grid = all blocks for a given launch
- Thread block is a group of threads that can:
  - Synchronize their execution
  - Communicate via shared memory

IDS AND DIMENSIONS

Threads

- 3D IDs, unique within a block

Blocks

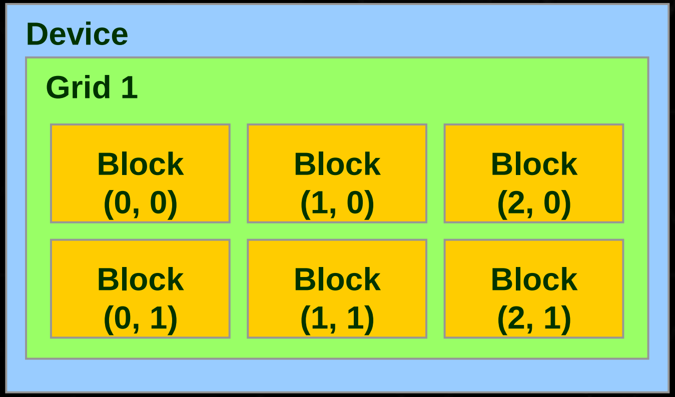
- 2D IDs, unique within a grid

Dimensions set at launch time

- Can be unique for each grid

Built-in variables

- threadIdx, blockIdx
- blockDim, gridDim



(Continued)

IDS AND DIMENSIONS

Threads

- 3D IDs, unique within a block

Blocks

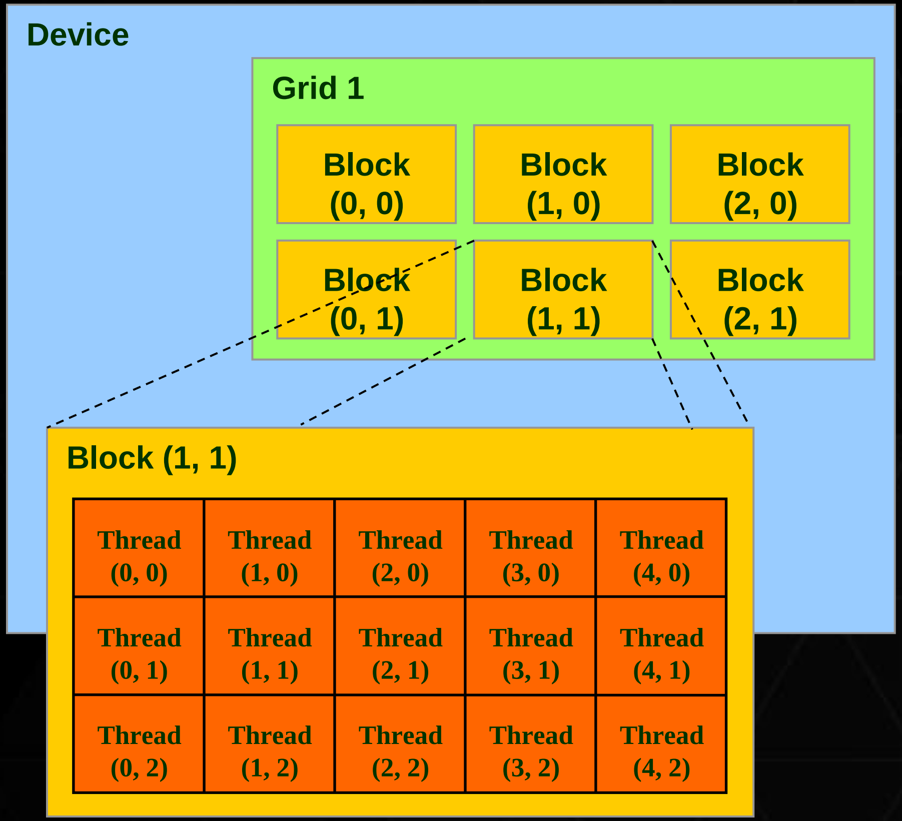
- 2D IDs, unique within a grid

Dimensions set at launch time

- Can be unique for each grid

Built-in variables

- threadIdx, blockIdx
- blockDim, gridDim



LAUNCHING KERNELS ON GPU

Launch parameters (triple chevron <<<>>> notation)

- grid dimensions (up to 2D), dim3 type
- thread-block dimensions (up to 3D), dim3 type
- shared memory: number of bytes per block
  - for extern smem variables declared without size
  - Optional, 0 by default
- stream ID
  - Optional, 0 by default

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block, 0, 0>>>(...);
kernel<<<32, 512>>>(...);
```

GPU KERNEL EXECUTION

- Kernel launches on a grid of blocks, <<<grid,block>>>(arg1,…)
- Each block is launched on one SM
  - A block is divided into warps of 32 threads each (think 32-way vector)
  - Warps in a block are scheduled and executed.
    - All threads in a warp execute same instruction simultaneously (think SIMD)
  - Number of blocks/SM determined by resources required by the block
    - Registers, shared memory, total warps, etc.
- Block runs to completion on SM it started on, no migration.

BLOCKS MUST BE INDEPENDENT

Any possible interleaving of blocks should be valid

- presumed to run to completion without pre-emption
- can run in any order
- can run concurrently OR sequentially

Blocks may coordinate but not synchronize

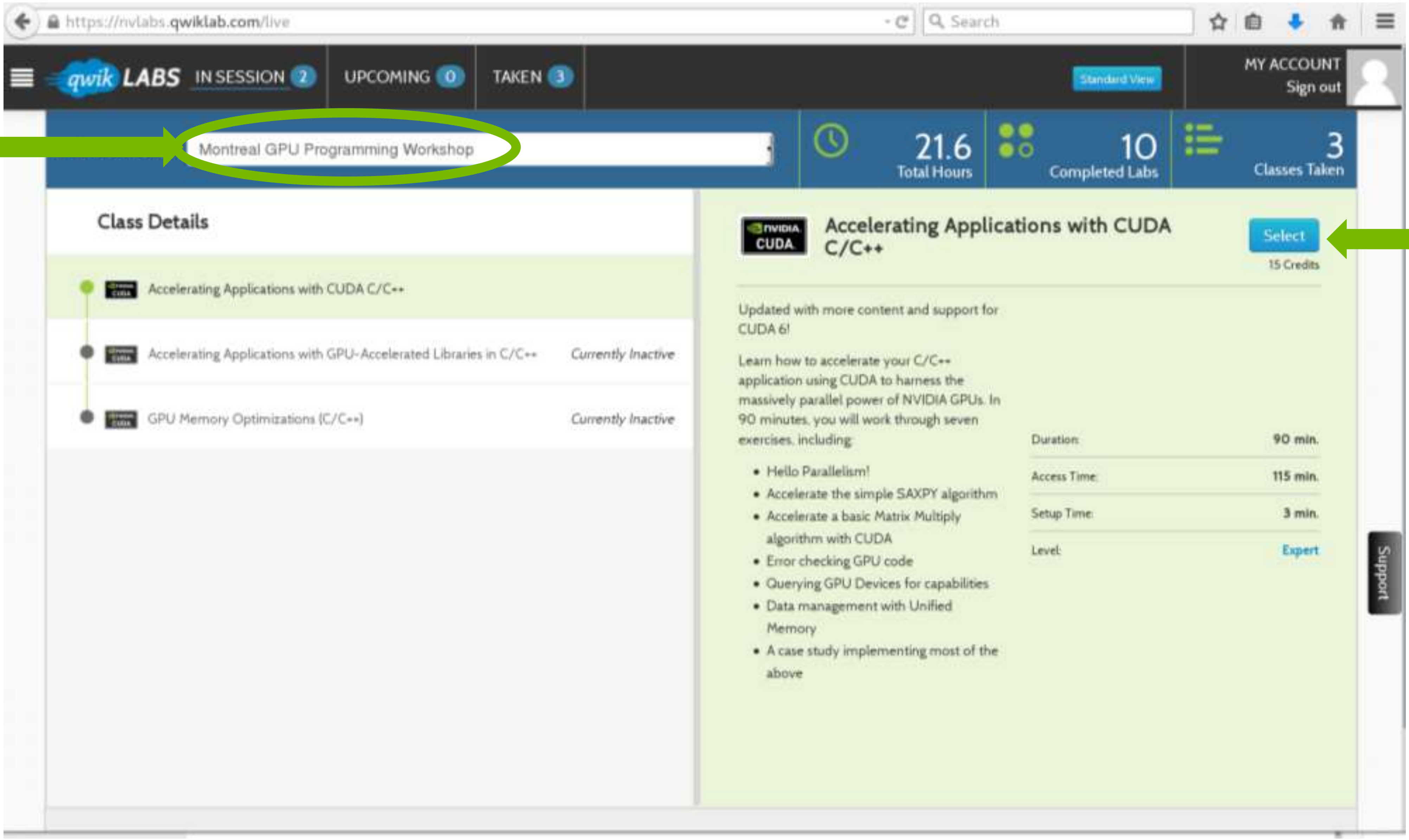
- shared queue pointer: OK
- shared lock: BAD … any dependence on order easily deadlocks

Independence requirement gives scalability

Hands-on labs

Prepare and Start AWS Instance

- Open a browser, go to nvlabs.qwiklab.com
  - Register (it's free) and Sign in.
  - Select the correct lab (Montreal GPU Programming Workshop) and once enabled press "Start Lab"
  - Instance can take up to 10 minutes to start.
- Three labs are available:
  - Accelerating Applications with CUDA C/C++
  - (optional) Accelerating Applications with GPU-Accelerated Libraries in C/C++
  - (optional) GPU Memory Optimizations (C/C++)



21.6 Total Hours
10 Completed Labs
3 Classes Taken

Class Details

- Accelerating Applications with CUDA C/C++
- Accelerating Applications with GPU-Accelerated Libraries in C/C++ *Currently Inactive*
- GPU Memory Optimizations (C/C++) *Currently Inactive*

Accelerating Applications with CUDA C/C++

Select

15 Credits

Updated with more content and support for CUDA 6!

Learn how to accelerate your C/C++ application using CUDA to harness the massively parallel power of NVIDIA GPUs. In 90 minutes, you will work through seven exercises, including:

- Hello Parallelism!
- Accelerate the simple SAXPY algorithm
- Accelerate a basic Matrix Multiply algorithm with CUDA
- Error checking GPU code
- Querying GPU Devices for capabilities
- Data management with Unified Memory
- A case study implementing most of the above

Duration:	90 min.
Access Time:	115 min.
Setup Time:	3 min.
Level:	Expert

Wrap up

Software

- GPU Driver
- CUDA toolkit
  - Includes all the software necessary for developers to write applications
    - Compiler (nvcc), libraries, profiler, debugger, documentation
- CUDA Samples
  - Samples illustrating GPU functionality and performance
  - Examples illustrating important programming constructs and techniques.
- www.nvidia.com/getcuda -- all above software is free

Want to try?

Links and resources

Deep Learning https://developer.nvidia.com/deep-learning

Hands-on labs https://nvidia.qwiklab.com/

Question? Email jbernauer@nvidia.com

COME DO YOUR LIFE’S WORK
JOIN NVIDIA

We are looking for great people at all levels to help us accelerate the next wave of AI-driven computing in Research, Engineering, and Sales and Marketing.

Our work opens up new universes to explore, enables amazing creativity and discovery, and powers what were once science fiction inventions like artificial intelligence and autonomous cars.

Check out our career opportunities:

- www.nvidia.com/careers
- Reach out to your NVIDIA social network or NVIDIA recruiter at DeepLearningRecruiting@nvidia.com