



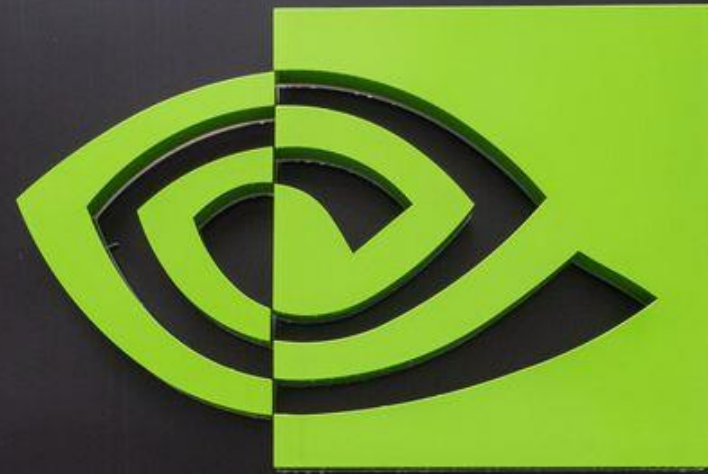
Accelerating AI with

 **nVIDIA**.solutions

Asher Fredman, Solution Architect

TALK AGENDA

- ❖ What and Why NVIDIA
- ❖ GPUs Vs. CPUs - the power parallel computing
- ❖ Introduce the CUDA programming model
- ❖ GPU architecture and how to utilize GPU capabilities
- ❖ GPU acceleration in DL
 - Matrix Multiplications
 - Tensor Cores and AMP
 - Inference Optimizations and Sparsity



nVIDIA

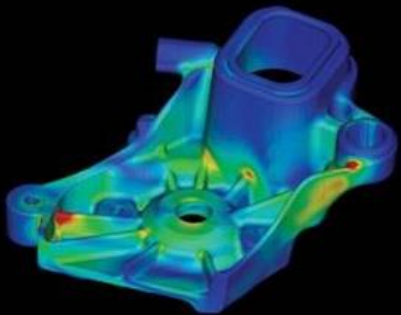


NVIDIA

From Computer Graphics to GPU Computing



GAMING



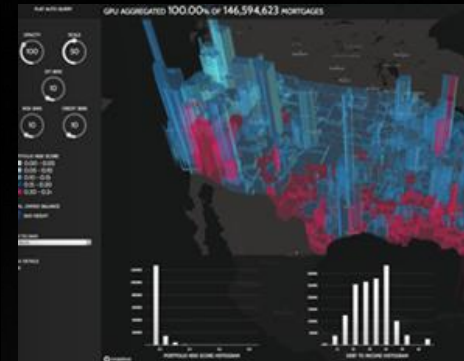
DESIGN



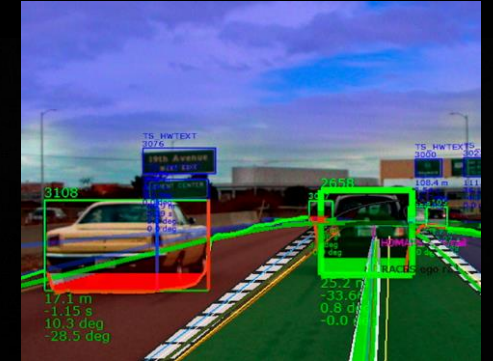
HPC



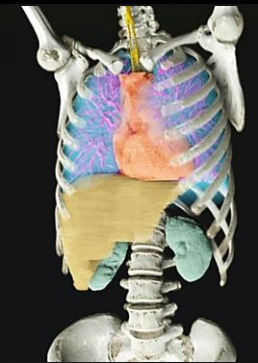
DEEP LEARNING



MACHINE LEARNING

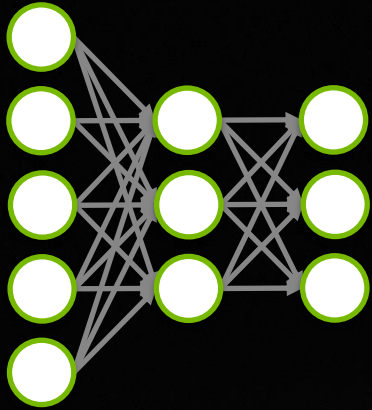


TRANSPORTATION



HEALTHCARE

THE BIG BANG IN AI



DNN



BIG DATA



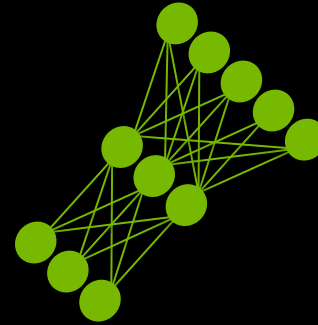
GPU

DEEP LEARNING REVOLUTIONIZING COMPUTING

Image Classification, Object Detection,
Localization, Action Recognition



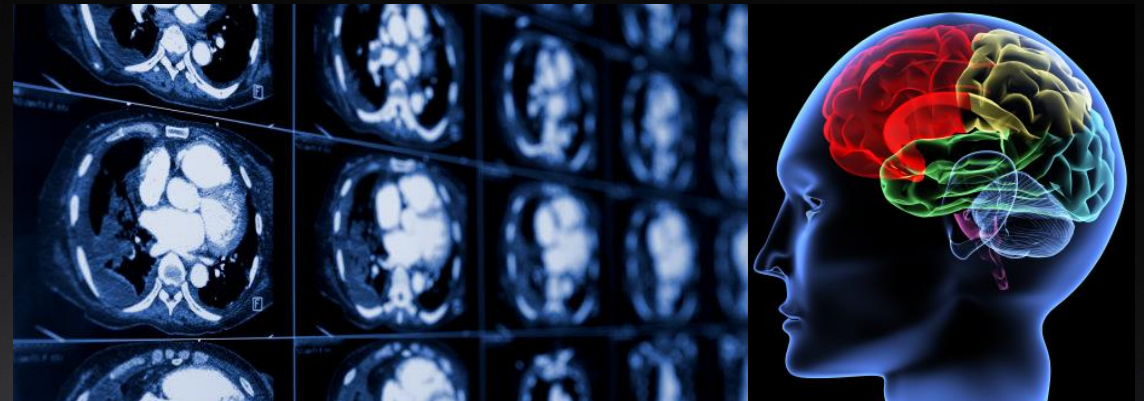
Speech Recognition, Speech Translation,
Natural Language Processing



Pedestrian Detection, Lane Detection,
Traffic Sign Recognition



Breast Cancer Cell Mitosis Detection,
Volumetric Brain Image Segmentation





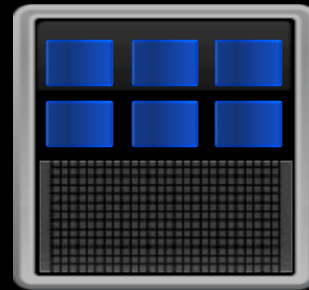
GPU and CPU

POWERING ALL INDUSTRIES

With a single innovation...

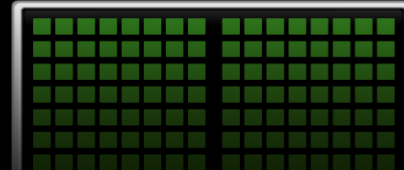
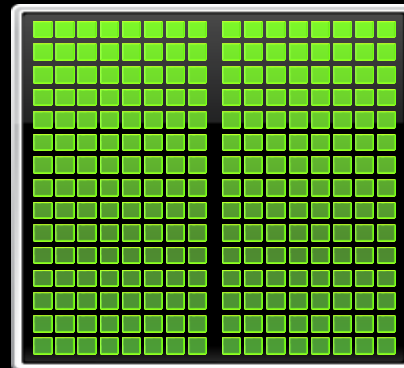
CPU

Optimized for
Serial Tasks

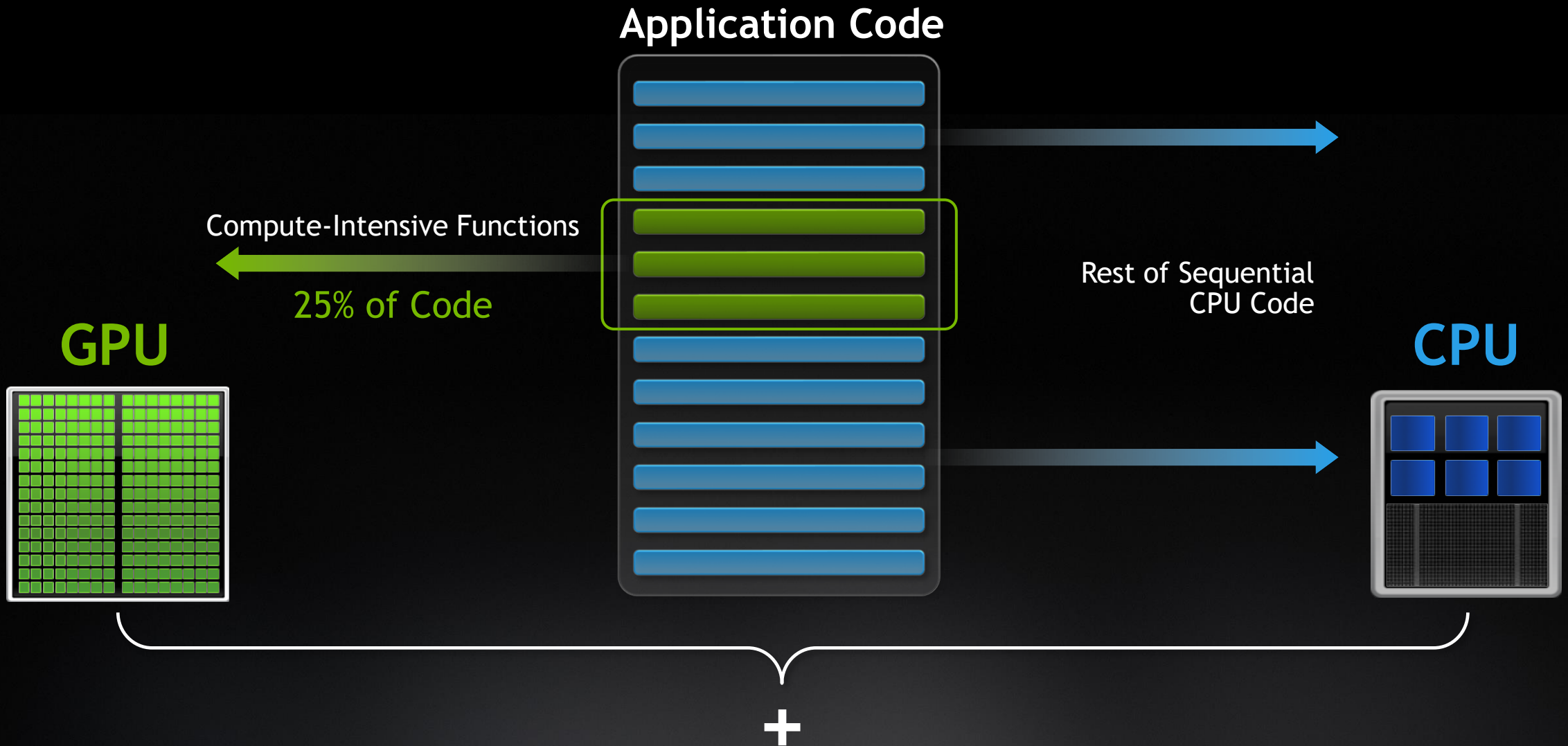


GPU Accelerator

Optimized for
Parallel Tasks

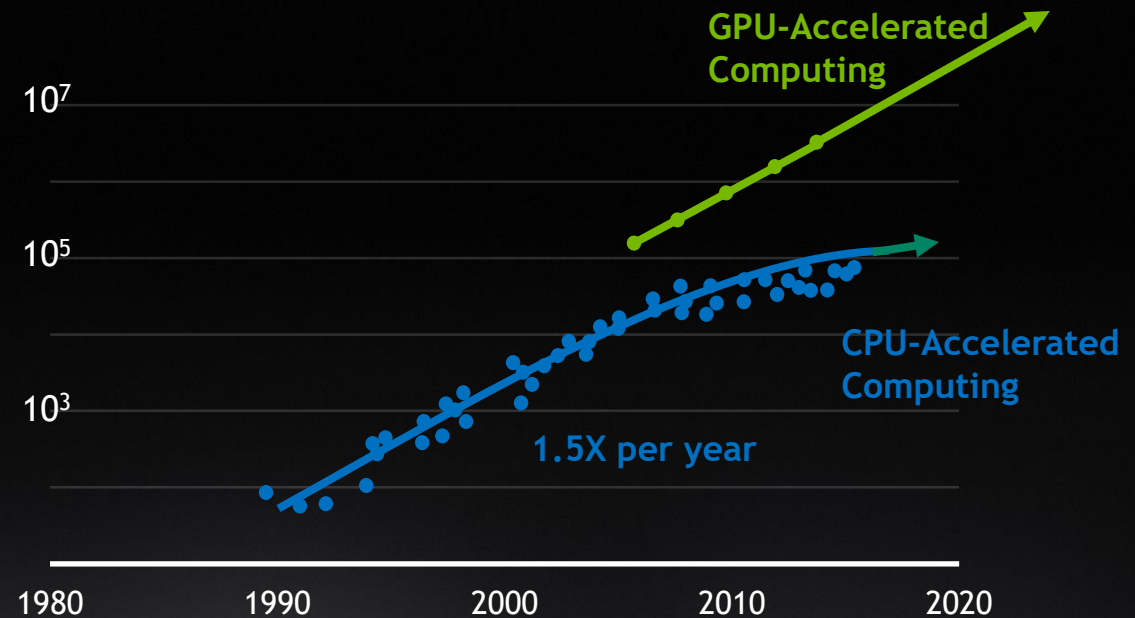
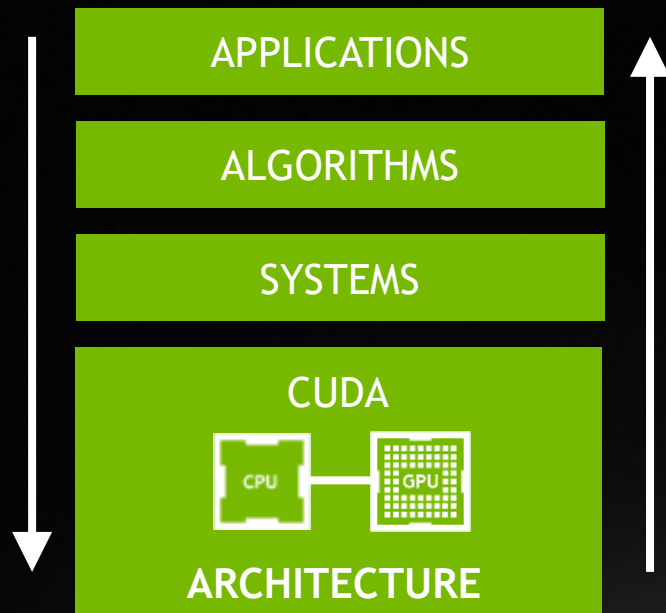


SMALL CHANGES, BIG SPEED-UP



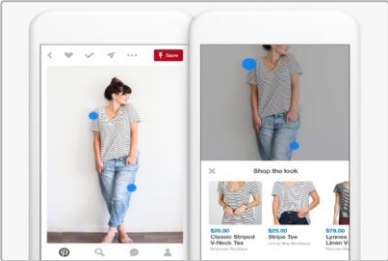
A SUPERCHARGED COMPUTING MODEL

To power the next advances in technology...



NVIDIA END TO END AI PLATFORM

Recommendation



MERLIN

Image & Video



TLT/DS

Speech, NLP, Conversation



RIVA

Search



AI

User Analytics



RAPIDS/SPARK

NGC

Software Hub

Pre-trained Models

SDKs

Validated Systems

Deep Learning Frameworks



Data Science Community



Deep Learning SDK

Data Science SDK

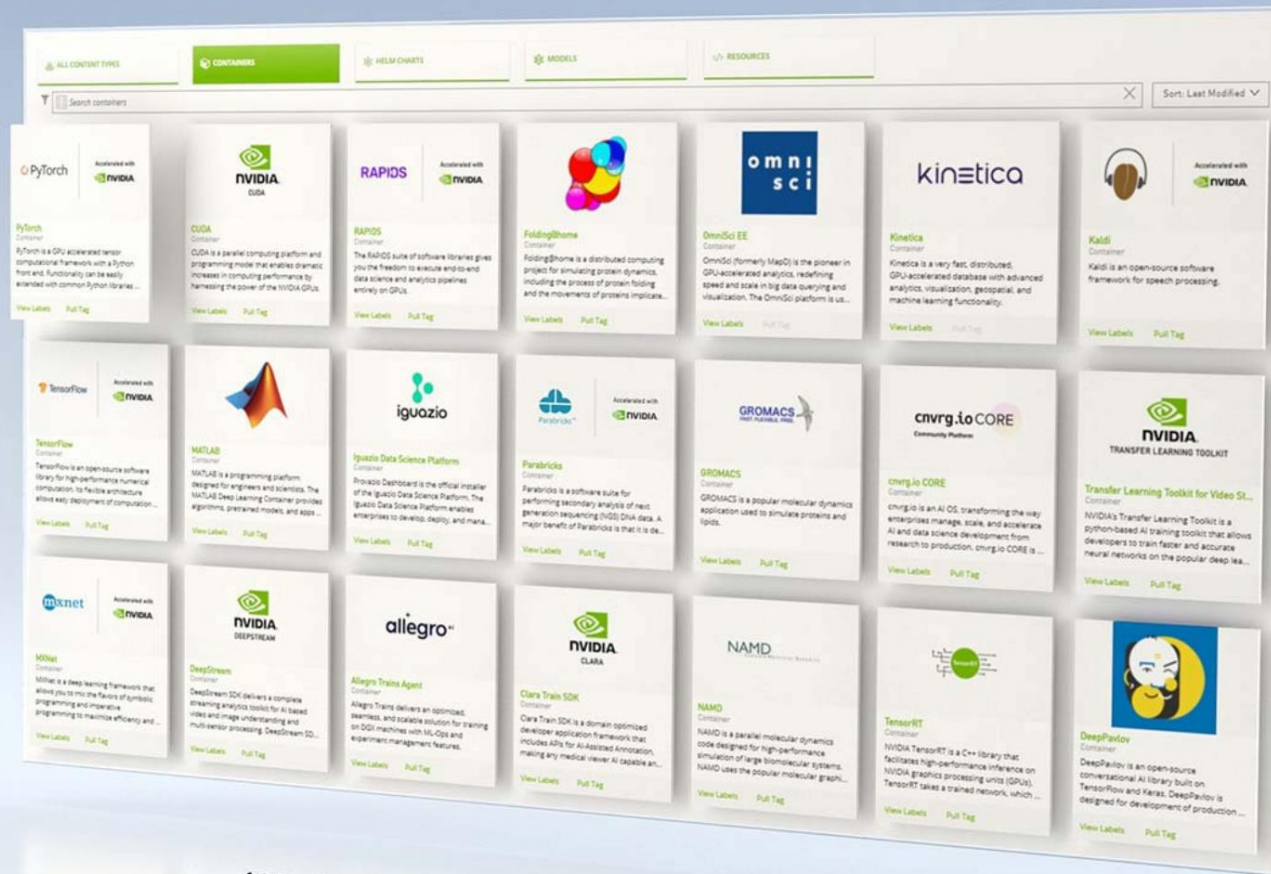
Data Center Tools

CUDA

GPU HGX CONNECT-X6 DGX EGX EVERY MAJOR CLOUD

BUILD AI FASTER. DEPLOY ANYWHERE WITH NGC

ngc.nvidia.com



100+ CONTAINERS | 250K USERS | 1M DOWNLOADS

CONTINUOUS PERFORMANCE IMPROVEMENT

Developers' Software Optimizations Deliver Better Performance on the Same Hardware

Monthly DL Framework Updates & Stack Optimizations Drive Performance

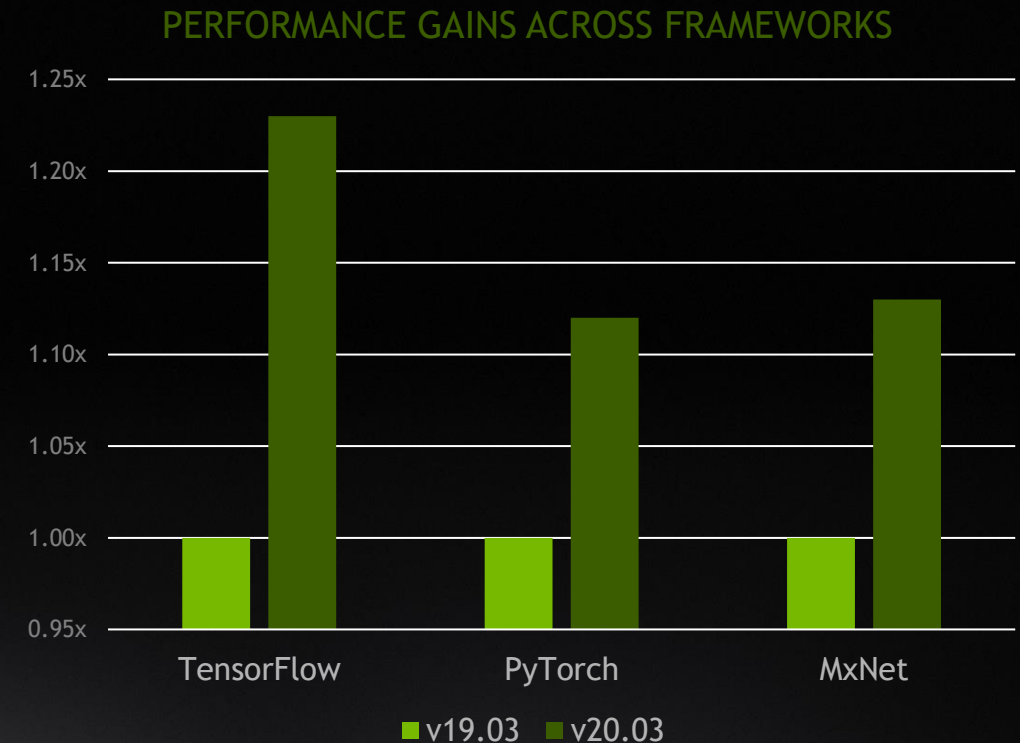
cuDNN - Highly tuned standard training routines

cuBLAS - Highly tuned matrix multiplication

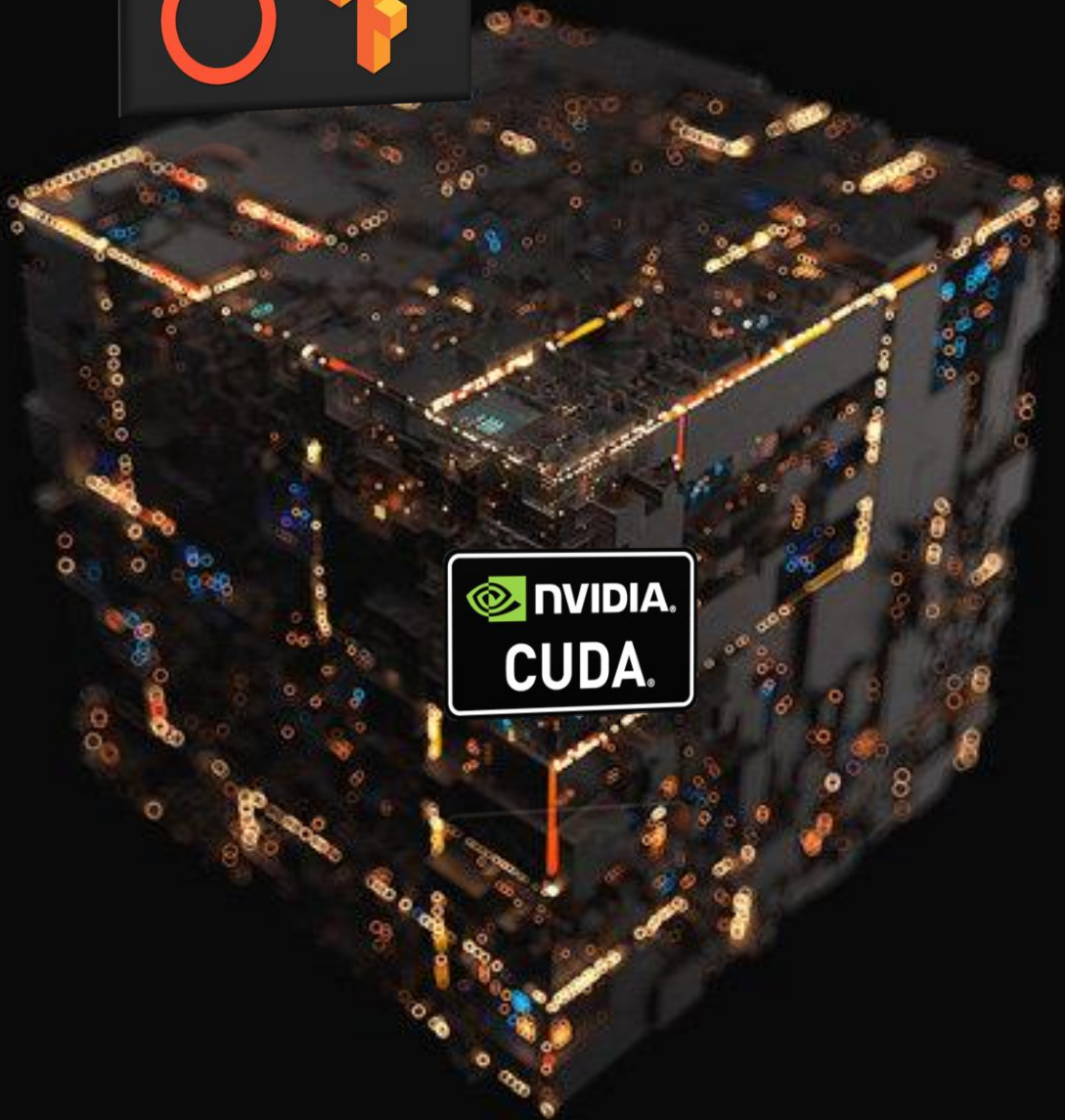
DALI - Moves compute intensive pre-processing to GPUs

NCCL - Faster training across multi-GPU architecture

Framework - Latest versions w/ newest features and superior perf



512 Batch Size for TF & PyT, 256 Batch size for MxNet | ResNet-50 Training v1.5 | 16x V100 | DGX-2



 NVIDIA.
CUDA.

CUDA C/C++ BASICS



What is CUDA?



- **A general-purpose parallel computing platform and programming model.**
- General purpose - one ring to rule them all
- Parallel computing via minimal extensions to familiar environments
- GPU abstractions to optimize code using HW capabilities

3 WAYS TO ACCELERATE APPLICATIONS

Applications

Libraries

“Drop-in”
Acceleration

OpenACC
Directives

Easily Accelerate
Applications

Programming
Languages

Maximum
Flexibility

Introduction to CUDA C/C++

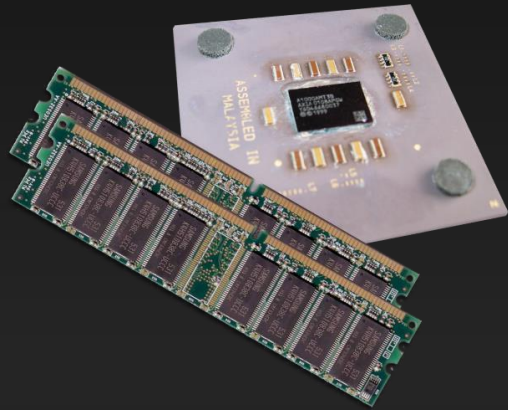


- What will you learn in this section?
 - Start with vector addition
 - Write and launch CUDA C/C++ kernels
 - Manage GPU memory

Heterogeneous Computing



- Terminology:
 - *Host* The CPU and its memory (host memory)
 - *Device* The GPU and its memory (device memory)

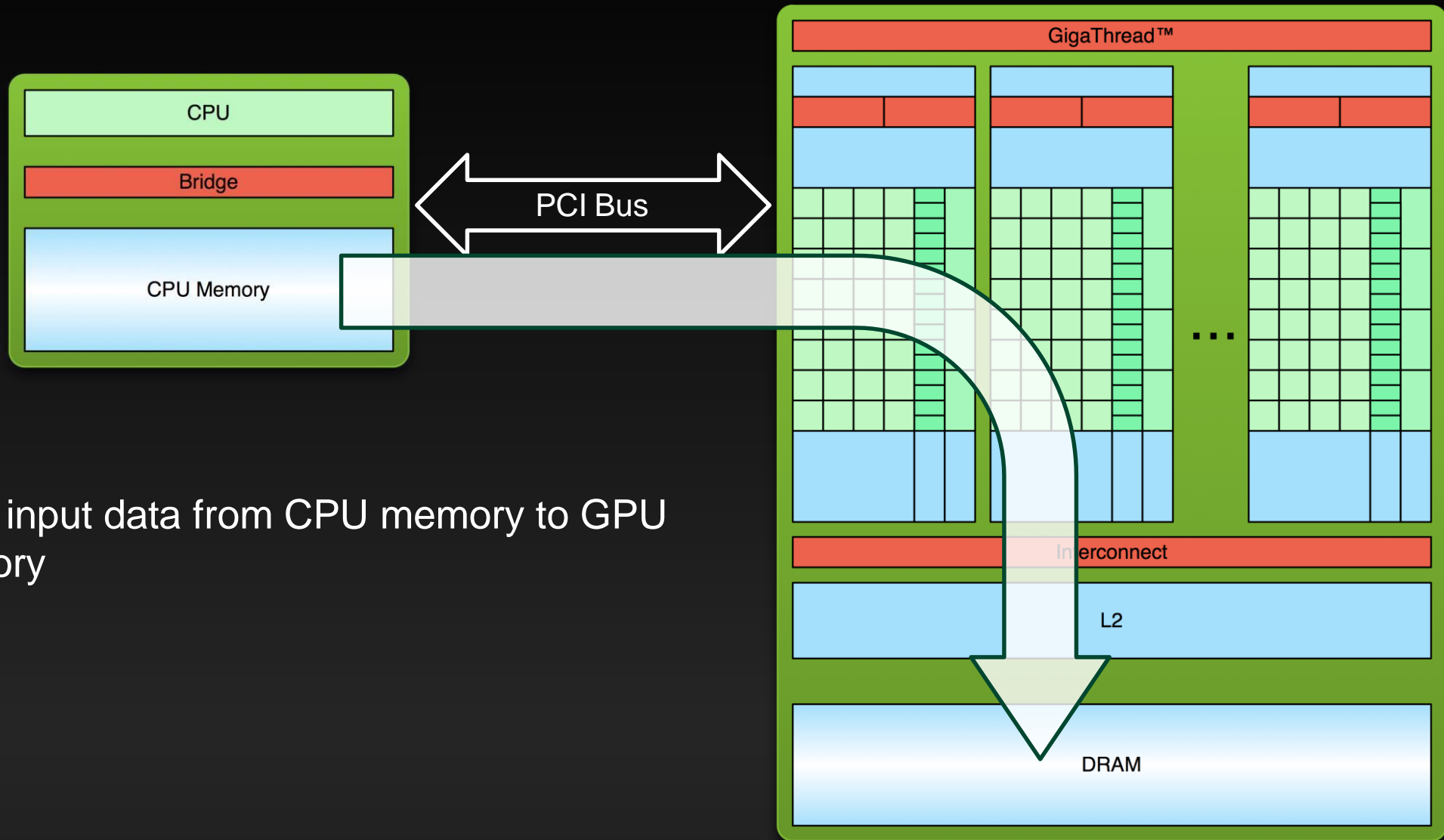


Host



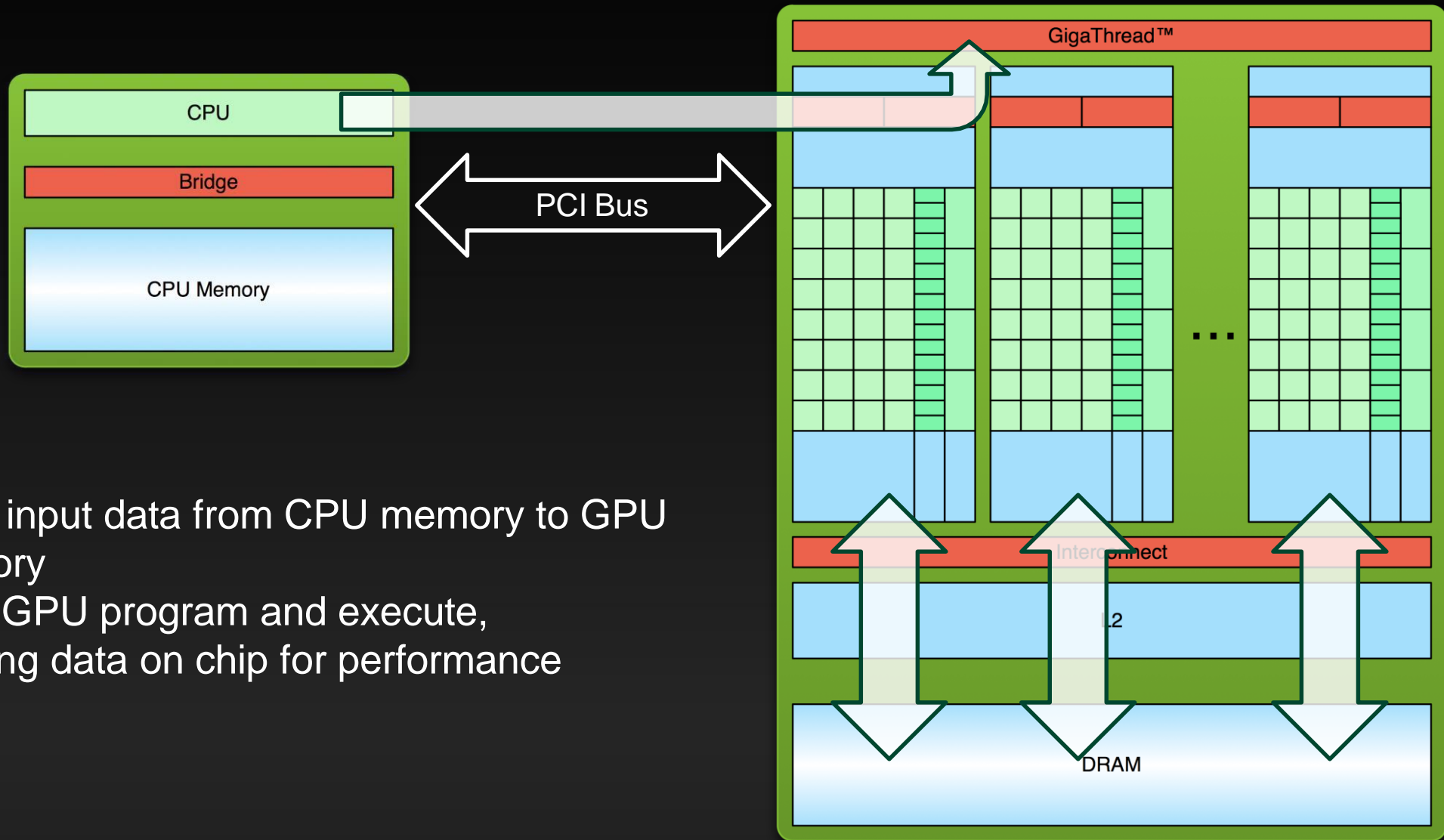
Device

Simple Processing Flow



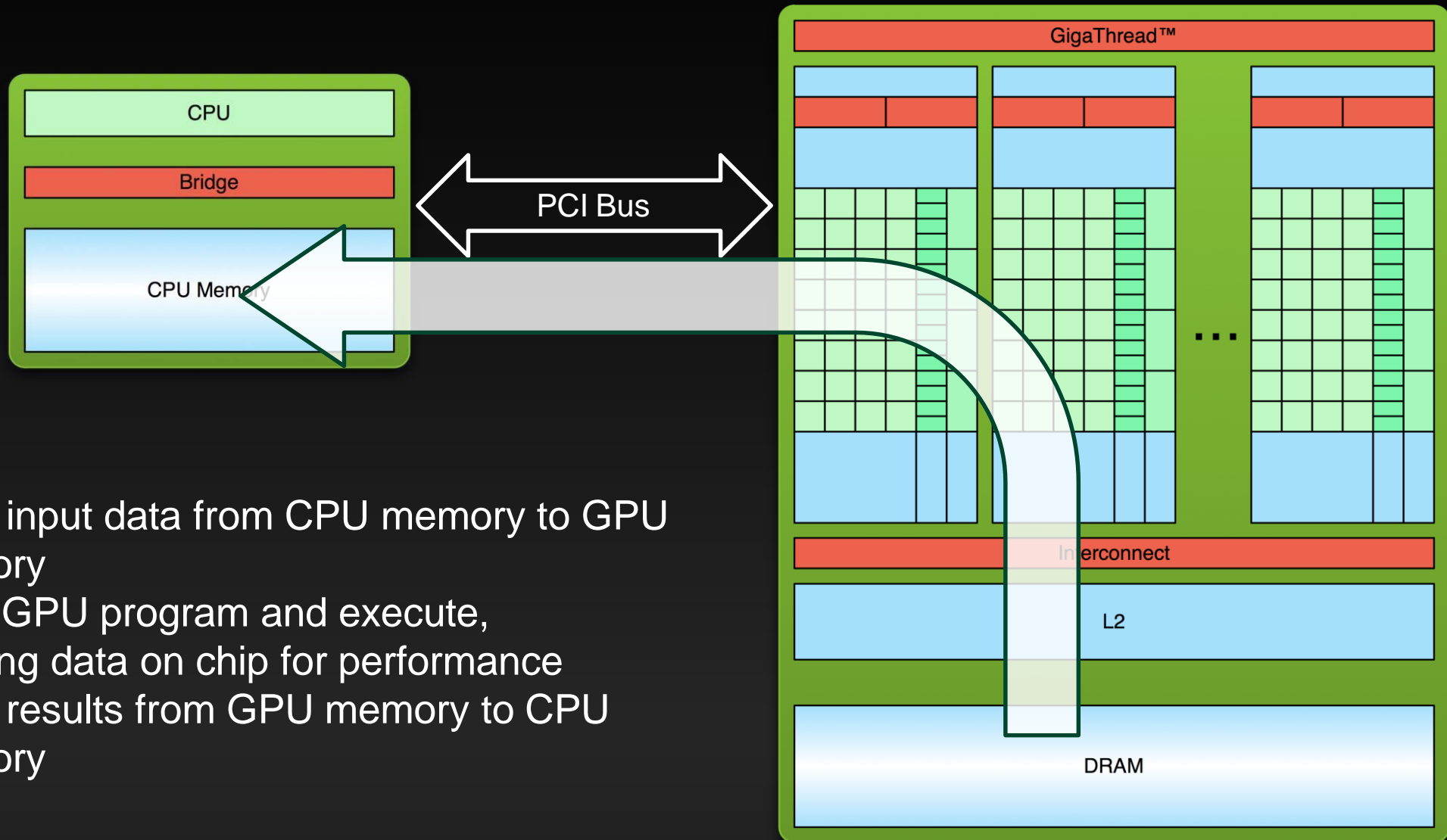
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



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

Simple Processing Flow



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

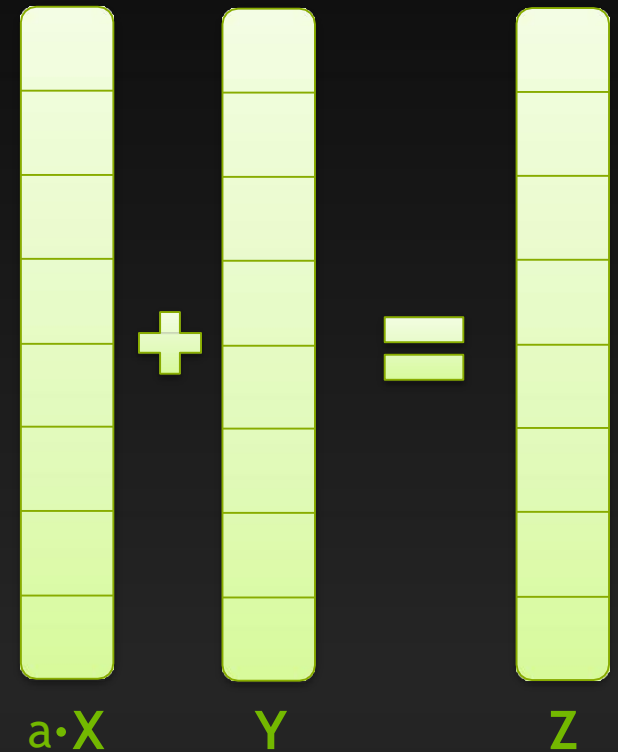
Parallel SAXPY



$$z = \alpha x + y$$

x, y, z : vector
 α : scalar

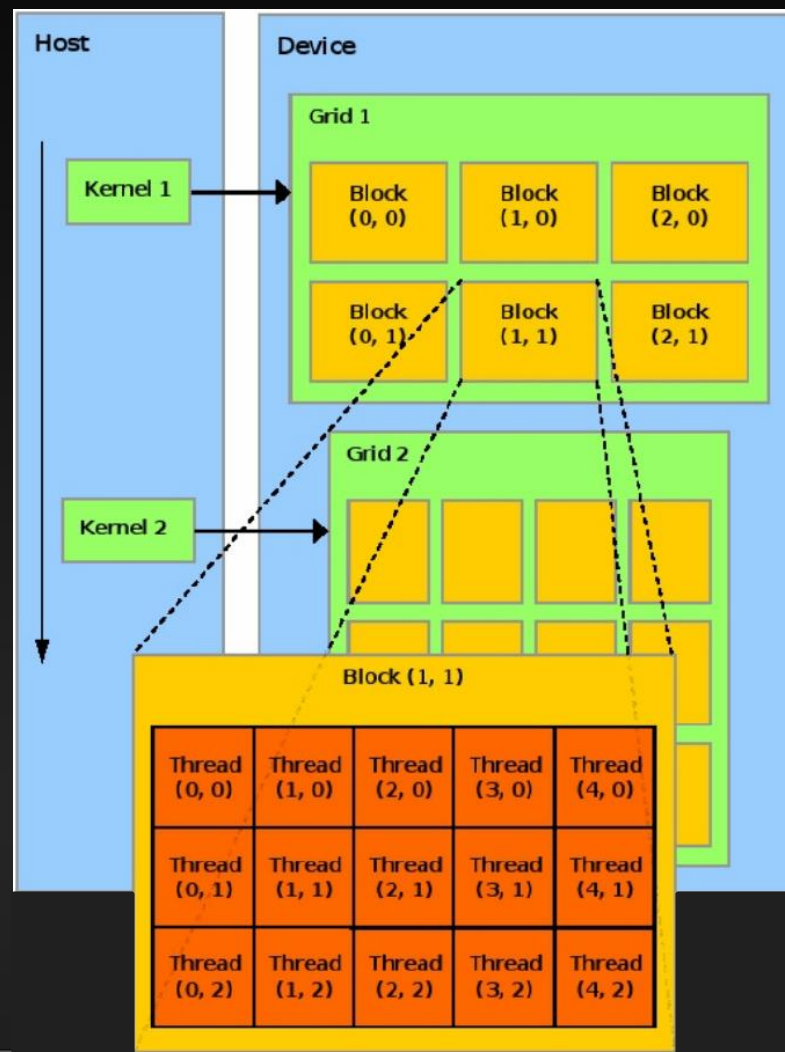
- GPU computing is about massive parallelism!
- We need an interesting example...
- SAXPY** stands for “Single-Precision A·X Plus Y”.



CUDA KERNEL EXECUTION



Grid → Block → Thread



CUDA code

```
saxyp_serial(N, 2.0, d_x, d_y);
```

```
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];
}
```

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];
}
```

Parallel C Code

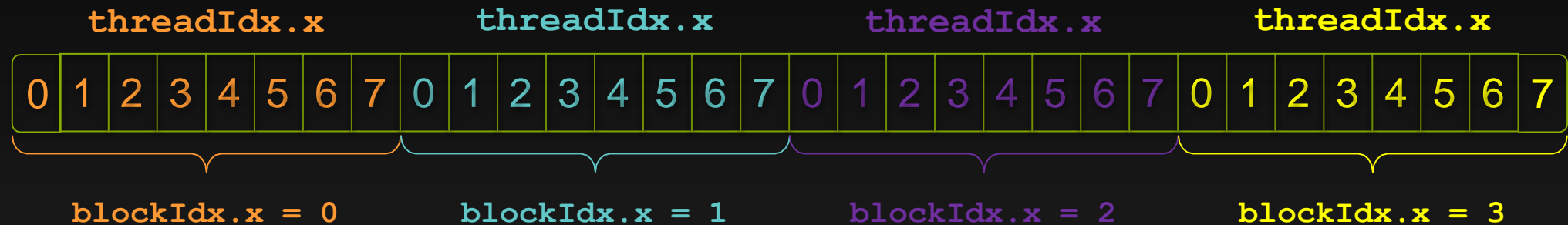
```
saxyp_parallel<<<n_blocks,n_threads>>>(N, 2.0, d_x, d_y);
```

N = n_blocks x n_threads

Indexing Arrays with Blocks and Threads



- No longer as simple as using `blockIdx.x` and `threadIdx.x`
 - Consider indexing an array with one element per thread (8 threads/block)



```
int index = threadIdx.x + blockIdx.x * M;  
          = 5 + 2 * 8;  
          = 21;
```


Why Bother with Blocks of Threads?

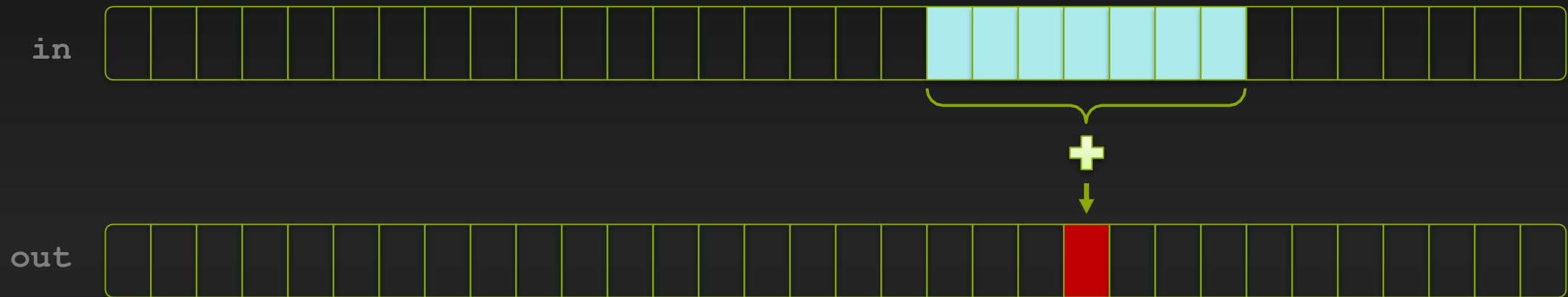


- **Blocks seem unnecessary**
 - They add a level of complexity
 - What do we gain?
- **Unlike parallel blocks, threads have mechanisms to:**
 - Communicate
 - Synchronize
- **See stencil computations for an example**

1D Stencil



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



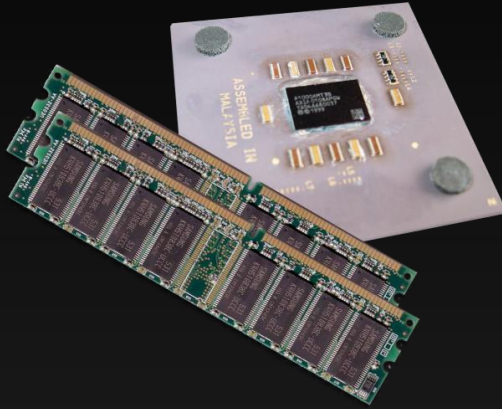
1D Stencil



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



Recap



Host



Device

- Launching parallel kernels on device __global__
 - Launch N copies of `add()` with `add<<<N/M,M>>> (...)` ;
 - Use `blockIdx.x` to access block index
 - Use `threadIdx.x` to access thread index within block

Handling Arbitrary Vector Sizes



- Typical problems are not friendly multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

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

- Update the kernel launch:

```
add<<<(N + M-1) / M, M>>>(d_a, d_b, d_c, N);
```

The background features a dark, textured surface with several large, curved, metallic-looking bands that sweep across the frame, creating a sense of depth and movement. The lighting highlights the metallic sheen and the fine grid-like texture of the surface.

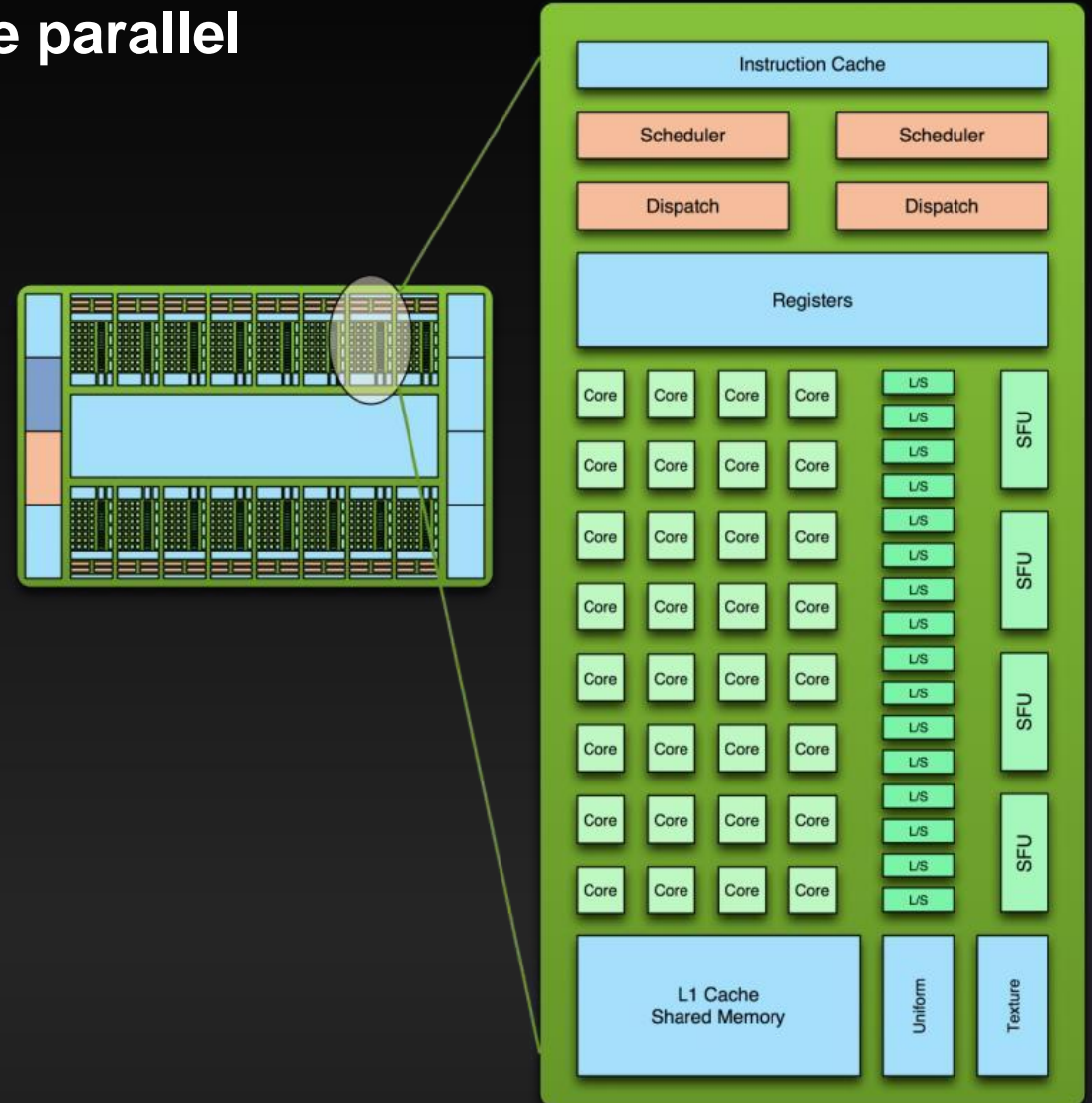
GPU Architectures and CUDA Optimization



20-Series Architecture (Fermi)



- 512 **Scalar Processor (SP) cores** execute parallel thread instructions
- 16 **Streaming Multiprocessors (SMs)** each contains
 - 32 scalar processors
 - 32 fp32 / int32 ops / clock,
 - 16 fp64 ops / clock
 - 4 Special Function Units (SFUs)
 - **Shared register file (128KB)**
 - 48 KB / 16 KB Shared memory
 - 16KB / 48 KB L1 data cache
 - **6 GB of DRAM**



Pascal/Volta cc6.0/7.0

- 64 SP units (“cores”)
- 32 DP units
- LD/ST units
- FP16 @ 2x SP rate
- cc7.0: TensorCore
- 4 warp schedulers
- Each warp scheduler is dual-issue capable
- P100: 50 SM’s, 16GB
- V100: 80 SM’s, 16/32GB



Thread Hierarchy and Execution Model



Software

Hardware



Thread

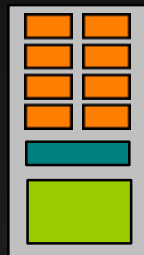


Scalar
Processor

Threads are executed by scalar processors



Thread
Block

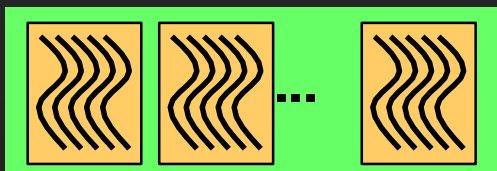


Multiprocessor

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)



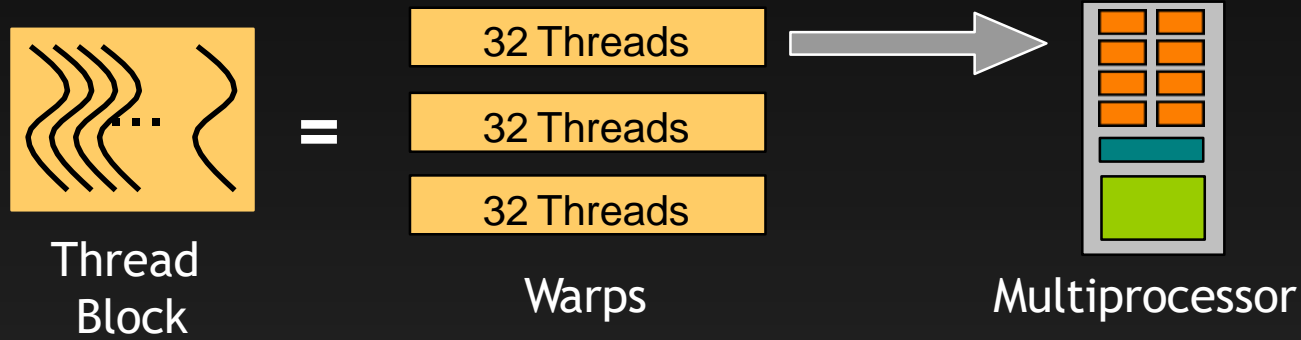
Grid



Device

A kernel is launched as a grid of thread blocks

Warps



A thread block consists of 32-thread warps

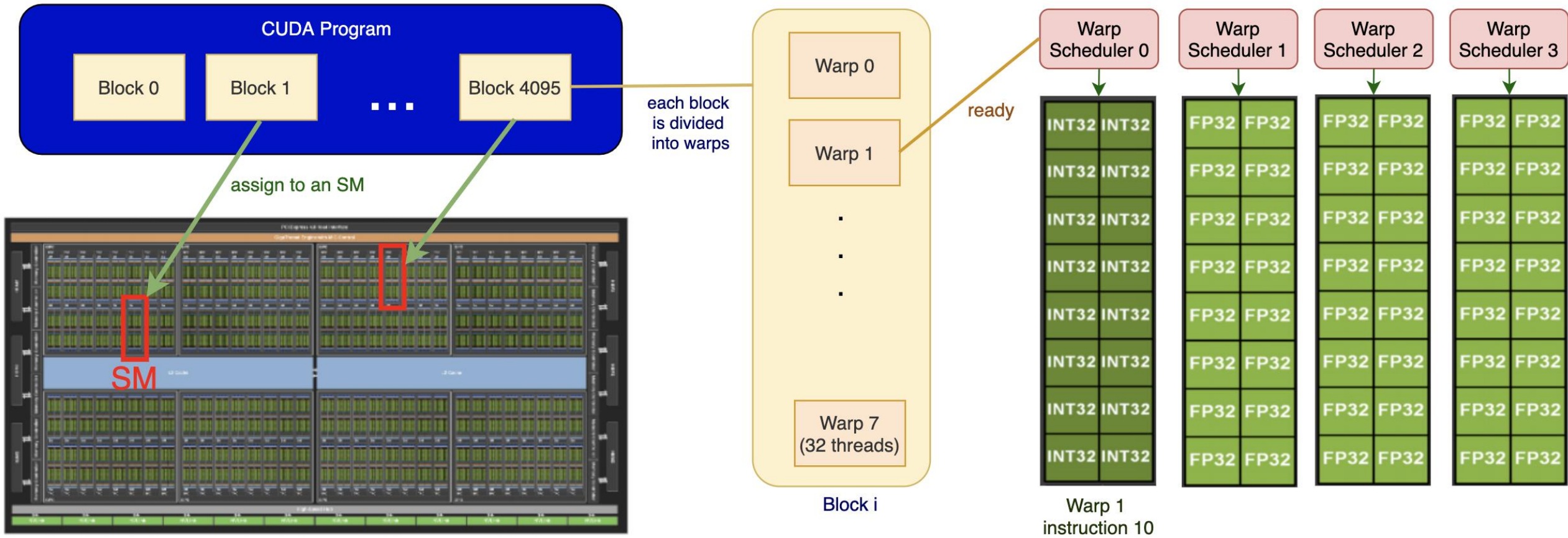
A warp is executed physically in parallel (SIMD) on a multiprocessor

Execution Model Ampere

This CUDA application uses 256 threads per block

each warp contains 32 threads

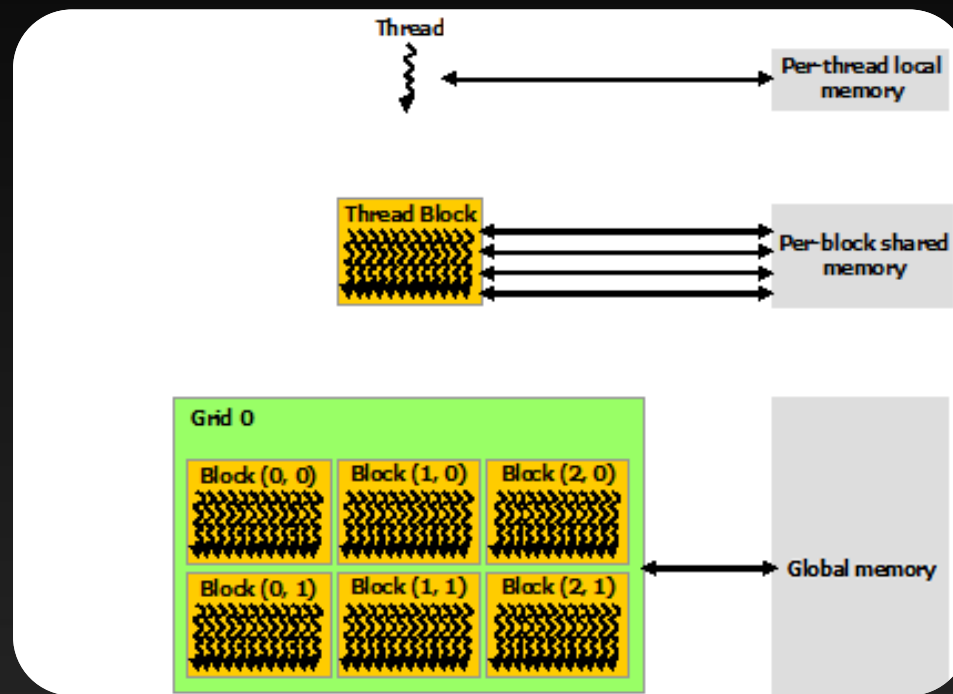
4 Warp schedulers per SM



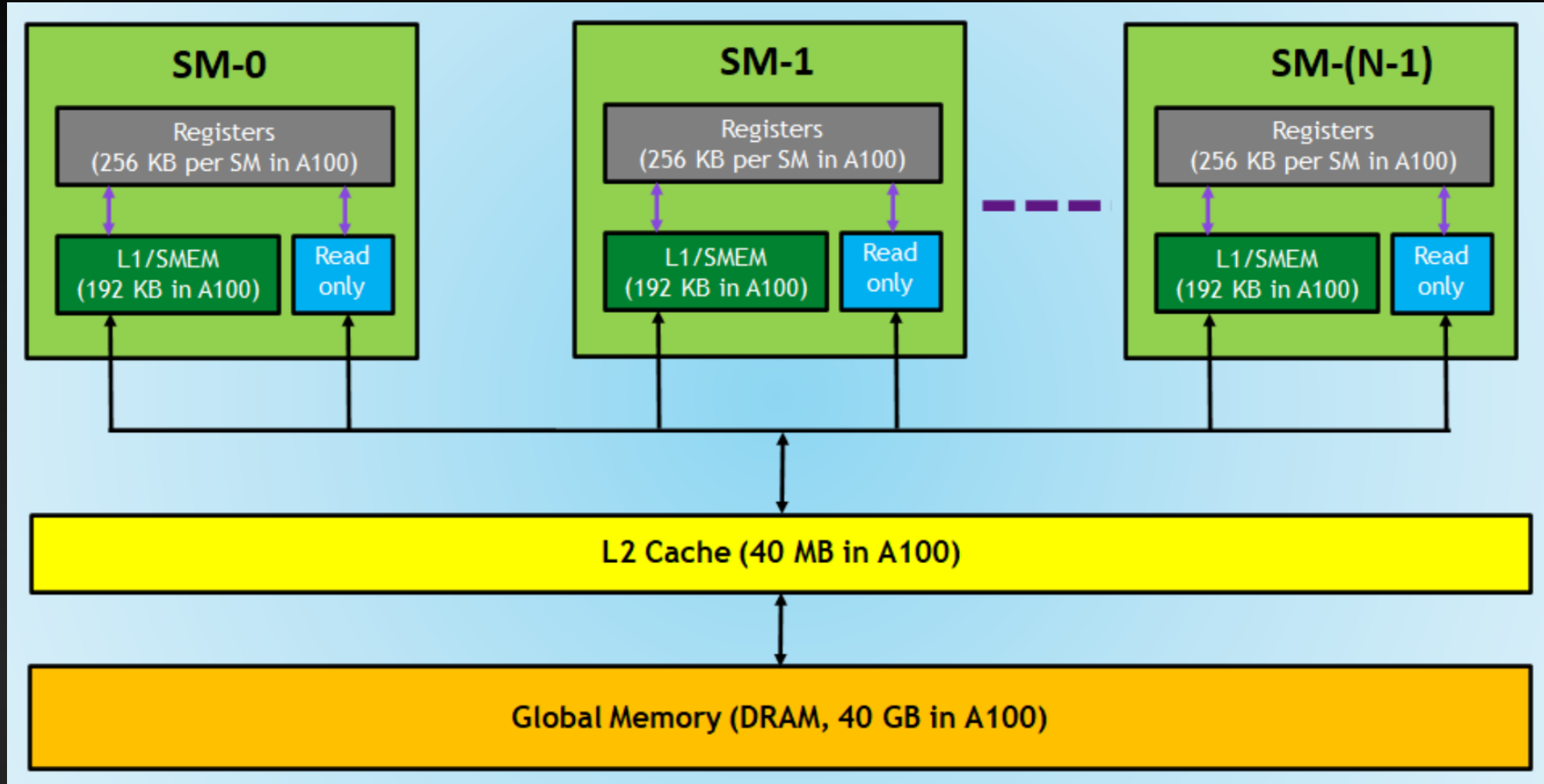
Memory Hierarchy



Memory model



Memory hierarchy in GPUs



Launch Configuration



Hiding Latency - Launch Configuration

- Key to understanding:
 - Instructions are issued in order
 - A thread stalls when one of the operands isn't ready:
 - Memory read by itself doesn't stall execution
 - Latency is hidden by switching threads
 - GMEM latency: ~400 cycles
- How many threads/threadblocks to launch?
- Conclusion:
 - Need enough threads to hide latency

GPU Latency Hiding



- In CUDA C source code:
- `int idx = threadIdx.x+blockDim.x*blockIdx.x;`
- `c[idx] = a[idx] * b[idx];`

- In machine code:
- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1

GPU Latency Hiding - inside the SM



- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1

clock cycles:

C₀ C₁ C₂ C₃ C₄ C₅ C₆ C₇ C₈ C₉ C₁₀ C₁₁ C₁₂ C₁₃ C₁₄ C₁₅ C₁₆ C₁₇ C₁₈ C₁₉ ...

warps

W0: I0

W1:

W2:

W3:

W4:

W5:

W6:

W7:

W8:

W9:

...

GPU Latency Hiding - inside the SM



- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1

clock cycles:

C₀ C₁ C₂ C₃ C₄ C₅ C₆ C₇ C₈ C₉ C₁₀ C₁₁ C₁₂ C₁₃ C₁₄ C₁₅ C₁₆ C₁₇ C₁₈ C₁₉ ...

warps

W0: I0 I1

W1:

W2:

W3:

W4:

W5:

W6:

W7:

W8:

W9:

...

GPU Latency Hiding - inside the SM



- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1

clock cycles:

C₀ C₁ C₂ C₃ C₄ C₅ C₆ C₇ C₈ C₉ C₁₀ C₁₁ C₁₂ C₁₃ C₁₄ C₁₅ C₁₆ C₁₇ C₁₈ C₁₉ ...

warps

W0: I0 I1

W1:

W2:

W3:

W4:

W5:

W6:

W7:

W8:

W9:

...

GPU Latency Hiding - inside the SM



- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1

clock cycles:

C₀ C₁ C₂ C₃ C₄ C₅ C₆ C₇ C₈ C₉ C₁₀ C₁₁ C₁₂ C₁₃ C₁₄ C₁₅ C₁₆ C₁₇ C₁₈ C₁₉ ...

warps

W0:

I0 I1

W1:

I0

W2:

W3:

W4:

W5:

W6:

W7:

W8:

W9:

...

GPU Latency Hiding - inside the SM

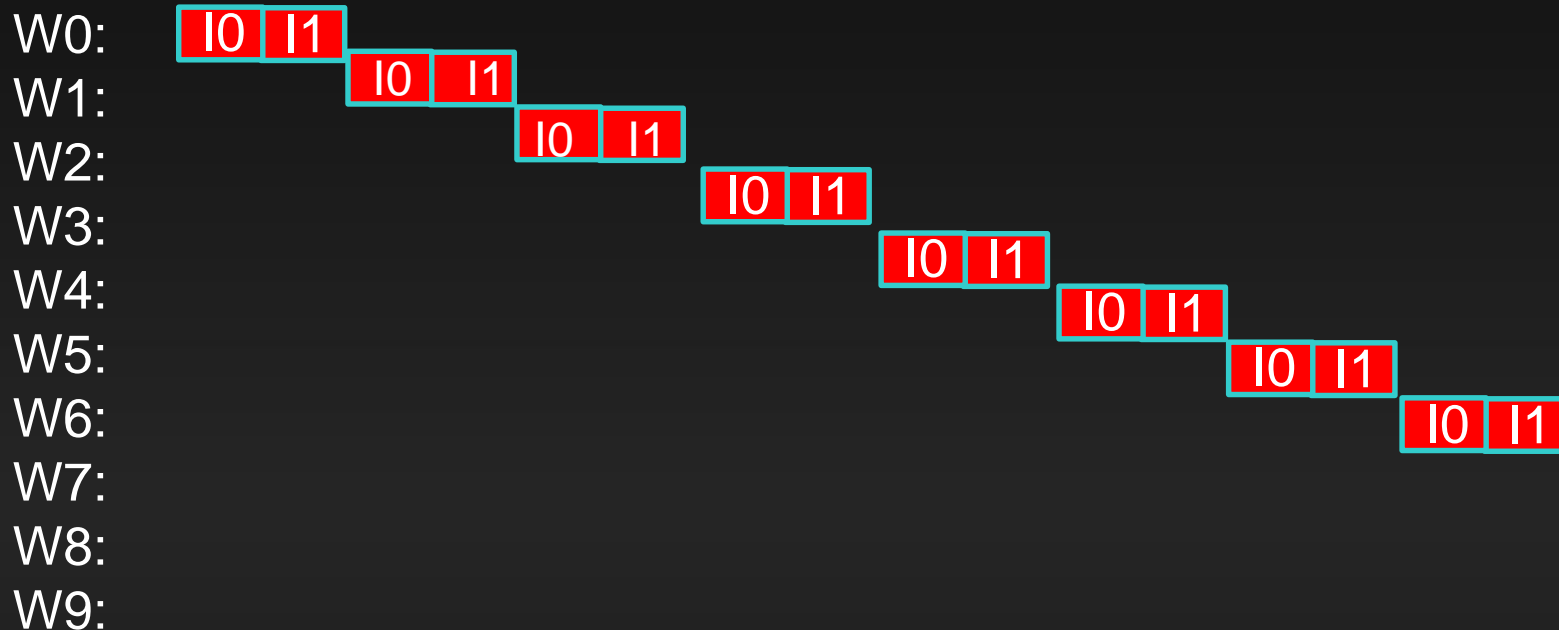


- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1

clock cycles:

C₀ C₁ C₂ C₃ C₄ C₅ C₆ C₇ C₈ C₉ C₁₀ C₁₁ C₁₂ C₁₃ C₁₄ C₁₅ C₁₆ C₁₇ C₁₈ C₁₉ ...

warps

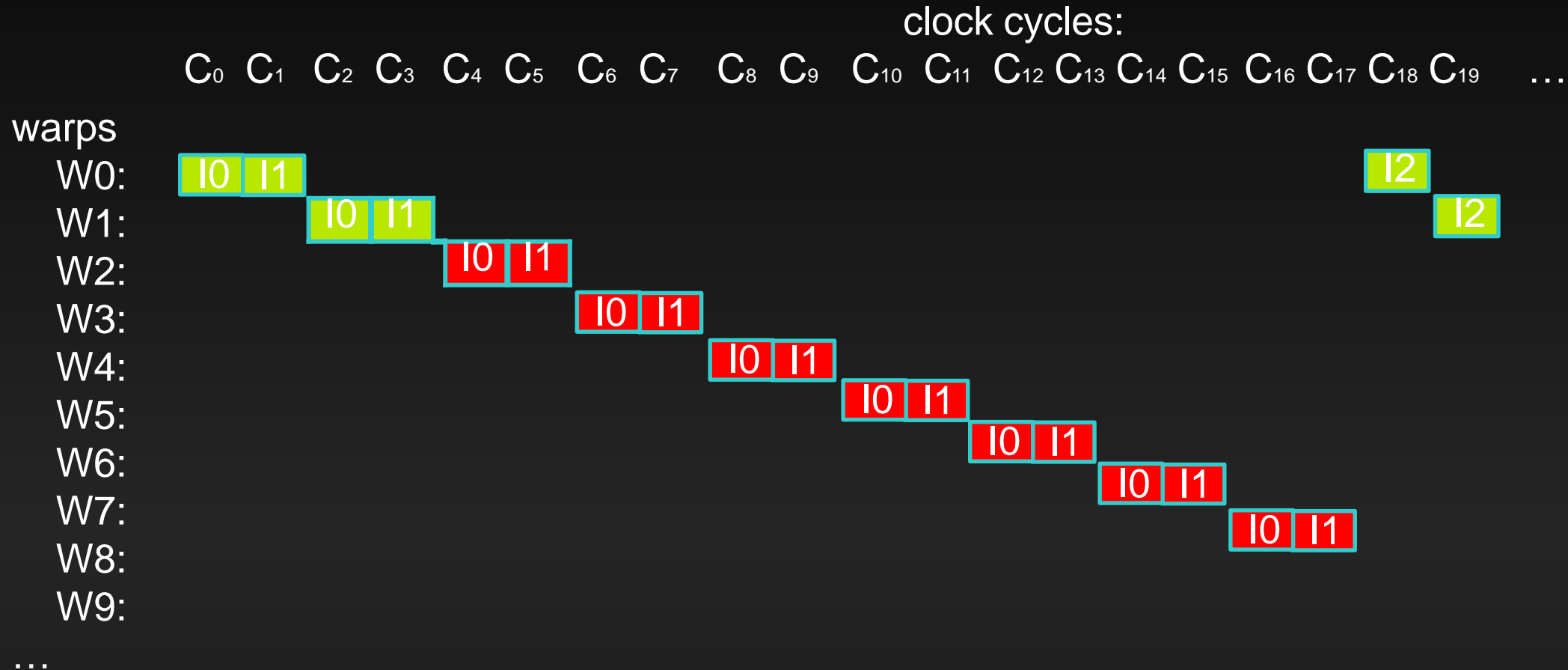


...

GPU Latency Hiding - inside the SM



- I0: LD R0, a[idx];
- I1: LD R1, b[idx];
- I2: MPY R2,R0,R1



Launch Configuration: Summary



- Need enough total threads to keep GPU busy
 - Typically, you'd like **512+ threads** per SM (aim for 2048 - maximum "occupancy")
 - More if processing one fp32 element per thread
 - Of course, exceptions exist
- Threadblock configuration
 - Threads per block should be a **multiple of warp size (32)**
 - SM can concurrently execute **up to 16** thread blocks
 - Really small thread blocks prevent achieving good occupancy
 - Really large thread blocks are less flexible
 - Generally, use **128-256 threads/block**, but use whatever is best for the application
- For more details:
 - Vasily Volkov's GTC2010 talk "Better Performance at Lower Occupancy"
(http://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf)



EFFICIENT GEMM
IMPLEMENTATIONS

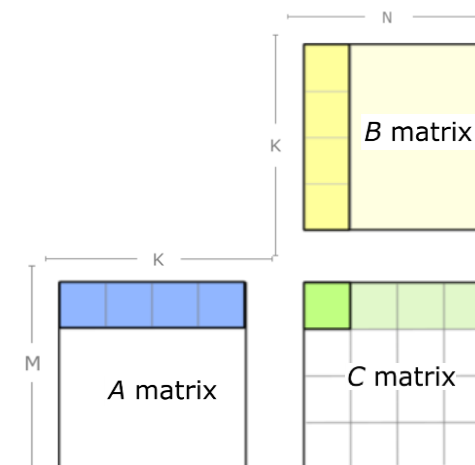
GENERAL MATRIX PRODUCT

Basic definition

General matrix product

$$C = \alpha \text{op}(A) * \text{op}(B) + \beta C$$

C is M -by- N , $\text{op}(A)$ is M -by- K , $\text{op}(B)$ is K -by- N



Compute independent dot products

```
// Independent dot products
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        for (int k = 0; k < K; ++k)
            C[i][j] += A[i][k] * B[k][j];
```

Inefficient due to large working sets to hold parts of A and B

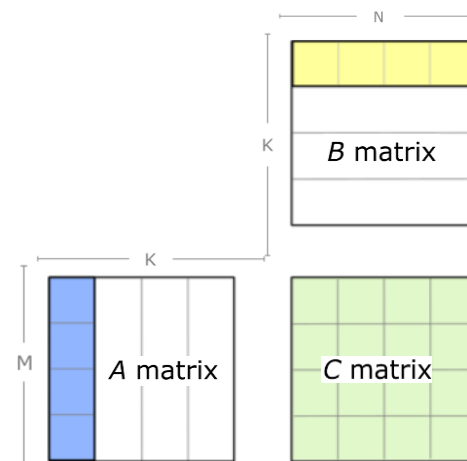
GENERAL MATRIX PRODUCT

Accumulated outer products

General matrix product

$$C = \alpha \text{op}(A) * \text{op}(B) + B C$$

C is M -by- N , $\text{op}(A)$ is M -by- K , $\text{op}(B)$ is K -by- N



~~Compute independent dot products~~

```
// Independent dot products
for (int i = 0; i < M; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < K; ++k)
      C[i][j] += A[i][k] * B[k][j];
```

Permute loop nests

```
// Accumulated outer products
for (int k = 0; k < K; ++k)
  for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

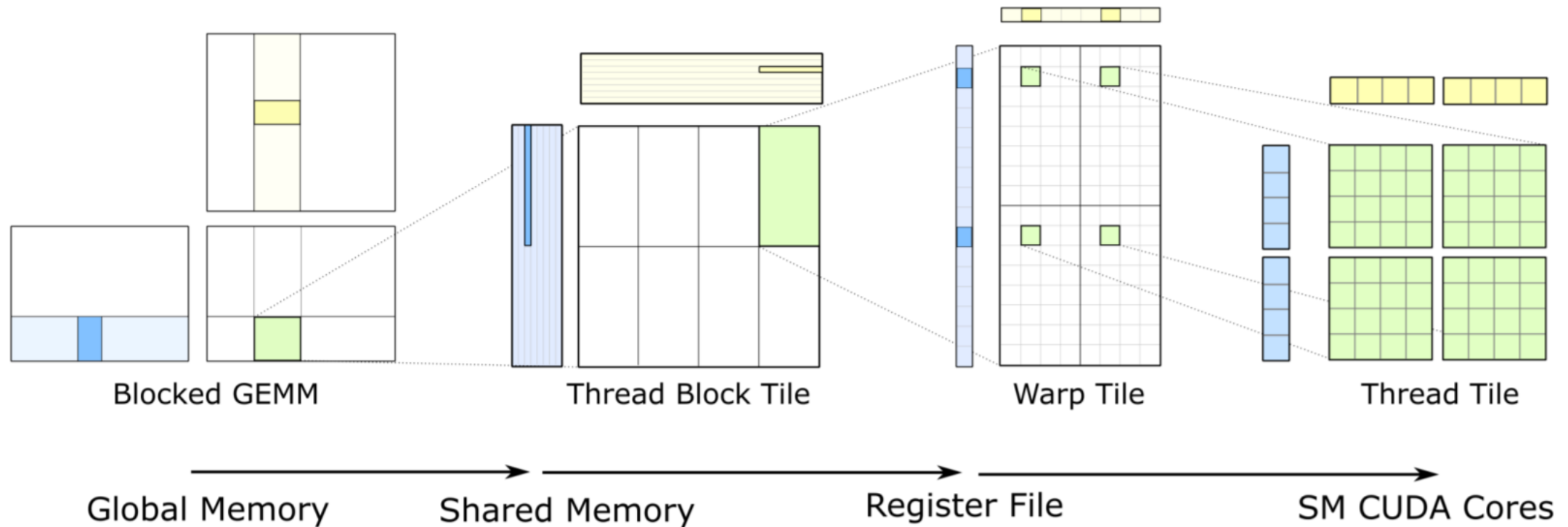
Load elements of A and B exactly once

CUTLASS

CUDA TEMPLATE LIBRARY FOR DENSE LINEAR ALGEBRA AT ALL LEVELS AND SCALE

COMPLETE GEMM HIERARCHY

Data reuse at each level of the memory hierarchy



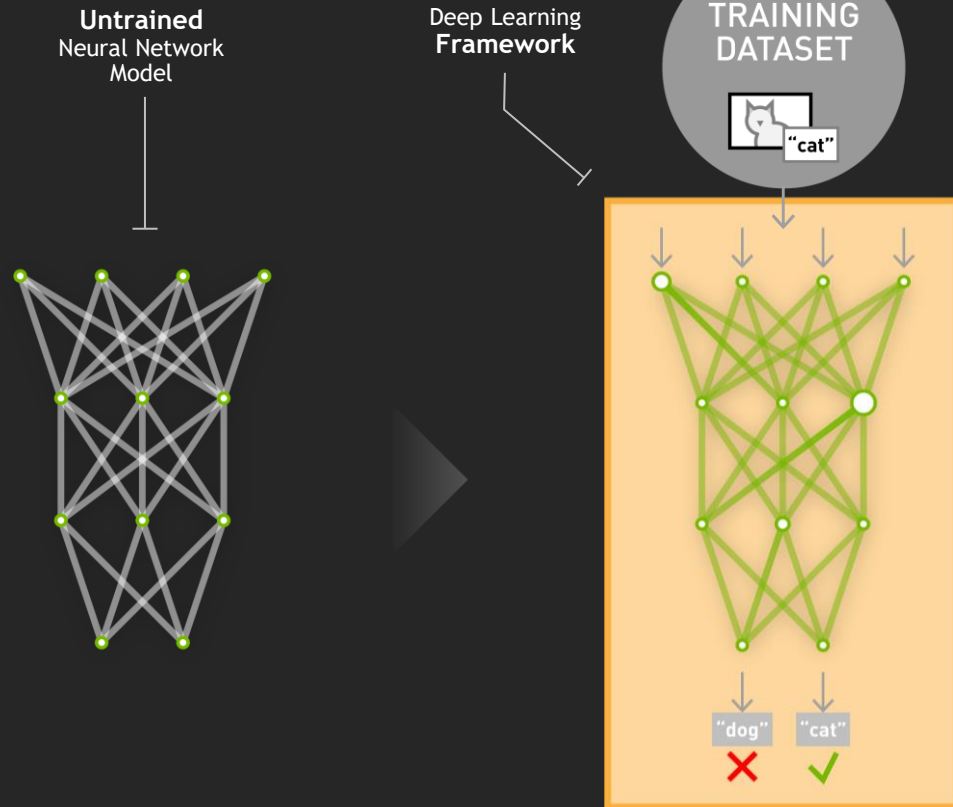


**ACCELERATING
TRAINING
AND
INFERRNCING**

DEEP LEARNING APPLICATION DEVELOPMENT

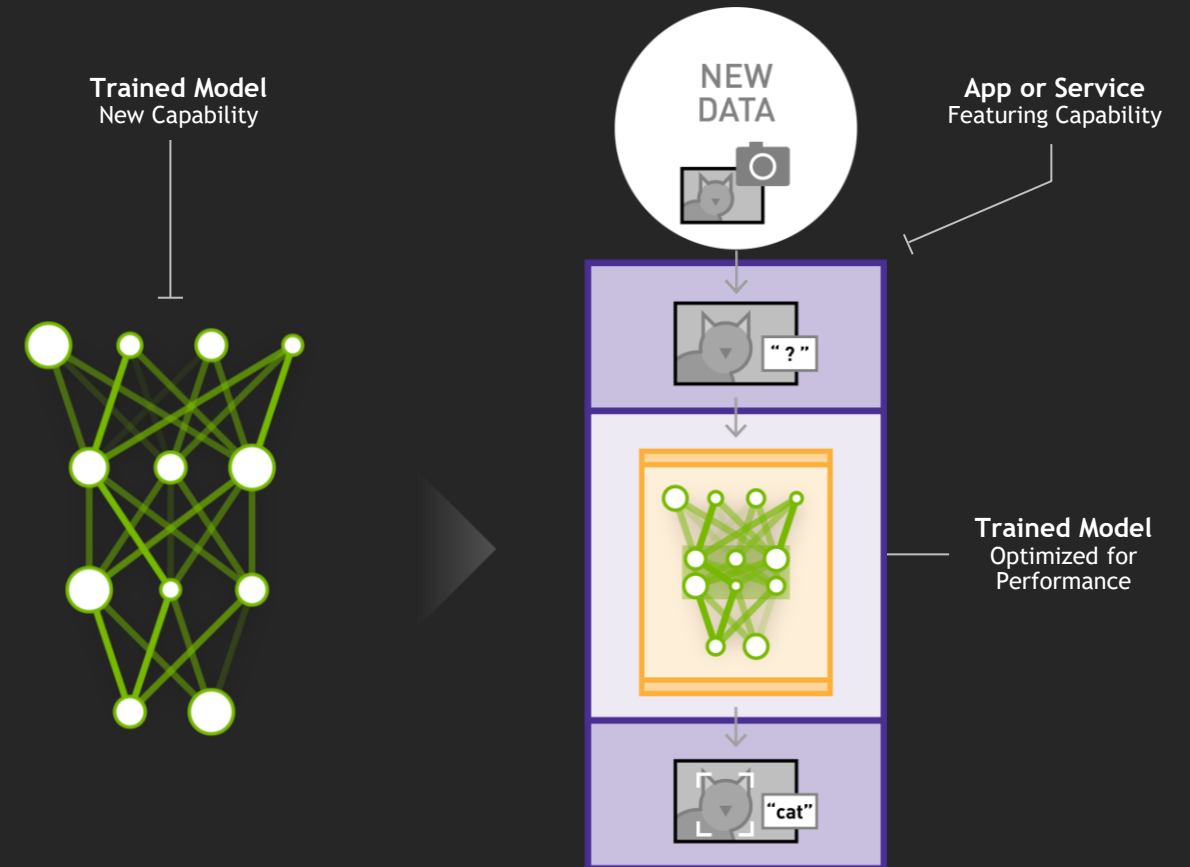
TRAINING

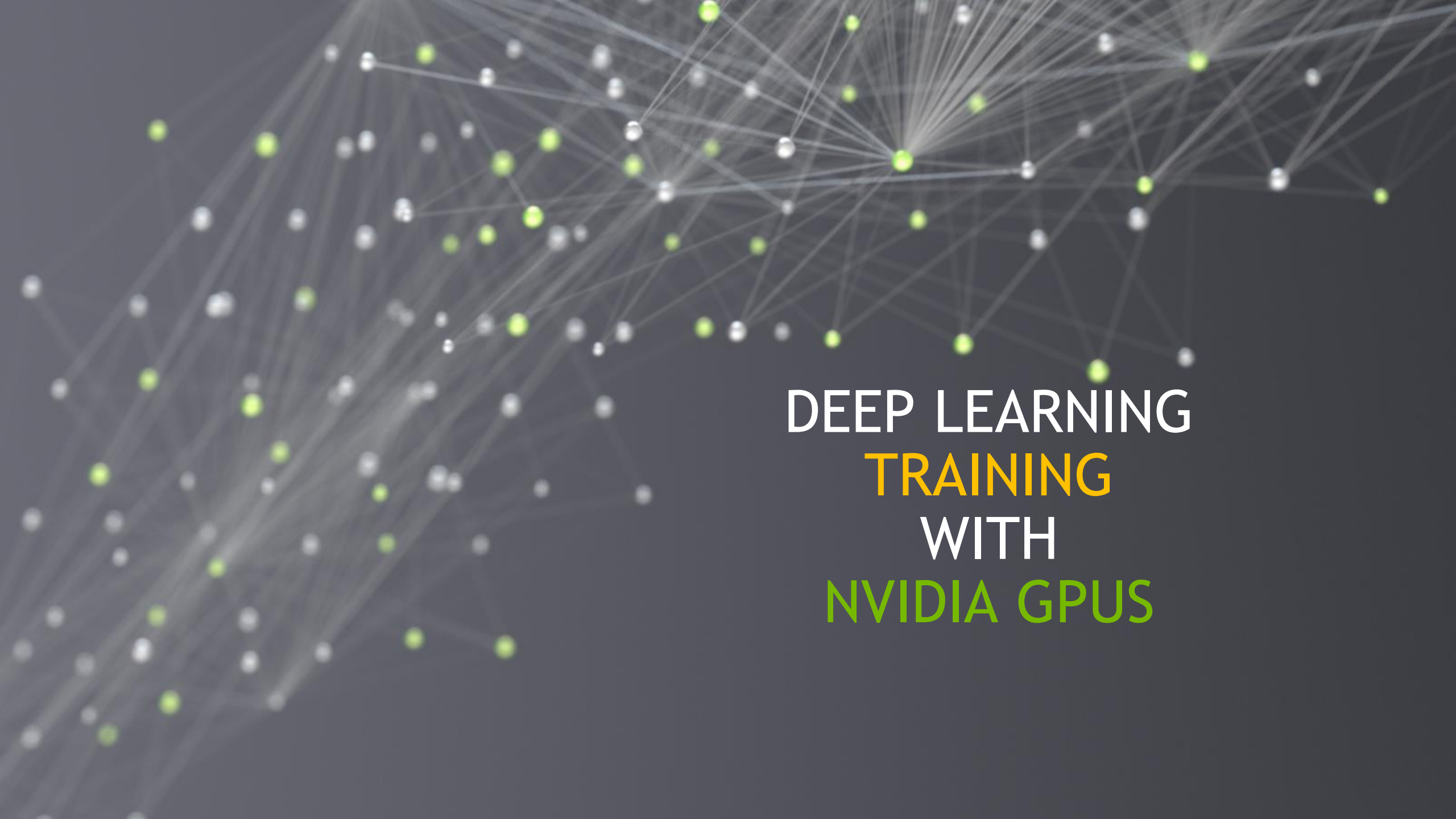
Learning a new capability
from existing data



INFERENCE

Applying this capability
to new data





DEEP LEARNING
TRAINING
WITH
NVIDIA GPUS



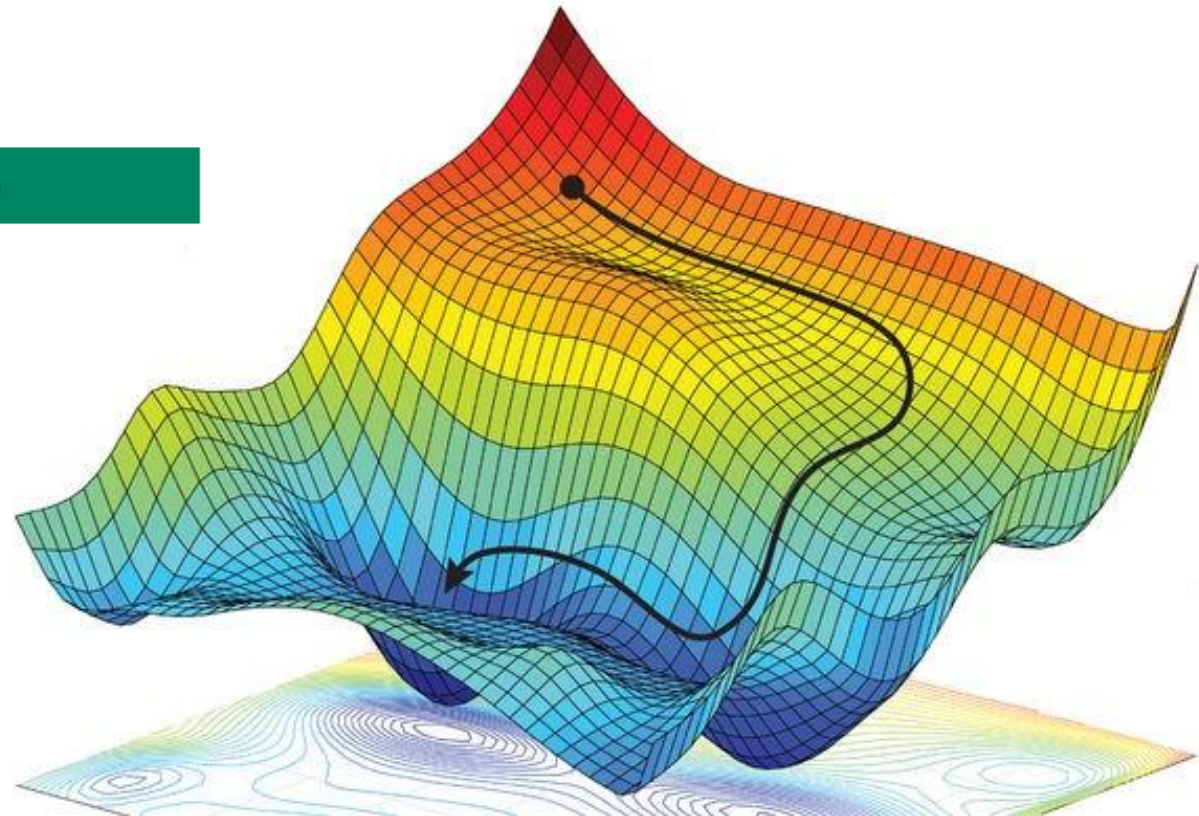
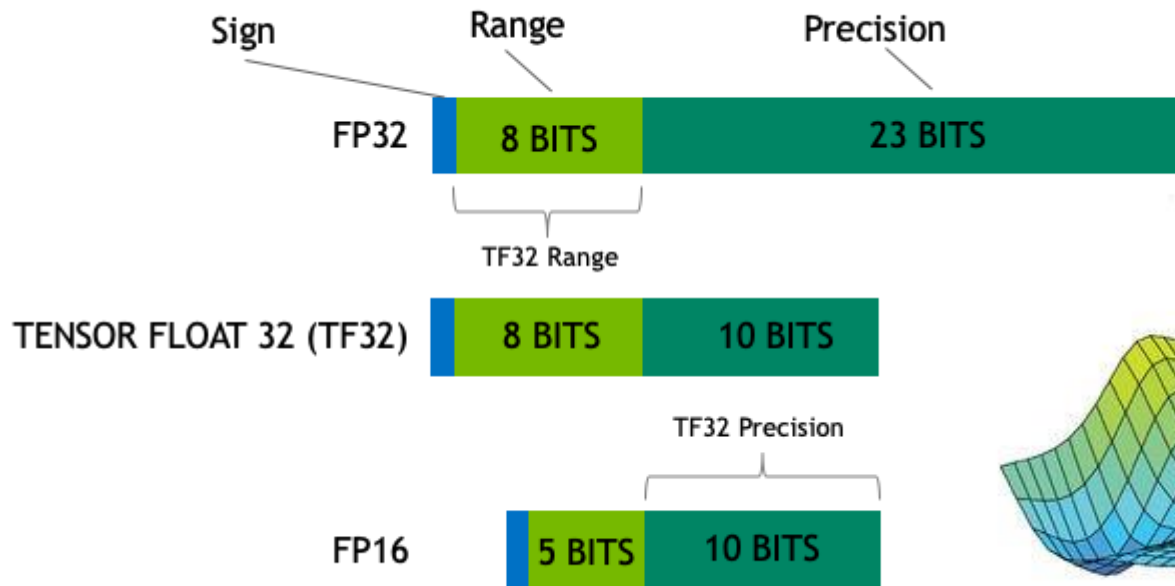
AMP
AUTOMATIC MIXED
PRECISION

THE IMPORTANCE OF FP32

$$1230000 = 1.23 \times 10^6$$

Mantissa Exponent

	Dynamic Range	Min Positive Value
FP32	$-3.4 \times 10^{38} \sim +3.4 \times 10^{38}$	1.4×10^{-45}
FP16	$-65504 \sim +65504$	5.96×10^{-8}
INT8	$-128 \sim +127$	1

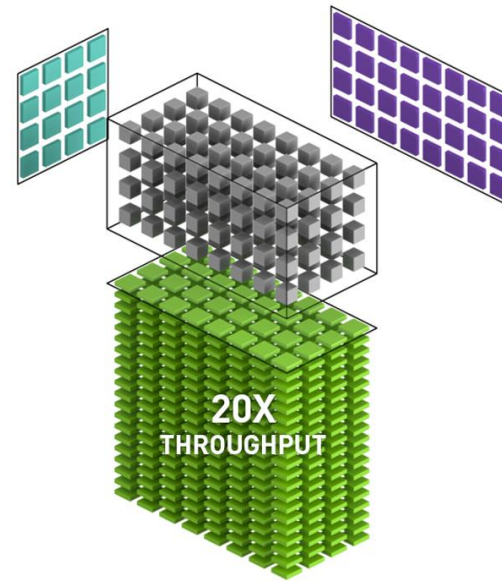
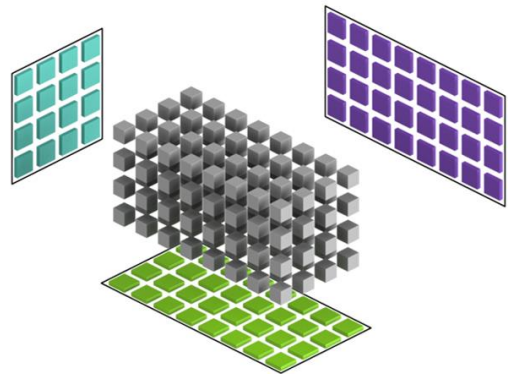


NEW TF32 TENSOR CORES ON A100

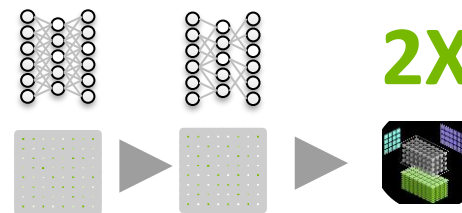
20X Higher FLOPS for AI, Zero Code Change

NVIDIA V100 FP32

NVIDIA A100 Tensor Core TF32 with Sparsity



TENSOR FLOAT 32 (TF32)

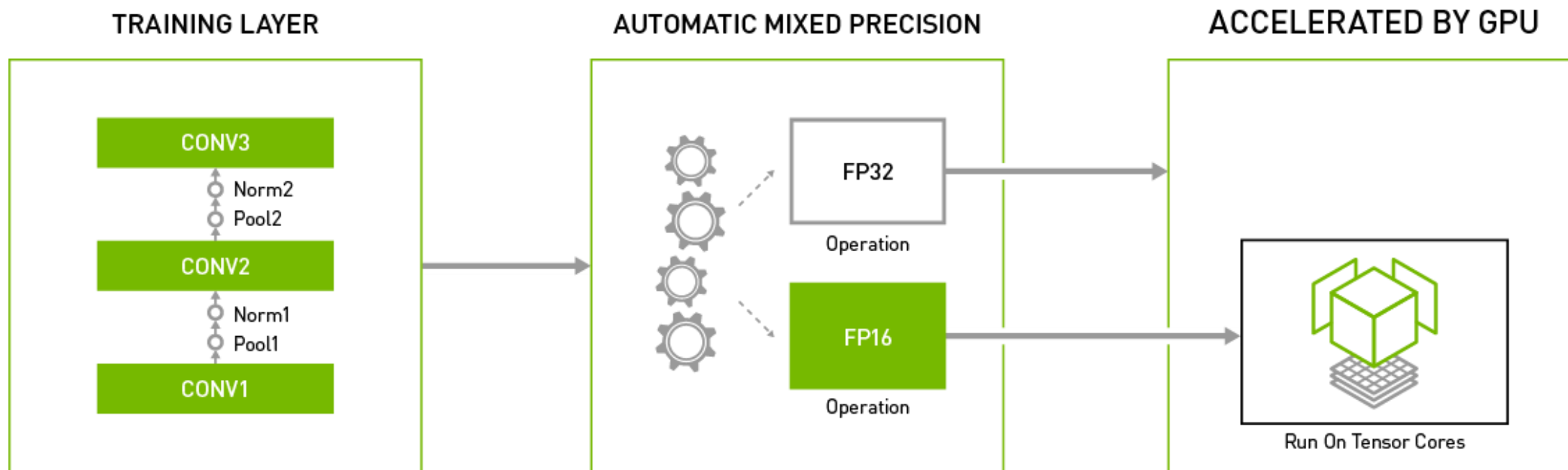


Works like FP32 for AI with Range of FP32 and Precision of FP16

DNN Sparsity Matrix

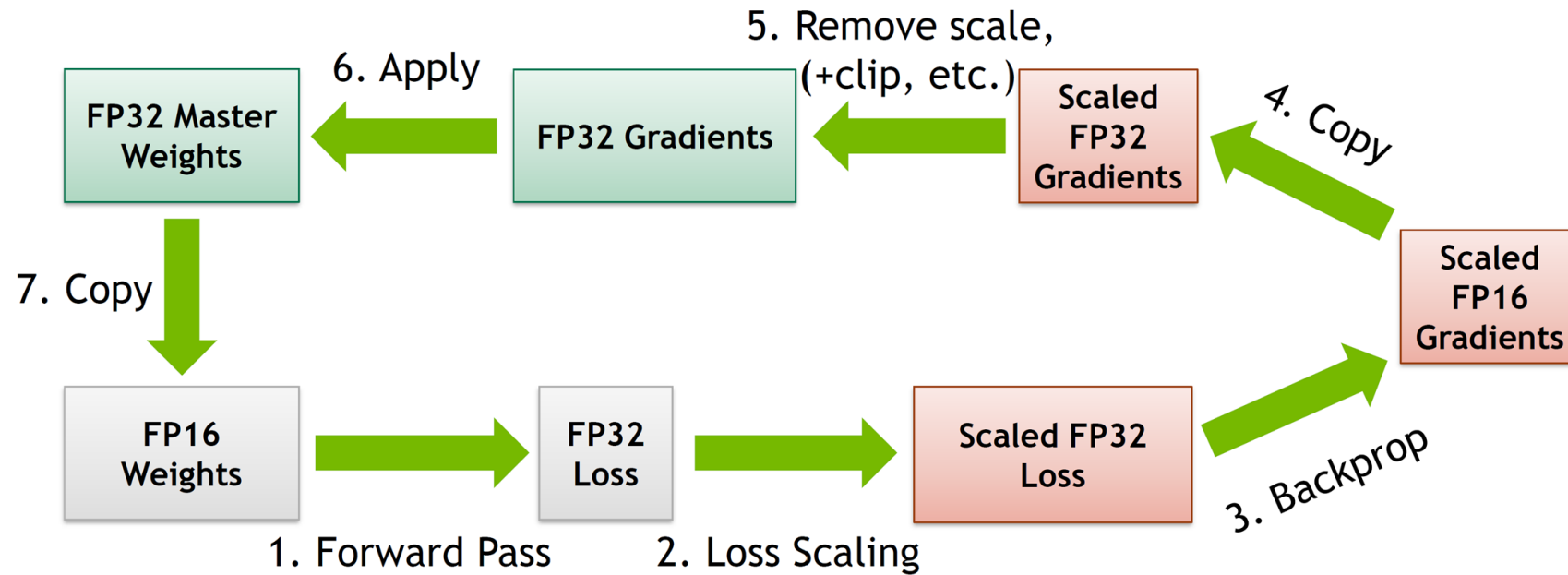
AMP

Utilizing tensor cores with 3 lines of code



AMP

Automatic Mixed Precision



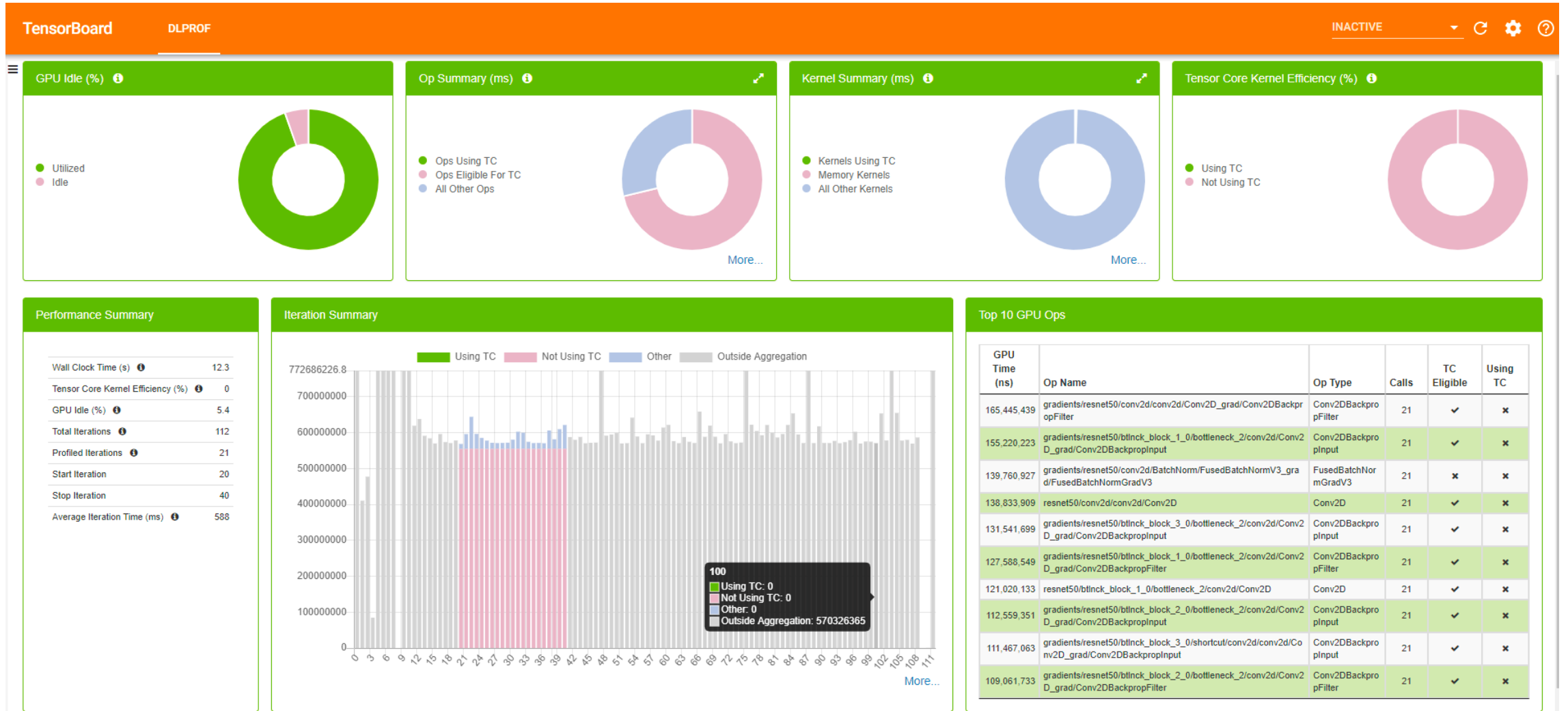
ENABLING AUTOMATIC MIXED PRECISION

Add Just A Few Lines of Code

- PyTorch
 - Two steps: initialization and wrapping backpropagation

```
from apex import amp
model = ...
optimizer = SomeOptimizer(model.parameters(), ...)
# ...
model, optimizer = amp.initialize(model, optimizer, opt_level="O1")
# ...
for train_loop():
    loss = loss_fn(model(x), y)
    with amp.scale_loss(loss, optimizer) as scaled_loss:
        scaled_loss.backward()
    # Can manipulate the .grads if you'd like
    optimizer.step()
```

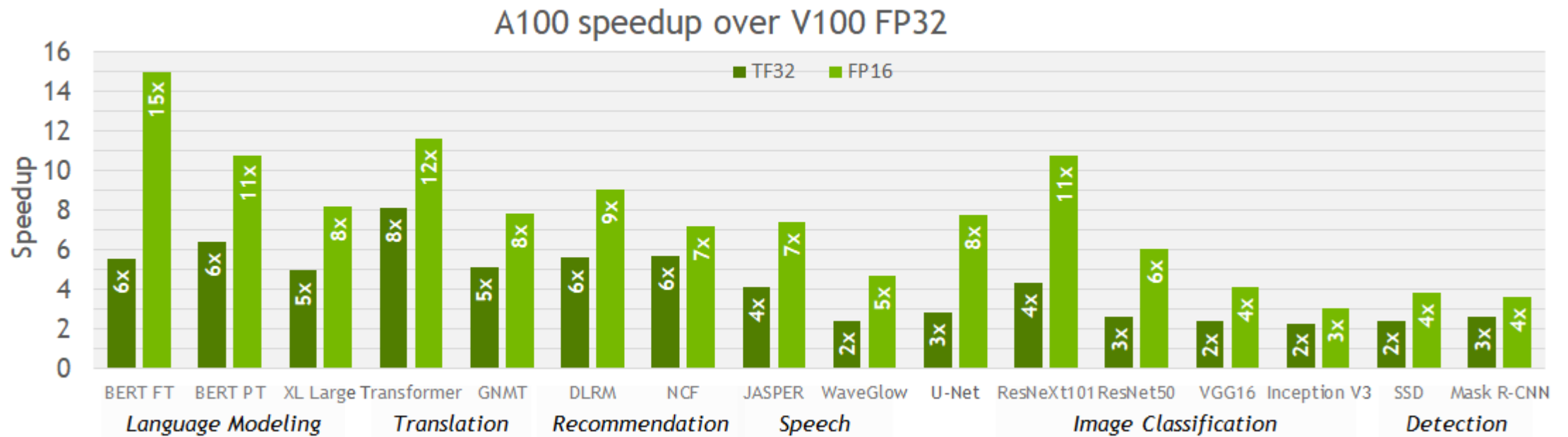
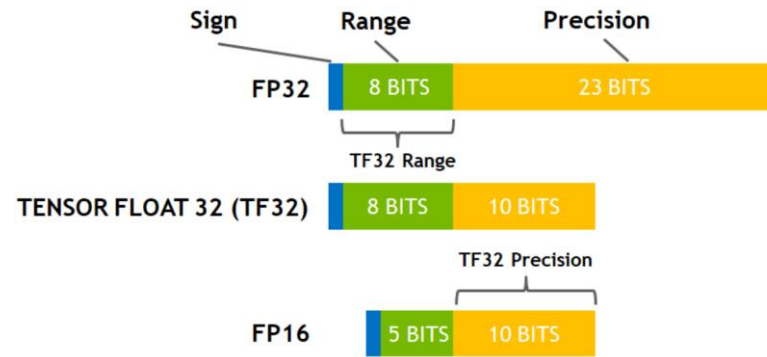

NVIDIA DLPROF



MULTIPLY-ADD OPERATIONS PER CLOCK PER SM

NVIDIA Architecture	CUDA Cores				Tensor Cores					
	FP64	FP32	FP16	INT8	FP64	TF32	FP16	INT8	INT4	INT1
Volta	32	64	128	256			512			
Turing	2	64	128	256			512	1024	2048	8192
Ampere (A100)	32	64	256	256	64	512	1024	2048	4096	16384
Ampere, sparse						1024	2048	4096	8192	

TF32 NUMERICAL REPRESENTATIONS

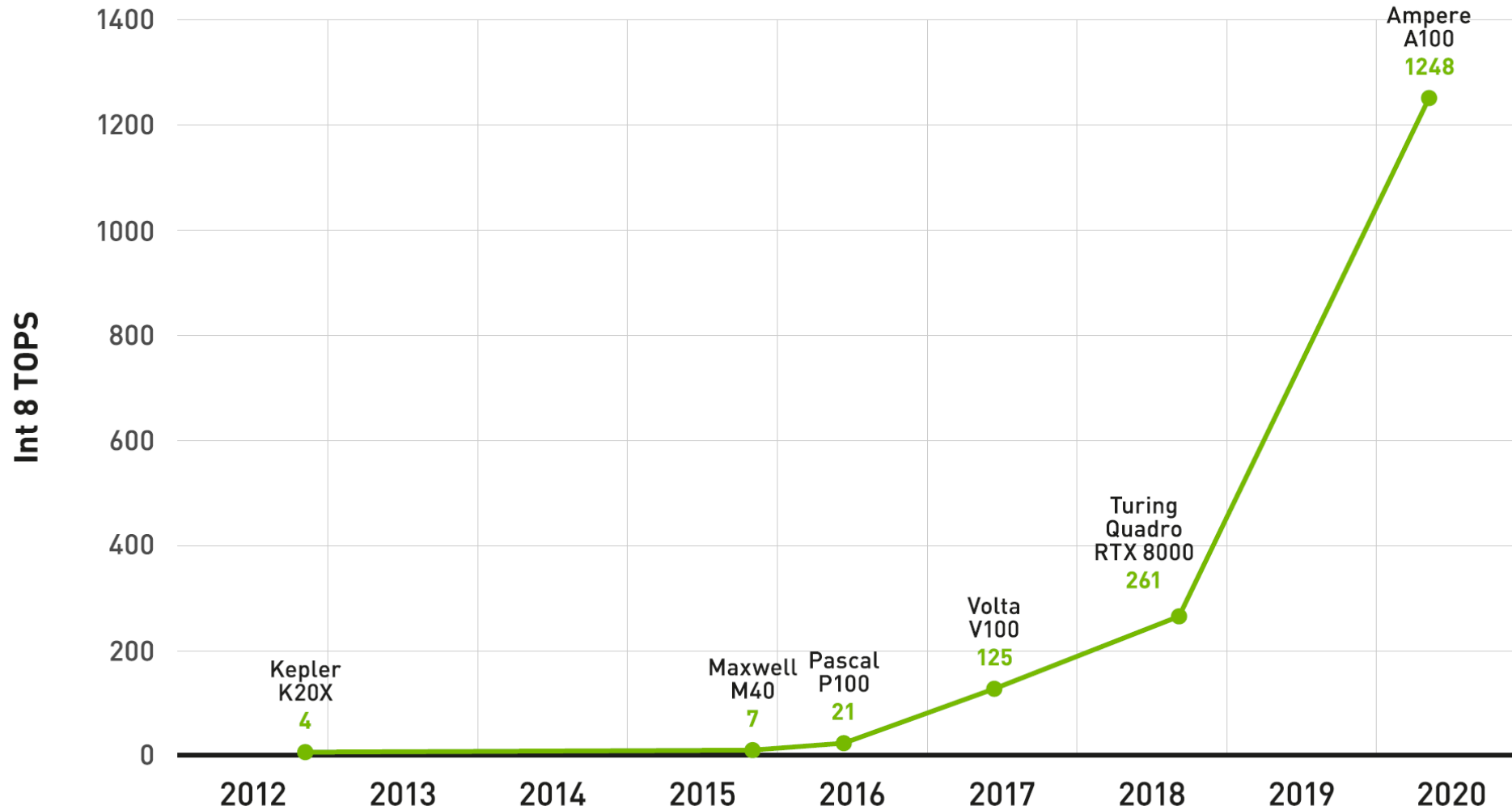




DEEP LEARNING
INFERENCE
WITH
NVIDIA GPUS

HUANG'S LAW

Single-Chip Inference Performance - 317X in 8 years

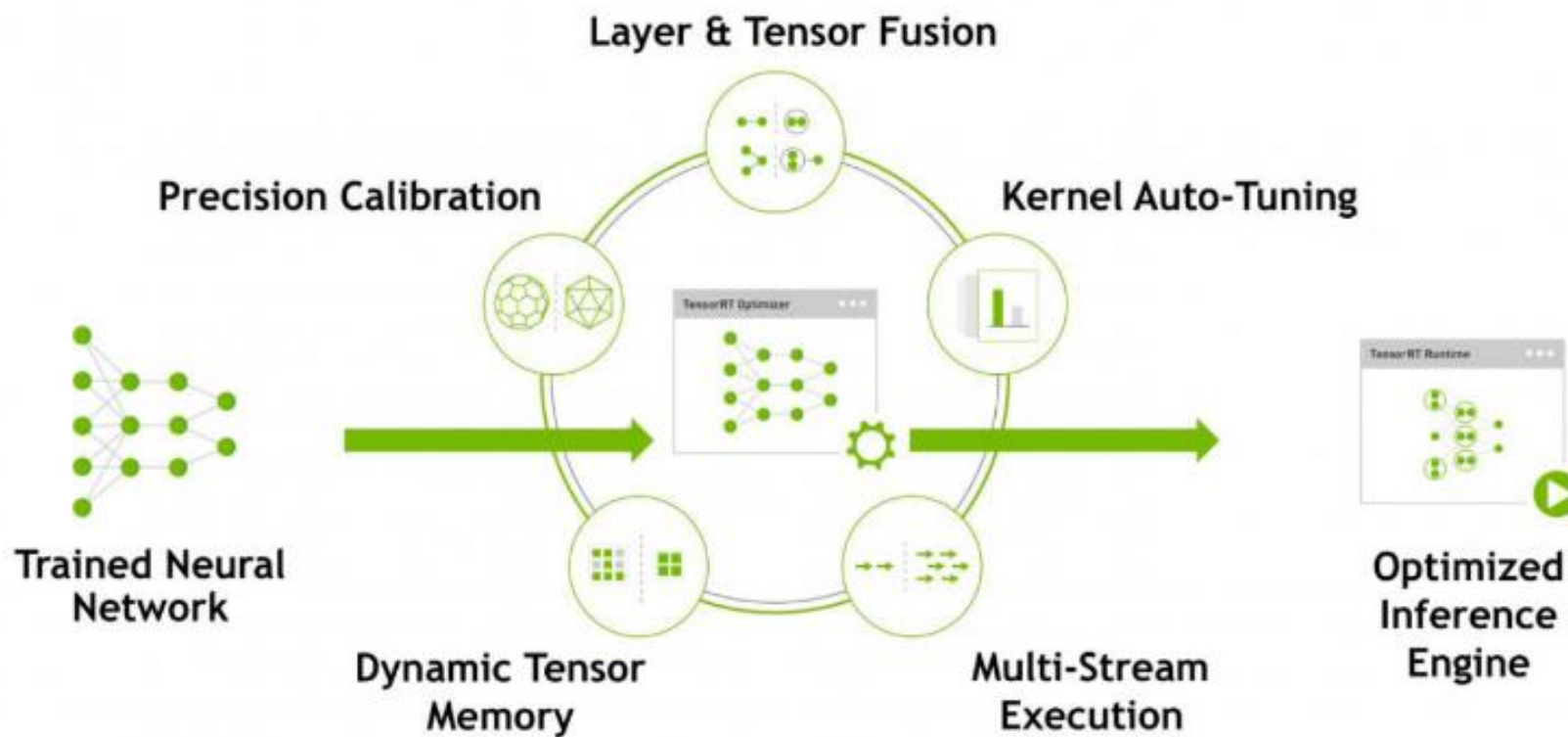




TENSOR-RT

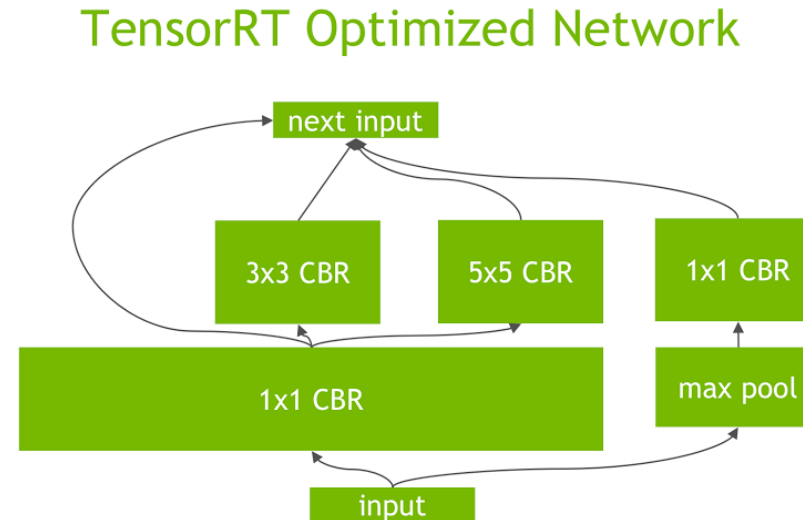
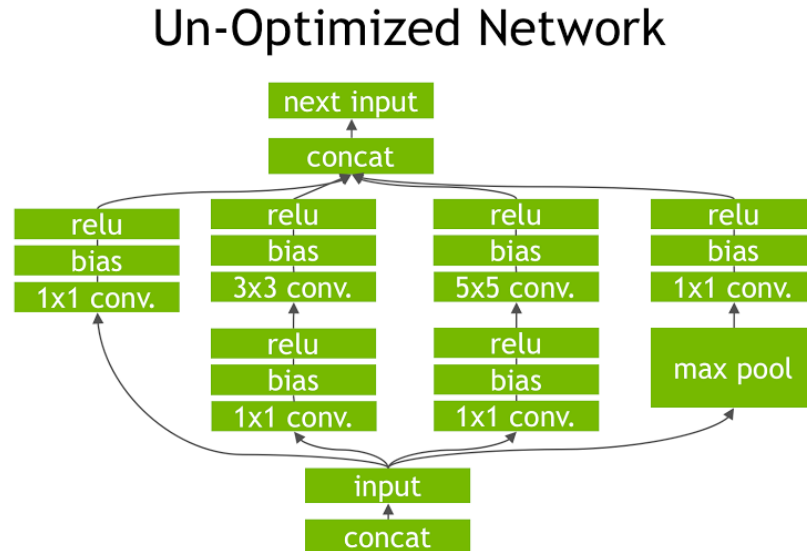
TENSORRT

Optimizations



KERNEL FUSION

- Improve GPU utilization - less kernel launch overhead, better memory usage and bandwidth
- Vertical fusion = Combine sequential kernel calls
- Horizontal fusion = Combine same kernels that have common input but different weights



KERNEL AUTO-TUNING

- There are multiple low-level algorithms/implementations for common operations
- TensorRT selects the optimal kernels based on your parameters e.g. batch size, filter-size, input data size
- TensorRT selects the optimal kernel based on your target platform

CONVOLUTION ALGORITHMS

128x128x128x128 convolution, FP32, NCHW, Quadro GV100

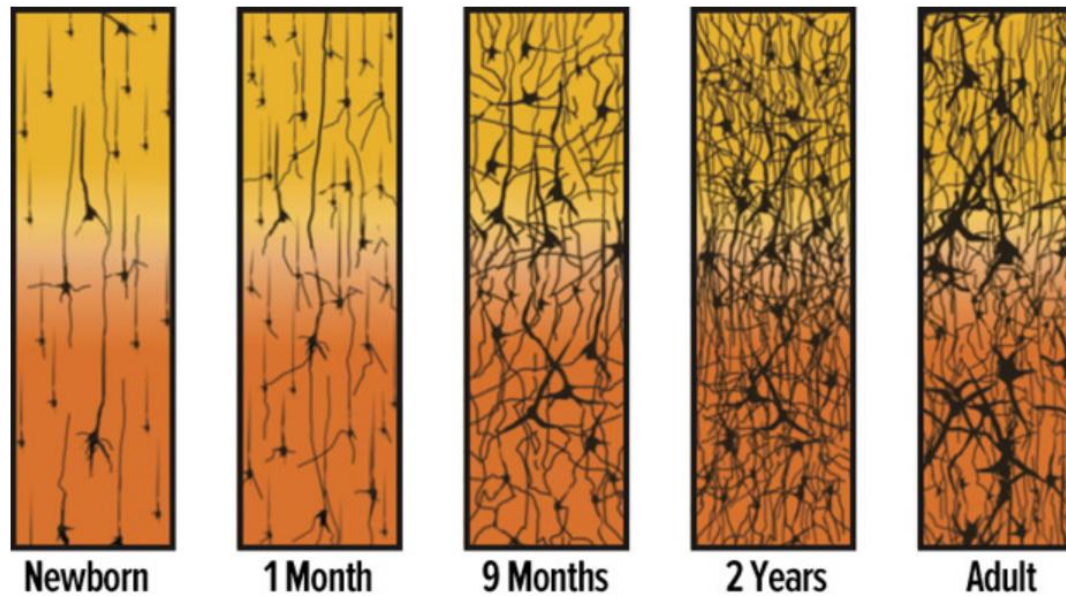
CUDNN_CONVOLUTION_FWD_ALGO_ ...	3 x 3		11 x 11	
	Performance	Workspace	Performance	Workspace
CUDNN_CONVOLUTION_FWD_ALGO_GEMM	0.76 ms	72 MB	8.47 ms	968 MB
CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM	0.62 ms	None	6.82 ms	None
CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM	0.47 ms	0.01 MB	6.58 ms	0.01 MB
CUDNN_CONVOLUTION_FWD_ALGO_FFT	45.3 ms	8322 MB	44.7 ms	8328 MB
CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING	3.69 ms	70 MB	5.13 ms	70 MB
CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD	0.26 ms	1.56 MB	Unsupported	
CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED	2.73 ms	578 MB	Unsupported	



PRUNING

SPARSE NEURAL NETWORKS

Synapse density over time



Synapse Density Over Time FIGURE 3

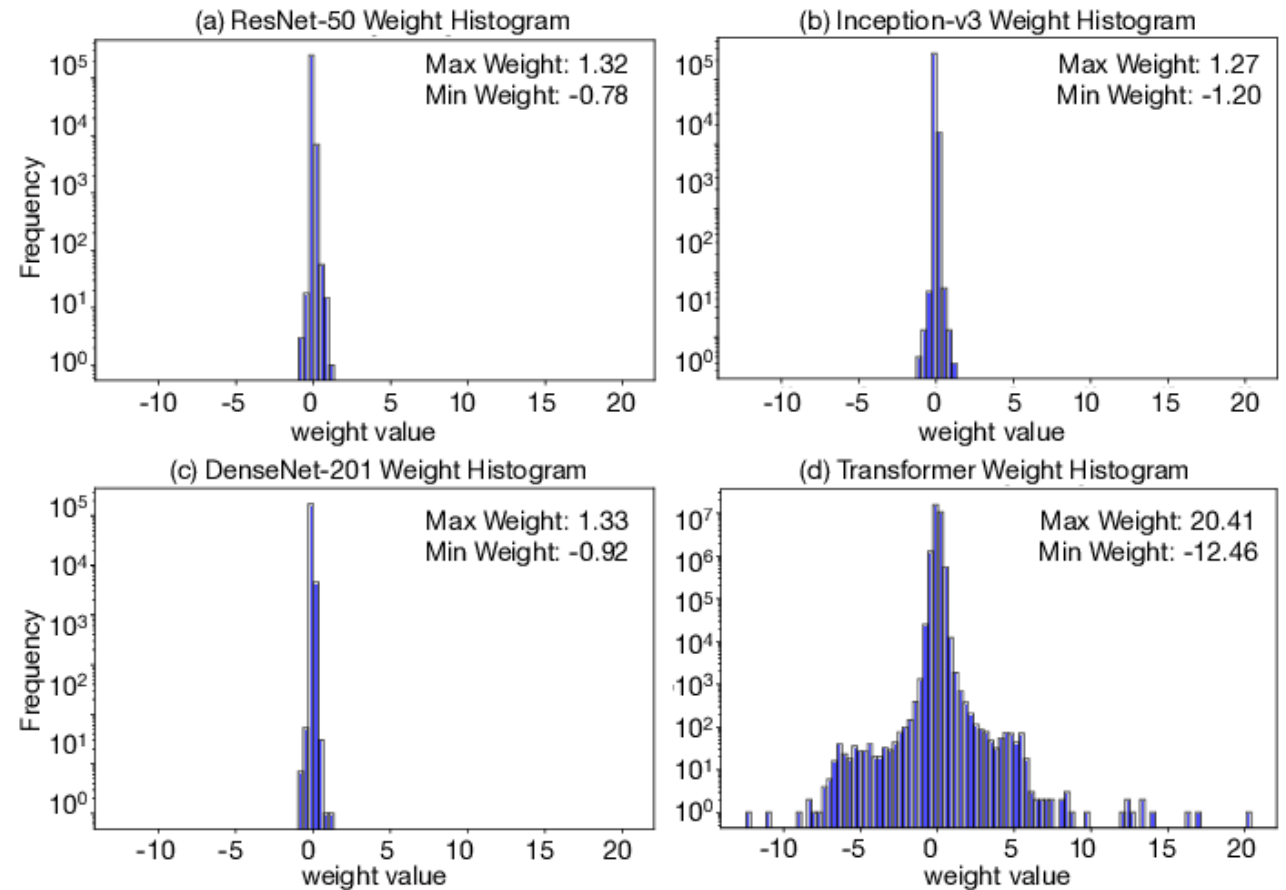
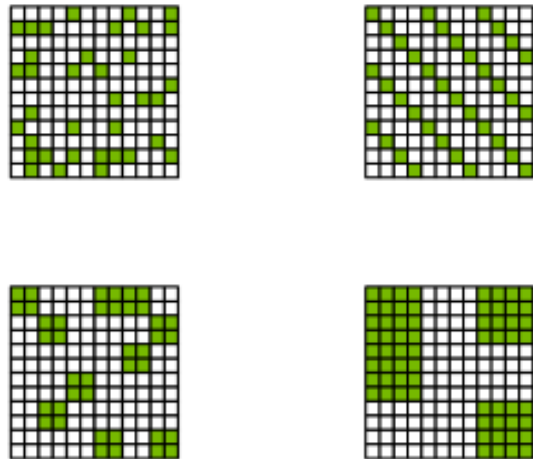
Source: Adapted from Corel, J.L. The postnatal development of the human cerebral cortex. Cambridge, MA: Harvard University Press; 1975.

PRUNING

The idea

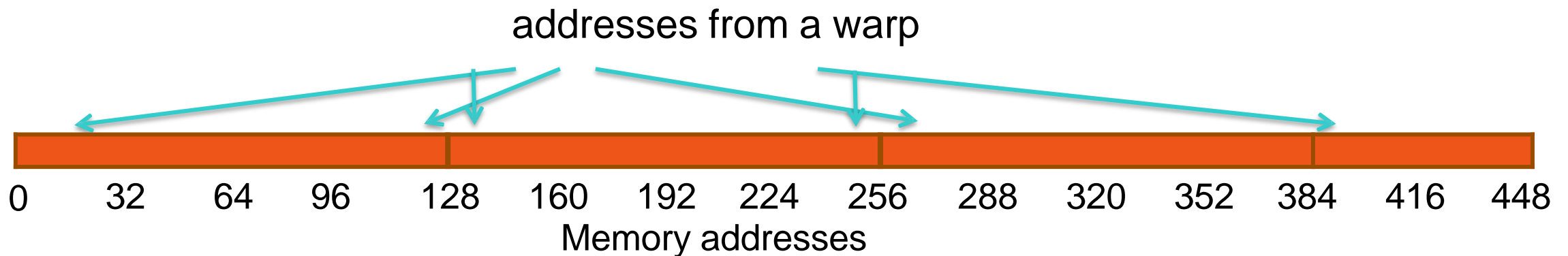
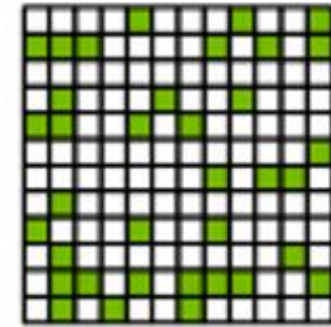
The opportunity:

- Reduced memory bandwidth
- Reduced memory footprint
- Acceleration (especially in presence of hardware acceleration)



CHERRY PICKING IN SPARSE MATRICES

- Memory operations are issued per warp (32 threads)
 - Just like all other instructions
- If only a single byte is needed -
 - 32 bytes will be issued, and only 1 will be used.



GOALS FOR A TRAINING RECIPE

Maintains accuracy

Is applicable across various tasks, network architectures, and optimizers

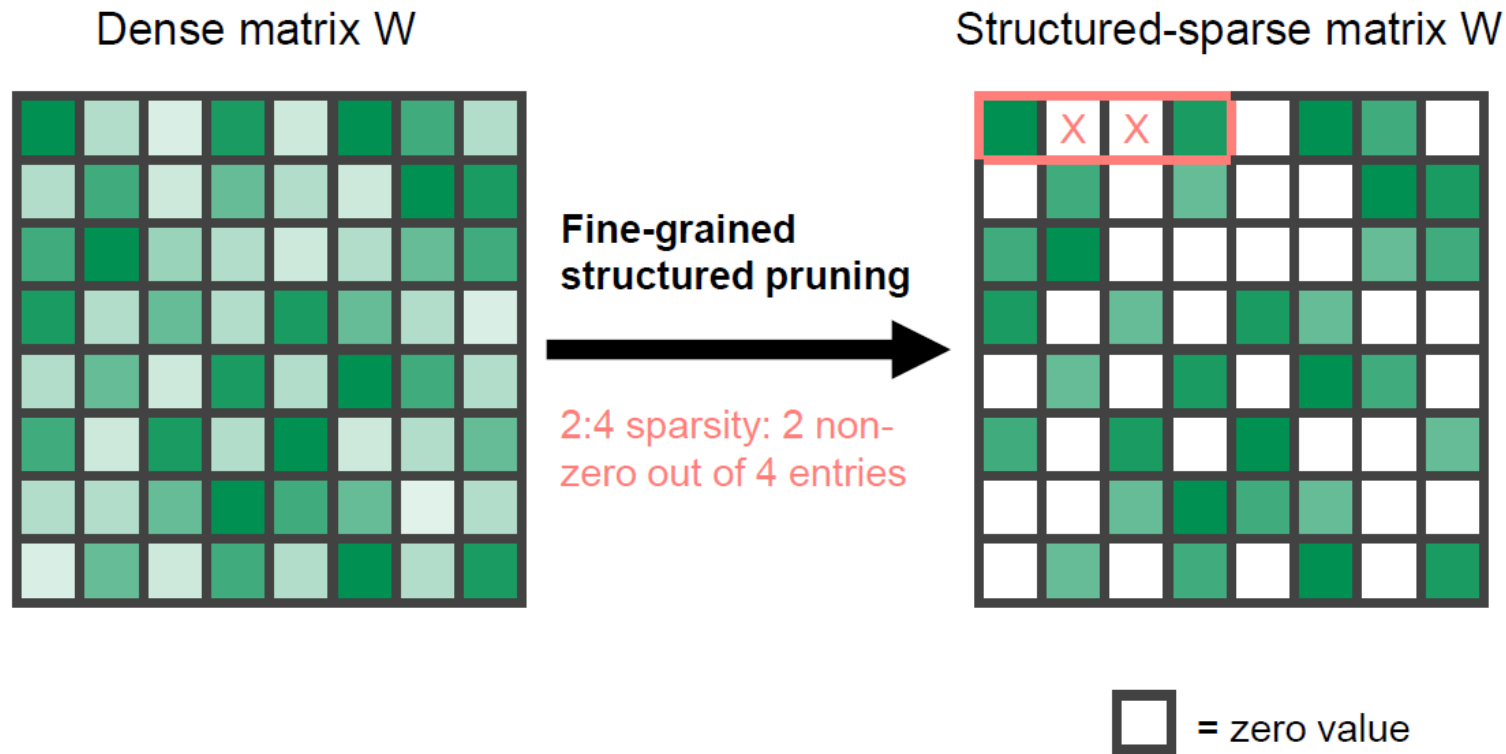
Does not require hyper-parameter searches



STRUCTURED SPARSITY

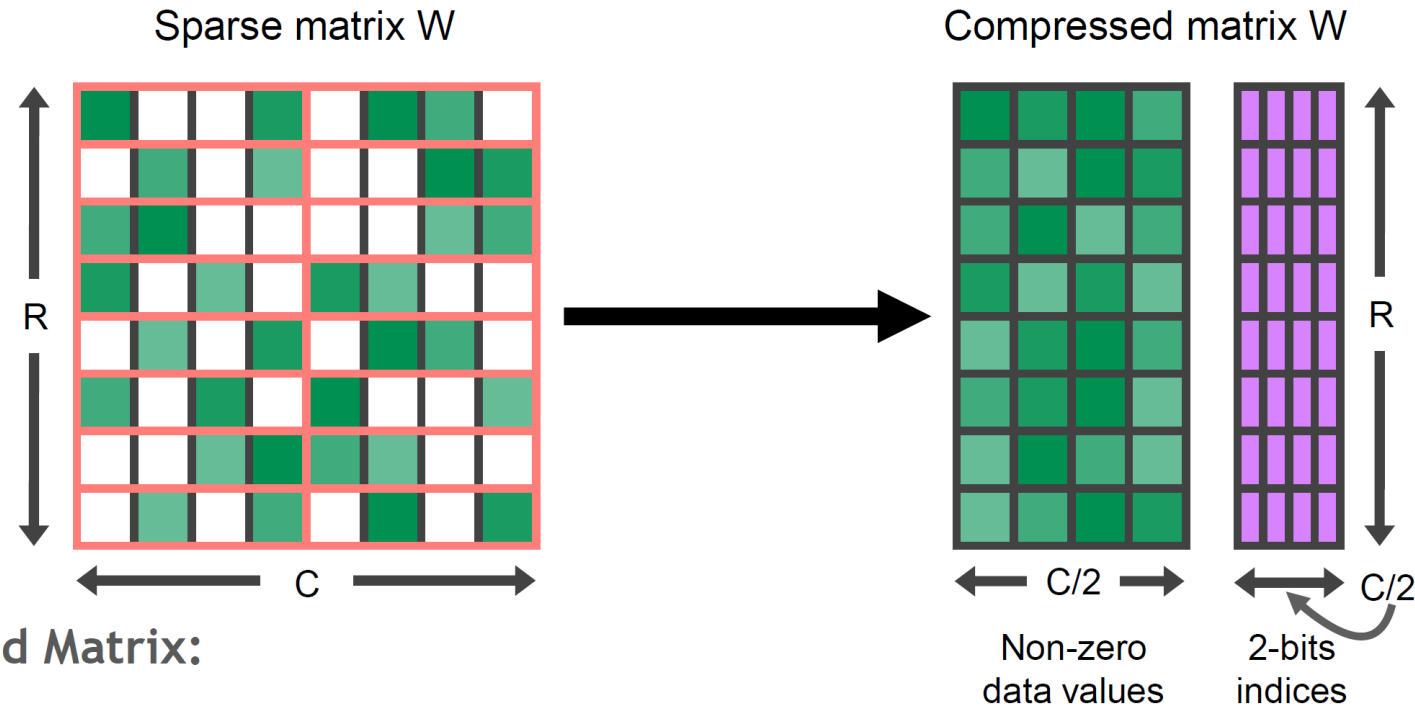
SPARSITY IN AMPERE

At Most 2 Non-zeros in Every Contiguous Group of 4 Values



2:4 COMPRESSED MATRIX FORMAT

At most 2 non-zeros in every contiguous group of 4 values



Compressed Matrix:

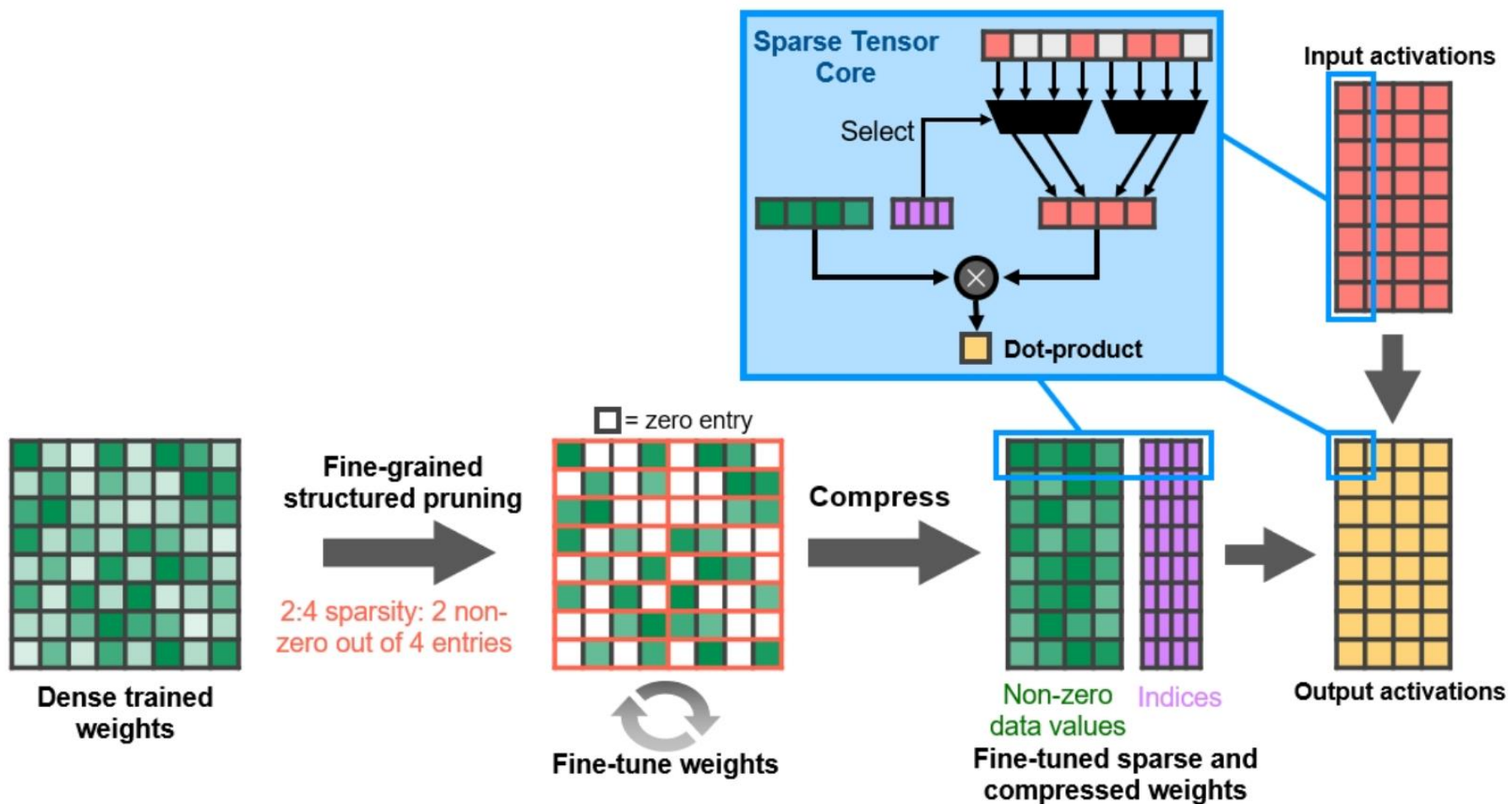
Data: $\frac{1}{2}$ size

Metadata: 2b per non-zero element

16b data => 12.5% overhead

8b data => 25% overhead

FINE-GRAINED STRUCTURED SPARSITY IN AMPERE



SPARSITY IN AMPERE GPUS

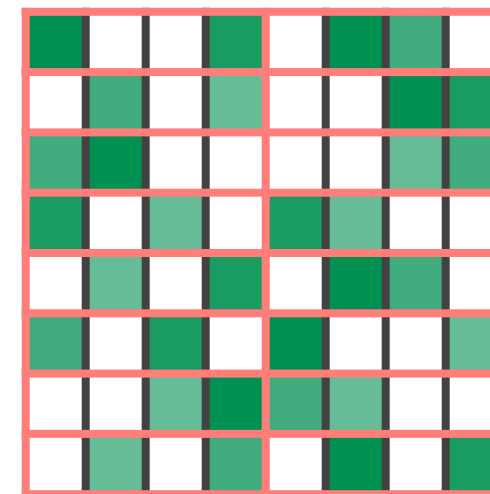
Fine-grained structured sparsity for Tensor Cores

- 50% fine-grained sparsity
- 2:4 pattern: 2 values out of each contiguous block of 4 must be 0

Addresses the 3 challenges:

- **Accuracy:** maintains accuracy of the original, unpruned network
 - Medium sparsity level (50%), fine-grained
- **Training:** a recipe shown to work across tasks and networks
- **Speedup:**
 - Specialized Tensor Core support for sparse math
 - Structured: lends itself to efficient memory utilization

2:4 structured-sparse matrix



□ = zero value

NLP - LANGUAGE MODELING

Transformer-XL, BERT

Network	Task	Metric	Dense FP16	Accuracy			
				Sparse FP16		Sparse INT8	
Transformer-XL	enwik8	BPC	1.06	1.06	-	-	-
BERT-Base	SQuAD v1.1	F1	87.6	88.1	0.5	88.1	0.5
BERT-Large	SQuAD v1.1	F1	91.1	91.5	0.4	91.5	0.4

GENERATE A STRUCTURED SPARSE NETWORK

APEX's Automatic SParsity: ASP

```
import torch
from apex.contrib.sparsity import ASP
device = torch.device('cuda')

model = TheModelClass(*args, **kwargs) # Define model structure
model.load_state_dict(torch.load('dense_model.pth'))

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9) # Define optimizer

ASP.prune_trained_model(model, optimizer)

x, y = DataLoader(...) #load data samples and labels to train the model
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

torch.save(model.state_dict(), 'pruned_model.pth') # checkpoint has weights and masks
```

Init mask buffers, tell optimizer
to mask weights and gradients,
compute sparse masks:
Universal Fine Tuning

PyTorch sparse fine-tuning loop

Summary

- ❖ NVIDIA is an accelerated computing platform
- ❖ Optimizing the entire stack from HW to applications
- ❖ “CUDA Everywhere” - One Ring to Rule Them All!
- ❖ Hardware <-> Software Interactions for Optimal Performance



THANK YOU!