ORTA DOĞU TEKNİK ÜNİVERSİTESİ
MIDDLE EAST TECHNICAL UNIVERSITY

Deployment Tools and Advanced Concepts

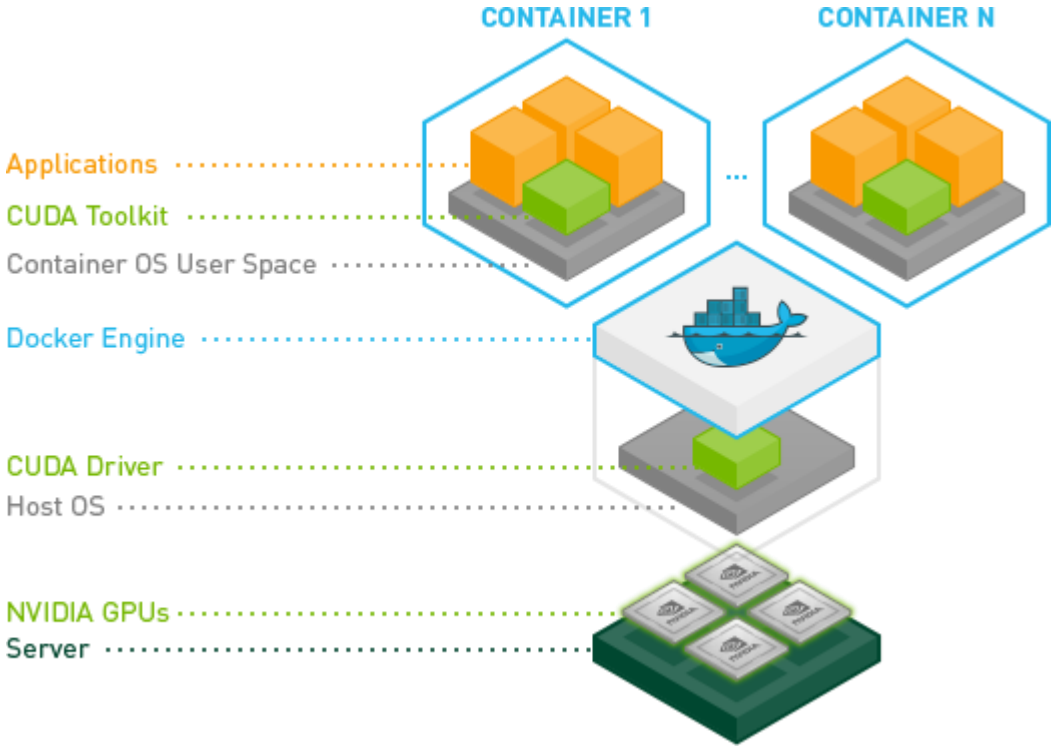
Prof. Dr. Alptekin Temizel
Graduate School of Informatics, METU

Docker containers

- Docker® containers are often used to seamlessly deploy CPU-based applications on multiple machines.
- Containers are both hardware-agnostic and platform-agnostic.
- This is obviously not the case when using NVIDIA GPUs since it is using specialized hardware and it requires the installation of the NVIDIA driver.
- As a result, Docker Engine does not natively support NVIDIA GPUs with containers.

NVIDIA Docker

- To make the Docker images portable while still leveraging NVIDIA GPUs, nvidia-docker makes the images agnostic of the NVIDIA driver.
- The required character devices and driver files are mounted when starting the container on the target machine.



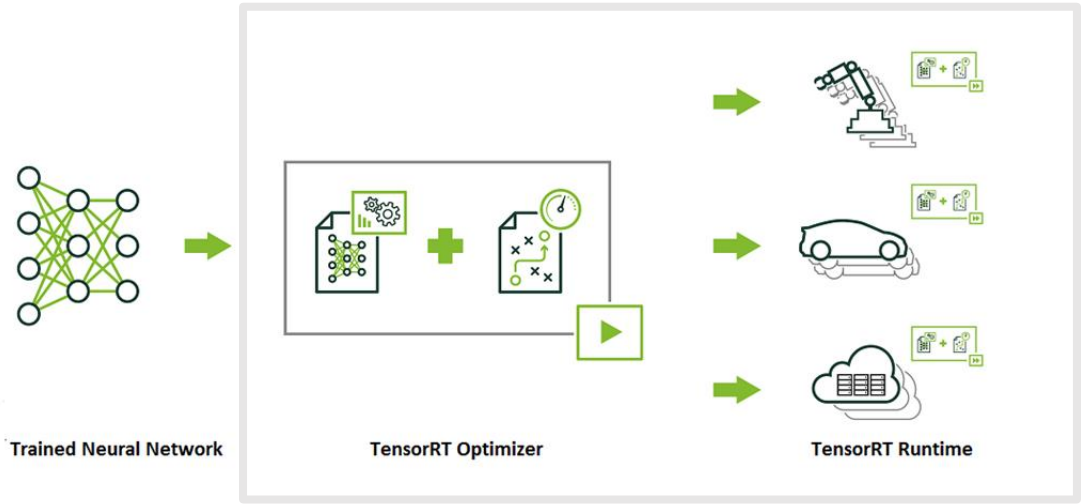
GPU containerization

Containerizing GPU applications provides several benefits,:

- Reproducible builds
- Ease of deployment
- Isolation of individual devices
- Run across heterogeneous driver/toolkit environments
- Requires only the NVIDIA driver to be installed
- Enables "fire and forget" GPU applications
- Facilitate collaboration

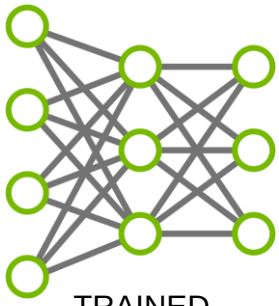
TensorRT

- Inference engine for production deployment of deep learning applications



- Allows developers to focus on developing AI powered applications
  - TensorRT ensures optimal inference performance

TensorRT Optimizer



TRAINED
NEURAL
NETWORK

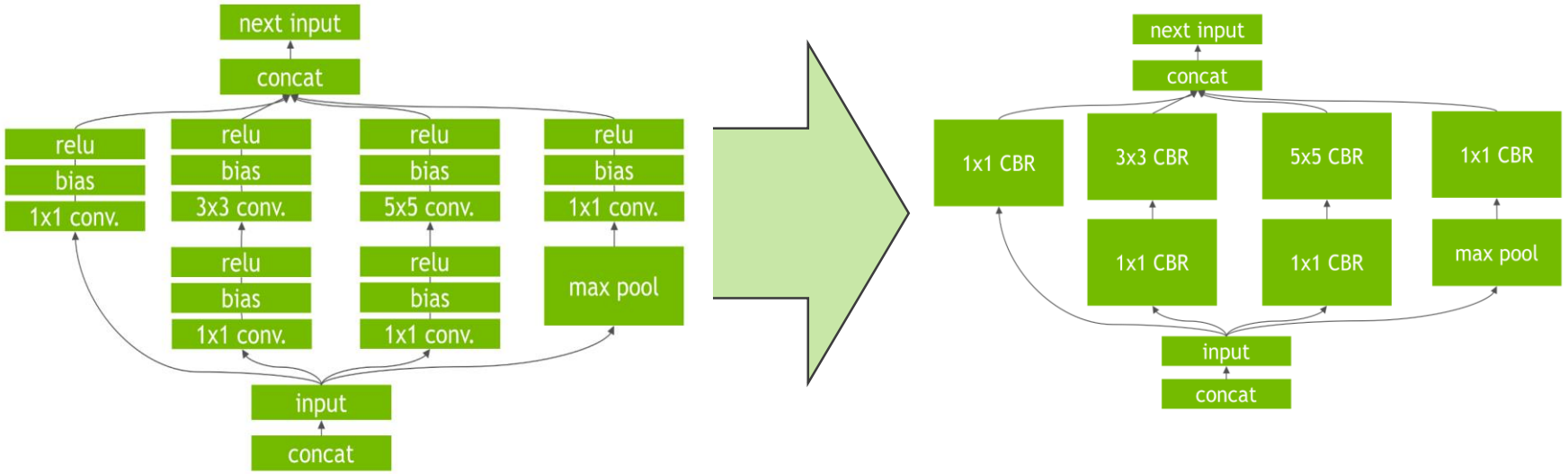
- Weight & Activation Precision Calibration (INT8)
- Layer and Tensor Fusion
- Auto-tuning for target platform
- Select optimal tensor layout
- Batch size tuning
- Dynamic tensor memory

OPTIMIZED
INFERENCE
RUNTIME

developer.nvidia.com/tensorrt

TensorRT Optimizer

Vertical Layer Fusion

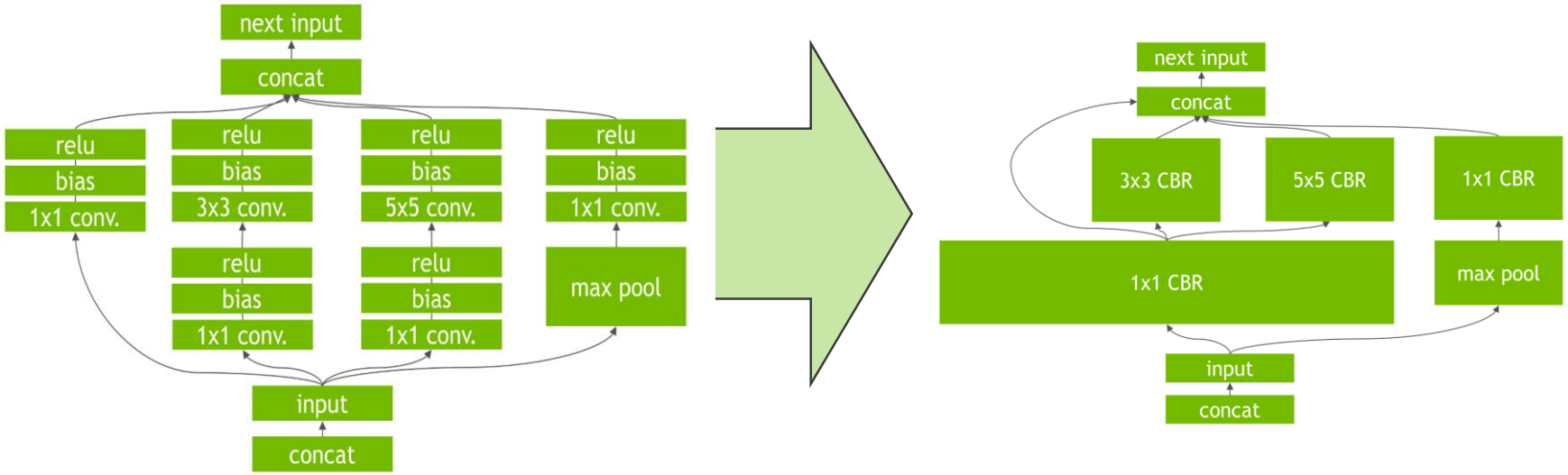


CBR = Convolution, Bias and ReLU

developer.nvidia.com/tensorrt

TensorRT Optimizer

Horizontal Layer Fusion (Layer Aggregation)



CBR = Convolution, Bias and ReLU

developer.nvidia.com/tensorrt

TensorRT Optimizer

Supported layers

- Convolution: 2D
- Activation: ReLU, tanh and sigmoid
- Pooling: max and average
- ElementWise: sum, product or max of two tensors
- LRN: cross-channel only
- Fully-connected: with or without bias
- SoftMax: cross-channel only
- Deconvolution

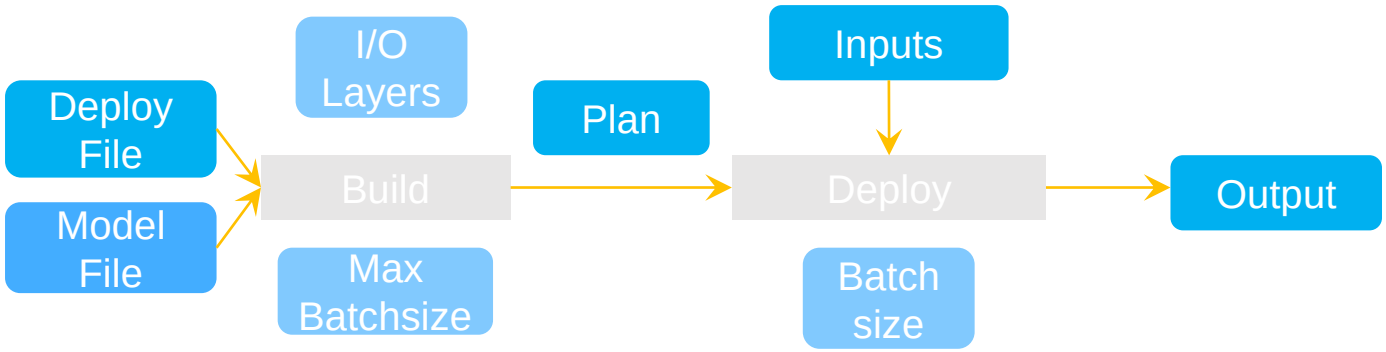
TensorRT Optimizer

- Scalability:
  - Output/Input Layers can connect with other deep learning framework directly
    - Caffe, Theano, Torch, TensorFlow
- Reduced Latency:
  - INT8 or FP16
    - INT8 delivers 3X more throughput compared to FP32
    - INT8 uses 61% less memory compared to FP32

TensorRT Runtime

Two Phases

- **Build:** optimizations on the network configuration and generates an optimized plan for computing the forward pass
- **Deploy:** Forward and output the inference result



TensorRT Runtime

- No need to install and run a deep learning framework on the deployment hardware
- **Plan** is a runtime (serialized) object
  - smaller than the combination of model and weights
  - Ready for immediate use. Alternatively, state can be serialized and saved to disk or to an object store for distribution.
- Three files needed to deploy a classification neural network:
  - Network architecture file (deploy.prototxt)
  - Trained weights (net.caffemodel)
  - Label file to provide a name for each output class

PyCUDA

- Maps all of CUDA into Python.
- Enables run-time code generation (RTCG) for flexible, fast, automatically tuned codes.
- Added robustness: automatic management of object lifetimes, automatic error checking
- Added convenience: comes with ready-made on-GPU linear algebra, reduction, scan. Add-on packages for FFT and LAPACK available.
- Fast. Near-zero wrapping overhead.

PyCUDA

```
import pycuda.driver as drv
import pycuda.autoinit
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
 __global__ void multiply_them(float *dest, float *a, float *b)
{
const int i = threadIdx.x;
dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

print dest-a*b
```


PyCUDA

```
import pycuda.driver as drv
import pycuda.autoinit
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
 __global__ void multiply_them(float *dest, float *a, float *b)
{
const int i = threadIdx.x;
dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

print dest-a*b
```

PyCUDA has compiled the CUDA source code and uploaded it to the card.

PyCUDA

```
import pycuda.driver as drv
import pycuda.autoinit
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
  __global__ void multiply_them(float *dest, float *a, float *b)
{
const int i = threadIdx.x;
dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

print dest-a*b
```

This code doesn't have to be a constant–you can easily have Python generate the code you want to compile. See Metaprogramming

PyCUDA

```
import pycuda.driver as drv
import pycuda.autoinit
import numpy

from pycuda.compiler import SourceM
mod = SourceModule("""
 __global__ void multiply_them(floa
{
const int i = threadIdx.x;
dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

print dest-a*b
```

PyCUDA's numpy interaction code has automatically allocated space on the device, copied the numpy arrays *a* and *b* over, launched a 400x1x1 single-block grid, and copied *dest* back.

PyCUDA

```
import pycuda.driver as drv
import pycuda.autoinit
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
 __global__ void multiply_them(float *dest, float *a, float *b)
{
const int i = threadIdx.x;
dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("

a = numpy.random.randn(400).astype
b = numpy.random.randn(400).astype

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a),
        block=(400,1,1), grid=(1,1))

print dest-a*b
```

You can keep your data on the card between kernel invocations.
No cleanup code is needed. PyCUDA will automatically infer what cleanup is necessary and do it for you.

Numba

- Numba compiles Python functions (not entire applications, and not parts of functions).
- Numba is a Python module that can turn a function into a (usually) faster function.
- Numba translates functions when they are first called. This ensures the compiler knows what argument types you will be using.
- Currently, Numba is focused on numerical data types, like int, float, and complex. There is very limited string processing support, and many string use cases are not going to work well on the GPU. To get best results with Numba, you will likely be using NumPy arrays.

https://github.com/ContinuumIO/gtc2017-numba

Pyculib

Pyculib is a package that provides access to several numerical libraries that are optimized for performance on NVidia GPUs.

Pyculib was originally part of Accelerate, developed by Anaconda, Inc.

The current version, 1.0.1, was released on July 27, 2017.

Features
Bindings to the following CUDA libraries:
cuBLAS
cuFFT
cuSPARSE
cuRAND
CUDA Sorting algorithms from the CUB and Modern GPU libraries

NVIDIA GPU REST Engine (GRE)

- Millions of people now share photos, video, and music or spoken words on the web daily.
- GPU powered micro services can process this data quickly to deliver great visual experiences and intelligent capabilities based on deep learning.

NVIDIA GPU REST Engine (GRE)

- NVIDIA GPU REST Engine (GRE) is a critical component for developers building low-latency web services.
- GRE includes a multi-threaded HTTP server that presents a RESTful web service and schedules requests efficiently across multiple NVIDIA GPUs.
- The overall response time depends on how much processing you need to do, but GRE itself adds very little overhead and can process null-requests in as little as 10 microseconds.

NVIDIA GPU REST Engine (GRE)

- The GRE powered web service takes incoming requests, fetches the required input data and schedules quanta of work to a work queue that is serviced asynchronously.
- A separate thread manages the work for each GPU and ensures that all GPUs in the system are continuously servicing requests from the queue.

Tensor Cores

- Tensor Cores, which give the Tesla V100 accelerator a peak throughput 12 times the 32-bit floating point throughput of the previous-generation Tesla P100.
- Tensor Cores are programmable matrix-multiply-and-accumulate units that can deliver up to 125 Tensor TFLOPS for training and inference applications.
- The Tesla V100 GPU contains 640 Tensor Cores: 8 per SM.

Tensor Cores

- Tensor Cores are already supported for deep learning frameworks: TensorFlow, PyTorch, MXNet, and Caffe2.
- Mixed-Precision Training Guide gives more information about enabling Tensor Cores when using these frameworks
- TensorRT 3.0 release also supports Tensor Cores for deep learning inference.
- cuBLAS uses Tensor Cores to speed up GEMM computations (GEMM is the BLAS term for a matrix-matrix multiplication)
- cuDNN uses Tensor Cores to speed up both convolutions and recurrent neural networks (RNNs).

Tensor Cores

- Artificial Intelligence researchers design deeper and deeper neural nets every year; the number of convolution layers in the deepest nets is now several dozen. Training DNNs requires the convolution layers to be run repeatedly, during both forward- and back-propagation.
- NVIDIA Developer Blog post shows you how to use Tensor Cores in your own application using CUDA Libraries as well as how to program them directly in CUDA C++ device code.

CUDA 10

- https://devblogs.nvidia.com/cuda-10-features-revealed/?ncid=so-ele-cd10-59652&_lrsc=5b61dc4d-e8b9-42fc-85a6-be93935f223e&ncid=so-lin-lt-798

Power Efficiency on Embedded Devices

- Jetson TX2 was designed for peak processing efficiency at 7.5W of power. This level of performance, referred to as Max-Q, represents the peak of the power/throughput curve.
- Every component on the module including the power supply is optimized to provide highest efficiency at this point.

- The Max-Q frequency for the GPU is 854 MHz, and for the ARM A57 CPUs it’s 1.2 GHz.

- The L4T BSP in JetPack 3.0 includes preset platform configurations for setting Jetson TX2 in Max-Q mode. JetPack 3.0 also includes a new command line tool called nvpmodel for switching profiles at run time.

Power Efficiency on Embedded Devices

- While Dynamic Voltage and Frequency Scaling (DVFS) permits Jetson TX2's Tegra "Parker" SoC to adjust clock speeds at run time according to user load and power consumption, the Max-Q configuration sets a cap on the clocks to ensure that the application is operating in the most efficient range only.

- Although most platforms with a limited power budget will benefit most from Max-Q behavior, others may prefer maximum clocks to attain peak throughput, albeit with higher power consumption and reduced efficiency.

Power Efficiency on Embedded Devices

- DVFS can be configured to run at a range of other clock speeds, including underclocking and overclocking. Max-P, the other preset platform configuration, enables maximum system performance in less than 15W.

- The Max-P frequency is 1122 MHz for the GPU and 2 GHz for the CPU when either ARM A57 cluster is enabled or Denver 2 cluster is enabled and 1.4 GHz when both the clusters are enabled.

- You can also create custom platform configurations with intermediate frequency targets to allow balancing between peak efficiency and peak performance for your application. Table 2 below shows how the performance increases going from Max-Q to Max-P and the maximum GPU clock frequency while the efficiency gradually reduces.

Power Efficiency on Embedded Devices

		NVIDIA Jetson TX1	NVIDIA Jetson TX2		
		Max Clock (998 MHz)	Max-Q (854 MHz)	max-P (1122 MHz)	Max Clock (1302 MHz)
GoogLeNet batch=2	Perf	141 FPS	138 FPS	176 FPS	201 FPS
	Power (AP+DRAM)	9.14 W	4.8 W	7.1 W	10.1 W
	Efficiency	15.42	28.6	24.8	19.9
GoogLeNet batch=128	Perf	204 FPS	196 FPS	253 FPS	290 FPS
	Power (AP+DRAM)	11.7 W	5.9 W	8.9 W	12.8 W
	Efficiency	17.44	33.2	28.5	22.7
AlexNet batch=2	Perf	164 FPS	178 FPS	222 FPS	250 FPS
	Power (AP+DRAM)	8.5 W	5.6 W	7.8 W	10.7 W
	Efficiency	19.3	32	28.3	23.3
AlexNet batch=128	Perf	505 FPS	463 FPS	601 FPS	692 FPS
	Power (AP+DRAM)	11.3 W	5.6 W	8.6 W	12.4 W
	Efficiency	44.7	82.7	69.9	55.8

Table 2. Power consumption measurements for GoogLeNet and AlexNet architectures for max-Q and max-P performance levels on NVIDIA Jetson TX1 and Jetson TX2. The table reports energy efficiency for all tests in images per second per Watt consumed.

Warp Operations

- Warp operations can be used to exchange information across a warp
  - No need to use shared memory
  - Synchronous in a warp
  - Dedicated operations
  - Very fast

Warp Operations

- The warp vote functions allow the threads of a given warp to perform a reduction-and-broadcast operation.
- These functions take as input an integer predicate from each thread in the warp and compare those values with zero.
- The results of the comparisons are combined (reduced) across the active threads of the warp, broadcasting a single return value to each participating thread.

Warp Vote

all, any, ballot

- int __all (int predicate) ;
  - evaluates predicate *for all active threads of the warp* and returns non-zero if and only if predicate evaluates to non-zero for all of them

- int __any(int predicate) ;
  - evaluates predicate and returns non-zero if and only if predicate evaluates to non-zero for any of them

- unsigned int __ballot (int predicate)
  - evaluates predicate and returns an integer whose Nth bit is set if and only if predicate evaluates to non-zero for the Nth thread of the warp

Warp Vote

all, any, ballot

- For each of these warp vote operations, the result excludes threads that are inactive (e.g., due to warp divergence).

- Inactive threads are represented by 0 bits in the value returned by __ballot() and are not considered in the reductions performed by __all() and __any().

Warp Operations - Shuffle

- The shuffle instruction allows a thread to read values stored in a register from a thread within the same warp without involving shared memory.
- Only available for architecture 3.x
- Exchanges 4 bytes of data across a warp.

Warp Operations - Shuffle

Why use the shuffle instruction instead of shared memory?

- You can use the shuffle instruction to free up shared memory to be used for other data or to increase your occupancy.

- It is faster than shared memory
  - only requires one instruction versus three for shared memory (write, synchronize, read).

- Relative to Fermi, shared memory bandwidth has doubled on Kepler devices but the number of compute cores has increased by 6x; therefore, the shuffle instruction provides another means to share data between threads and keep the CUDA cores busy with memory accesses that have low latency and high bandwidth.

- It can be used instead of warp-synchronous optimizations
  - removing __syncthreads()

Warp Operations - Shuffle

laneID: Index of a thread within a warp - calculated as thread index % 32.

```
float __shfl( float var,
              // Variable you want to read from source thread

              int srcLane,
             // laneID of the source thread

              int width=warpSize
             // Division of warp into segments of size width );
```


Warp Operations - Shuffle

integer

```
int __shfl(int var, int srcLane, int width=warpSize);

int __shfl_up(int var, unsigned int delta, int width=warpSize);

int __shfl_down(int var, unsigned int delta, int width=warpSize);

int __shfl_xor(int var, int laneMask, int width=warpSize);
```


Warp Operations - Shuffle

float

```
float __shfl(float var, int srcLane, int width=warpSize);

float __shfl_up(float var, unsigned int delta, int width=warpSize);

float __shfl_down(float var, unsigned int delta, int width=warpSize);

float __shfl_xor(float var, int laneMask, int width=warpSize);
```


Warp Operations - Shuffle

- **shuffle** returns the value of *var* held by the thread whose ID is given by *srcLane*.
  - If srcLane is outside the range [0:width-1], then the thread's own value of var is returned

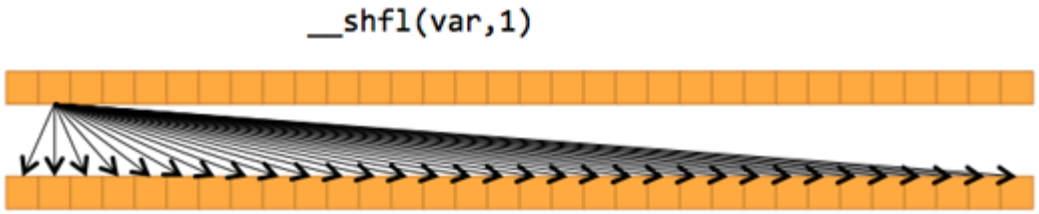
- **shuffle** up calculates a source lane ID by subtracting *delta* from the caller's lane ID.
  - The value of var held by the resulting lane ID is returned: in effect, *var* is shifted up the warp by delta lanes.
  - The source lane index will not wrap around the value of *width*, so effectively the lower delta lanes will be unchanged

Warp Operations - Shuffle

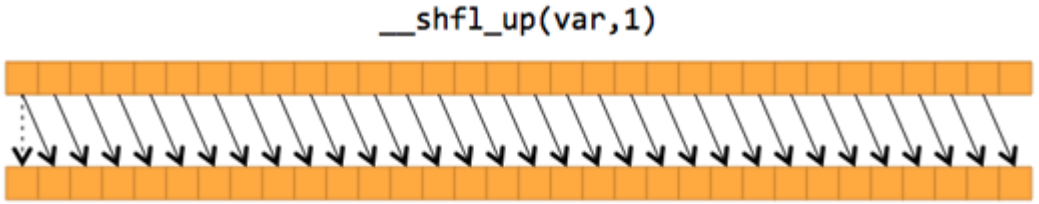
- **shuffle down** calculates a source lane ID by adding *delta* to the caller's lane ID.
  - The value of *var* held by the resulting lane ID is returned: this has the effect of shifting *var* down the warp by *delta* lanes.
  - As for __shfl_up(), the ID number of the source lane will not wrap around the value of width and so the upper delta lanes will remain unchanged

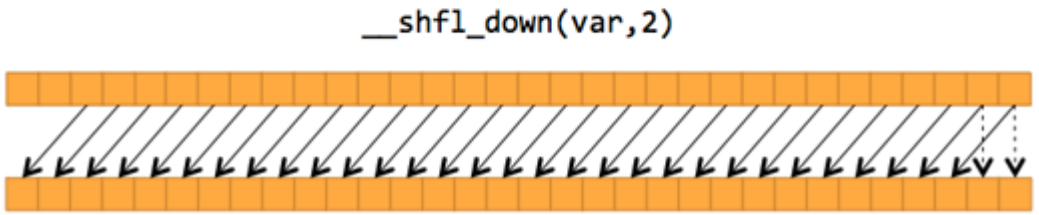
- **shuffle XOR** calculates a source line ID by performing a bitwise XOR of the caller's lane ID with laneMask:
  - the value of var held by the resulting lane ID is returned. If the resulting lane ID falls outside the range permitted by width, the thread's own value of var is returned.
  - This mode implements a butterfly addressing pattern such as is used in tree reduction and broadcast

Warp Operations - Shuffle

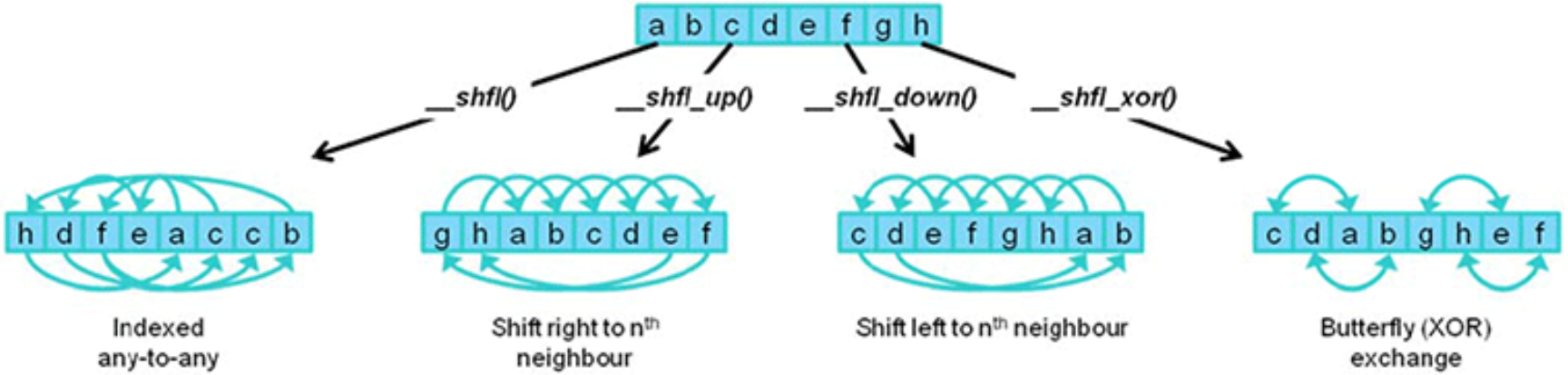


Shuffle instruction with constant srcLane broadcasts the value in a register from one thread to all threads in a warp





Warp Operations - Shuffle



Reduction using Shuffle

```
__global__ void warpReduce ( ) {

  // Seed starting value as inverse lane ID
  int  value = 31 - laneId ;

  // Use XOR mode to perform butterflyreduction
  for ( int i = 16; i >= 1; i /= 2)
     value += __ shfl_xor ( value, i, 32 ) ;

  // value now contains the sum across all threads
  printf ( Thread %d final value = %d\ n , threadIdx .x , value ) ;
}

 void main ( ) {
    warpReduce<<< 1 , 32 >>>() ;
    cudaDeviceSynchronize( ) ;
}
```


Reduction using Shuffle

Threads inside a warp are synchronous by definition
Can perform warp reduction inside a warp without shuffle

```
__global__ void warpReduce ( ) {
  __shared__ int smem [32] ;
  smem [threadIdx.x] = threadIdx.x ;

  if ( threadIdx.x < 16 ) smem [threadIdx.x] += smem [threadIdx.x + 16];
  if (threadIdx.x < 8 )    smem [threadIdx.x] += smem threadIdx.x + 8];
  if (threadIdx.x < 4 )    smem [threadIdx.x] += smem [threadIdx.x + 4];
  if (threadIdx.x < 2 )    smem [threadIdx.x] += smem threadIdx.x + 2];
  if (threadIdx.x < 1 )     smem [threadIdx.x] += smem [threadIdx.x + 1];

  printf ( Thread %d final value = %d\ n , threadIdx.x , smem [threadId.x] ) ;
}

void main ( ) {
warpReduce<<< 1 , 32 >>>() ;
}
```


Volta Compatibility

Independent Thread Scheduling Compatibility

- The Volta architecture introduces Independent Thread Scheduling among threads in a warp. If the developer made assumptions about warp-synchronicity, [1] this feature can alter the set of threads participating in the executed code compared to previous architectures.
- Please see *Compute Capability 7.0* in the *CUDA C Programming Guide* for details and corrective actions. To aid migration Volta developers can opt-in to the Pascal scheduling model with the following combination of compiler options:

```
nvcc -arch=compute_60 -code=sm_70 ...
```