cuBLAS / cuDNN

2023 NSF-sponsored Workshop on Deep Learning Systems in Advanced GPU Cyberinfrastructure

Dr. Iraklis Anagnostopoulos

Southern Illinois University

Table of contents

1. Overview
2. Introduction
3. cuBLAS
4. cuDNN

Overview

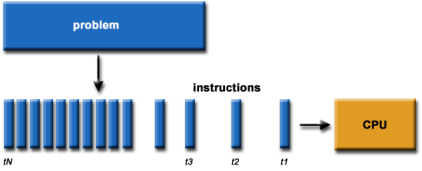
Overview

- Introduction
  - Brief introduction to CUDA, cuBLAS, and cuDNN
- cuBLAS Fundamentals
  - Overview of cuBLAS: What it is and why it's important
  - Basic operations in cuBLAS
  - Hands-on exercise: Implementing simple cuBLAS operations
- cuDNN Fundamentals
  - Overview of cuDNN: What it is and why it's important
  - Basic operations in cuDNN
  - Hands-on exercise: Implementing simple cuDNN operations
- Integration of cuBLAS and cuDNN
  - How cuBLAS and cuDNN work together
  - Hands-on exercise: Integrating cuBLAS and cuDNN in a simple project

Introduction

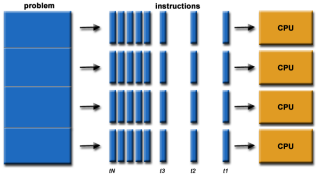
Traditional Computing

- Von Neumann architecture: instructions are sent from memory to the CPU
- Serial execution: Instructions are executed one after another on a single Central Processing Unit (CPU)
- Problems:
  - More expensive to produce
  - More expensive to run
  - Bus speed limitation



Parallel Computing

- Official-sounding definition: The simultaneous use of multiple compute resources to solve a computational problem.
- Benefits:
  - Economical – requires less power and cheaper to produce
  - Better performance – bus/bottleneck issue
- Limitations:
  - New architecture – Von Neumann is all we know!
  - New debugging difficulties – cache consistency issue

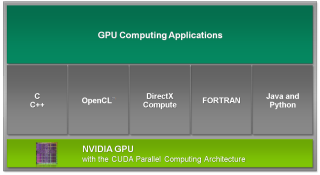


GPU

- GPUs are massively multithreaded many core chips
  - Hundreds of scalar processors
  - Tens of thousands of concurrent threads
  - Fine-grained data-parallel computation
- Users across science & engineering disciplines are achieving tenfold and higher speedups on GPU

What is CUDA?

- CUDA is NVIDIA's general purpose parallel computing architecture
- Designed for calculation-intensive computation on GPU hardware
- CUDA is not a language, it is an API

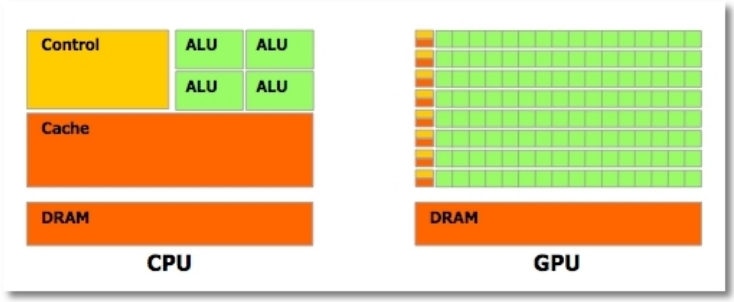


What is CUDA?

- CUDA is the acronym for Compute Unified Device Architecture
  - A parallel computing architecture developed by NVIDIA.
  - The computing engine in GPU.
  - CUDA can be accessible to software developers through industry standard programming languages.
- CUDA gives developers access to the instruction set and memory of the parallel computation elements in GPUs.

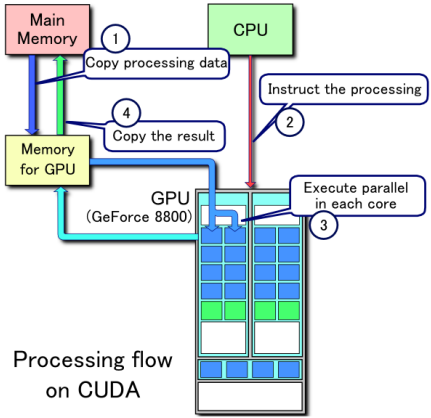
CUDA Goals

- Scale code to hundreds of cores running thousands of threads
- The task runs on the GPU independently from the CPU



Processing Flow

- Processing Flow of CUDA:
  - Copy data from main mem to GPU mem
  - CPU instructs the process to GPU
  - GPU execute parallel in each core
  - Copy the result from GPU mem to main mem



CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel

CUDA Programming Model: A Highly Multithreaded Coprocessor

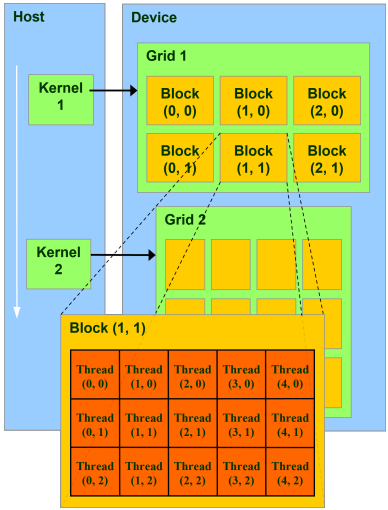
- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads

CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

CUDA programming model

- Device = GPU
- Host = CPU
- Kernel = function that runs on the device



CUDA highlights

- The API is an **extension to the ANSI C programming language**
- Low learning curve
- The hardware is **designed to enable lightweight runtime and driver**
- High performance

CUDA highlights

- The API is an **extension to the ANSI C programming language**
- Low learning curve
- The hardware is **designed to enable lightweight runtime and driver**
- High performance
- A kernel is executed by a grid of thread blocks
- A thread block is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution
- Threads from different blocks cannot cooperate

CUDA info

- Check your system:

```
$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Tue_May__3_18:49:52_PDT_2022
Cuda compilation tools, release 11.7, V11.7.64
Build cuda_11.7.r11.7/compiler.31294372_0
$ nvidia-smi
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 530.41.03              Driver Version: 530.41.03    CUDA Version: 12.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name                  Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf            Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce GTX 1050 Ti      Off| 00000000:01:00.0  On |                  N/A |
| 46%   37C    P0               N/A /  75W|    729MiB /  4096MiB |      5%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+
```

CUDA Example

- Check the provided `cuda_simple.cu`
- The code calculates the square of an array in CPU and GPU

Compilation and execution

```
$ nvcc cuda_simple.cu -o cuda_simple
$ ./cuda_simple
CUDA version: 11.7
CPU execution time: 0.306754 seconds
GPU execution time: 0.009643 seconds
```

- Compare and discuss on the output

Remeber Function Qualifiers

	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
  - Must return `void`
- `__device__` and `__host__` can be used together

cuBLAS

cuda library

- The cuda library consists of:
  - A minimal set of extensions to the C language that allow the programmer to target portions of the source code for execution on the device;
  - A runtime library split into:
    - A host component that runs on the host;
    - A device component that runs on the device and provides device-specific functions;
    - A common component that provides built-in vector types and a subset of the C standard library that are supported in both host and device code;

- cuda includes 2 widely used libraries
  - cuBLAS: BLAS implementation
  - cuFFT: FFT implementation

First of all, what is BLAS?

- Basic Linear Algebra Subprograms (BLAS) is a specification that prescribes a set of low-level routines for performing common linear algebra operations such as
  - vector addition,
  - scalar multiplication,
  - dot products,
  - linear combinations, and
  - matrix multiplication
- Official definition Wikipedia
- Libraries that implement BLAS exist in almost all major languages.

cuBLAS

- Implementation of BLAS (Basic Linear Algebra Subprograms) on top of CUDA driver:
- It allows access to the computational resources of NVIDIA GPUs.
- The basic model of using the cuBLAS library is:
  - Create matrix and vector objects in GPU memory space;
  - Fill them with data;
  - Call the cuBLAS functions;
  - Upload the results from GPU memory space back to the host;

cuBLAS - Information

- **GPU Acceleration**:faster execution of BLAS operations by leveraging the parallel processing power of NVIDIA GPUs.
- **High Performance**: highly optimized implementations of BLAS functions, significantly outperform CPU-only implementations.
- **Interoperability with CUDA**: cuBLAS is designed to interoperate seamlessly with CUDA, making it easier to build complex applications
- **Ease of Use**: cuBLAS provides a simple API that is similar to the standard BLAS API, making it easier for developers to transition from CPU-based BLAS to GPU-accelerated BLAS.

What is cuBLAS good for?

- Anything that uses heavy linear algebra computations
  - Graphics
  - Machine learning (this will be covered next week)
  - Computer vision
  - Physical simulations
  - Finance
  - etc...
- cuBLAS excels in situations where you want to maximize your performance by batching multiple kernels using streams.
  - Like making many small matrix-matrix multiplications on dense matrices

The various cuBLAS types

- All of the functions defined in cuBLAS have different versions which correspond to the different types of numbers in CUDA C
  - S, s : single precision (32 bit) real float
  - D, d : double precision (64 bit) real float
  - C, c : single precision (32 bit) complex float (implemented as a float2)
  - Z, z : double precision (64 bit) complex float
  - H, h : half precision (16 bit) real float

Sample cuBLAS function names w/ types

- `cublasIsamax` → cublas "I," s, amax
  - cublas : the cuBLAS prefix
  - I : stands for index. Cuda naming left over from Fortran
  - s : this is the single precision float variant
  - amax : finds a maximum
  - Returns smallest Index "i" of the earliest max element
- `cublasSgemm` → cublas S gemm
  - cublas : the prefix
  - S : single precision real float
  - gemm : general matrix-matrix multiplication

cuBLAS handle

- In cuBLAS, a handle (of type `cublasHandle_t`) is used to maintain the context for cuBLAS operations.
- The handle stores information about the computational resources being used, such as the CUDA device being used, the stream being used for asynchronous operations, and other settings.

cuBLAS handle

- In cuBLAS, a handle (of type `cublasHandle_t`) is used to maintain the context for cuBLAS operations.
- The handle stores information about the computational resources being used, such as the CUDA device being used, the stream being used for asynchronous operations, and other settings.
- **Initialization**: Before any cuBLAS operations can be performed, a handle must be created and initialized using the `cublasCreate()` function.
- **Usage**: The handle is passed as an argument to many cuBLAS functions. These functions use the handle to access the cuBLAS context and its associated computational resources.
- **Cleanup**: When the cuBLAS operations are complete, the handle should be destroyed using `cublasDestroy()`. This function releases the resources associated with the handle.

cuBLAS fucntions

- cuBLAS categorizes its functions in three levels
- **Level 1 Functions**: These are the simplest operations that involve two vectors.
- **Level 2 Functions**: These operations involve a matrix and a vector.
- **Level 3 Functions**: These are the most complex operations that involve two matrices.
- All functions in cuBLAS can be found here NVIDIA cuBLAS

cuBLAS - Function Levels - Level 1

- **Level 1 Functions (Vector-Vector Operations)**: These are the simplest operations that involve two vectors.
  - Basic building blocks for more complex operations.
  - They are the least computationally intensive of the three levels.
  - vector addition, dot product, and scalar multiplication.

cuBLAS - Function Levels - Level 1 Example

```
cublasStatus_t cublasSaxpy(cublasHandle_t handle, int n,
                           const float *alpha,
                           const float *x, int incx,
                           float *y, int incy)
```

- This function multiplies the vector **x** by the scalar *a* and adds it to the vector **y** overwriting the latest vector with the result: $y = a * \mathbf{x} + \mathbf{y}$

Param.	Memory	In/out	Meaning
handle		input	handle to the CUBLAS library context.
alpha	host or device	input	<type> scalar used for multiplication.
n		input	number of elements in the vector **x** and **y**.
x	device	input	<type> vector with **n** elements.
incx		input	stride between consecutive elements of **x**.
y	device	in/out	<type> vector with **n** elements.
incy		input	stride between consecutive elements of **y**.

cuBLAS - Function Levels - Level 2

- **Level 2 (Matrix-Vector Operations)**: Operations involve a matrix and a vector.
  - More complex operations that are commonly used in linear algebra
  - More computationally intensive than Level 1 operations
  - matrix-vector multiplication and triangular systems

cuBLAS - Function Levels - Level 2

- **Level 2 (Matrix-Vector Operations)**: Operations involve a matrix and a vector.
  - More complex operations that are commonly used in linear algebra
  - More computationally intensive than Level 1 operations
  - matrix-vector multiplication and triangular systems
- GEMV matrix-vector multiplication: $m \times n$ matrix $\mathbf{A}$, an n-dimensional vector $\mathbf{x}$, a m-dimensional vector $\mathbf{y}$, and for the scalars a and β: $y = a\mathbf{A}\mathbf{x} + \beta\mathbf{y}$

```
cublasStatus_t cublasSgemv(cublasHandle_t handle, cublasOperation_t trans,
                           int m, int n,
                           const float *alpha,
                           const float *A, int lda,
                           const float *x, int incx,
                           const float *beta,
                           float *y, int incy)
```

cuBLAS - Function Levels - Level 3

- **Level 3 Functions (Matrix-Matrix Operations)**: These are the most complex operations that involve two matrices.
  - Most complex and computationally intensive operations
  - matrix-matrix multiplication and systems of linear equations

cuBLAS - Function Levels - Level 3

- **Level 3 Functions (Matrix-Matrix Operations)**: These are the most complex operations that involve two matrices.
  - Most complex and computationally intensive operations
  - matrix-matrix multiplication and systems of linear equations
- Example, matrix-matrix multiplication: $\mathbf{C} = a\mathbf{AB} + \beta\mathbf{C}$

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float *alpha,
                           const float *A, int lda,
                           const float *B, int ldb,
                           const float *beta,
                           float *C, int ldc)
```

Example: Matrix multiplication

- Check the file `matmul.cu`. It performs matrix multiplication using the CPU, CUDA, and cuBLAS, and prints the execution time for each method

Compilation and execution

```
$ nvcc  -o matmul matmul.cu -lcublas
$ ./matmul
```

Example: Matrix multiplication

- Check the file `matmul.cu`. It performs matrix multiplication using the CPU, CUDA, and cuBLAS, and prints the execution time for each method

Compilation and execution

```
$ nvcc  -o matmul matmul.cu -lcublas
$ ./matmul
```

- Identify the following important parts:
  - The parameter that defines the size of the matrices
  - Memory allocation for CPU
  - Memory allocation for GPU
  - Memory transfers to GPU
  - Initialization of handle for cuBLAS
  - Identify the function (name) and the level of the used cublas function

Example: Matrix multiplication

For N = 1000

```
$ ./matmul
CPU time:    2.768519 seconds
CUDA time:   0.019095 seconds
cuBLAS time: 0.001316 seconds
```

For N = 1500

```
$ ./matmul
CPU time:    21.352445 seconds
CUDA time:   0.070569 seconds
cuBLAS time: 0.004040 seconds
```

Explain the differences in time and why cuBLAS is the fastest

Example: Matrix multiplication

- Why cuBLAS is the fastest?

Example: Matrix multiplication

- Why cuBLAS is the fastest?
- **Parallelism**: GPUs are inherently parallel processors, capable of running thousands of threads simultaneously.
- **Optimized Implementations**: The routines in cuBLAS are highly optimized for NVIDIA GPUs. They take advantage of specific features and architectures of these GPUs to achieve maximum performance.
- **Memory Hierarchy**: cuBLAS routines are designed to make efficient use of the GPU's memory hierarchy, including global memory, shared memory, and registers.
- **Asynchronous Execution**: cuBLAS operations can be performed asynchronously, meaning they can be launched and then return control to the CPU without waiting for the operation to complete.
- **Batched Operations**: cuBLAS supports batched operations, which allow for multiple operations to be performed simultaneously.

Activity: Calculating Work Done by a Force using cuBLAS

In physics, work done by a force is calculated as the dot product of the force vector and the displacement vector. In this activity, students are asked to use the `cublasSdot` function from cuBLAS to calculate the work done.

- Problem Statement: A force of 5 N is applied to an object, causing it to move in the same direction as the force for a distance of 10 m. Calculate the work done by the force.
- Vector Representation: Represent the force and displacement as vectors. Since the force and displacement are in the same direction, we can represent them as one-dimensional vectors:
  - Force vector, $F = [5]$
  - Displacement vector, $d = [10]$
- cuBLAS Implementation: Use the `cublasSdot` function to calculate the dot product of the force and displacement vectors.
- Use the provided template (`work_done.cu`)

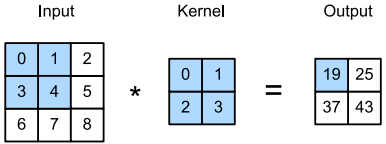
cuDNN

Convolutions for Images

- Let's see how convolutional layers work in practise
- CNNs are efficient to explore structures in image data $\Rightarrow$ images
- Convolution layer: an input tensor and a kernel tensor are combined to produce an output tensor through a cross-correlation operation

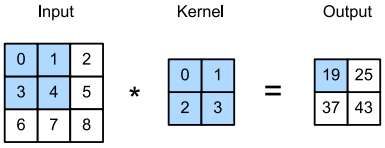
Convolutions for Images

- Let's see how convolutional layers work in practise
- CNNs are efficient to explore structures in image data ⇒ images
- Convolution layer: an input tensor and a kernel tensor are combined to produce an output tensor through a cross-correlation operation
- Let's ignore channels for now
  - Input: 2D tensor with shape 3×3
  - Kernel: 2×2 ⇒ kernel window (or convolution window)



Convolutions for Images

- Two-dimensional cross-correlation operation: start from upper-left
- Slide the **kernel window** across the input tensor, both from left to right and top to bottom
- Element-wise multiplication ⇒ single scalar value



$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$$

$$1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25$$

$$3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37$$

$$4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43$$

Convolution Operation

- Generally, the output size is given by
  - the input size $n_h \times n_w$ minus the
  - the size of the convolution kernel $k_h \times k_w$

$$(n_h - k_h + 1) \times (n_w - k_w + 1)$$

Convolution Operation

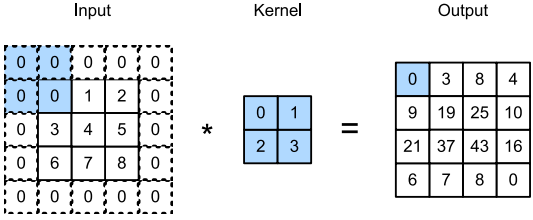
- Generally, the output size is given by
  - the input size $n_h \times n_w$ minus the
  - the size of the convolution kernel $k_h \times k_w$

$$(n_h - k_h + 1) \times (n_w - k_w + 1)$$

- This is the case since we need enough space to "shift" the convolution kernel across the image
- Next, we see how to keep the size unchanged by padding the image with zeros around its boundary so that there is enough space to shift the kernel

Padding

- Example:
  - An 3×3 image increased to 5×5 with padding
  - The corresponding output then increases to a 4×4 matrix

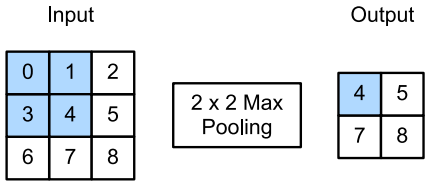


- The shaded portions are the first output element

$$0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$$

Max-pooling

- Start from the upper-left of the input tensor and sliding across the input tensor from left to right and top to bottom
- At each location that the pooling window hits, it computes the maximum



$$\max(0, 1, 3, 4) = 4$$

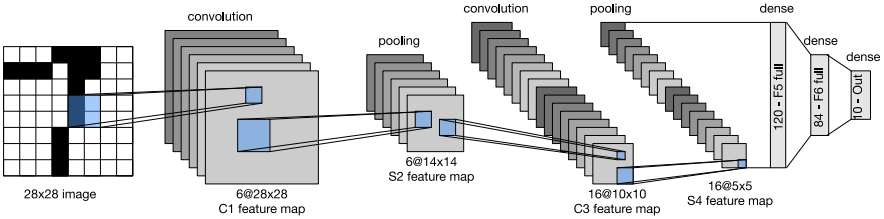
$$\max(1, 2, 4, 5) = 5$$

$$\max(3, 4, 6, 7) = 7$$

$$\max(4, 5, 7, 8) = 8$$

LeNet CNN

- Basic units in each convolutional block:
  - convolutional layer with 5×5 kernel
  - sigmoid activation function (ReLU was discovered later)
  - average pooling operation (max-pooling was discovered later)



cuDNN

- **Deep Learning Primitives**: cuDNN stands for CUDA Deep Neural Network library.
- It provides highly optimized primitives, or building blocks, for deep learning.
  - convolution
  - pooling
  - normalization
  - activation layers.

cuDNN

- **Deep Learning Primitives**: cuDNN stands for CUDA Deep Neural Network library.
- It provides highly optimized primitives, or building blocks, for deep learning.
  - convolution
  - pooling
  - normalization
  - activation layers.
- **GPU-Accelerated**: Like other CUDA libraries, cuDNN is designed to be run on NVIDIA GPUs.

cuDNN

- **Deep Learning Primitives**: cuDNN stands for CUDA Deep Neural Network library.
- It provides highly optimized primitives, or building blocks, for deep learning.
  - convolution
  - pooling
  - normalization
  - activation layers.
- **GPU-Accelerated**: Like other CUDA libraries, cuDNN is designed to be run on NVIDIA GPUs.
- **Integration with Frameworks**: cuDNN is used by many popular deep learning frameworks, including TensorFlow, PyTorch, and Caffe
  - when you train a deep learning model using one of these frameworks on an NVIDIA GPU, you're likely using cuDNN under the hood.

What is cuDNN good for?

- **Accelerating Deep Learning**: cuDNN is designed to accelerate the training and inference of deep neural networks.
- **Supporting Various Neural Network Architectures**: cuDNN supports many different types of neural networks, including convolutional neural networks (CNNs), recurrent neural networks (RNNs), and transformer networks.
- **Integration with Deep Learning Frameworks**: cuDNN is integrated into many popular deep learning frameworks, such as TensorFlow, PyTorch, and Caffe.
- **Efficient Use of GPU Resources**: cuDNN is designed to make efficient use of the parallel processing capabilities of NVIDIA GPUs.
- **Ease of Use**: While cuDNN provides a lot of flexibility and control, it's also designed to be easy to use.
- Official documentation

cuDNN program structure 1/2

- **Include the cuDNN library**: In your source code, you need to include the cuDNN header file using `#include <cudnn.h>`.
- **Allocate memory**: This will involve both CPU memory (using something like `malloc` in C) and GPU memory (using `cudaMalloc`).
- **Initialize cuDNN**: Create a cuDNN handle with `cudnnCreate`. This handle is used in subsequent calls to cuDNN functions.
- **Set up tensor descriptors**: Use the `cudnnCreateTensorDescriptor` function to create a descriptor for each tensor you will be using (input, output, etc.).
- **Set up convolution descriptors**: If you're doing a convolution operation, use `cudnnCreateConvolutionDescriptor` and `cudnnSetConvolution2dDescriptor` or similar functions to set up the convolution operation.

cuDNN program structure 2/2

- **Choose an algorithm**: For some operations, such as convolution, cuDNN provides several algorithms that can be used. Use `cudnnGetConvolutionForwardAlgorithm` or a similar function to choose an algorithm.
- **Allocate workspace**: Some cuDNN functions require a workspace: an area of GPU memory that they can use for intermediate calculations. Use `cudnnGetConvolutionForwardWorkspaceSize` or a similar function to determine how much workspace is needed, and then allocate that much memory with `cudaMalloc`.
- **Perform the operation**: Call the cuDNN function that performs the operation you want, such as `cudnnConvolutionForward` for a convolution operation.
- **Retrieve the results**: If the operation produces output (like a convolution operation does), use `cudaMemcpy` to copy the results from GPU memory back to CPU memory.
- **Clean up**: When you're done with cuDNN, use `cudnnDestroyTensorDescriptor`, `cudnnDestroyConvolutionDescriptor`, and cudnnDestroy to free up the resources you've used. Also use `cudaFree` to free any GPU memory you've allocated.

Why are descriptors important in cuDNN

- **Data Abstraction**: Descriptors provide an abstract representation of the data, allowing cuDNN functions to operate on a wide variety of data layouts and types without needing to know the specifics of the data.
- **Memory Efficiency**: By describing the data without holding the actual data, descriptors enable efficient use of memory.
- **Performance Optimization**: Descriptors allow cuDNN to make performance optimizations based on the properties of the data. For example, cuDNN can choose different algorithms for different tensor shapes or data types, leading to faster execution times
- **Code Simplification**: Using descriptors can simplify the code by encapsulating complex data properties.
- **Consistency**: Descriptors provide a consistent interface across different cuDNN functions.

Example: Simple example

- Check the file `cudnn_simple.cu`. It is a simple example of a program that uses cuDNN to create a tensor descriptor and set its dimensions

Compilation and execution

```
$ nvcc  -o cudnn_simple cudnn_simple.cu -lcudnn
$ ./cudnn_simple
```

Example: Simple example

- Check the file `cudnn_simple.cu`. It is a simple example of a program that uses cuDNN to create a tensor descriptor and set its dimensions

Compilation and execution

```
$ nvcc  -o cudnn_simple cudnn_simple.cu -lcudnn
$ ./cudnn_simple
```

- Initializes cuDNN with `cudnnCreate`
- Creates a tensor descriptor with `cudnnCreateTensorDescriptor`.
- Sets the dimensions of the tensor to $1 \times 1 \times 1 \times 1$ with `cudnnSetTensor4dDescriptor`.
  - `CUDNN_TENSOR_NCHW` parameter specifies the order (batch size, number of channels, height, width),
- Cleans up by destroying the tensor descriptor and the cuDNN handle

Example: 1D convolution

- Check the file `conv1d.cu`. It performs 1D convolution using the CPU, CUDA, and cuDNN, and prints the execution time for each method

Compilation and execution

```
$ nvcc  -o conv1d conv1d.cu -lcudnn
$ ./conv1d
```

Example: 1D convolution

- Check the file `conv1d.cu`. It performs 1D convolution using the CPU, CUDA, and cuDNN, and prints the execution time for each method

Compilation and execution

```
$ nvcc  -o conv1d conv1d.cu -lcudnn
$ ./conv1d
```

- Identify the following important parts:
  - The parameter that defines the size of the matrices
  - Memory allocation for CPU
  - Memory allocation for GPU
  - Memory transfers to GPU
  - Initialization of descriptors for cuDNN
  - Identify the functions used in cuDNN

Example: 1D convolution

N = 1000000, M = 100

```
$ ./conv1d
CUDA Time:  0.002651
cuDNN Time: 0.000019
CPU Time:   0.248088
```

N = 10000000, M = 1000

```
$ ./conv1d
CUDA Time:  0.158909
cuDNN Time: 0.000011
CPU Time:   24.136710
```

Explain the differences in time and why cuDNN is the fastest

Activity

- Find the **activity** file `cublas_cudnn.cu`
- This activity combines cuBLAS and cuDNN
- The goal is to get two matrices
  - multiply them using cuBLAS
  - apply ReLU activation function at the result using cuDNN

Questions?