Setup

ssh to DAVINCI and grab a GPU node (if you don't already have one):

```
$ source /projects/k2i/hpc/scripts/login-gpu.sh
```

Setup your environment once you have a GPU node (***__this will take 10-15 minutes__***):

```
$ source /projects/k2i/hpc/scripts/gpu-day-two-setup.sh
```

You should already have the code from yesterday, but if not you can grab it from Github:

```
$ git clone git@github.com:agrippa/hpc-bootcamp.git $HOME/bootcamp-gpu
```

Useful API references for today's projects:

1. Numba: http://numba.pydata.org/numba-doc/latest/index.html
2. Tensorflow: https://www.tensorflow.org/api_docs/python

Slides are accessible in your browser in Github and Box:

Github: https://github.com/agrippa/hpc-bootcamp Box: http://bit.ly/hpc-bc19

GPU Accelerated Computing (Day 2)
Productive GPU Programming

Max Grossman
Habanero Extreme Scale Software Research Group, Rice University
Principal & Co-Founder, 7pod Technologies
Author, Professional CUDA C Programming

Outline

1. Cons of CUDA
2. CUDA Libraries
3. Numba
4. Tensorflow

About Me

Habanero Research Group

- Runtime Systems for HPC (“Big Compute”)
- Data Analytics (“Big Data”)
- Compilers and Programmer Tools

Co-Founder, 7pod Technologies (7pod.tech)

- Consulting on new applications of data analytics and high performance computing
- HPC and data analytics training

Cons of CUDA

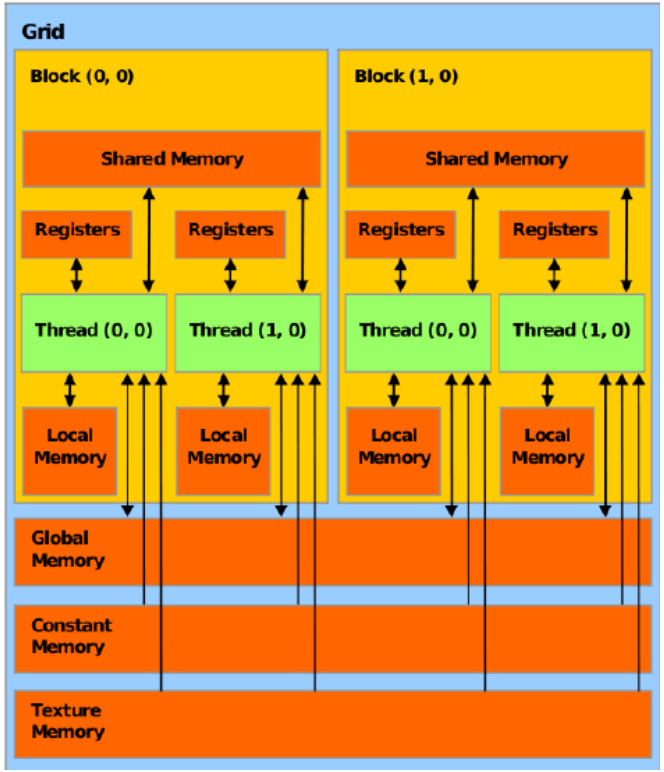
Massive Number of New Concepts

Memory management – manually managing multiple non-coherent address spaces.

Memory hierarchy – what the &#^$ is texture memory? Shared? Constant?

Thread hierarchy - Thread blocks, grids of thread blocks.

…



The most common CUDA illustration… even it is bewilderingly complex.

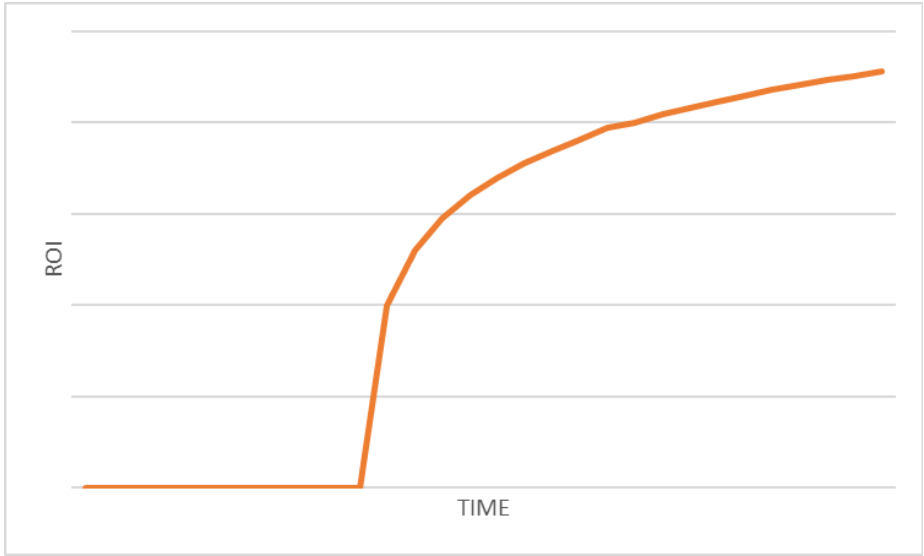
Massive Initial Investment

Investment for porting non-trivial kernels is a step function.

- Initial ports are often much slower than their CPU counterparts.
- Performance benefits only realized with determined, profile-driven optimization

Return-on-investment often unclear.

- The most common CUDA question: will kernel foo run faster on a GPU?
- For simple kernels the answer can be obvious, for more complex ones less so.



Language

Much/most of the world's application code and application programmers do not use C/C++.

Makes it harder to find CUDA developers, harder to integrate CUDA kernels with the rest of your code base.

2018 Stack Overflow Developer Survey

Productive GPU Programming

Many commercial, open source, and research products available today that make GPU programming easier

- Generally there is a tradeoff in either features or generalizability

Today we'll explore three different approaches with hands-on exercises for each:

1. **CUDA Libraries**: NVIDIA-supported domain-specific libraries
2. **Numba**: Program CUDA devices in Python
3. **Tensorflow**: Machine learning library commonly used to target GPUs

CUDA Libraries

CUDA Libraries

CUDA Libraries offer high-level, high performance, domain-specific APIs for GPUs.

Examples:

1. CUSPARSE: Sparse linear algebra
2. CUBLAS: Dense linear algebra
3. CUDNN: Deep Neural Networks
4. CUFFT: Fast Fourier Transforms

Take advantage of expert-tuned kernels without writing a line of kernel code.

Many CUDA libraries emulate their host counterparts (e.g. CUBLAS and BLAS).

Common Library Workflow

1. Create a library-specific handle that manages contextual information useful for the library's operation.
   - Many CUDA Libraries have the concept of a handle which stores opaque library-specific information on the host which many library functions access
   - Programmer's responsibility to manage this handle
   - For example: `cublasHandle_t`, `cufftHandle`, `cusparseHandle_t`, `curandGenerator_t`

1. Allocate device memory for inputs and outputs to the library function.
   - Use `cudaMalloc` as usual

Common Library Workflow

3. If inputs are not already in a library-supported format, convert them to be accessible by the library.
   - Many CUDA Libraries only accept data in a specific format
   - For example: column-major vs. row-major arrays

4. Populate the pre-allocated device memory with inputs in a supported format.
   - In many cases, this step simply implies a `cudaMemcpy` or one of its variants to make the data accessible on the GPU
   - Some libraries provide custom transfer functions, for example: `cublasSetVector` optimizes strided copies for the CUBLAS library

Common Library Workflow

5. Configure the library computation to be executed.
   - In some libraries, this is a no-op
   - Others require additional metadata to execute library computation correctly
   - In some cases this configuration takes the form of extra parameters passed to library functions, others set fields in the library handle

6. Execute a library call that offloads the desired computation to the GPU.
   - No GPU-specific knowledge required

Common Library Workflow

7. Retrieve the results of that computation from device memory, possibly in a library-determined format.
   - Again, this may be as simple as a `cudaMemcpy` or require a library-specific function

8. If necessary, convert the retrieved data to the application's native format.
   - If a conversion to a library-specific format was necessary, this step ensures the application can now use the calculated data
   - In general, it is best to keep the application format and library format the same, reducing overhead from repeated conversions

Common Library Workflow

9. Release CUDA resources.
   - Includes the usual CUDA cleanup (`cudaFree`, `cudaStreamDestroy`, etc) plus any library-specific cleanup

10. Continue with the remainder of the application.

Common Library Workflow

- Not all libraries follow this workflow, and not all libraries require every step in this workflow
  - In fact, for many libraries many steps are skipped
  - Keeping this workflow in mind will help give you context on what the library might be doing behind the scenes and where you are in the process

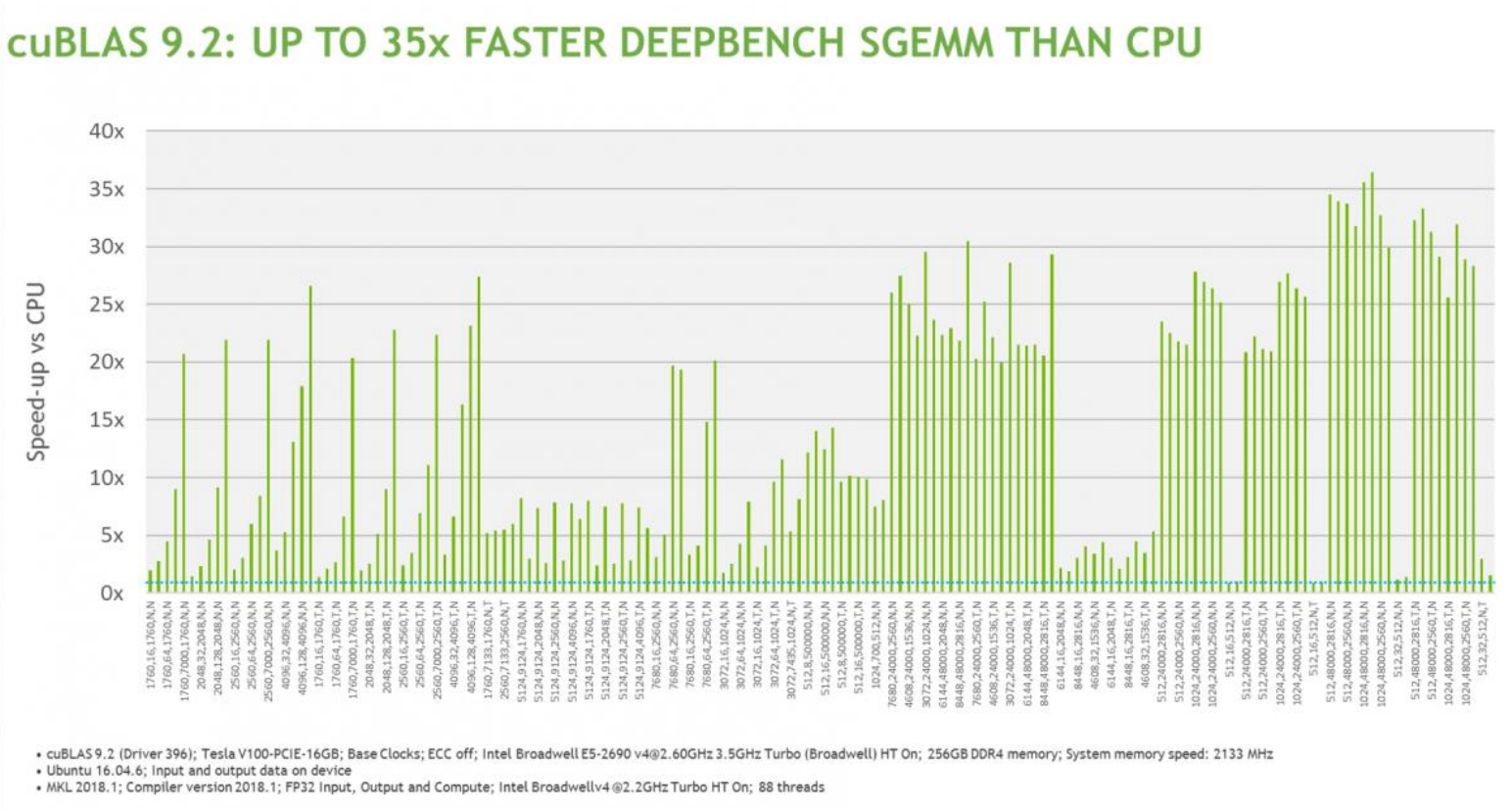
- Next, we'll look at one simple illustration of this workflow

cuBLAS

- cuBLAS is a port of a popular linear algebra library, BLAS

- cuBLAS (like BLAS) splits its subroutines into multiple levels based on data types processed:
  - Level 1: vector-only operations (e.g. vector addition)
  - Level 2: matrix-vector operations (e.g. matrix-vector multiplication)
  - Level 3: matrix-matrix operations (e.g. matrix multiplication)

cuBLAS Performance



https://developer.nvidia.com/cublas

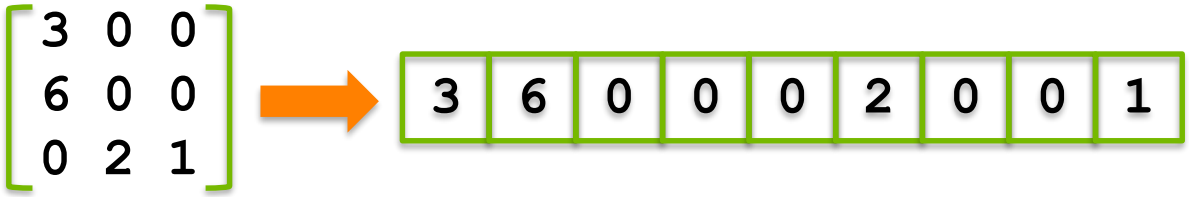
cuBLAS Portability

Porting to cuBLAS from BLAS is a straightforward process. In general, it requires:

- Adding device memory allocation/freeing (`cudaMalloc`, `cudaFree`)
- Adding device transfer functions (`cublasSetVector`, `cublasSetMatrix`, etc)
- Transform library routine calls from BLAS to cuBLAS (e.g. `cblas_sgemv` ➔ `cublasSgemv`)

cuBLAS Ideosyncracies

- For legacy compatibility, cuBLAS operates on column-major matrices



- cuBLAS also has a legacy API which was dropped in CUDA 4.0, we'll be looking at the new cuBLAS API
  - If you find cuBLAS code that doesn't quite match up, you may be looking at the old cuBLAS API

cuBLAS Data Management

Device memory in cuBLAS is allocated as you're used to: `cudaMalloc`

Transferring data to the device uses cuBLAS-specific functions:
`cublasGetVector`/`cublasSetVector` and
`cublasGetMatrix`/`cublasSetMatrix`

cuBLAS Data Management

- Example:

```
cublasStatus_t cublasSetVector(int n, int elemSize,
    const void *x, int incx, void *y, int incy);
```

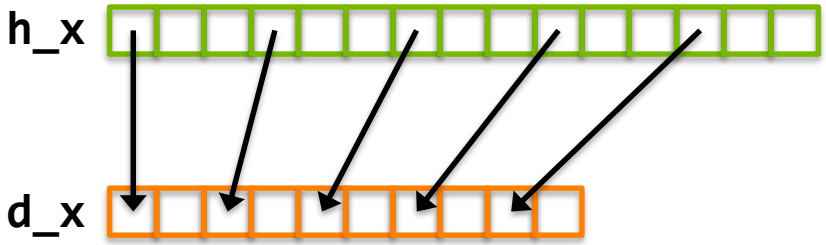
where:

- n is the number of elements to transfer to the GPU
- `elemSize` is the size of each element (e.g. `sizeof(int)`)
- x is the vector on the host to copy from
- `incx` is a stride in `x` of the array cells to transfer to
- y is the vector on the GPU to copy to
- `incy` is a stride in `y` of the array cells to transfer to

cuBLAS Data Management

- Example:

```
cublasSetVector(5, sizeof(int), h_x, 3, d_x, 2);
```



cuBLAS Data Management

Similarly:

```
cublasStatus_t cublasSetMatrix(int rows, int cols,
    int elemSize, const void *A, int lda, void *B, int ldb);
```

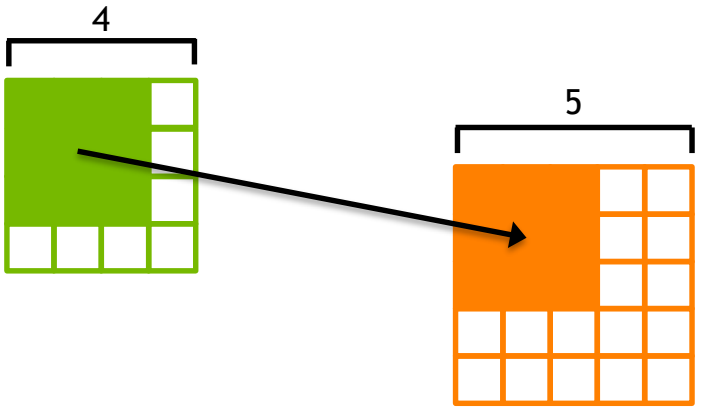
where:

- `rows` is the number of rows in a matrix to copy
- `cols` is the number of cols in a matrix to copy
- `elemSize` is the size of each cell in the matrix (e.g. `sizeof(int)`)
- `A` is the source matrix on the host
- `lda` is the number of rows in the underlying array for `A`
- `B` is the destination matrix on the GPU
- `ldb` is the number of rows in the underlying array for `B`

cuBLAS Data Management

- Similarly:

```
cublasSetMatrix(3, 3, sizeof(int), h_A, 4, d_A, 5);
```



Hands On - cuBLAS

Let's take a look at a simple cuBLAS example that performs a matrix-vector multiplication. This example uses 6 of the 10 steps in the common library workflow:

1. Create a cuBLAS handle using `cublasCreateHandle`
2. Allocate device memory for inputs and outputs using `cudaMalloc`
3. Populate device memory using `cublasSetVector`, `cublasSetMatrix`
4. Call `cublasSgemv` to run matrix-vector multiplication on the GPU
5. Retrieve results from the GPU using `cublasGetVector`
6. Release CUDA and cuBLAS resources using `cudaFree`, `cublasDestroy`

Hands On - cuBLAS

The `10_cublas/` folder contains a template program with helpful TODOs.

Try completing these TODOs.

Compile and run the program to verify the correctness of the output – this example will print a percent error on the generated output. Any value below 3% is acceptable.

Review – CUDA Libraries

CUDA Libraries offer high-level, high performance, domain-specific APIs for GPUs.

Deliberately designed to facilitate porting from legacy, host-only libraries.

LIBRARY NAME	DOMAIN
NVIDIA cuFFT	Fast Fourier Transforms
NVIDIA cuBLAS	Linear Algebra (BLAS Library)
CULA Tools	Linear Algebra
MAGMA	Next-gen Linear Algebra
IMSL Fortran Numerical Library	Mathematics and Statistics
NVIDIA cuSPARSE	Sparse Linear Algebra
NVIDIA CUSP	Sparse Linear Algebra and Graph Computations
AccelerEyes ArrayFire	Mathematics, Signal and Image Processing, and Statistics
NVIDIA cuRAND	Random Number Generation
NVIDIA NPP	Image and Signal Processing
NVIDIA CUDA Math Library	Mathematics
Thrust	Parallel Algorithms and Data Structures
HiPLAR	Linear Algebra in R
Geometry Performance Primitives	Computational Geometry
Paralution	Sparse Iterative Methods
AmgX	Core Solvers

Numba

Numba Overview

Numba enables programming CUDA devices in Python.

Uses Numpy arrays for data structures.

Uses just-in-time CUDA code generation from Python to create kernels.

```
from numba import cuda
import numpy as np

@cuda.jit
def vecadd(A, B, C):
    tid = cuda.blockIdx.x * cuda.blockDim.x + \
        cuda.threadIdx.x
    C[tid] = A[tid] + B[tid]

threads_per_block = 256
blocks_per_grid = 10
N = blocks_per_grid * threads_per_block

A = np.random.rand(N)
B = np.random.rand(N)
C = np.zeros(N)

vecadd[blocks_per_grid, threads_per_block](A, B, C)
```

Quick Numpy Introduction

Foundational library for numerical computing in Python.

Most commonly used for creating, slicing, manipulating N-dimensional arrays.

Includes other utilities, e.g. random number generation.

Numba uses Numpy arrays to store data copied to and from CUDA devices.

```
import numpy as np

# Create a 100-element array of zeros
arr = np.zeros(100)

# Read the 4th element of arr
… = arr[3]

# Create a 100x100 array of zeros
arr = np.zeros((100, 100))

# Write an element in the array
arr[4, 4] = …

# Copy the 4th column to the 3rd row
arr[2, :] = arr[:, 3]
```

Writing Numba Kernels

Declared as regular Python methods.

Decorated with `@cuda.jit`

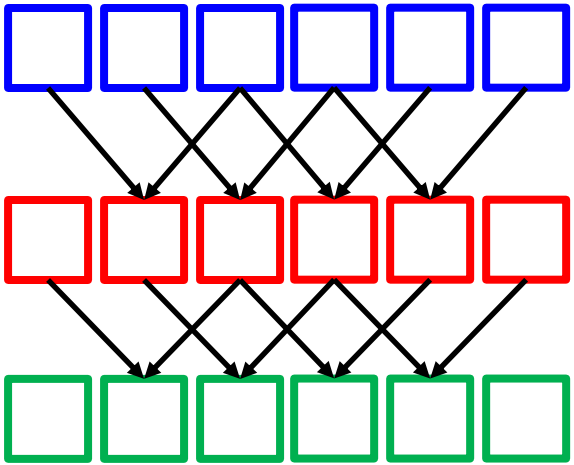
Launched with threads per block, blocks per grid.

Have access to special `blockIdx`, `blockDim`, `threadIdx` variables.

```
@cuda.jit
def vecadd(A, B, C):
    tid = cuda.blockIdx.x * cuda.blockDim.x + \
        cuda.threadIdx.x
    C[tid] = A[tid] + B[tid]
```

Hands On – 1D Iterative Averaging

```
for iter in range(niters):
    for i in range(1, N + 1):
        next[i] = curr[i - 1] + curr[i + 1] / 2.0
    tmp = curr
    curr = next
    next = tmp
```



Hands On – 1D Iterative Averaging

The `11_numba_baseline/` folder contains a reference Python implementation of 1D iterative averaging.

```
$ python 1d_iter_avg.py <N> <niters> # Try N=8000, niters=5000
```

Let's try a first port to Numba. There are helpful TODOs in `1d_iter_avg.py`.

`1d_iter_avg.py` writes an image file img.png plotting the results of the iterations, use these images to verify your port is correct (`eog img.png`).

What kind of performance do you see (try repeating each experiment multiple times)?

Hands On – 1D Iterative Averaging

Dataset	Performance	Improvement Relative to Python
Python	195.64 iters / s	
Numba	420.28 iters / s	2.15x
CUDA	353,085.61 iters / s	1,804.77x

Massive performance discrepancy between CUDA and numba.

nvprof works just as well on numba programs!

Let's start with nvprof's Summary Mode:

```
$ nvprof python 1d_iter_avg.py 8000 5000
```

Hands On – 1D Iterative Averaging

	CUDA memcpy DtoH	CUDA memcpy HtoD	Kernel
numba	43.50%	51.40%	5.10%
CUDA	0.10%	0.11%	99.79%

numba is spending the majority of time copying data around.

Numba Memory Management

Numba is by default conservative in its memory management.

```
for iter in range(niters):
    kernel[blocks_per_grid, threads_per_block](nxt, curr, len(curr) - 2)
```

```
for iter in range(niters):
    cudaMemcpy(d_nxt, nxt, len(nxt), cudaMemcpyHostToDevice)
    cudaMemcpy(d_curr, curr, len(curr), cudaMemcpyHostToDevice)

    kernel[blocks_per_grid, threads_per_block](nxt, curr, len(curr) - 2)

    cudaMemcpy(nxt, d_nxt, len(nxt), cudaMemcpyDeviceToHost)
    cudaMemcpy(curr, d_curr, len(curr), cudaMemcpyDeviceToHost)
```

Numba Memory Management

Numba exposes simple APIs for explicit memory management of CUDA devices.

Copy to device:

```
# Create a new numpy array and
# copy it to the device
arr = np.arange(100)
d_arr = cuda.to_device(arr)
```

Copy from device:

```
# Copy into a new host array
h_arr = d_arr.copy_to_host()

# Copy into an existing host array
existing_arr = np.zeros(100)
d_arr.copy_to_host(existing_arr)
```

Device array objects can then be passed to Numba kernels to avoid transfers:

```
kernel[blocks_per_grid, threads_per_block](d_arr)
```

Hands On – 1D Iterative Averaging

Try extending the template in the `12_numba_mem/` folder to use explicit memory management in numba.

Re-measure performance with timers and nvprof, what changes do we see?

Hands On – 1D Iterative Averaging

Try extending the template in the `12_numba_mem/` folder to use explicit memory management in numba.

Dataset	Performance	Speedup Relative to Python
Python	195.64 iters / s	
Numba	2,595.46 iters / s	13.27x
CUDA	353,085.61 iters / s	1,804.77x

	CUDA memcpy DtoH	CUDA memcpy HtoD	Kernel
numba	0.09%	0.21%	99.70%
CUDA	0.10%	0.11%	99.79%

Hands On – 1D Iterative Averaging

Performance has clearly improved as a result of reduced data transfer, but still is much slower than CUDA.

Let's look at the following metrics in nvprof for both numba and CUDA, in order to start diagnosing the performance delta

1. ipc
2. sm_efficiency
3. alu_fu_utilization
4. gld_transactions
5. gst_transactions
6. gld_transactions_per_request
7. gst_transactions_per_request
8. gld_efficiency
9. gst_efficiency

Hands On – 1D Iterative Averaging

Metric	numba	CUDA
ipc	0.829075	0.463110
sm_efficiency	30.76%	21.68%
alu_fu_utilization	Mid (4)	Low (2)
gld_transactions	1680	840
gst_transactions	1120	560
gld_transactions_per_request	3.36	1.68
gst_transactions_per_request	4.48	2.24
gld_efficiency	59.52%	59.52%
gst_efficiency	80.00%	80.00%

gld/gst efficiency is actually reasonable for auto-generated code…

Lower IPC for CUDA likely due to more concise kernel.

Nothing that explains 100x performance difference.

Hands On – 1D Iterative Averaging

Let’s take a look at the performance outside the kernel.

Analyze the sequence of kernel calls and copies performed by numba and CUDA using:

```
$ nvprof --print-gpu-trace ...
```

Hands On – 1D Iterative Averaging

Let's take a look at the performance outside the kernel.

CUDA

Start	Duration	Grid Size	Block Size	Regs*	Name
202.91ms	1.7930us	(8 1 1)	(256 1 1)	8	kernel(float*, float*, int)
202.92ms	1.0720us	(8 1 1)	(256 1 1)	8	kernel(float*, float*, int)
202.93ms	1.0560us	(8 1 1)	(256 1 1)	8	kernel(float*, float*, int)

0.0090 ms between kernels

numba

Start	Duration	Grid Size	Block Size	Regs*	Name
371.09ms	2.5560us	(8 1 1)	(256 1 1)	12	cudapy::__main__::kernel$241
371.40ms	1.5950us	(8 1 1)	(256 1 1)	12	cudapy::__main__::kernel$241
371.67ms	1.5530us	(8 1 1)	(256 1 1)	12	cudapy::__main__::kernel$241

0.2639 ms between kernels

Review – Numba

Numba makes rapid prototyping of CUDA-accelerated programs easy.

Quick path from Python to GPUs, with incremental, profile-driven optimization possible.

Dataset	Performance	Speedup Relative to Python
Python	195.64 iters / s	
Numba	2,595.46 iters / s	13.27x
CUDA	353,085.61 iters / s	1,804.77x

Most relevant for:

- Quick validation that GPUs may yield speedup
- Long-running kernels, where overheads of Python can be hidden
- Heavy Python kernels that do not make use of advanced Python features

Tensorflow

Tensorflow

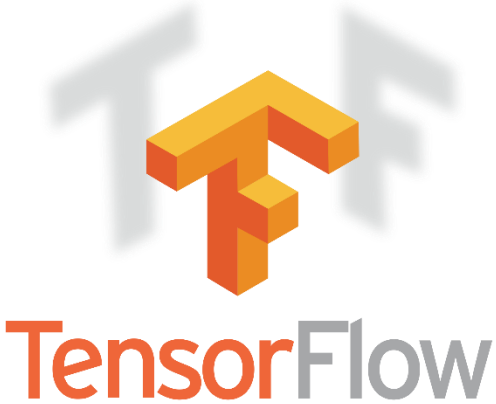
Open source, free, portable library from Google – expresses computation as a graph of operations.

Supports multiple backends to target GPUs, CPUs, TPUs, etc.

Supports an easy-to-use Python frontend.

Most commonly used for training and deploying neural nets.

Supports execution on everything from mobile devices to supercomputers.



https://github.com/tensorflow/tensorflow

Tensorflow

Core abstraction of Tensorflow is the computation graph.

- Defines the operations to perform and the dependencies between them.

Let's consider a simple 3-component dot product example:

```
a = (0, 1, 2)
b = (3, 4, 5)

c0 = a[0] * b[0]
c1 = a[1] * b[1]
c2 = a[2] * b[2]

s = c0 + c1 + c2
```

Procedural Python

Tensorflow

Core abstraction of Tensorflow is the computation graph.

- Defines the operations to perform and the dependencies between them.

Let's consider a simple 3-component dot product example:

```
a = (0, 1, 2)
b = (3, 4, 5)

c0 = a[0] * b[0]
c1 = a[1] * b[1]
c2 = a[2] * b[2]

s = c0 + c1 + c2
```

Procedural Python

```
a = (0, 1, 2)
b = (3, 4, 5)

c1 = a[1] * b[1]
c0 = a[0] * b[0]
c2 = a[2] * b[2]

s = c0 + c1 + c2
```

Does the answer change?

Tensorflow

Core abstraction of Tensorflow is the computation graph.

- Defines the operations to perform and the dependencies between them.

Let's consider a simple 3-component dot product example:

```
a = (0, 1, 2)
b = (3, 4, 5)

c0 = a[0] * b[0]
c1 = a[1] * b[1]
c2 = a[2] * b[2]

s = c0 + c1 + c2
```

Procedural Python

```
a = (0, 1, 2)
b = (3, 4, 5)

c1 = a[1] * b[1]
c0 = a[0] * b[0]
c2 = a[2] * b[2]

s = c0 + c1 + c2
```

Does the answer change?

No!

Indicates that the fundamental computation graph is over-restricted by step-by-step execution.

Tensorflow

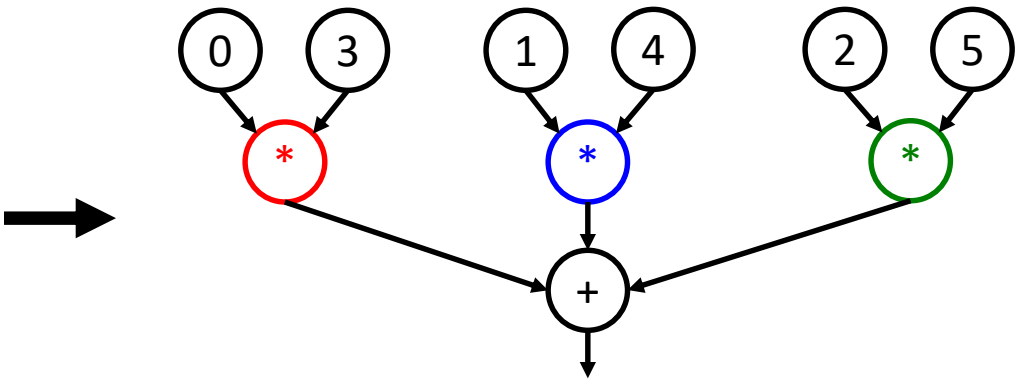
Core abstraction of Tensorflow is the computation graph.

- Defines the operations to perform and the dependencies between them.

Let's consider a simple 3-component dot product example:

```
a = (0, 1, 2)
b = (3, 4, 5)

c0 = a[0] * b[0]
c1 = a[1] * b[1]
c2 = a[2] * b[2]

s = c0 + c1 + c2
```

Procedural Python



Computation Graph

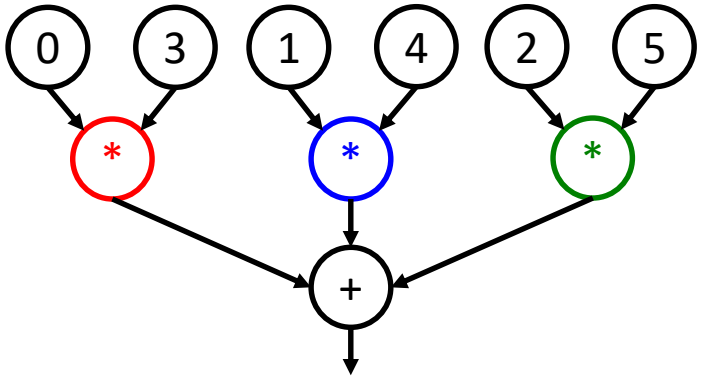
Tensorflow

A TF Graph is constructed once (i.e. compiled).

A TF Graph can then be executed multiple times with different inputs.

TF runtime will transparently execute this graph on available compute resources (e.g. GPU, CPU, TPU) without the programmer writing any kernels or data transfer.

While you can explicitly create graph objects to add operations to, TF also has a default graph which is always present (similar to the default CUDA stream).



Graph

Tensorflow

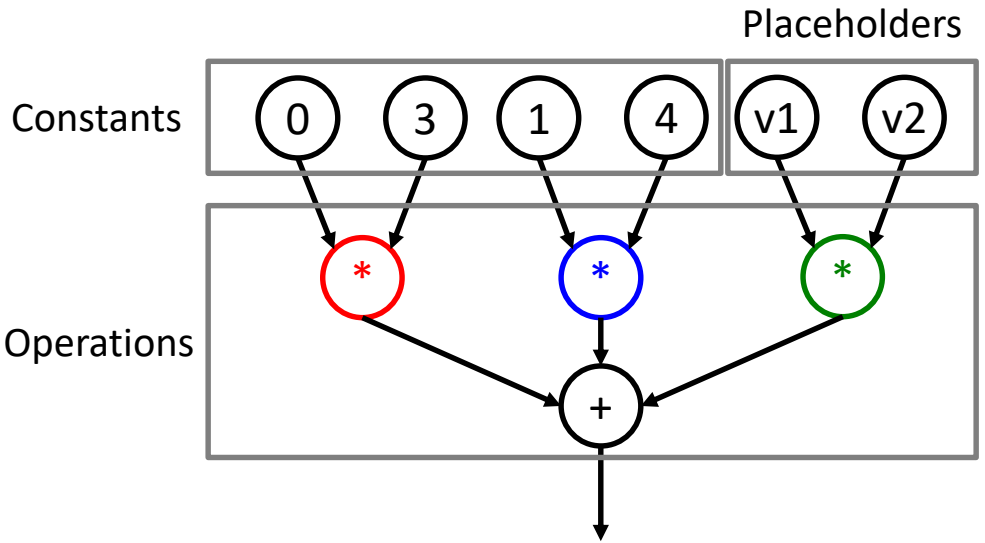
A TF graph is composed of four types of nodes: constants, placeholders, operations, variables.

Constant: zero inputs, fixed output.

Placeholder: placeholder for a value that must be supplied at execution.

Operation: accepts 1 or more inputs and produces an output.

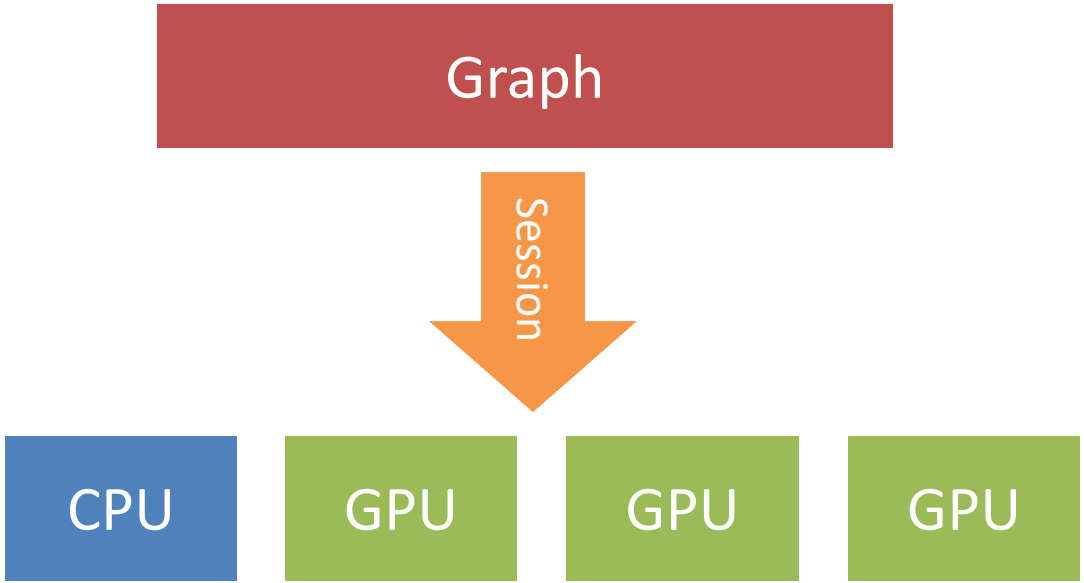
(Variables discussed later)



Tensorflow

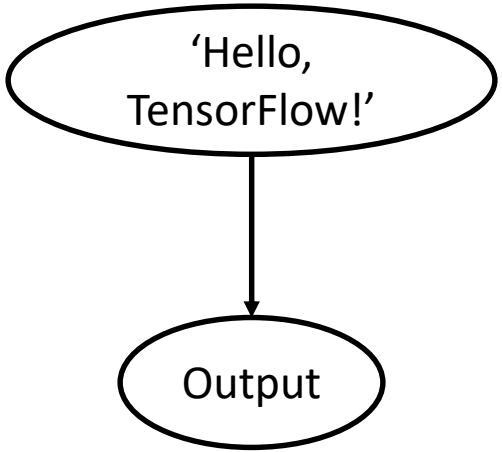
A TF Session encapsulates the state and operations of a TF graph.

- Responsible for mapping that graph to hardware resources.
- A TF graph cannot be run without a TF session



Hands On – Hello World in TF

```
import tensorflow as tf

hello = tf.constant('Hello, TensorFlow!')

sess = tf.Session()

print(sess.run(hello))
```



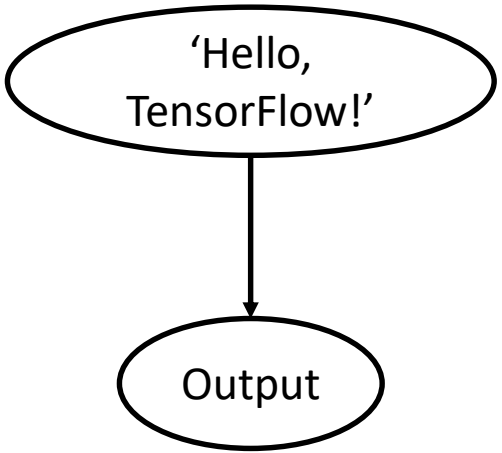
Inspired by: https://github.com/aymericdamien/TensorFlow-Examples

Hands On – Hello World in TF

To validate your environment, try running the provided `helloworld.py` under `13_tf_hello_world/`.

```
$ python helloworld.py
```

Python will print a couple of warnings, these can be safely ignored.



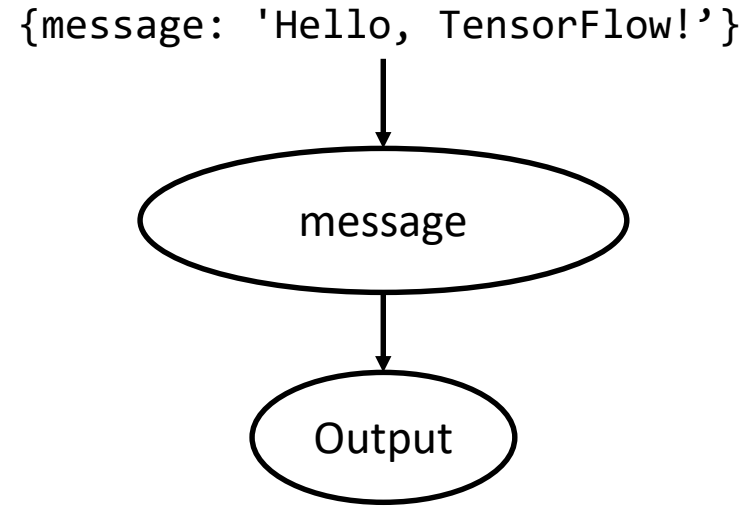
Caveat: While TF can run on GPUs, today we’ll simply be running it on CPUs. The code is the same, but unfortunately DAVINCI’s GPUs are not supported by TF.

Inspired by: https://github.com/aymericdamien/TensorFlow-Examples

Hands On – Hello World in TF

Consider `helloworld_placeholder.py` under `13_tf_hello_world/`.

Semantically identical to helloworld.py, but uses a placeholder to set the printed message.

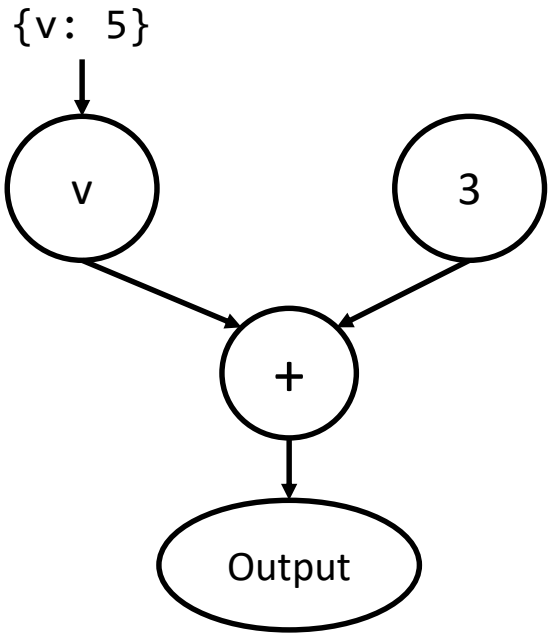



```
message = tf.placeholder(dtype = tf.string,
                         name = 'message')

sess = tf.Session()

print(sess.run(message, feed_dict = {message: 'Hello, TensorFlow!'}))
```

Inspired by: https://github.com/aymericdamien/TensorFlow-Examples

Hands On – Hello World in TF

Consider `helloworld_op.py` under `13_tf_hello_world/`.



Demonstrates the definition of an operation (+).

```
c = tf.constant(3)

v = tf.placeholder(dtype = tf.int32, name = 'v')

sum = tf.add(c, v)

sess = tf.Session()

sess.run(sum, feed_dict = {v: 5})
```

Inspired by: https://github.com/aymericdamien/TensorFlow-Examples

Machine Learning w/ Tensorflow

While Tensorflow can be used to execute general purpose computation graphs, it is most commonly used to perform supervised training of machine learning models.

A **supervised machine learning model** seeks to approximate some unknown function through observations of its inputs (X) and outputs (Y).

Given the following, what would you guess W and B are?

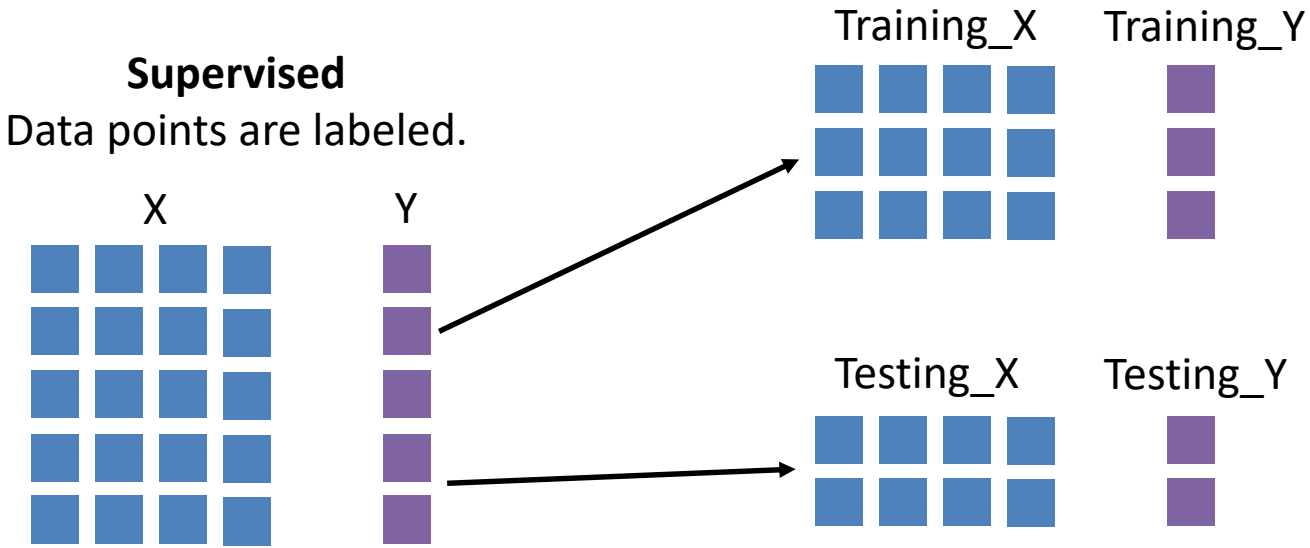
```
Y = F(X) = W * X + B
```

X	Y
1	2
2	4
3	6
4	8
5	10

Machine Learning w/ Tensorflow

In supervised learning, labeled data points are split into training and testing datasets.

- Training data teaches the model what the expected outputs are
- Testing data measures how well the model is approximating the hidden function



Machine Learning w/ Tensorflow

Supervised machine learning in TF generally takes the following steps:

1. Define the structure of a model as a TF graph.
2. Load our training dataset.
3. For a given number of epochs/iterations:
    1. Compute error of the model on the training data
    2. Update the model to reduce error

1. Y = W * X + B
2. X = [0, 1, 3, 6, 10]
   Y = [0, 2, 6, 12, 20]
3. err = MSE(calc_Y, Y)

   Update W, B

Machine Learning w/ Tensorflow

Supervised machine learning in TF generally takes the following steps:

1. Define the structure of a model as a TF graph.	1. Y = W * X + B	Uses TF Graph APIs
2. Load our training dataset.	2. X = [0, 1, 3, 6, 10] Y = [0, 2, 6, 12, 20]	Generally loaded into numpy arrays
3. For a given number of epochs/iterations:		
1. Compute error of the model on the training data	3. err = MSE(calc_Y, Y)	Easy to hand-code or use TF utilities
2. Update the model to reduce error	Update W, B	Handled by TF optimizers

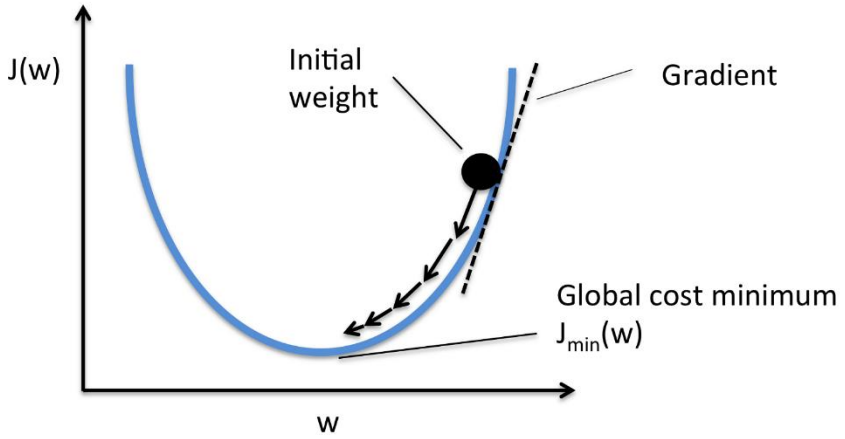
Tensorflow Optimizers

More generally, optimizers are used to minimize/maximize a function given constraints.

In ML, we generally want to minimize the error between our labels/expected values and the values computed by our model.

Optimization frameworks are used to update the variables of our model.

TF offers a variety of optimizers for just this.

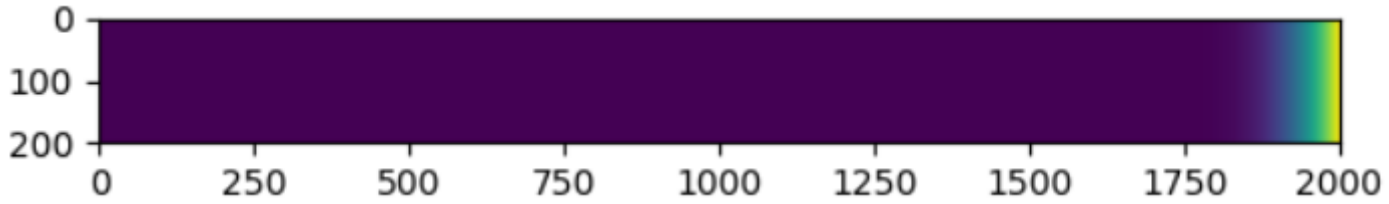


Hands On – Linear Regression in TF

Recall in the 1D Iterative Averaging example, each cell was updated according to:

```
arr[i] = 1/2 * arr[i - 1] + 1/2 * arr[i + 1]
```

1D iterative averaging has properties similar to some physical systems (e.g. heat propagation along a metal cylinder).



What if I gave you experimental temperature data from such a system? Could you use machine learning to recover/approximate the rules of that system?

Hands On – Linear Regression in TF

Consider the contents of the `14_tf_regress/` folder.

1. `X.bin`: Data sampled from an execution of `1d_iter_avg.py`, in the form `arr[i - 1]`, `arr[i]`, `arr[i + 1]`.
2. `Y.bin`: Data sampled from an execution of `1d_iter_avg.py`, the value of `arr[i]` computed from the corresponding values in `X.bin`.
3. `linear_regression.py`: An example TF program that loads these data files and trains a linear model with the structure `W1 * arr[i - 1] + W2 * arr[i + 1]`.

Let's try walking through the code together, and then train the model:

```
$ python linear_regression.py
```

Inspired by: https://github.com/aymericdamien/TensorFlow-Examples

Hands On – Linear Regression in TF

```
$ python linear_regression.py
...
Epoch: 39800 cost= 0.000000092 W= 0.52370435 0.48045552
Epoch: 39900 cost= 0.000000090 W= 0.5235196 0.4806075
Epoch: 40000 cost= 0.000000089 W= 0.52333724 0.4807585
Optimization Finished!
Training cost= 8.899919e-08 W= 0.52333724 0.4807585

Testing... (Mean square loss Comparison)
Testing cost= 2.9966046e-08
Absolute mean square loss difference: 5.903314e-08
```

0.52333724 * arr[i-1] * 0.4807585 * arr[i+1]

Low training cost, optimizer did a good job minimizing

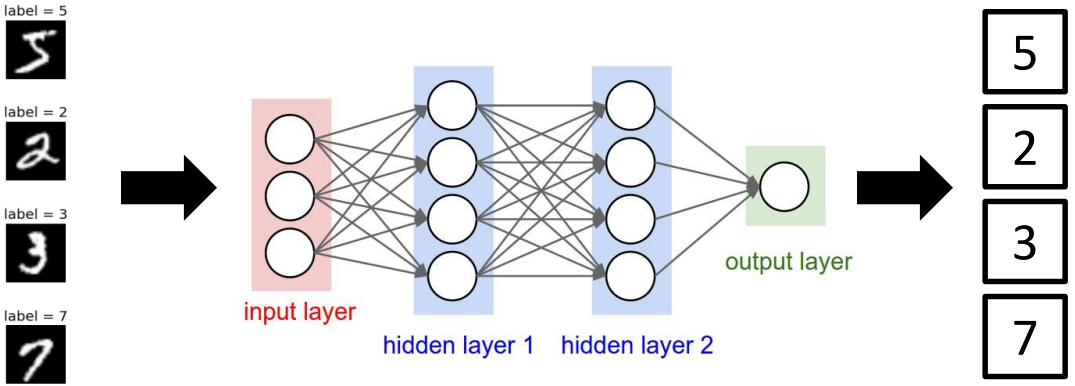
Low testing cost, suggests the model is general enough to work on unseen data

Tensorflow Neural Nets

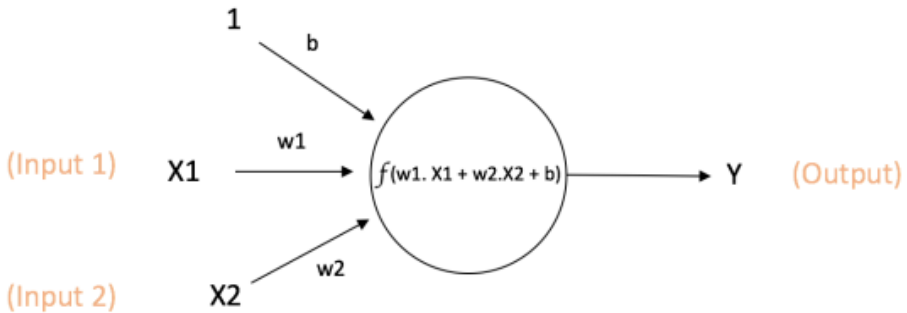
No TF tutorial is complete without neural nets…

Going into detail on all the variants of neural nets would require more than 1 day.

We'll look at a simple example of using dense neural nets to predict handwritten digits in the MNIST dataset.



What is a Neural Net?



Output of neuron = Y= f(w1. X1 + w2.X2 + b)

A single neuron

X = neuron's inputs
Y = neuron's output
B = bias
W = weights

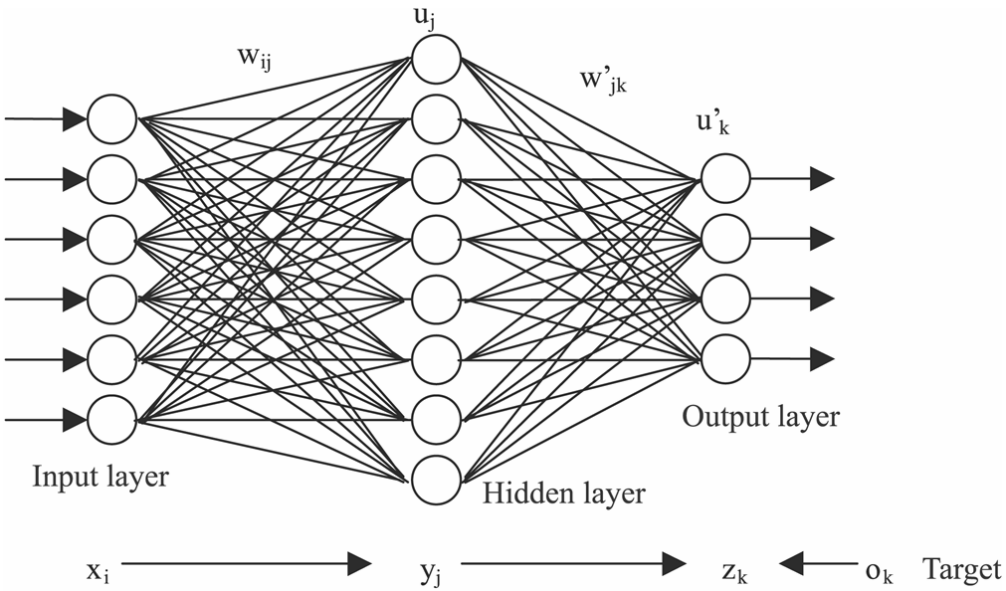
The Renaissance of Neural Nets

A neural net is a directed graph of neurons.

Power to accurately approximate a wide range of functions[1].

Only recently has training practically useful nets become feasible:

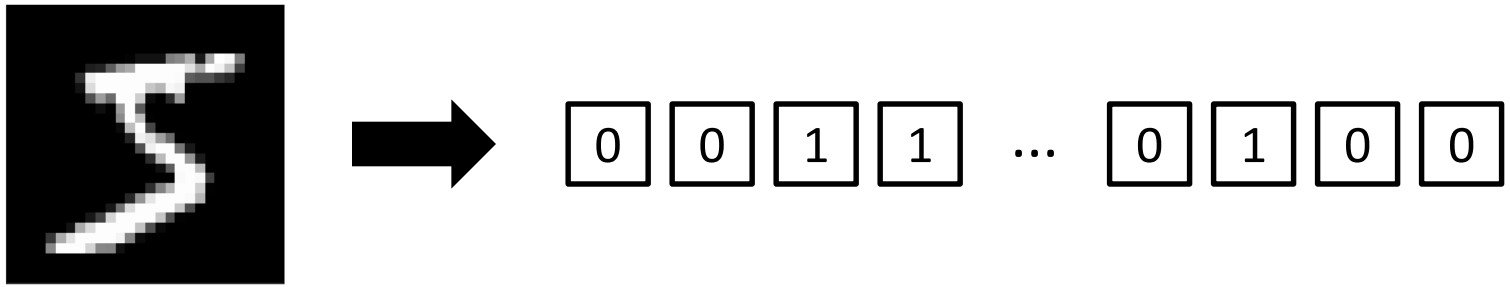
- Large, clean data sets.
- Improved computational and memory throughput, via GPUs.



1. "Universal Function Approximation by Deep Neural Nets with Bounded Width and ReLU Activations". Boris Hanin. https://arxiv.org/abs/1708.02691

Tensorflow Neural Nets

The MNIST dataset consists of handwritten digits (encoded as pixels) and their expected labels (i.e. the digit itself).



We want to train a neural net to predict the number these flattened vectors represent.

Hands On – Neural Nets in TF

`15_tf_nn/` contains an example code for training a neural net on the MNIST dataset:

```
$ python neural_network.py
```

This script will print a percent accuracy at completion (i.e. how many labels in the testing dataset match).

What accuracies are you able to achieve?

Hands On – Neural Nets in TF

Try experimenting with network structure.

The current model uses two layers, each of 256 neurons.

```
x = tf.placeholder('float', shape = (batch_size, num_input), name = 'images')
layer_1 = tf.layers.dense(x, n_hidden_1)
layer_2 = tf.layers.dense(layer_1, n_hidden_2)
out_layer = tf.layers.dense(layer_2, num_classes)
```

Does adding layers help? Does increasing the size of current layers help?

```
layer_2 = tf.layers.dense(layer_1, n_hidden_2)
```

Input layer

# of neurons in this layer

Hands On – Neural Nets in TF

Model	Epochs	Accuracy	Time
256 x 256	20	91.306%	41.60 s
256 x 256	80	91.677%	166.07 s
256 x 256 x 256	80	91.386%	211.22 s
256	80	91.907%	122.22 s
128	80	91.957%	81.76 s
16	80	92.037%	63.50 s
8	80	91.577%	51.57 s
4	80	87.049%	45.49 s
2	80	69.481%	47.54 s

Increasing epochs yields a slight improvement in accuracy.

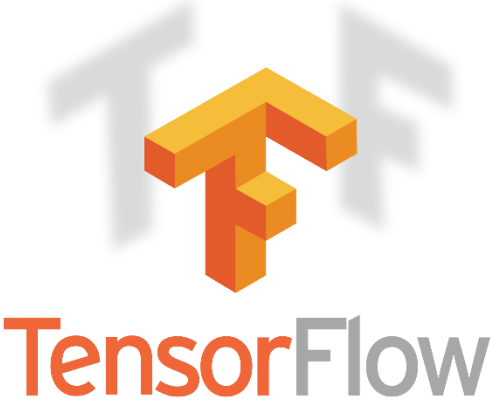
Increased model complexity doesn't yield improved results – complex models may yield better results, but with more variables they take longer to train.

Reduced model complexity is more accurate up to a point – fewer variables learn quicker.

Review – Tensorflow

High level library that supports general purpose computation, but is most commonly used to train and deploy supervised machine learning models.

Can transparently offload computation to GPUs – generally yields massive training performance improvements.



Supports a wide range of features we didn't cover:

- Different types of layers: CNN, RNN, pooling, deconv
- Distributed training
- Estimators
- …

Summary

Productive GPU Programming

Many options for productive GPU programming above the level of CUDA

- High level, domain specific libraries
- Ahead-of-time or just-in-time compilers/interpreters

In general, trade off performance/generality for usability/time-to-solution.

Full list of CUDA libraries across deep learning, signal processing, linear algebra, and more: https://developer.nvidia.com/gpu-accelerated-libraries