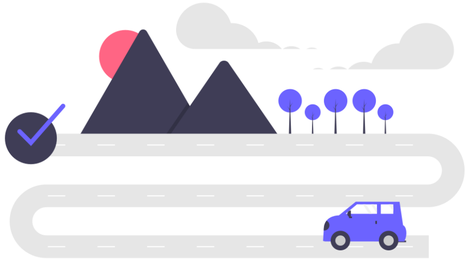
CME 213, ME 339 Spring 2023
Introduction to parallel computing using MPI, openMP, and CUDA

Eric Darve, ICME, Stanford

"Physics is the universe's operating system." (Steven R Garman)

Stanford University

Recap



- CUDA programming
- Performance and optimization
- CUDA profiler

Final Project

Goal

- Implement a neural network with 2 layers
- Densely connected layers
- Requires computing matrix-matrix products
- Parallelized with 4 GPUs using MPI (distributed memory)

Logistics

Turn in	Date	Grade weight
Preliminary report + working code (not optimized)	Friday, June 2nd	20%
Final report (4 pages) + code + optimization	Monday, June 12th	80%

Preliminary report

- Goal is to get a **working code**
- This is an important skill in programming; how to write **correct code first** without worrying about performance
- What is the **simplest** and most straightforward approach you can think of?
- Can you choose an approach that will **minimize chances of making a mistake?**
- Short report with outline of what you have done.

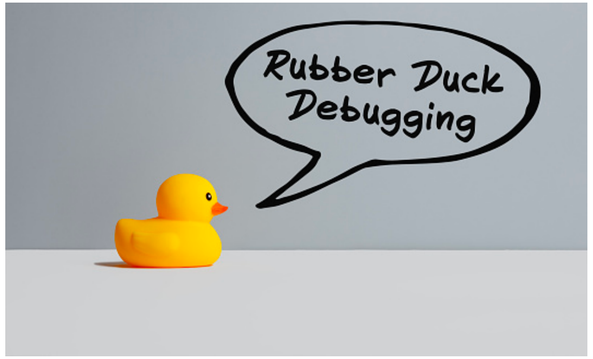
Final report

- Correctness: tests and results
- Performance of code (optional leaderboard)
- Quality of report: organization, clarity
- Quality of plots and figures: labels, legend, font, visual quality
- Quality of profiling, analysis, data gathering, performance
  - What are the performance bottlenecks in your code?
  - How did you address them? How can they be addressed (future work)?

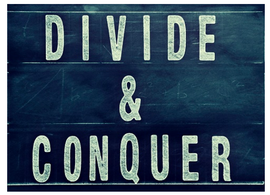
Correctness and debugging

- Ways to test your code are provided
- Output of CPU is saved at various points to files
- Files are subsequently loaded and content is compared against GPU code.
- This is a basic testing framework, but you will need to write additional tests to make sure your code works correctly.

Tips for debugging

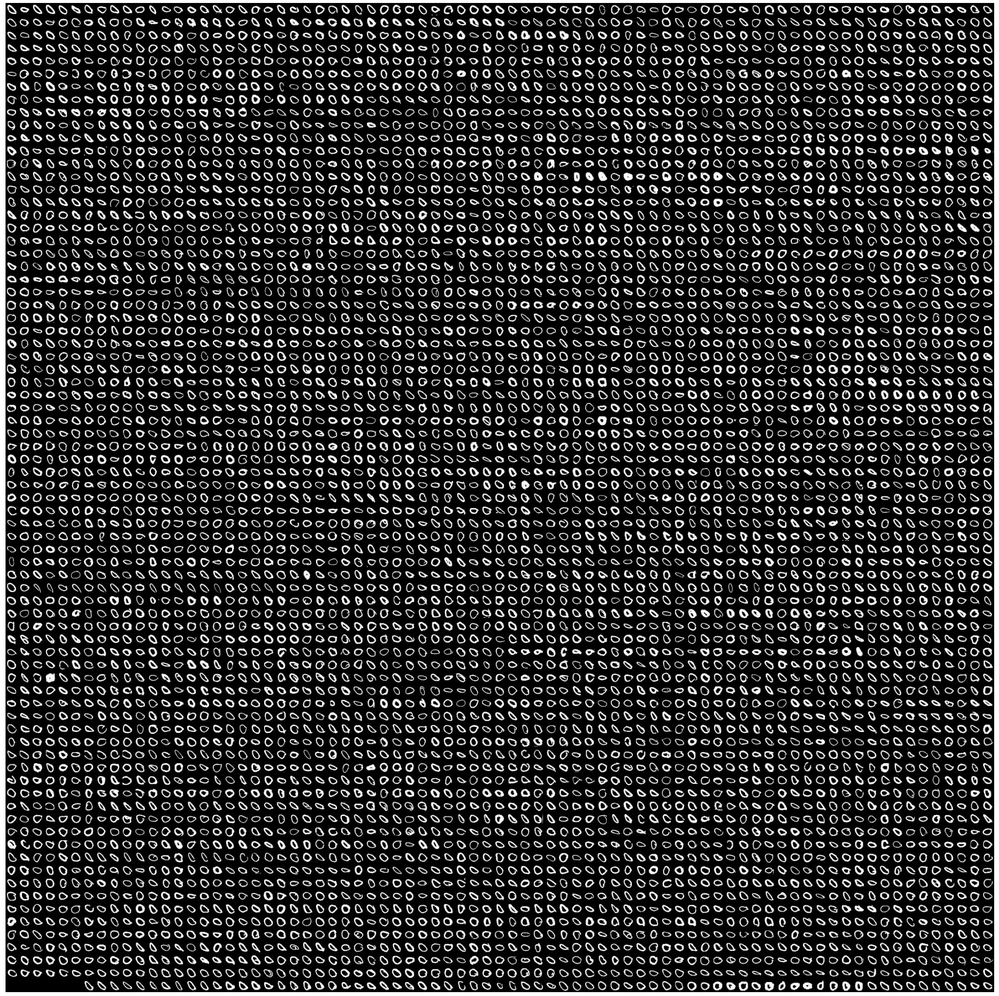


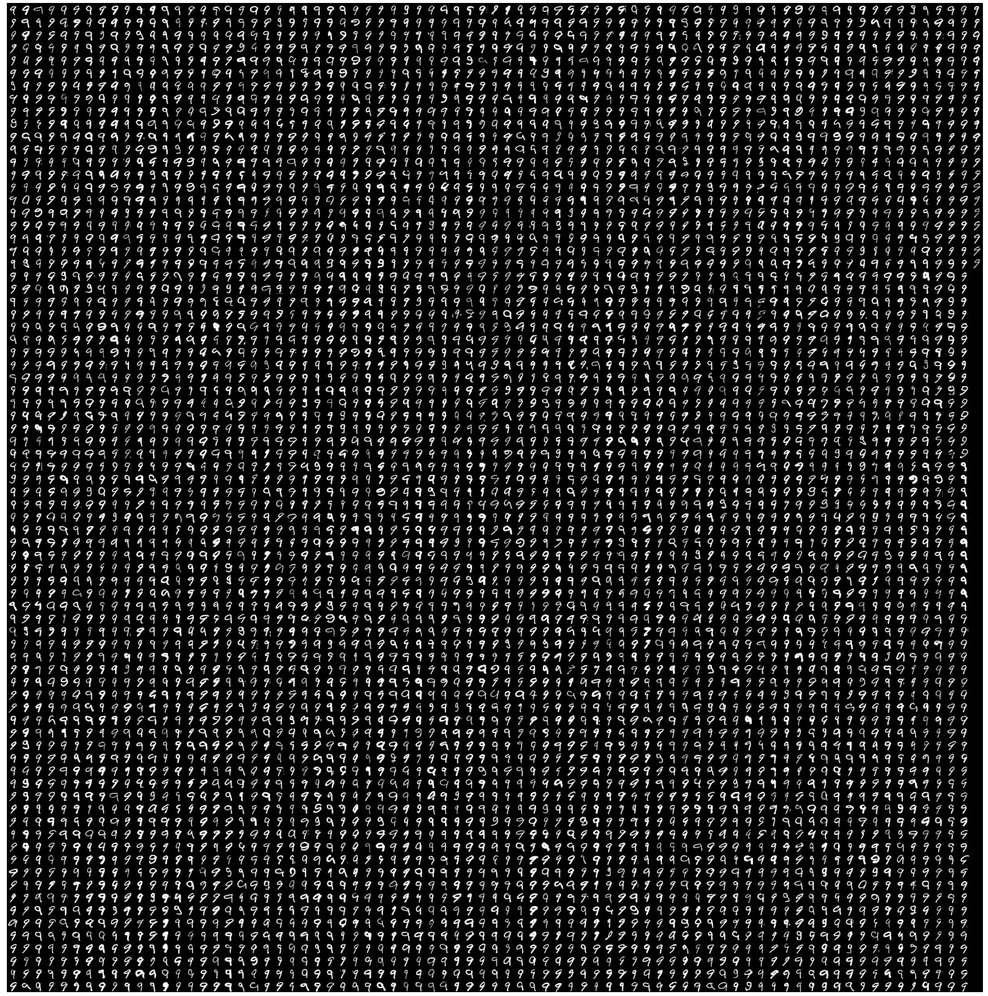
- Too many to list!
- Use a strategy! The worst approach is the **random** search…
- **Unit** tests: every time you write a small block of code, you should test it to make sure it is correct.
- Assume by default that **every line of code has an error.** This is your null hypothesis.
- Use **divide and conquer:** narrow down the location of the bug using a **dichotomy search.**



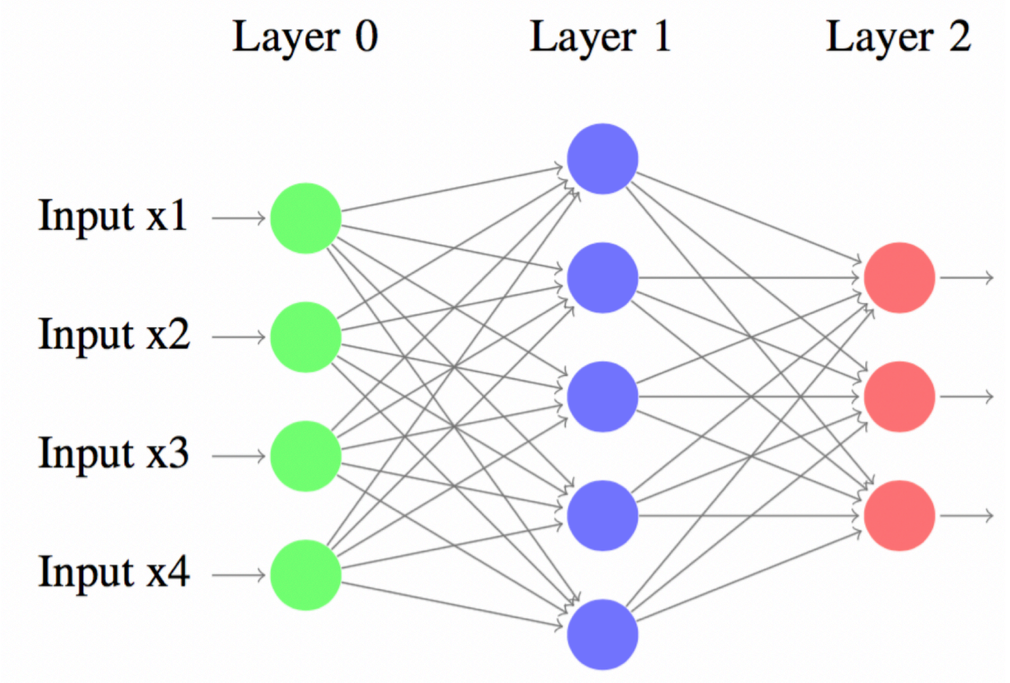
- Spend time thinking about ways to check the code: known mathematical result or inequality, result can obtained through other means (e.g., **CPU,** by hand, analytically).
- Test small cases that can be verified by hand.
- Main tools: **`assert` and `printf`.**
- Spend time reading compiler error messages; they are actually trying to tell you something useful!
- **Look for:** out of bound memory accesses, loop bounds, integer divisions (e.g., the problem size does not divide evenly), partitioning (e.g., partitioning is not even in some cases).
- Test using **single and double precisions** to make sure this is not a roundoff error. We provide a framework to do that.
- **Golden rule of debugging:** Time spent writing tests, adding asserts, and print statements = 0
  Time spent debugging without a strategy = ∞

The task; MNIST dataset





What is a neural network?



Inputs and outputs

- **Input layer:** a set of images organized as a matrix
- **Hidden layer:** –n `num`; variable size
- **Output layer:** softmax vector with 10 digits
- The output corresponds to the “probability” of the digit as estimated by the DNN.

softmax

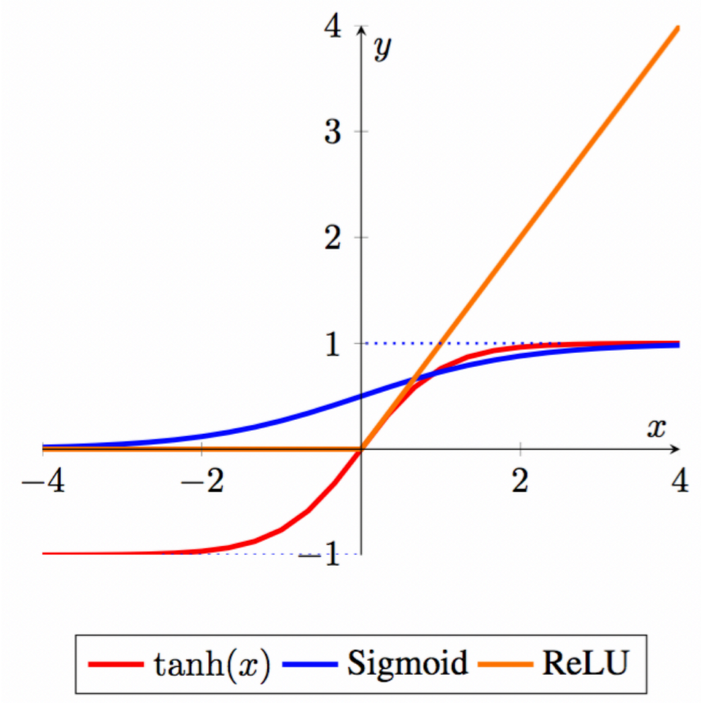
A function commonly used to map a vector of real numbers to a probability vector:

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{k=0}^{9} \exp(z_k)}$$

DNN layer

- A linear transformation followed by a non-linear map.
- Linear map: $z = Wx + b$
- W: dense matrix (weights); b: vector (bias)
- Usually a convolution matrix is used, but for this project we use a regular dense matrix.
- Output of layer: $a_i = \sigma(z_i)$; element-wise non-linear function

Example of non-linear activation functions



We will use the sigmoid.

Training the DNN

- We need to optimize the weights and biases so that the output of the DNN is correct.
- The first step is to define an error metric.
- In this case, we use the cross-entropy.
- The cross-entropy allows computing the dissimilarity between two distributions.

Cross-entropy explained

- Generate a sequence of realizations where event i has probability y_i.
- You don't know y_i but you have some approximation $\hat{y}_i$ that you use to estimate the probability of the sequence.
- For a very long sequence of size N, the estimate is:

$$\mathscr{P} = \prod_i \hat{y}_i^{N y_i} \qquad -\frac{\ln \mathscr{P}}{N} \to -\sum_i y_i \ln \hat{y}_i = H(y, \hat{y})$$

Cross-entropy

Cross-entropy: $H(y, \hat{y}) = -\sum_i y_i \ln \hat{y}_i \sim -\frac{\ln \mathscr{P}}{N}$

When $\hat{y}_i = y_i$, $\mathscr{P}$ is **maximum** and the cross-entropy is **minimum**.

The more dissimilar $\hat{y}$ and y are the larger the cross-entropy.

Cross-entropy and KL divergence

More maths if you are familiar with this topic

$$H(y, \hat{y}) = H(y) + D_{\mathrm{KL}}(y \,||\, \hat{y})$$

$D_{\mathrm{KL}}(y \,||\, \hat{y}) \geq 0$ and $D_{\mathrm{KL}}(y \,||\, \hat{y}) = 0$ when $\hat{y} = y$.

So when $H(y, \hat{y})$ is minimum (over $\hat{y}$) we have $\hat{y} = y$.

MNIST dataset

Given some input with digit c, we define:

$$y_i = 0 \text{ if } i \neq c \text{ and } y_c = 1.$$

Then we optimize the weights and biases of the DNN such that

$H(y, \hat{y})$ is minimum, where $\hat{y}$ is the output of the DNN (a vector of length 10 with the softmax layer).

$$H(y, \hat{y}) = -\sum_i y_i \ln \hat{y}_i = -\ln \hat{y}_c$$

The DNN tries to make $\hat{y}_c \sim 1$.

Loss function

Consider a large number of input images.

We now define the loss function using all images indexed by i:

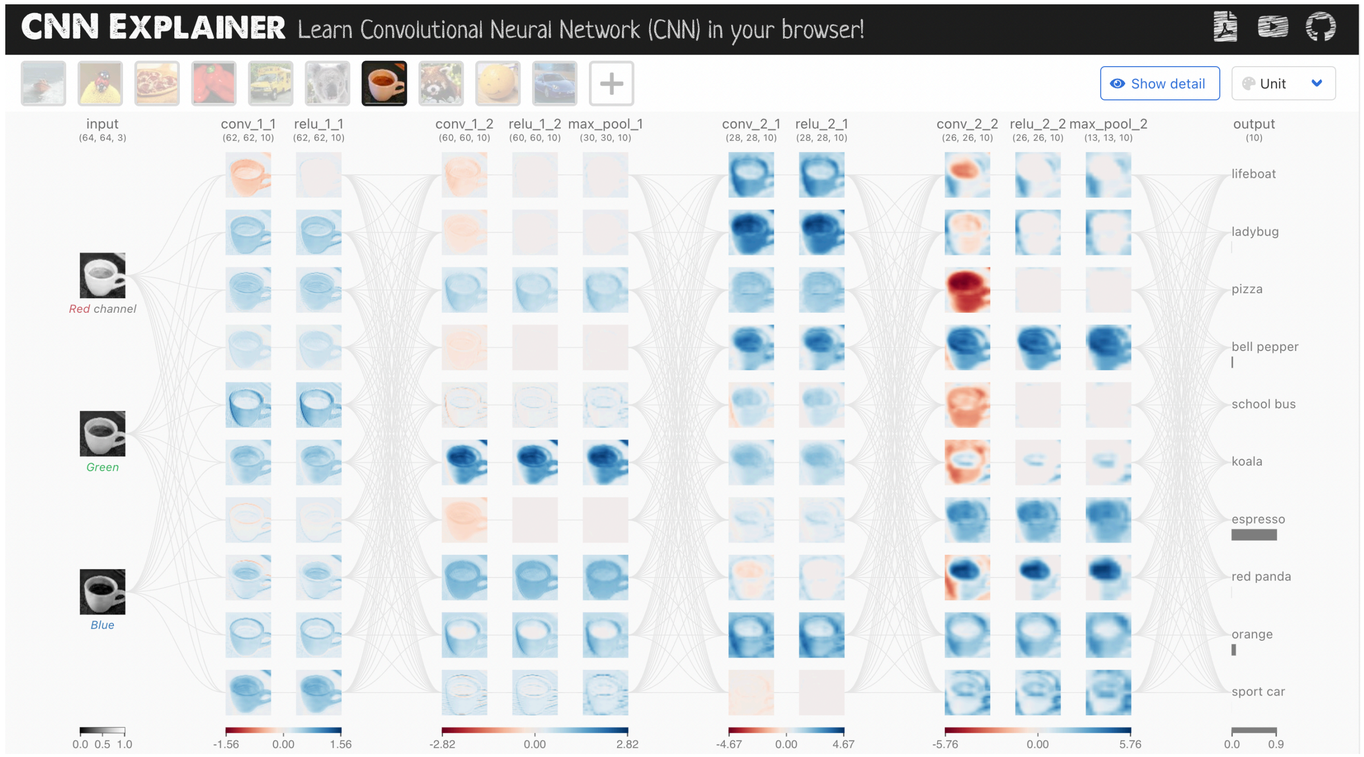
$$J(p) = \frac{1}{N} \sum_{i=1}^{N} H(y_i, \hat{y}_i)$$

p weights and biases of network = all parameters; N: number of images

Optimize p by minimizing $J(p)$.

CNN explainer

https://poloclub.github.io/cnn-explainer/



How to train a neural network

We use a gradient descent algorithm:

$$p \leftarrow p - \alpha \nabla_p J$$

Gradient is computed by repeated application of the chain rule

= Backpropagation

See final project write-up for details.

Stochastic gradient descent

Full batch:

$$p \leftarrow p - \frac{\alpha}{N} \nabla_p \sum_{i=1}^{N} H(y_i, \hat{y}_i(p))$$

Stochastic gradient

$$p \leftarrow p - \frac{\alpha}{N} \nabla_p \sum_{i \in \text{rand subset}} H(y_i, \hat{y}_i(p))$$

Benefits of stochastic gradient descent

One epoch: once all the input images have been processed.

If we use a small random subset, this allows more updates of the DNN coefficients per epoch

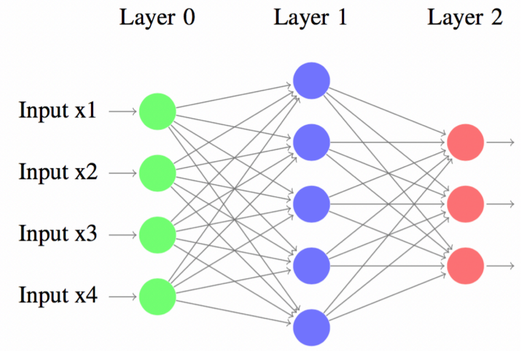
→ more accurate

Randomness of subset selection allows avoiding local minima and escaping saddle points

→ better convergence

Sequence of training steps

- Forward pass = left to right
  - DNN prediction
  - Compare with label
- Backward propagation = right to left
  - Chain rule
  - Compute gradient and update DNN weights and biases.
- Iterate until convergence



Regularization

We add the following term to the loss function:

$$J(p) = \frac{1}{N}\sum_{i=1}^{N} H(y_i, \hat{y}_i) + \frac{1}{2}\lambda \|p\|_2^2$$

Gradient:

$$-\nabla_p J(p) = -\frac{1}{N}\sum_{i=1}^{N} \nabla_p H(y_i, \hat{y}_i(p)) - \lambda\, p$$

Effect is to reduce the size of the DNN weights and biases p.

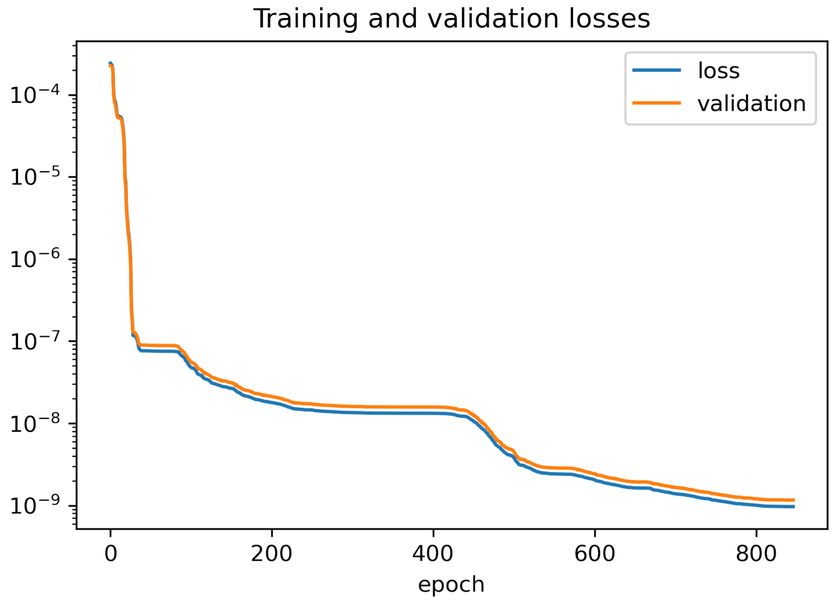
Magnitude of weights

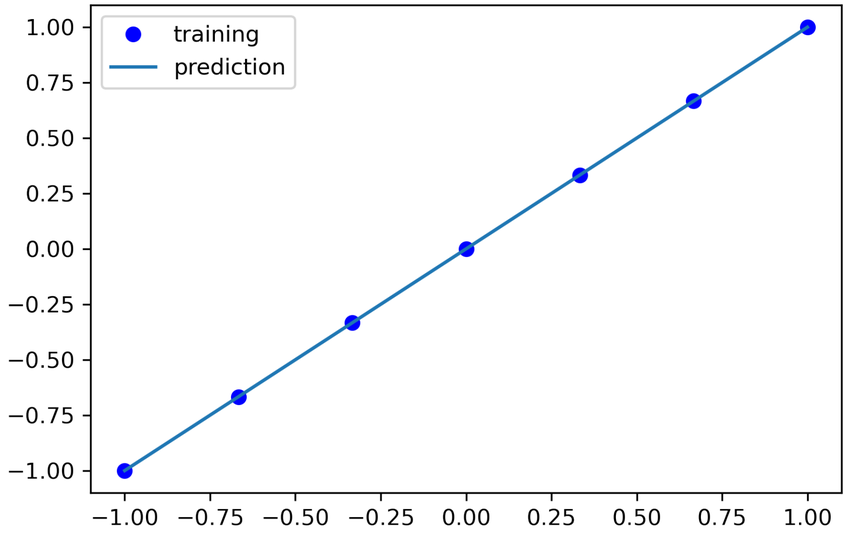
- Small weights and biases result in a DNN that is more **linear.**
- Large weights and biases allow the DNN to approximate sharp jumps in the function.
- Because of over-fitting, regularization is needed to improve the accuracy of the DNN.

How we can figure out how much regularization is needed?

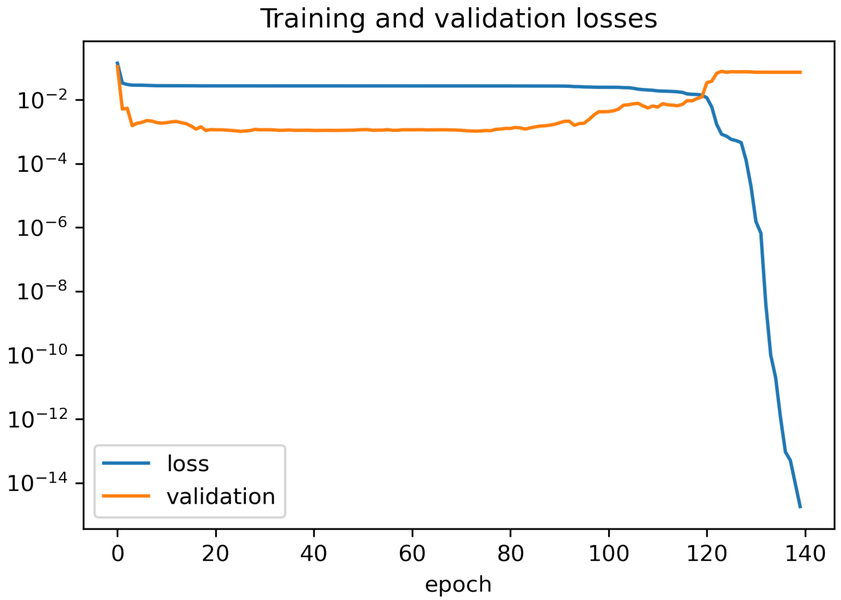
- Training set: used to minimize loss; involved in defining the gradient.
- Validation set: used to evaluate the model; how accurate is it? This allows avoiding overfitting.
- Over-fitting is similar to what happens when fitting a high-order polynomial in certain situations: we can get wild oscillations.

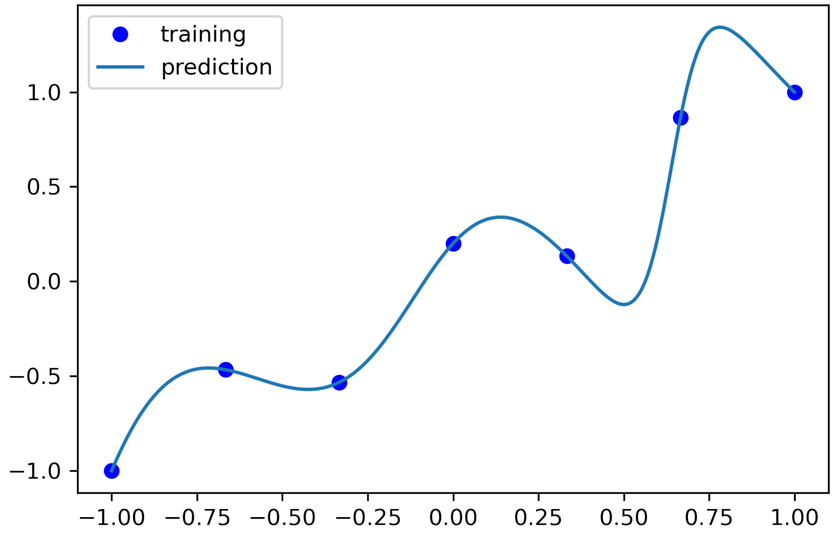
Example: small 2-layer DNN with width 8



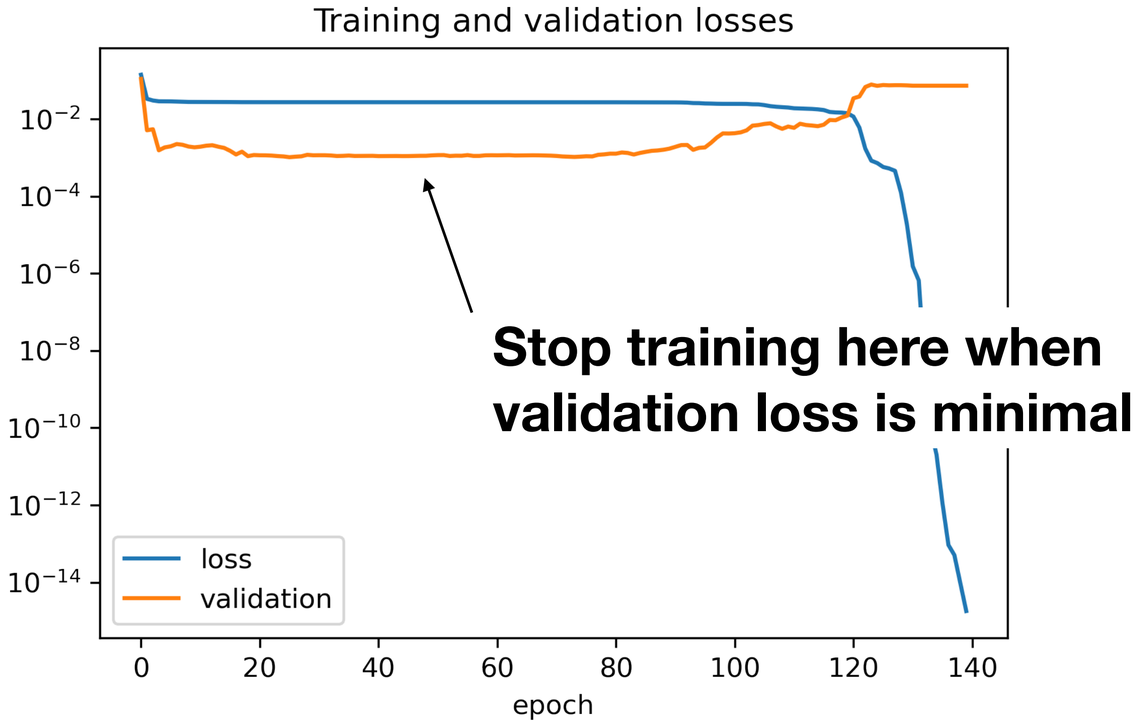


With noise added to data

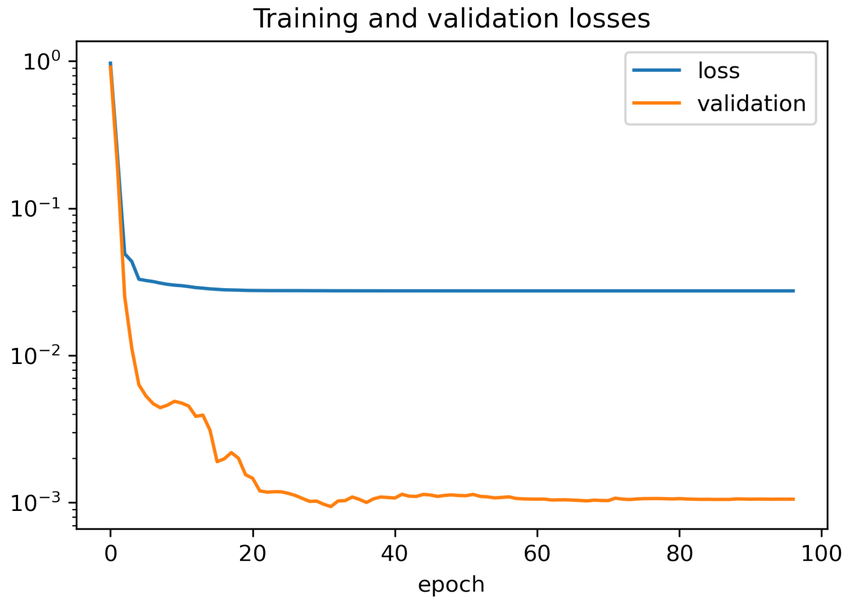


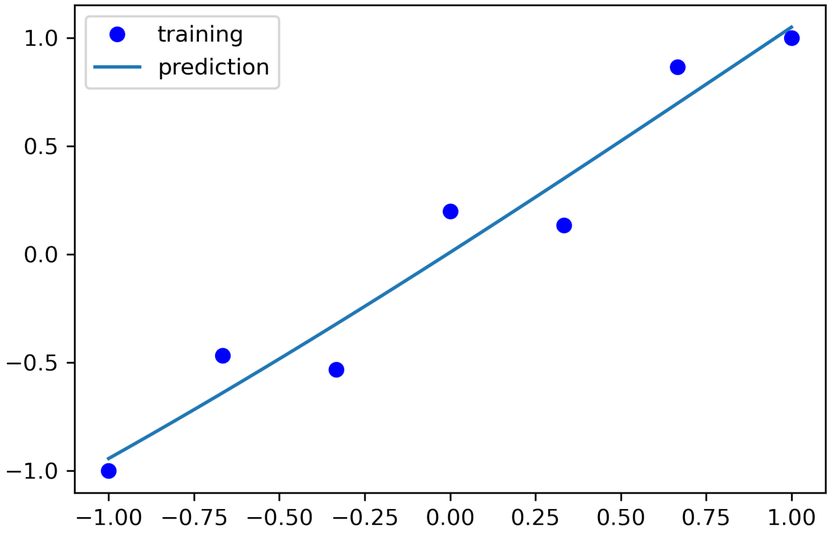


Fix 1: early stopping

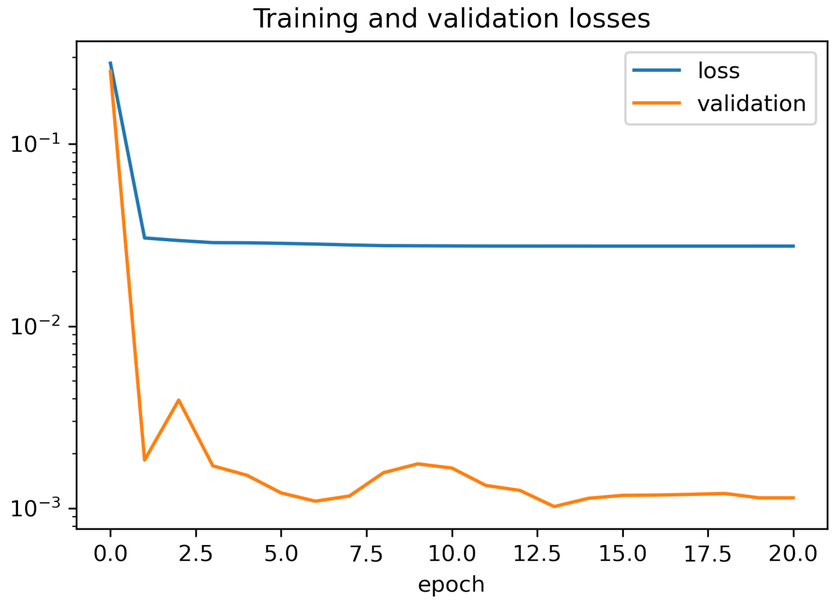


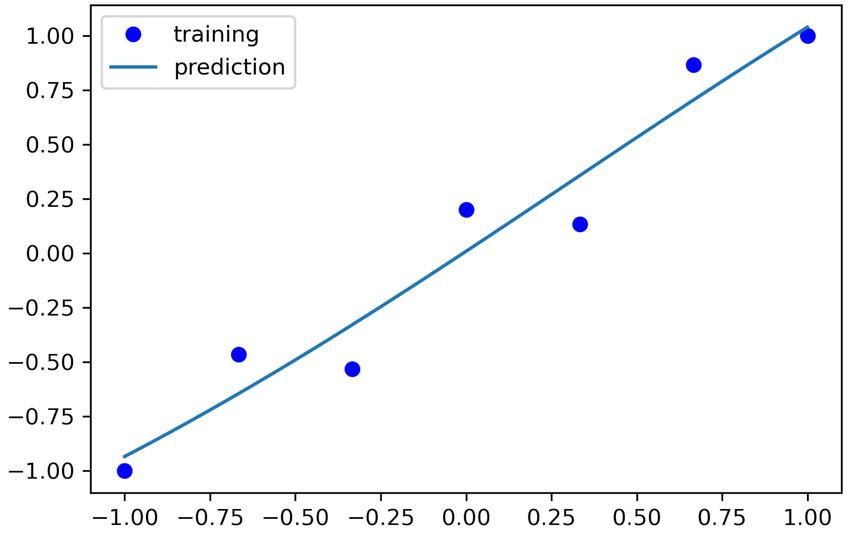
Fix 2: reduce the size of the DNN
Example with DNN with width 1



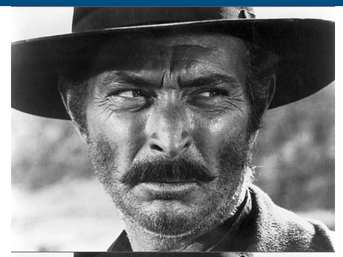
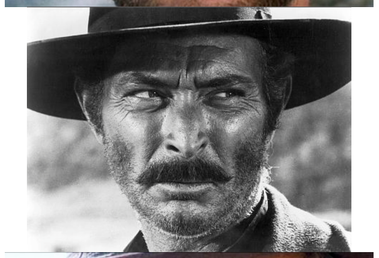
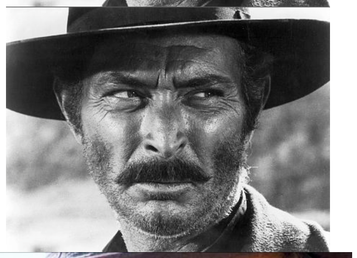
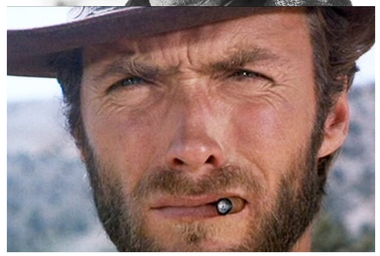


Fix 3: add regularization, e.g., $\lambda = 10^{-3}$





Adjusting the regularization

Training error	Validation error	Diagnostic
		Overfitting; increase λ
		Too much regularization; decrease λ
		Just right; regularization is good

Learning sets

- **Training set:** optimize DNN parameters.
- **Validation set:** optimize regularization (and other hyperparameters not covered by the gradient descent algorithm).
- **Test set:** unseen data used to evaluate the final accuracy of the DNN.

Main tasks in project: CUDA

1. Implement a matrix-matrix product (GEMM) algorithm
2. Implement CUDA kernels for the activation functions
3. Implement the forward and backward passes

A reference CPU code is provided.

Main tasks in project: MPI

Implement the MPI algorithm for distributed memory

1. Distribute the images in the mini-batch across the 4 GPUs
2. Compute a partial gradient for each image
3. Perform a reduction (sum) to calculate the gradient
4. Update the weights and biases by applying the gradient

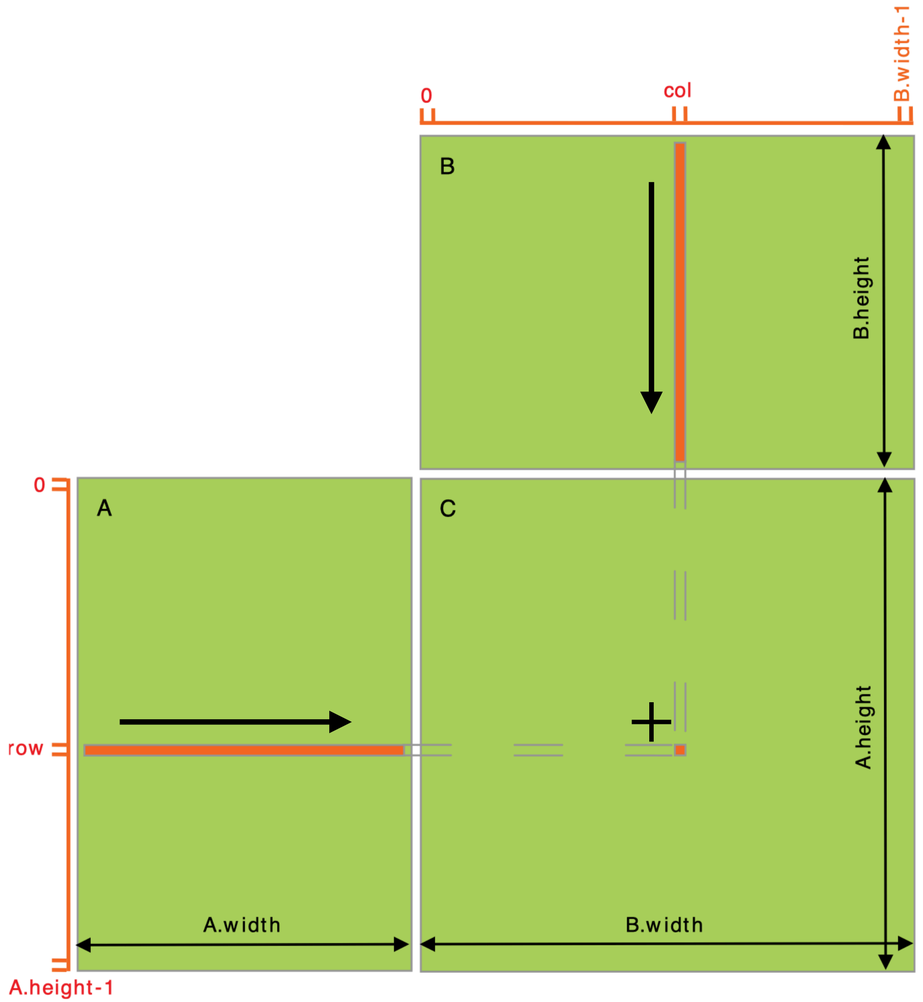
Matrix-matrix products in CUDA

Simplest algorithm

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

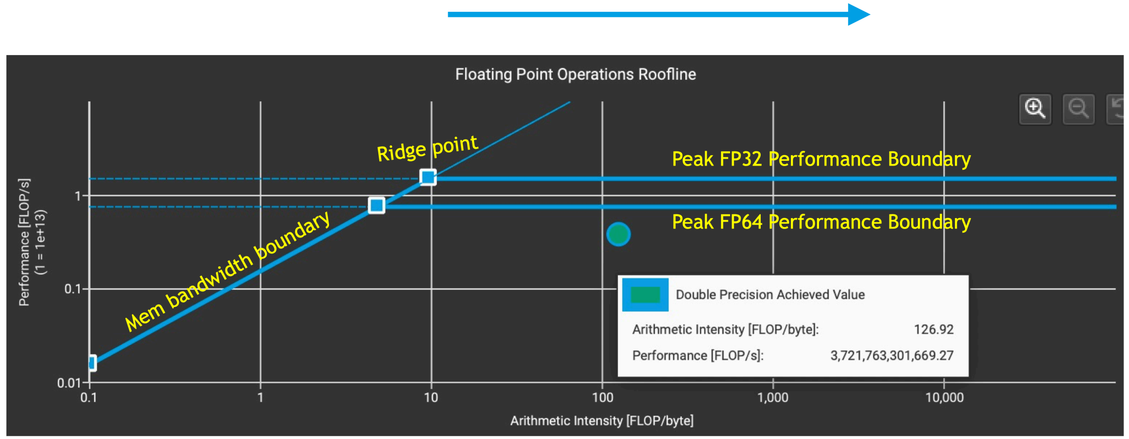
One thread per matrix entry

Generates memory traffic and limited instruction level parallelism.



Increasing performance

- How can we reduce the memory traffic?
- This would allow us to move to the right on the roofline plot.
- For this we need to understand the pattern of memory accesses.



Outer form of mat-mat product

$c_{ij} \leftarrow c_{ij} + a_{ik} b_{kj}$

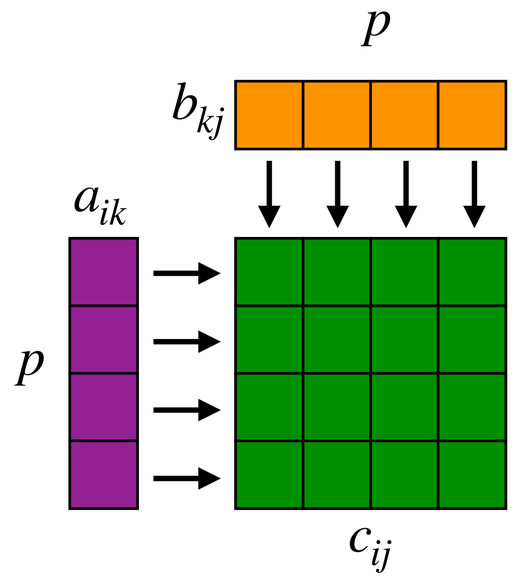
Load a_{ik} and b_{kj} for $i_0 \leq i \leq i_1$ and $j_0 \leq j \leq j_1$.

Outer product form.

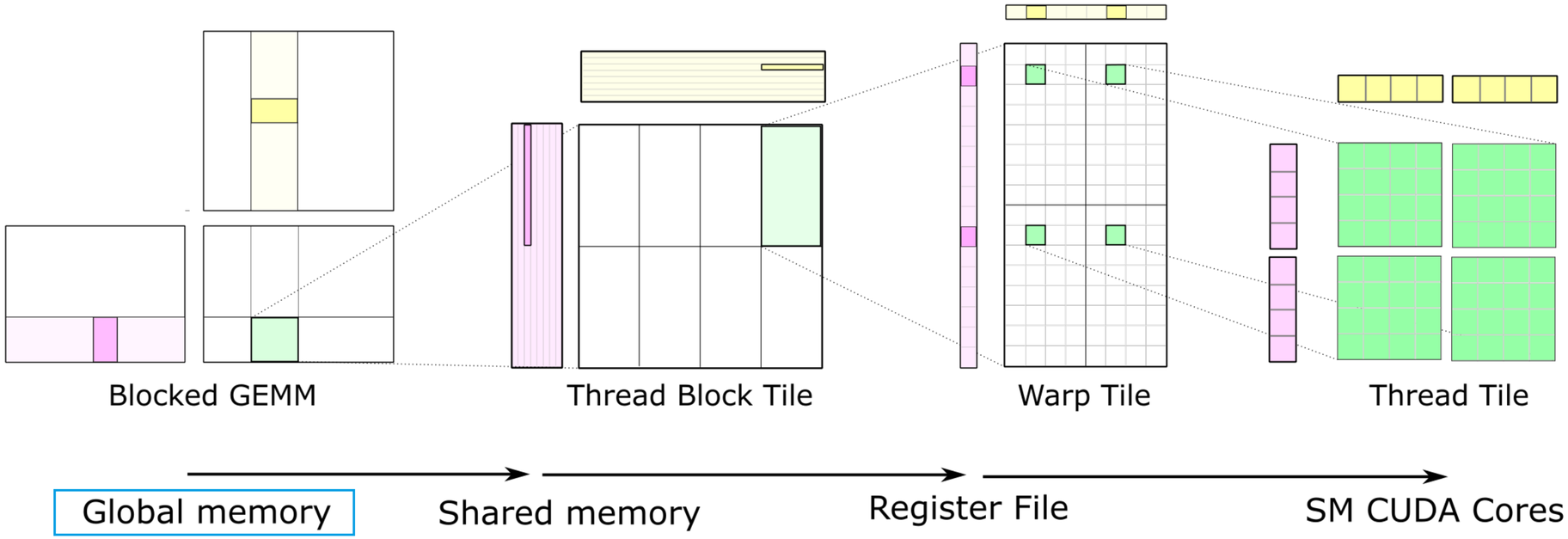
Reads: $2p$

Flops: $2p^2$

With p big enough, we generate enough flops.



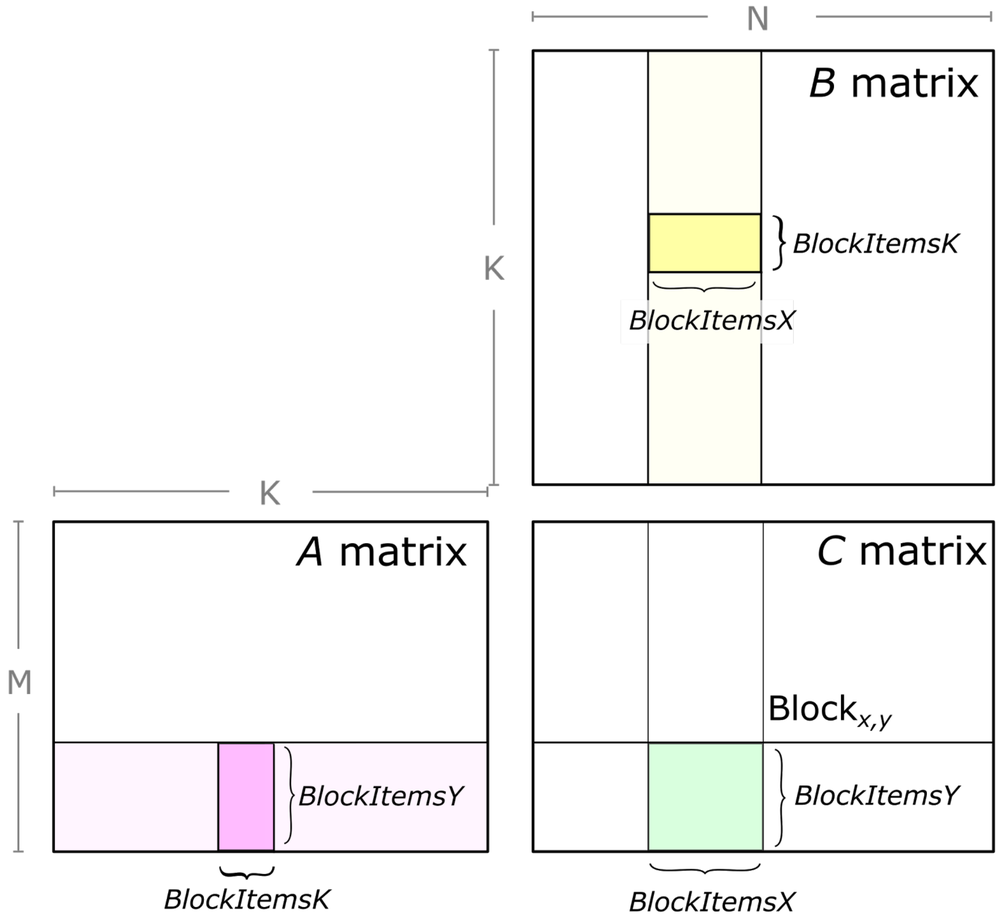
Hierarchical decomposition of the product



The algorithm is mapped to the memory hierarchy of the GPU with L2 cache, shared memory, and registers.

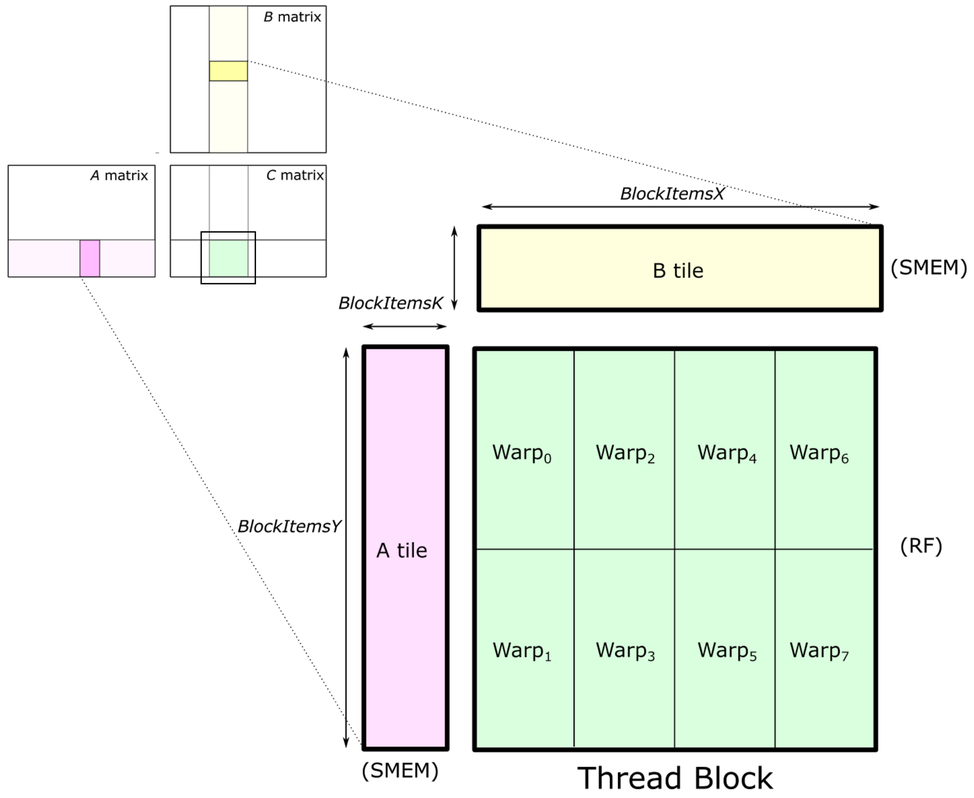
Global memory

- A thread-block calculates a large block of C.
- Rectangular blocks of matrix A and B are loaded.
- Reads: $K(X + Y)$
- Flops: $2XY$.
- Choose X and Y sufficiently large.



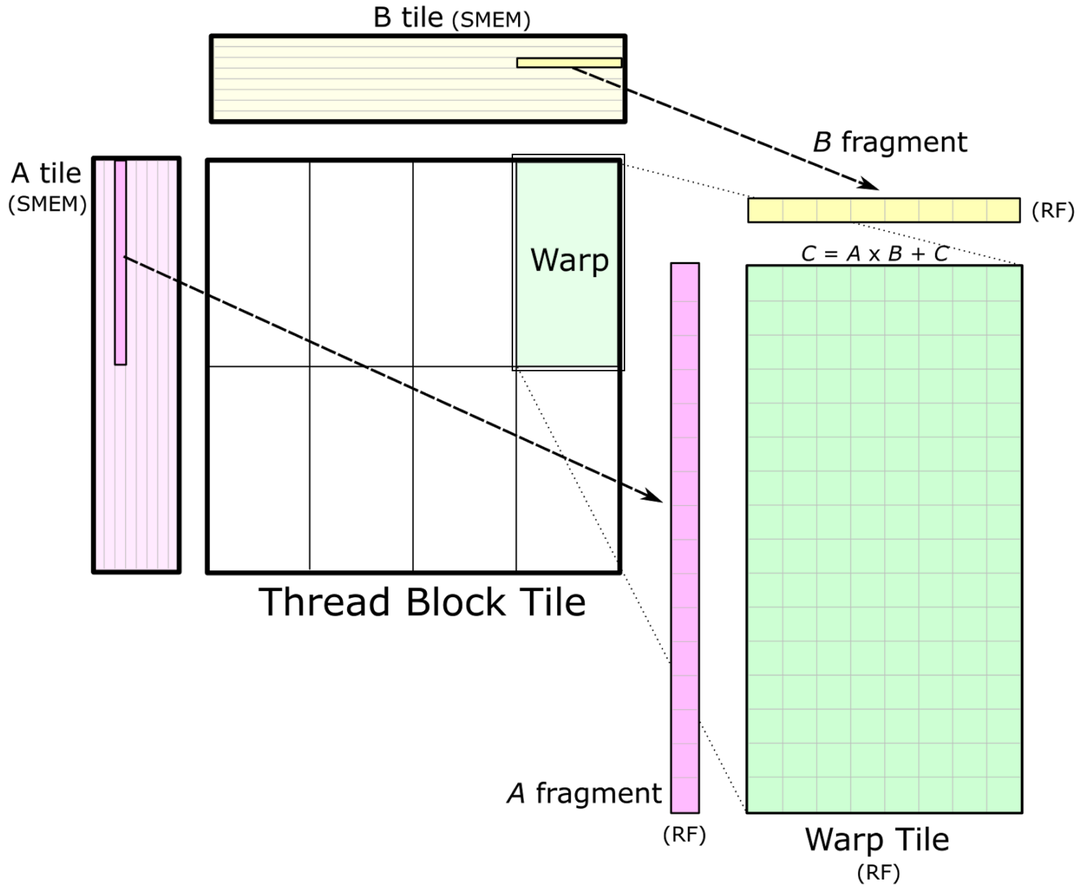
Thread block workload

- Data is loaded into shared memory for fast access by all warps.
- Loading is performed by all warps.
- Each warp computes a rectangle in the output C.
- Accumulation uses registers.



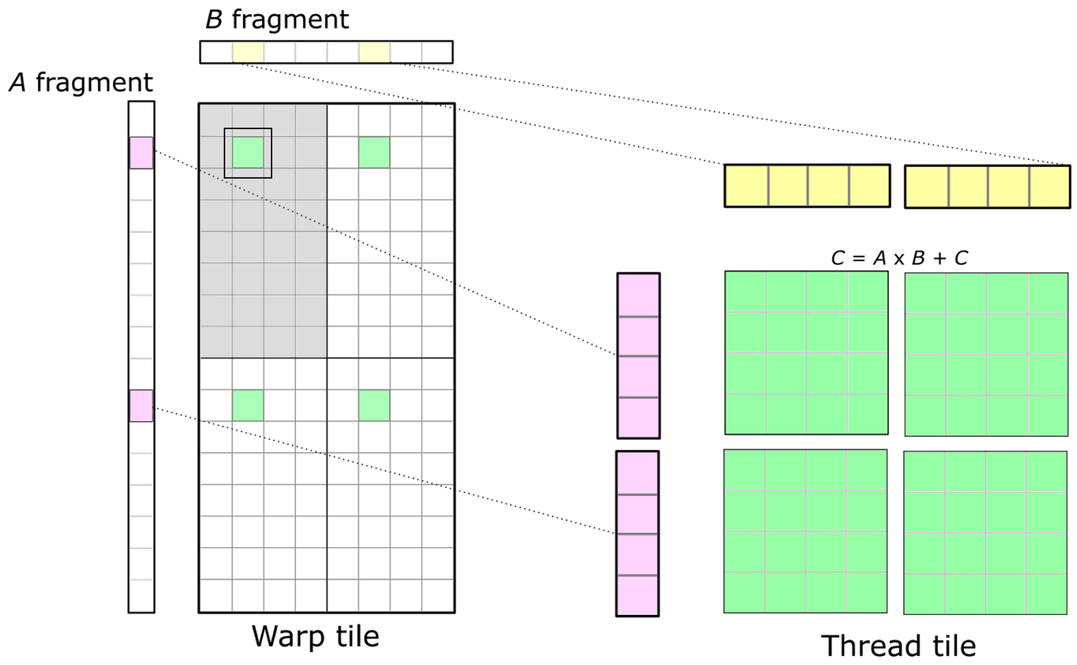
Warp level workload

- Each warp loads one column of A and one row of B. Then accumulates the product into a block of C.
- The data is copied from shared memory to register files.



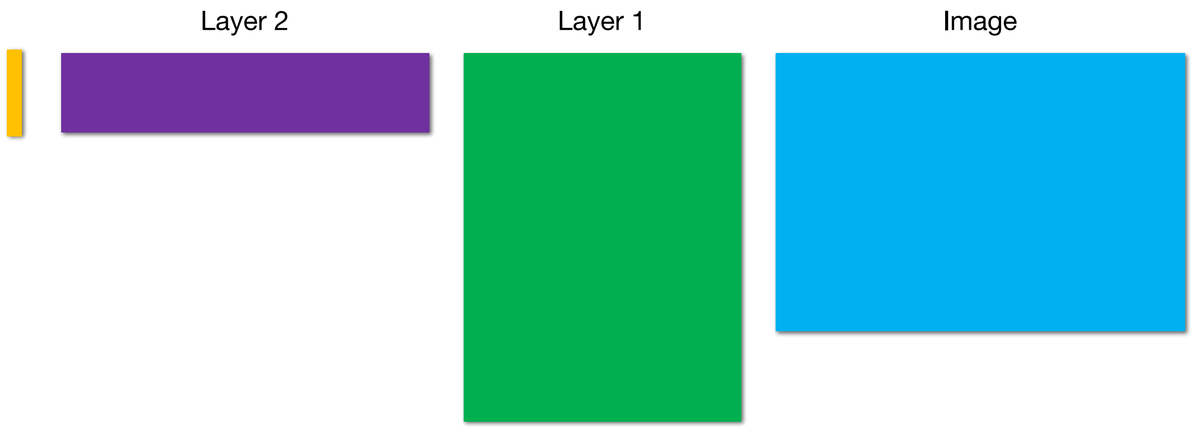
Thread level workload

- Each thread computes multiple entries in the output C.
- Because each entry is updated independently of the others, we can generate a lot of **instruction level parallelism.**
- This further increases the performance.

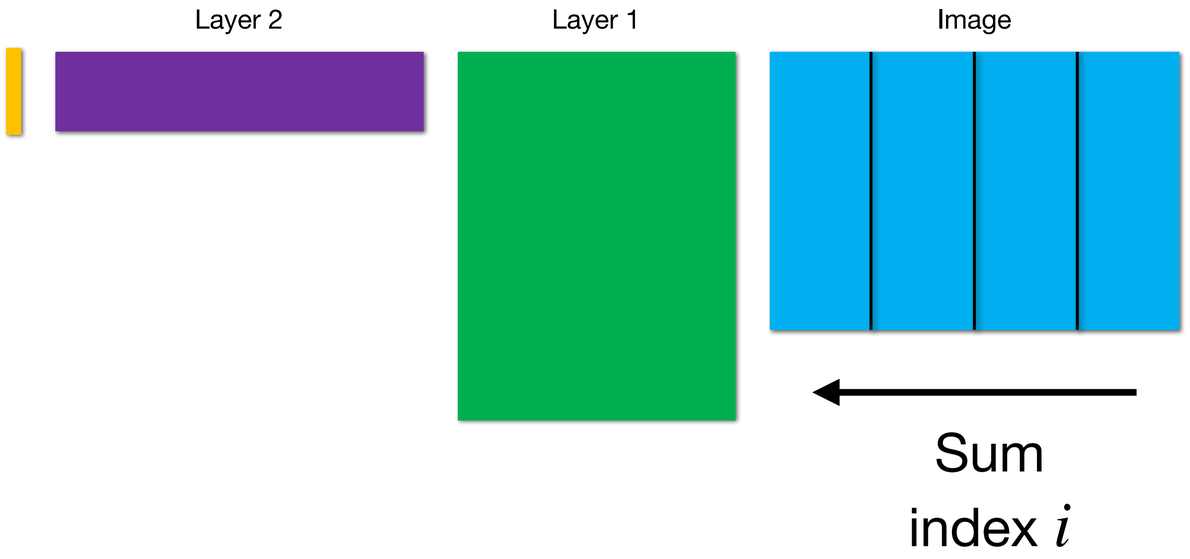


Distributed MPI

- We will cover this topic in the next few weeks. But the basic concepts required for the project will be covered shortly.
- Input: images
- Output: vector of probabilities of size 10.



Partitioning

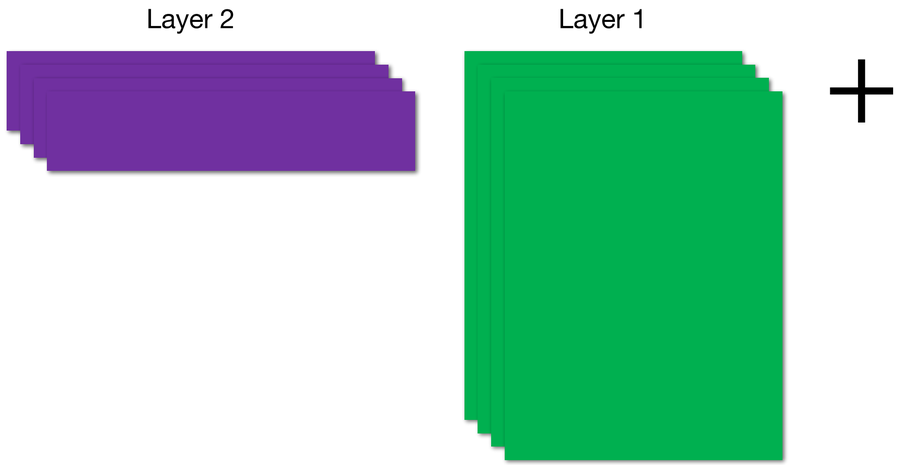


Partition the input across the columns.

$$J(p) = \frac{1}{N} \sum_{i=1}^{N} H(y_i, \hat{y}_i)$$

Each GPU computes a partial gradient → sum/reduction → apply full gradient to weights and biases.

Reduction



- Reduction of all weights and biases requires sending the data across the network between the different computing nodes.
- This can be time consuming even on a very fast network.
- As a bonus problem, we suggest a better way to do this.

Files and debugging

Debugging

- To evaluate your code, we will run it and check the accuracy of the calculation.
- Your code will be checked at several points: the value of the weights and biases at various epochs and their value at the end.
- Let’s go over the details.

Sequential code

- The CPU calculation is used as a reference result.
- Results of the calculation at every epoch are saved.
- Edit the direction location in `main.cpp:`

```
string cpu_save_dir = “/home/darve/fp_output”;
string cpu_load_dir = “/home/darve/fp_output”;
```

- Use `–s –d` option to generate this data.
- This takes a while. This is why the data is saved to files.

Code output

```
darve@icme-gpu1:~/final_project$ srun -n 4 -G 4 --mpi=pmi2 -p CME ./main -s -d
Number of MPI processes = 4
Number of CUDA devices = 4
Grading mode off
Number of neurons 1000
Number of epochs 1
Batch size 32
Regularization          0.0001
Learning rate           0.001
The sequential code will be run
The debug option is on
Output directory is Outputs
The CPU results will be saved to /home/darve/fp_output
The CPU results will be loaded from /home/darve/fp_output
Loading training data
Training data information:
Size of x_train, N =  60000
Size of label_train = 60000
Loading testing data
Start Sequential Training
Saving W0 CPU data to file /home/darve/fp_output/seq_epoch-W0-0.mat
Saving W1 CPU data to file /home/darve/fp_output/seq_epoch-W1-0.mat
Saving b0 CPU data to file /home/darve/fp_output/seq_epoch-b0-0.mat
Saving b1 CPU data to file /home/darve/fp_output/seq_epoch-b1-0.mat
Time for Sequential Training: 60.19 seconds
Saving to file /home/darve/fp_output/seq_nn-W0.mat
Saving to file /home/darve/fp_output/seq_nn-b0.mat
Saving to file /home/darve/fp_output/seq_nn-W1.mat
Saving to file /home/darve/fp_output/seq_nn-b1.mat
Precision on validation set for sequential training =   0.9153333306312561
Precision on testing set for sequential training =   0.8973000049591064

Start Parallel Training
Loading CPU data W0 from file /home/darve/fp_output/seq_epoch-W0-0.mat
Loading CPU data W1 from file /home/darve/fp_output/seq_epoch-W1-0.mat
Loading CPU data b0 from file /home/darve/fp_output/seq_epoch-b0-0.mat
Loading CPU data b1 from file /home/darve/fp_output/seq_epoch-b1-0.mat
Time for Parallel Training: 6.80323 seconds
Precision on validation set for parallel training =   0.9153333306312561
Precision on testing set for parallel training =   0.8973000049591064

Grading mode on. Now checking for correctness...
Loading from file /home/darve/fp_output/seq_nn-W0.mat
Loading from file /home/darve/fp_output/seq_nn-b0.mat
Loading from file /home/darve/fp_output/seq_nn-W1.mat
Loading from file /home/darve/fp_output/seq_nn-b1.mat

Max norm of diff b/w seq and par: W[0]: 4.08162e-06, b[0]: 8.65353e-06
l2  norm of diff b/w seq and par: W[0]: 2.85522e-06, b[0]: 2.66075e-06
Max norm of diff b/w seq and par: W[1]: 4.00552e-07, b[1]: 1.88942e-06
l2  norm of diff b/w seq and par: W[1]: 4.42264e-07, b[1]: 1.99009e-06
darve@icme-gpu1:~/final_project$
```

Files generated

At each epoch →

At the end →

```
darve@icme-gpu1:~/final_project$ ls -1 ~/fp_output/*
/home/darve/fp_output/seq_epoch-b0-0.mat
/home/darve/fp_output/seq_epoch-b1-0.mat
/home/darve/fp_output/seq_epoch-W0-0.mat
/home/darve/fp_output/seq_epoch-W1-0.mat
/home/darve/fp_output/seq_nn-b0.mat
/home/darve/fp_output/seq_nn-b1.mat
/home/darve/fp_output/seq_nn-W0.mat
/home/darve/fp_output/seq_nn-W1.mat
darve@icme-gpu1:~/final_project$
```

Debug mode

- Assuming you have previously saved the CPU result, you can compare against your GPU result using `–d`.
- **Don’t use `–s` again. Instead just use the data previously saved to file.**
- **This will reduce the runtime considerably.**
- If getting errors, it’s probably because the CPU files were not generated correctly.
- The results of the test are saved to the directory `Outputs/`.

Code output

```
darve@icme-gpu1:~/final_project$ srun -n 4 -G 4 --mpi=pmi2 -p CME ./main -d
Number of MPI processes = 4
Number of CUDA devices = 4
Grading mode off
Number of neurons 1000
Number of epochs 1
Batch size 32
Regularization          0.0001
Learning rate           0.001
The sequential code will not be run
The debug option is on
Output directory is Outputs
The CPU results will be loaded from /home/darve/fp_output
Loading training data
Training data information:
Size of x_train, N =  60000
Size of label_train = 60000
Loading testing data

Start Parallel Training
Loading CPU data W0 from file /home/darve/fp_output/seq_epoch-W0-0.mat
Loading CPU data W1 from file /home/darve/fp_output/seq_epoch-W1-0.mat
Loading CPU data b0 from file /home/darve/fp_output/seq_epoch-b0-0.mat
Loading CPU data b1 from file /home/darve/fp_output/seq_epoch-b1-0.mat
Time for Parallel Training: 6.75543 seconds
Precision on validation set for parallel training =   0.9153333306312561
Precision on testing set for parallel training =   0.8973000049591064

Grading mode on. Now checking for correctness...
Loading from file /home/darve/fp_output/seq_nn-W0.mat
Loading from file /home/darve/fp_output/seq_nn-b0.mat
Loading from file /home/darve/fp_output/seq_nn-W1.mat
Loading from file /home/darve/fp_output/seq_nn-b1.mat

Max norm of diff b/w seq and par: W[0]: 4.08162e-06, b[0]: 8.65353e-06
l2  norm of diff b/w seq and par: W[0]: 2.85522e-06, b[0]: 2.66075e-06
Max norm of diff b/w seq and par: W[1]: 4.00552e-07, b[1]: 1.88942e-06
l2  norm of diff b/w seq and par: W[1]: 4.42264e-07, b[1]: 1.99009e-06
darve@icme-gpu1:~/final_project$
```

Files generated

```
darve@icme-gpu1:~/final_project$ ls Outputs
CpuGpuDiff-4.txt  NNErrors-4.txt
darve@icme-gpu1:~/final_project$
```

```
darve@icme-gpu1:~/final_project$ cat Outputs/CpuGpuDiff-4.txt
Iteration       Max Err W0      Max Err W1      Max Err b0      Max Err b1      L2 Err W0       L2 Err W1       L2 Err b0       L2 Err b1
0               1.31434e-08     1.15806e-09     3.8074e-06      2.07388e-07     2.17838e-08     8.73694e-09     2.00689e-06     1.40444e-07
darve@icme-gpu1:~/final_project$ cat Outputs/NNErrors-4.txt
Mismatches for W[0]
No errors were found
Mismatches for b[0]
No errors were found
Max norm of diff b/w seq and par: W[0]: 4.08162e-06, b[0]: 8.65353e-06
l2  norm of diff b/w seq and par: W[0]: 2.85522e-06, b[0]: 2.66075e-06
Mismatches for W[1]
No errors were found
Mismatches for b[1]
No errors were found
Max norm of diff b/w seq and par: W[1]: 4.00552e-07, b[1]: 1.88942e-06
l2  norm of diff b/w seq and par: W[1]: 4.42264e-07, b[1]: 1.99009e-06
darve@icme-gpu1:~/final_project$
```

Tip

- When testing your code, make sure that you use **the same options** (batch size, number of epochs, number of neurons, etc.) when running the **sequential code** to generate the reference results and when running the **GPU code** for testing.

- If different options are used, the result will be invalid, and the code may fail to run.

Grading modes: –g option

- These options will be used to test your code for correctness during grading.
- 1, 2, 3 : runs the deep learning training with different learning rates and number of epochs; use –g `{1,2,3}`
- 4 : run GEMM tests only; use –g `4`

Example GEMM test output

```
darve@icme-gpu1:~/final_project$ srun -n 4 -G 4 --mpi=pmi2 -p CME ./main -g 4
Number of MPI processes = 4
Number of CUDA devices = 4

Entering GEMM Benchmarking mode! Stand by.

Starting GEMM 1: M = 8000; N = 10000; K = 7840
GEMM matched with reference successfully! Rel diff = 0
Time for reference GEMM implementation: 0.427576 seconds
Time for my GEMM implementation: 1.34342 seconds
Completed GEMM 1

Starting GEMM 2: M = 8000; N = 1000; K = 10000
GEMM matched with reference successfully! Rel diff = 5.08913e-08
Time for reference GEMM implementation: 0.051957 seconds
Time for my GEMM implementation: 0.367644 seconds
Completed GEMM 2

Starting GEMM 3: M = 8000; N = 100; K = 10000
GEMM matched with reference successfully! Rel diff = 1.07542e-06
Time for reference GEMM implementation: 0.00660718 seconds
Time for my GEMM implementation: 0.0393743 seconds
Completed GEMM 3
darve@icme-gpu1:~/final_project$
```

Running in double precision

- Checking roundoff errors is critical.
- Is a small error due to roundoff or a small bug?
- For this we have the ability to run the code in single or double precision.
- But you need to follow our convention to make this work.

File `common.h`

```
#ifndef USE_DOUBLE

typedef float nn_real;
#define MPI_FP MPI_FLOAT
#define cublas_gemm cublasSgemm

#else

typedef double nn_real;
#define MPI_FP MPI_DOUBLE
#define cublas_gemm cublasDgemm

#endif
```

Compiling option -DUSE_DOUBLE
See Makefile_double

Use these types

`nn_real` : use this type for all floating point numbers!
`MPI_FP` : use this type for MPI communications.

Example: GEMM test output in double precision

```
darve@icme-gpu1:~/final_project$ srun -n 4 -G 4 --mpi=pmi2 -p CME ./main -g 4
Number of MPI processes = 4
Number of CUDA devices = 4

Entering GEMM Benchmarking mode! Stand by.

Starting GEMM 1: M = 3200; N = 4000; K = 3136
GEMM matched with reference successfully! Rel diff = 0
Time for reference GEMM implementation: 0.75962 seconds
Time for my GEMM implementation: 0.573063 seconds
Completed GEMM 1

Starting GEMM 2: M = 3200; N = 400; K = 4000
GEMM matched with reference successfully! Rel diff = 9.44085e-17
Time for reference GEMM implementation: 0.1047 seconds
Time for my GEMM implementation: 0.0753251 seconds
Completed GEMM 2

Starting GEMM 3: M = 3200; N = 40; K = 4000
GEMM matched with reference successfully! Rel diff = 7.62202e-15
Time for reference GEMM implementation: 0.0158935 seconds
Time for my GEMM implementation: 0.00775442 seconds
Completed GEMM 3
darve@icme-gpu1:~/final_project$
```

Testing scripts

- `run_seq.sh` : runs the sequential code to generate the reference results and files; **edit the file to fix the path** for your account.
- `simple.sh` : runs the code with some default settings
- `test.sh` : runs all the grading tests with a given number of processes
- `test_all_N_all_modes.sh` : runs all the grading tests with N=1, 2, 3, 4. This will run the code 12 times.

test_all_N_all_modes.sh

```
darve@icme-gpu1:~/final_project$ ls -1 Outputs
CpuGpuDiff-1-1.txt
CpuGpuDiff-1-2.txt
CpuGpuDiff-1-3.txt
CpuGpuDiff-2-1.txt
CpuGpuDiff-2-2.txt
CpuGpuDiff-2-3.txt
CpuGpuDiff-3-1.txt
CpuGpuDiff-3-2.txt
CpuGpuDiff-3-3.txt
CpuGpuDiff-4-1.txt
CpuGpuDiff-4-2.txt
CpuGpuDiff-4-3.txt
NNErrors-1-1.txt
NNErrors-1-2.txt
NNErrors-1-3.txt
NNErrors-2-1.txt
NNErrors-2-2.txt
NNErrors-2-3.txt
NNErrors-3-1.txt
NNErrors-3-2.txt
NNErrors-3-3.txt
NNErrors-4-1.txt
NNErrors-4-2.txt
NNErrors-4-3.txt
darve@icme-gpu1:~/final_project$
```

CpuGpuDiff-[no of processes]-[grading mode].txt

Example output in single precision

```
darve@icme-gpu1:~/final_project/Outputs$ cat CpuGpuDiff-4-1.txt
Iteration       Max Err W0      Max Err W1      Max Err b0      Max Err b1      L2 Err W0       L2 Err W1       L2 Err b0       L2 Err b1
0               7.07739e-10     3.98307e-10     1.88914e-06     6.67562e-07     5.05204e-09     6.178e-09       1.16514e-06     5.37949e-07
600             6.43032e-08     8.0993e-08      3.43646e-07     5.38849e-07     7.74904e-08     9.42907e-08     3.2624e-07      5.93339e-07
1200            9.12034e-08     1.08122e-07     5.71564e-07     1.11143e-06     1.06173e-07     1.15545e-07     4.02672e-07     8.38676e-07
1800            1.14456e-07     1.24401e-07     7.93401e-07     6.96815e-07     1.31647e-07     1.32361e-07     4.82591e-07     6.03463e-07
2400            1.4416e-07      1.43386e-07     7.05875e-07     1.14509e-06     1.5006e-07      1.44997e-07     5.39451e-07     9.50739e-07
darve@icme-gpu1:~/final_project/Outputs$
```

```
darve@icme-gpu1:~/final_project/Outputs$ cat NNErrors-4-1.txt
Mismatches for W[0]
No errors were found
Mismatches for b[0]
No errors were found
Max norm of diff b/w seq and par: W[0]: 1.53509e-07, b[0]: 7.68691e-07
l2  norm of diff b/w seq and par: W[0]: 1.58221e-07, b[0]: 5.41653e-07
Mismatches for W[1]
No errors were found
Mismatches for b[1]
No errors were found
Max norm of diff b/w seq and par: W[1]: 1.52698e-07, b[1]: 1.04114e-06
l2  norm of diff b/w seq and par: W[1]: 1.52444e-07, b[1]: 9.65748e-07
darve@icme-gpu1:~/final_project/Outputs$
```

Output of single precision test

```
*** Summary ***

2400            1.51533e-07    1.60584e-07    8.71963e-07    4.58036e-07    1.53121e-07    1.66675e-07    5.55846e-07    5.46478e-07
2400            1.52991e-07    1.42864e-07    1.20414e-06    1.22143e-06    1.51702e-07    1.48576e-07    5.61592e-07    1.03886e-06
2400            1.30362e-07    1.39192e-07    8.30441e-07    1.14509e-06    1.46052e-07    1.42407e-07    5.39342e-07    8.25964e-07
2400            1.4416e-07     1.43386e-07    7.05875e-07    1.14509e-06    1.5006e-07     1.44997e-07    5.39451e-07    9.50739e-07
600             8.50934e-08    1.23335e-07    5.2083e-07     6.19716e-07    1.08243e-07    1.34629e-07    4.10451e-07    5.52405e-07
600             8.25992e-08    1.10324e-07    4.68747e-07    8.26288e-07    1.08989e-07    1.20275e-07    4.112e-07      6.95321e-07
600             8.41509e-08    1.05962e-07    4.68747e-07    8.26288e-07    1.07864e-07    1.17327e-07    4.1562e-07     6.20857e-07
600             8.63552e-08    1.08404e-07    4.94789e-07    6.88573e-07    1.08151e-07    1.18419e-07    4.06341e-07    7.03131e-07
60              3.76119e-08    6.21742e-08    4.9341e-07     2.34095e-07    5.8519e-08     8.05621e-08    2.99763e-07    2.31579e-07
60              3.6535e-08     4.73395e-08    4.31734e-07    2.7311e-07     5.82998e-08    6.38923e-08    3.11167e-07    2.20679e-07
60              3.76807e-08    4.4873e-08     4.31734e-07    1.56063e-07    5.86922e-08    6.28102e-08    3.07064e-07    1.18657e-07
60              3.76524e-08    4.55069e-08    4.00896e-07    3.51142e-07    5.83543e-08    6.37026e-08    3.06847e-07    2.70896e-07

*** Grading mode 4 ***

main -g 4
Number of MPI processes = 4
Number of CUDA devices = 4

Entering GEMM Benchmarking mode! Stand by.

Starting GEMM 1: M = 8000; N = 10000; K = 7840
GEMM matched with reference successfully! Rel diff = 0
Time for reference GEMM implementation: 0.436547 seconds
Time for my GEMM implementation: 1.34342 seconds
Completed GEMM 1

Starting GEMM 2: M = 8000; N = 1000; K = 10000
GEMM matched with reference successfully! Rel diff = 5.08913e-08
Time for reference GEMM implementation: 0.0523902 seconds
Time for my GEMM implementation: 0.371896 seconds
Completed GEMM 2

Starting GEMM 3: M = 8000; N = 100; K = 10000
GEMM matched with reference successfully! Rel diff = 1.07542e-06
Time for reference GEMM implementation: 0.00658449 seconds
Time for my GEMM implementation: 0.0398121 seconds
Completed GEMM 3

*** Tests are complete ***
```

Example output in double precision

```
darve@icme-gpu1:~/final_project/Outputs$ cat CpuGpuDiff-4-1.txt
Iteration       Max Err W0      Max Err W1      Max Err b0      Max Err b1      L2 Err W0       L2 Err W1       L2 Err b0       L2 Err b1
0               1.8916e-18      1.67152e-18     2.45916e-15     9.56485e-16     9.70233e-18     1.66601e-17     2.29228e-15     8.76719e-16
600             1.07044e-16     1.36448e-16     6.93431e-16     7.16916e-16     1.40019e-16     1.60736e-16     6.19144e-16     7.09163e-16
1200            1.61694e-16     1.7445e-16      9.67836e-16     1.13214e-15     1.98578e-16     1.97264e-16     7.40081e-16     8.73672e-16
1800            2.01067e-16     2.07826e-16     1.10837e-15     1.01979e-15     2.39746e-16     2.27093e-16     8.58286e-16     1.14171e-15
2400            2.28528e-16     2.31712e-16     1.00543e-15     1.27974e-15     2.7219e-16      2.45211e-16     9.07254e-16     1.35153e-15
darve@icme-gpu1:~/final_project/Outputs$
```

```
darve@icme-gpu1:~/final_project/Outputs$ cat NNErrors-4-1.txt
Mismatches for W[0]
No errors were found
Mismatches for b[0]
No errors were found
Max norm of diff b/w seq and par: W[0]: 2.44395e-16, b[0]: 1.14544e-15
l2  norm of diff b/w seq and par: W[0]: 2.90717e-16, b[0]: 9.34657e-16
Mismatches for W[1]
No errors were found
Mismatches for b[1]
No errors were found
Max norm of diff b/w seq and par: W[1]: 2.45286e-16, b[1]: 1.55142e-15
l2  norm of diff b/w seq and par: W[1]: 2.58128e-16, b[1]: 1.71034e-15
darve@icme-gpu1:~/final_project/Outputs$
```

Output of double precision test

```
*** Summary ***

2400           2.42896e-16    2.57093e-16    1.23745e-15    7.82061e-16    2.75504e-16    2.75834e-16    9.3875e-16     7.60842e-16
2400           2.39221e-16    2.36368e-16    1.39213e-15    1.27974e-15    2.76257e-16    2.50582e-16    9.16374e-16    1.11895e-15
2400           2.38719e-16    2.2733e-16     1.46948e-15    1.13754e-15    2.73876e-16    2.45198e-16    9.34674e-16    1.11021e-15
2400           2.28528e-16    2.31712e-16    1.00543e-15    1.27974e-15    2.7219e-16     2.45211e-16    9.07254e-16    1.35153e-15
600            1.9313e-16     2.12373e-16    1.30966e-15    1.41082e-15    2.10763e-16    2.35401e-16    7.56632e-16    8.99852e-16
600            1.85314e-16    1.81103e-16    1.35817e-15    1.28257e-15    2.02469e-16    2.05549e-16    7.77569e-16    1.08986e-15
600            1.80601e-16    1.80084e-16    1.06713e-15    1.15431e-15    2.04892e-16    2.07799e-16    7.56676e-16    8.45257e-16
600            1.76346e-16    1.80586e-16    1.26116e-15    1.15431e-15    1.98622e-16    1.99969e-16    7.83286e-16    8.96843e-16
60             9.93503e-17    1.03392e-16    1.52217e-15    2.9069e-16     1.19902e-16    1.41782e-16    6.18216e-16    2.57816e-16
60             9.37556e-17    8.82419e-17    1.32113e-15    5.08708e-16    1.17042e-16    1.16667e-16    6.16084e-16    4.21166e-16
60             1.06466e-16    8.01509e-17    1.55089e-15    3.27027e-16    1.20311e-16    1.10646e-16    6.15539e-16    3.23161e-16
60             1.00213e-16    8.41269e-17    1.57961e-15    4.36035e-16    1.18585e-16    1.13091e-16    6.28574e-16    4.2362e-16

*** Grading mode 4 ***

main -g 4
Number of MPI processes = 4
Number of CUDA devices = 4

Entering GEMM Benchmarking mode! Stand by.

Starting GEMM 1: M = 3200; N = 4000; K = 3136
GEMM matched with reference successfully! Rel diff = 0
Time for reference GEMM implementation: 0.801072 seconds
Time for my GEMM implementation: 0.580784 seconds
Completed GEMM 1

Starting GEMM 2: M = 3200; N = 400; K = 4000
GEMM matched with reference successfully! Rel diff = 9.44085e-17
Time for reference GEMM implementation: 0.106404 seconds
Time for my GEMM implementation: 0.0771477 seconds
Completed GEMM 2

Starting GEMM 3: M = 3200; N = 40; K = 4000
GEMM matched with reference successfully! Rel diff = 7.62202e-15
Time for reference GEMM implementation: 0.0146576 seconds
Time for my GEMM implementation: 0.00786759 seconds
Completed GEMM 3

*** Tests are complete ***
```

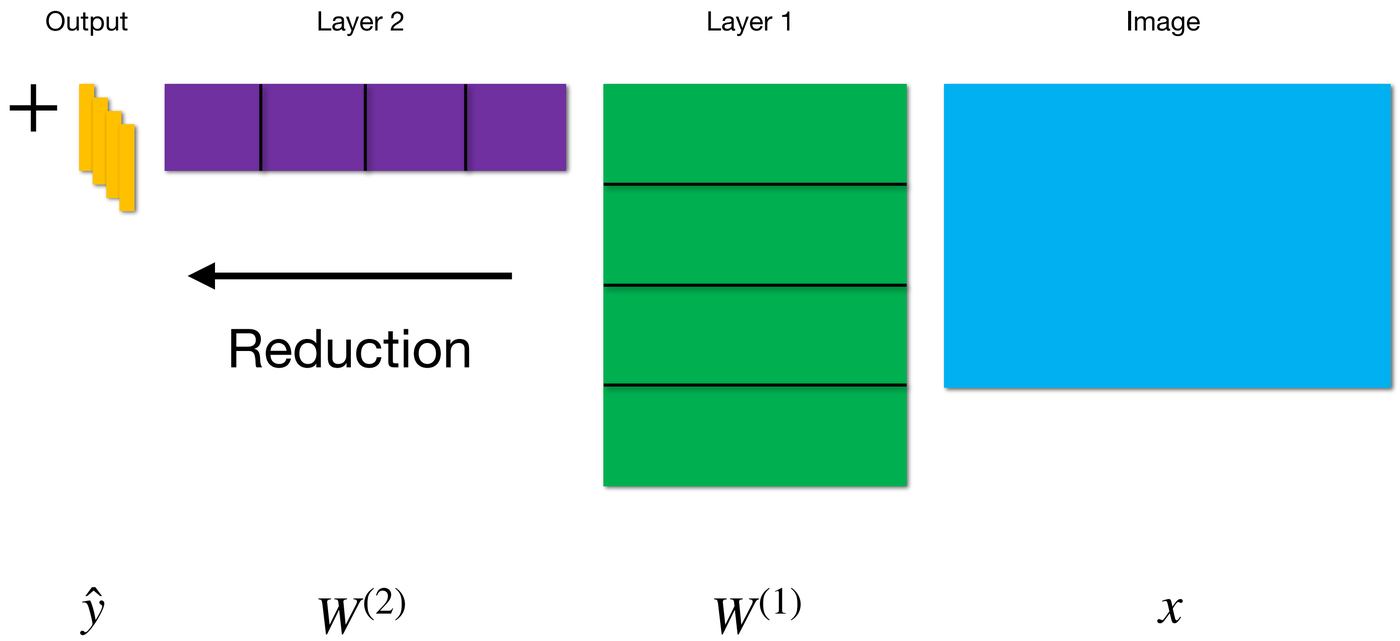
Additional scripts

- `run_nsys.sh` : NVIDIA Nsight Systems; `nsys profile`
- `run_cu.sh` : NVIDIA Nsight Compute; `ncu --set full`
- Generate reports on the cluster.
- Visualize locally on your machine.

Extra Credit
Can we do better?

Extra credit 1:
Better MPI with lower communication

Better partitioning

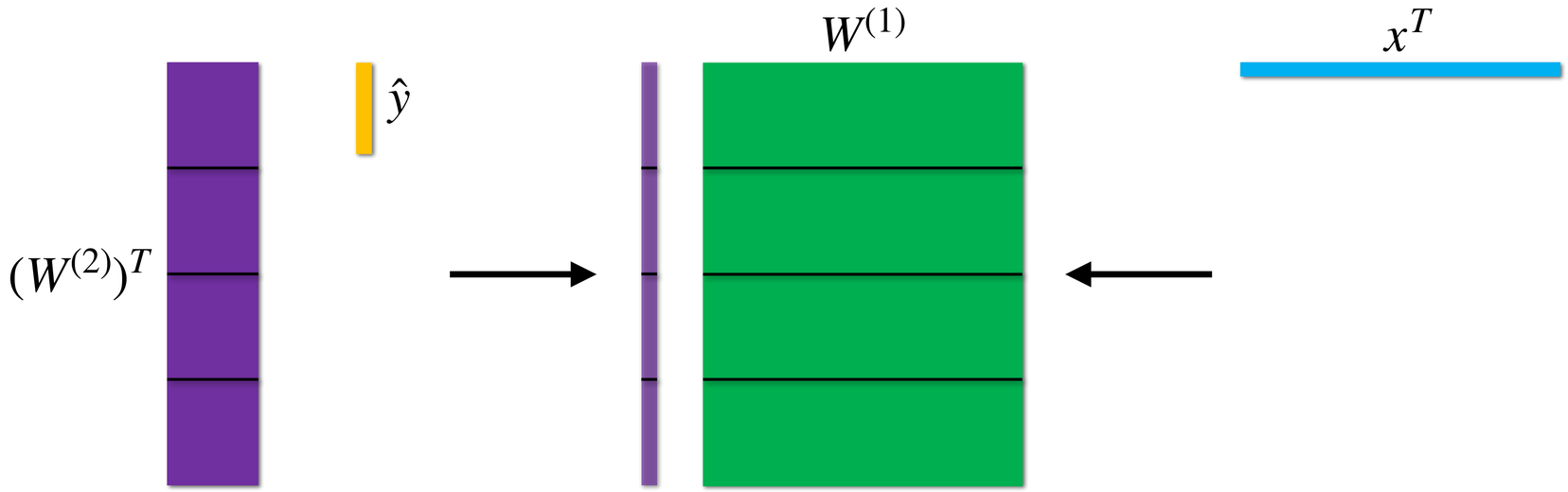


Final reduction is over a much smaller dataset.

Backpropagation

- See Final Project write-up.
- Backpropagation is used to compute the gradient of the loss function with respect to the weights and biases.
- The algorithms proceeds from the output to the input (reverse of the forward pass).
- Applying the chain rule to compute the gradient leads to multiplication with the transpose of the weight matrices.

Backpropagation steps



$(W^{(2)})^T\hat{y}$ $\quad$ $W^{(1)} \leftarrow W^{(1)} - \alpha[(W^{(2)})^T\hat{y}]x^T$ $\quad$ x^T

- Equations are simplified; the non-linear activation functions are not included. But the pattern of computation is the same.

- **No communication is required with this modified algorithm!**

Extra credit 2:
Use Tensor Cores

Tensor cores

- See instructions and information in the final project PDF for details.
- **The reduced precision means that you won't be able to get the same result as in single or double precision.**
- However, you can still converge the DNN to a good solution in this setting.
- Discuss how you implemented your code and what performance improvements you observed.
- Has this change affected the precision of your DNN model?

Final advice

- Start early
- Focus on testing and debugging **strategy**
- Create appropriate **unit tests** for each section of code you write. Don't just rely on the grading tests.
- Watch for corner cases (loop bounds, array sizes, integer divisions)
- Validate your code before proceeding to the next step
- Start with the simplest working implementation