CME 213, ME 339 Spring 2022
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

- CUDA programming
- Streaming multiprocessor, warp, thread block, grid
- Memory subsystem, cache, bandwith
- Performance and optimization
- CUDA profiler
- OpenACC

Final Project

Goal

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

Logistics

Turn in	Date	Grade weight
Preliminary report + working code (not optimized)	Friday, May 27th	20%
Final report (4 pages) + code + optimization	Tuesday, June 7th	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 to outline what you have done.

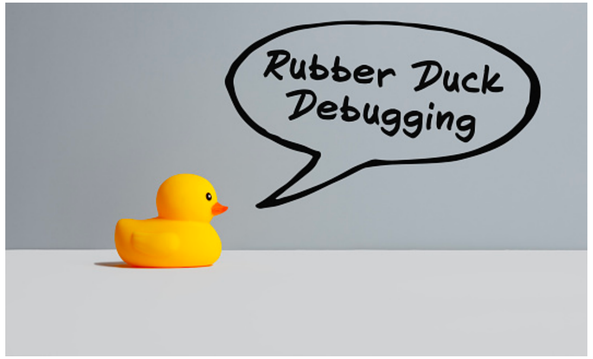
Final report

- 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)?

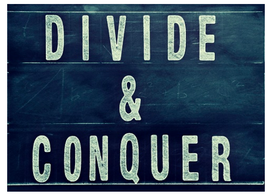
Correctness and debugging

- 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 the basic testing framework, but you will need to write additional tests to make sure your code works correctly.

Tips for debugging

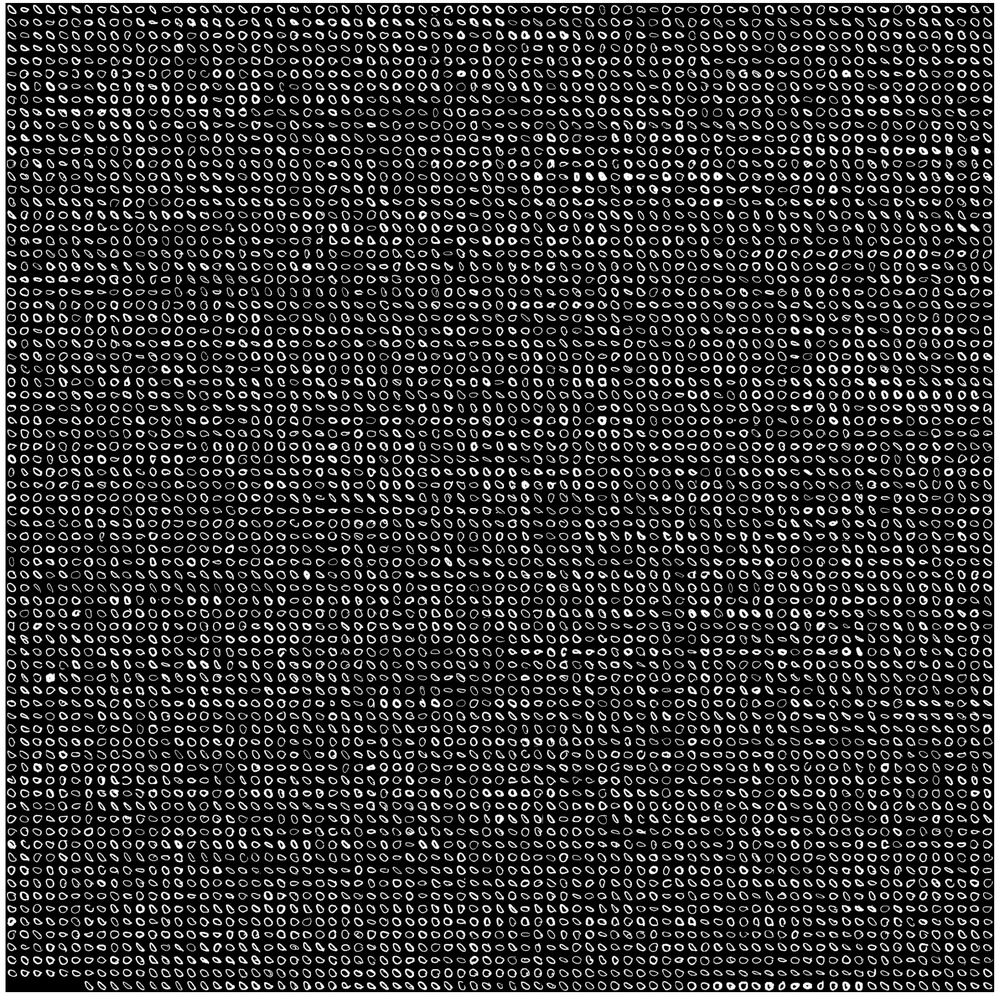


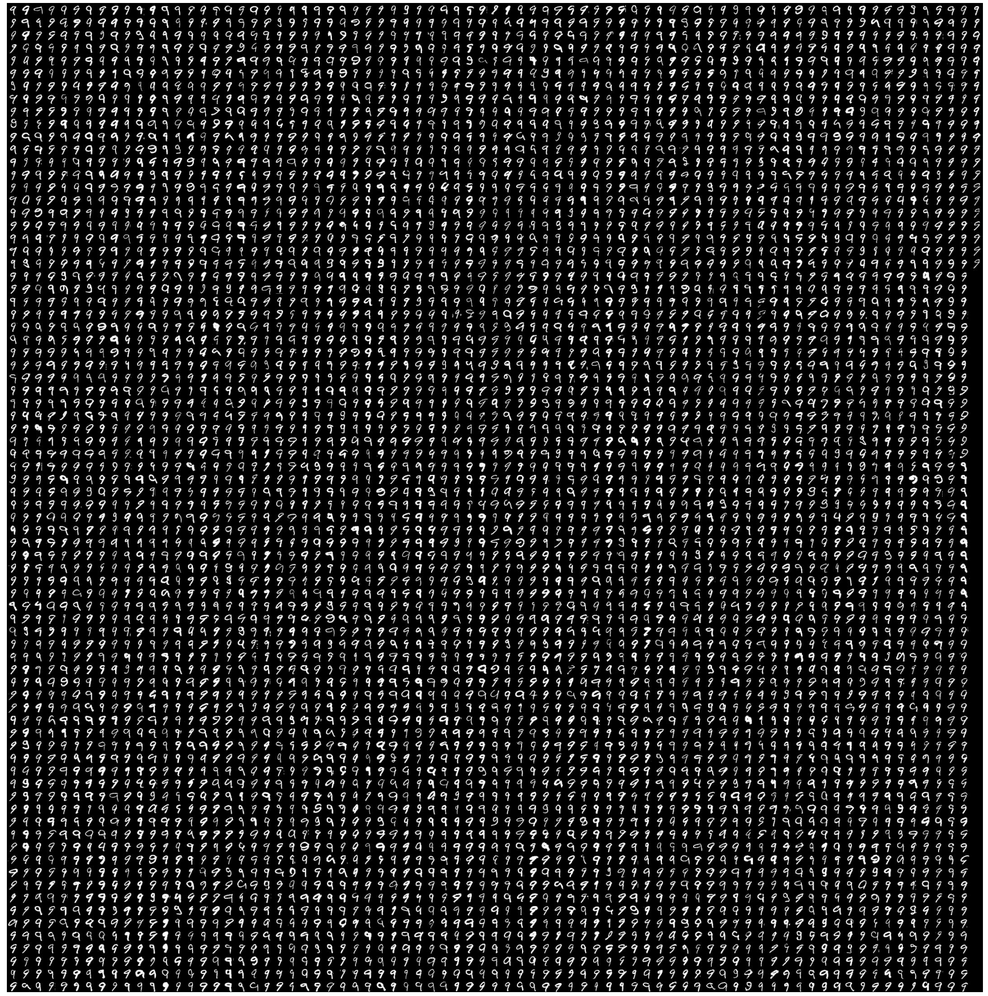
- Too many to list!
- Use a strategy! The worst approach is the **random** search…
- **Unit** tests: every time you write a small block of code, you should test it to make sure it is correct.
- Assume by default that **every line of code has an error.** This is your null hypothesis.
- Use **divide and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



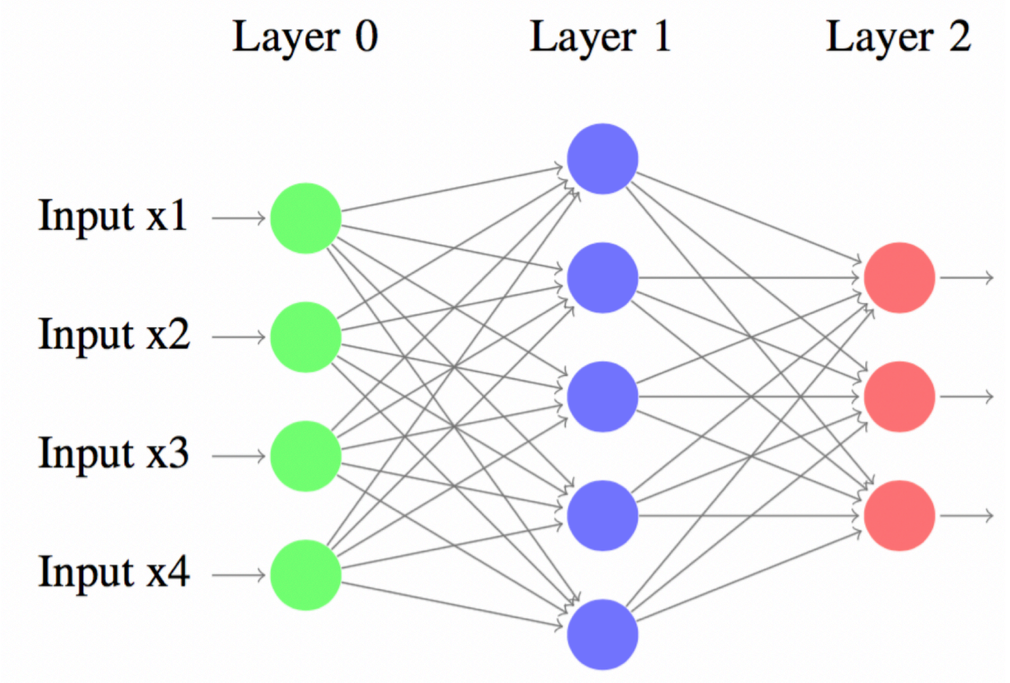
- Spend time thinking about ways to check the code: known mathematical result or inequality, result can obtained through other means (e.g., **CPU** or with pencil and paper).
- Test small cases that can be verified by hand.
- Main tools: **`assert` and `print`.**
- Spend time reading compiler error messages; it is 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.
- **Golden rule of debugging:** time spent writing tests, adding asserts, and print statements = 0! Time spent debugging without a strategy = $+\infty$.

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

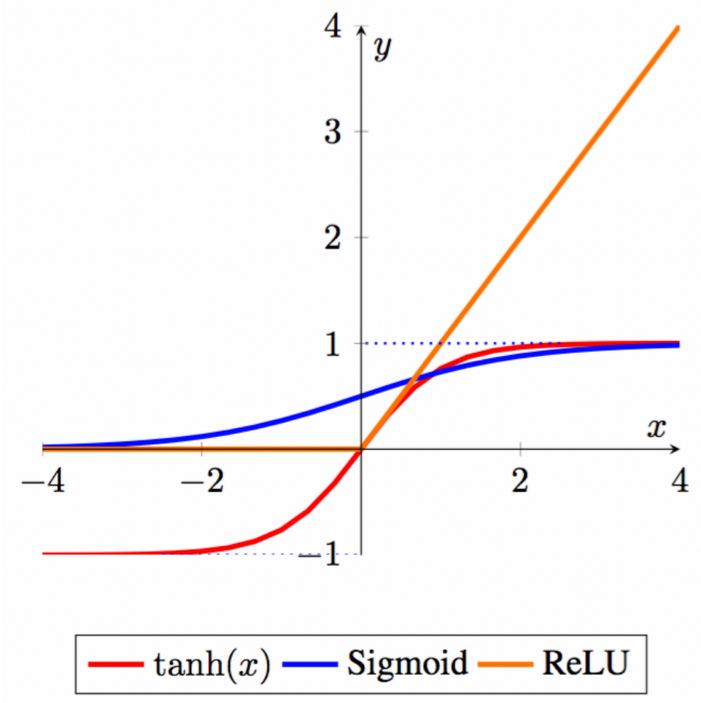
A function commonly used to map a vector of real numbers to a probability vector.

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

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 for this project we use a regular dense matrix.
- Output of layer: $a_i = \sigma(z_i)$

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 some error metric.
- In this case, we use the cross-entropy.
- The cross-entropy allows computing the dissimilarity between two distributions.

Cross-entropy

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

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:

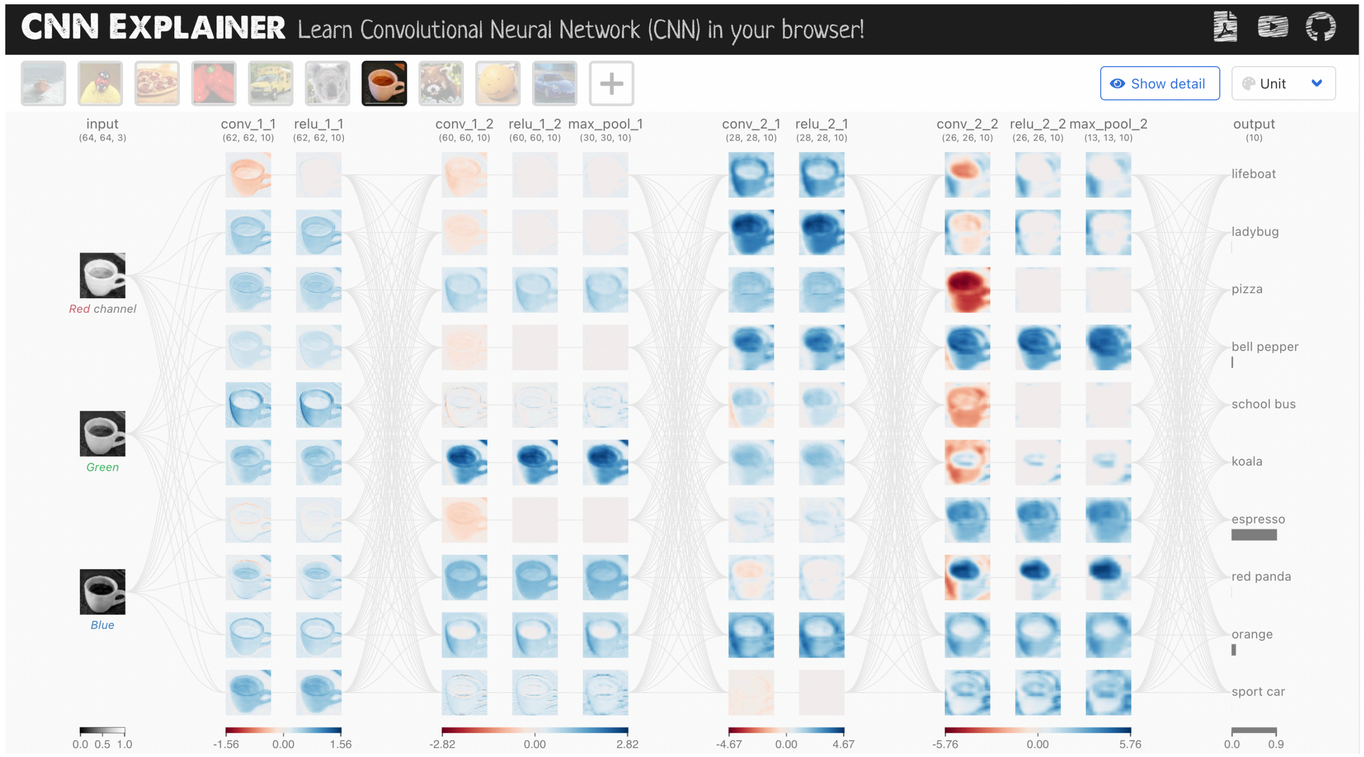
$$J(p) = \frac{1}{N} \sum_{i=1}^{N} H(y_i, \hat{y}_i)$$

p weights and biases of network = all parameters

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:

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

Stochastic gradient

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

Benefits of stochastic gradient descent

One epoch: process all the input images.

If we use a small 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
- 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:

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

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

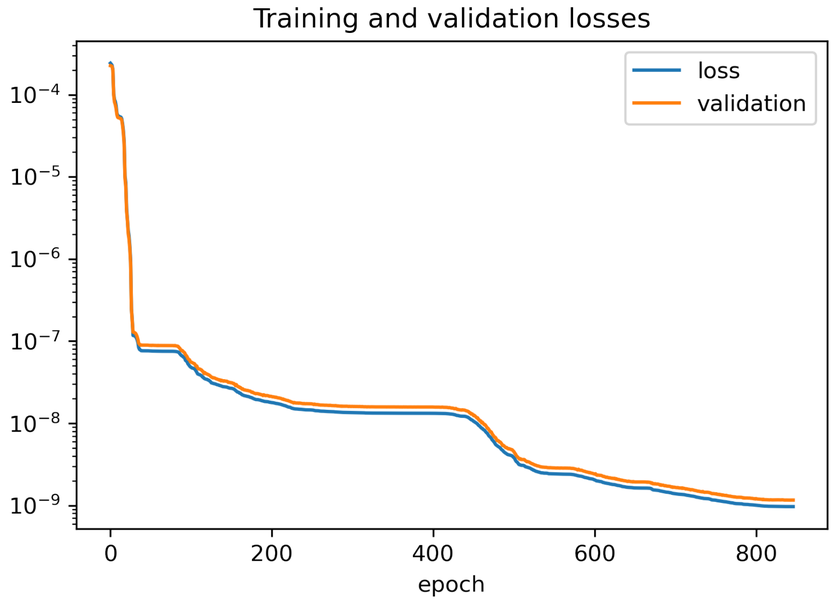
Magnitude of weights

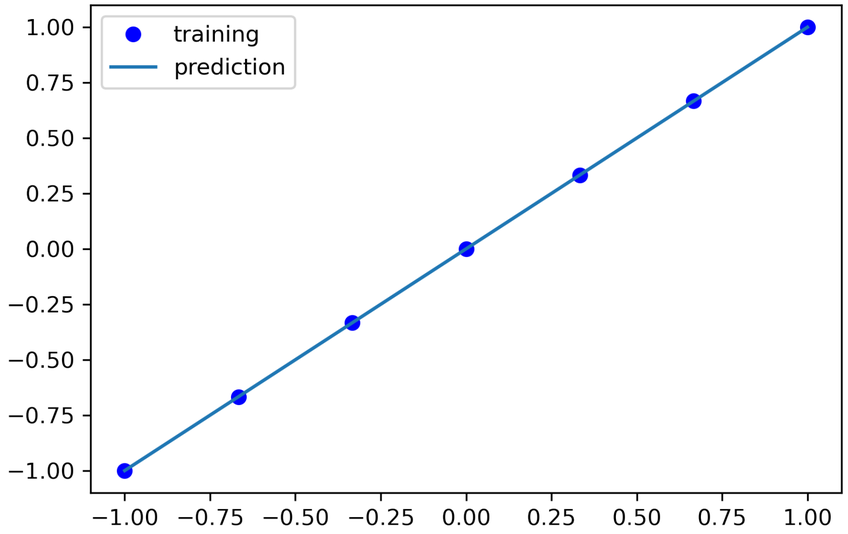
- Small weights and biases result in the DNN that is more linear
- Large weights and biases all the DNN to approximate sharp jump in the function.
- Regularization is needed to improve the accuracy of the DNN.

How we can figure out how much regularization is needed

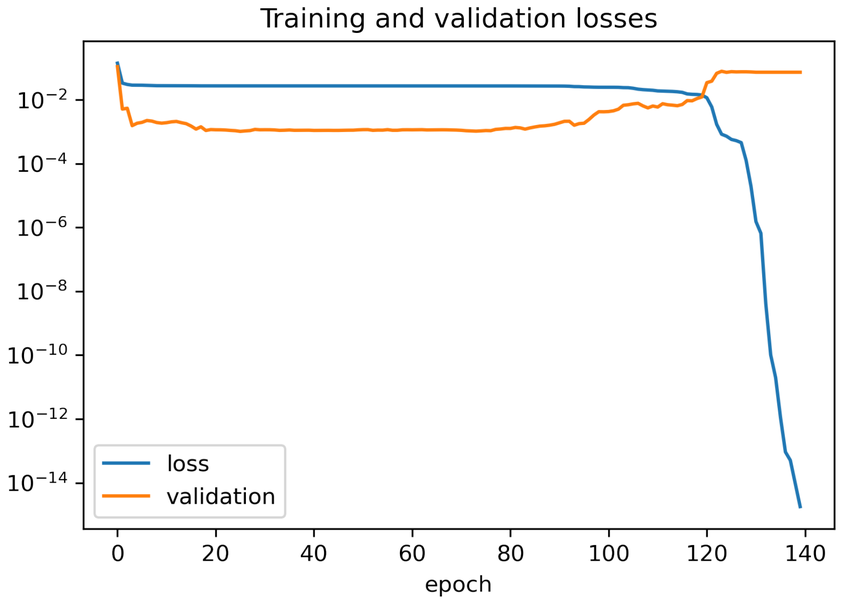
- Training set: used to minimize loss; involved in defining the gradient
- Validation set: used to evaluate model; how accurate is it? Avoids overfitting
- Over-fitting is similar to what happens when fitting a high-order polynomial in certain situations: we can get wild oscillations.

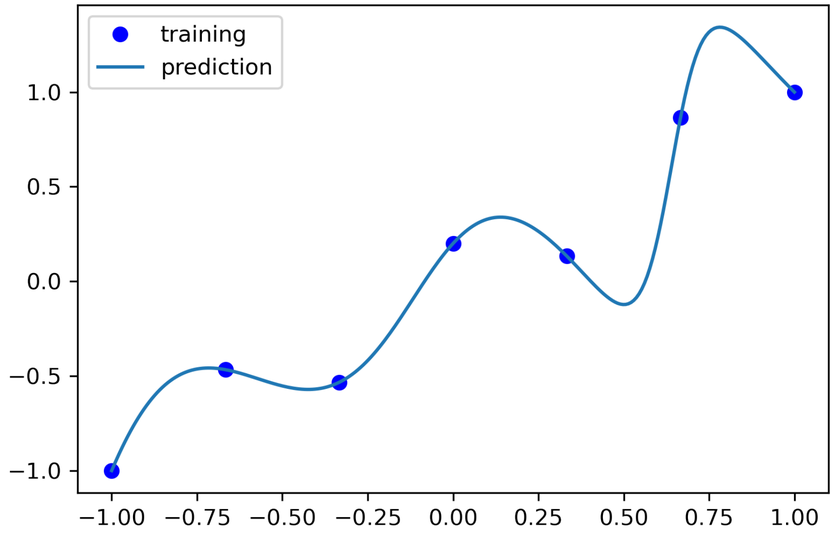
Example: small 2-layer DNN with width 8



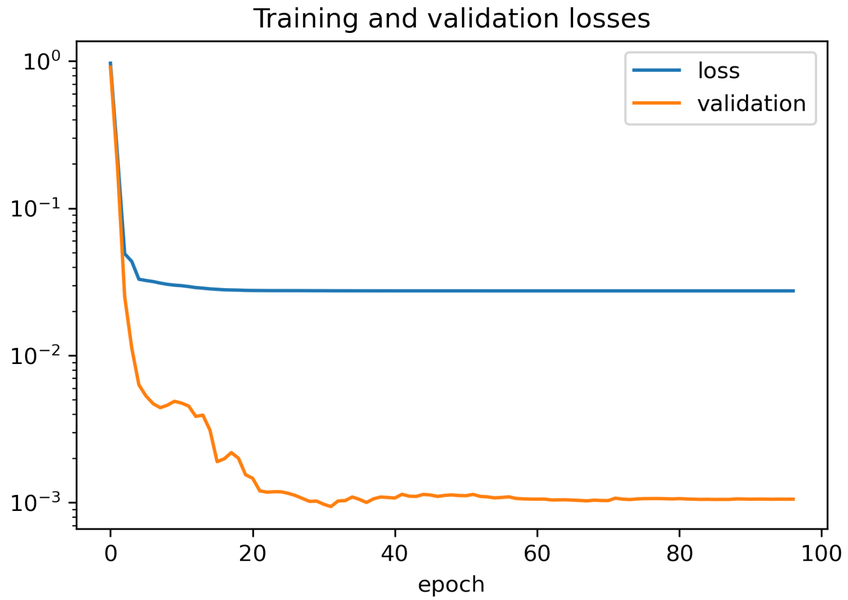


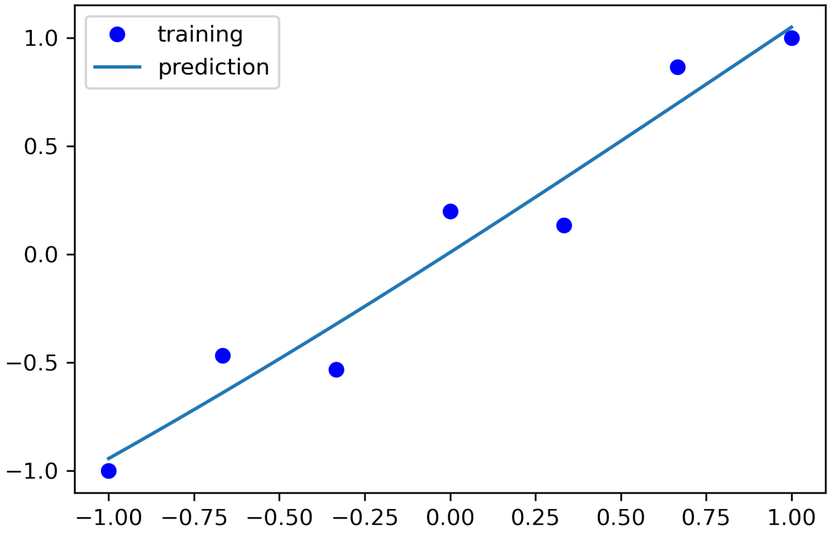
With noise added to data



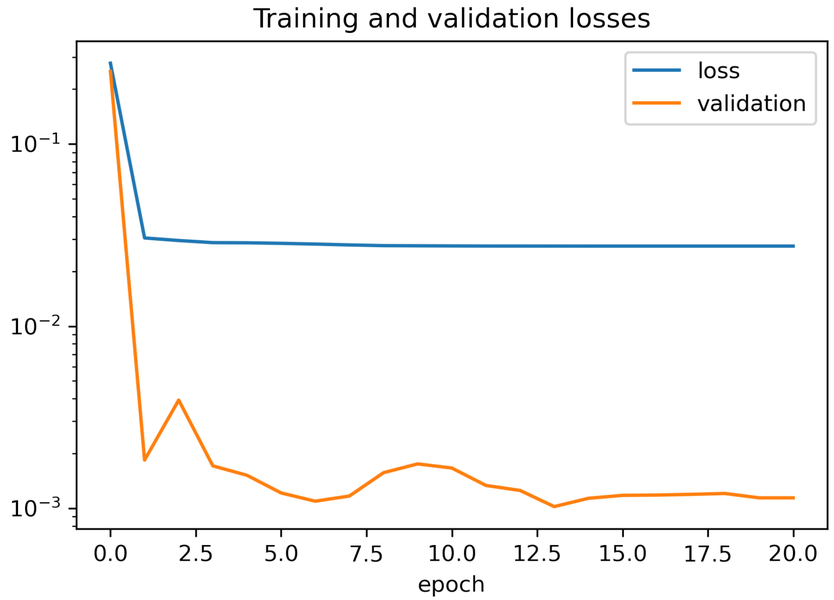


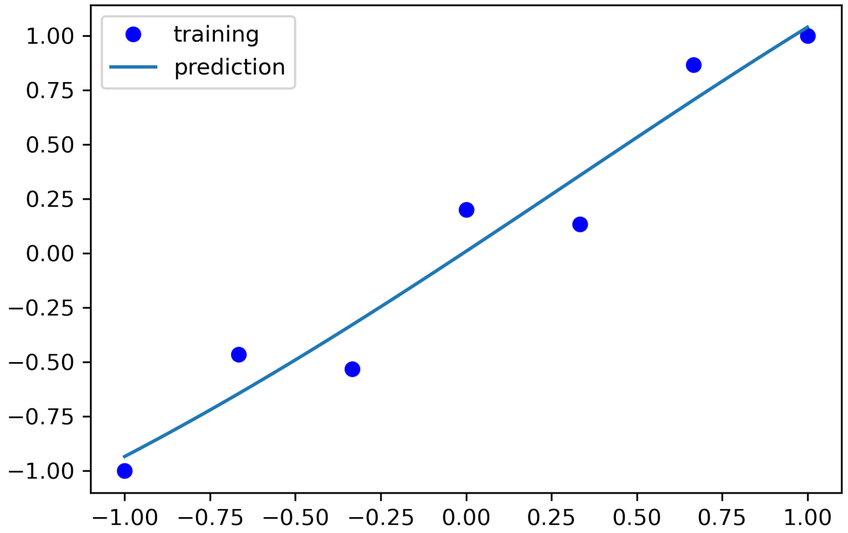
Fix 1: reduce the size of the DNN; for example with width 1



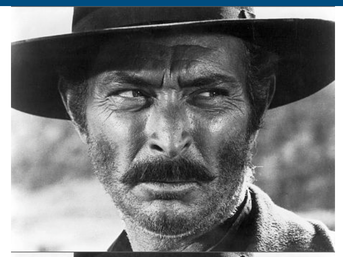
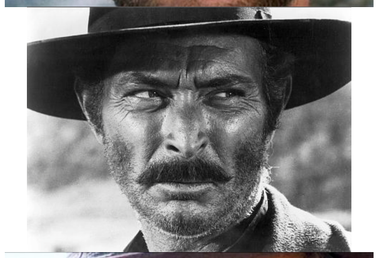
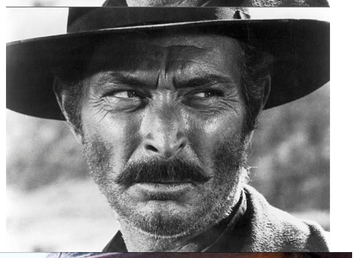
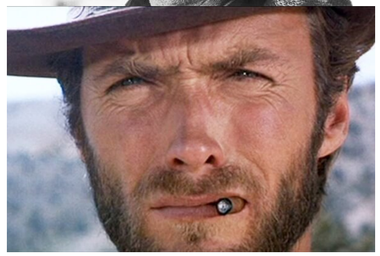


Fix 2: add regularization, e.g., $\lambda = 10^{-3}$





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 no 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 minibatch 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

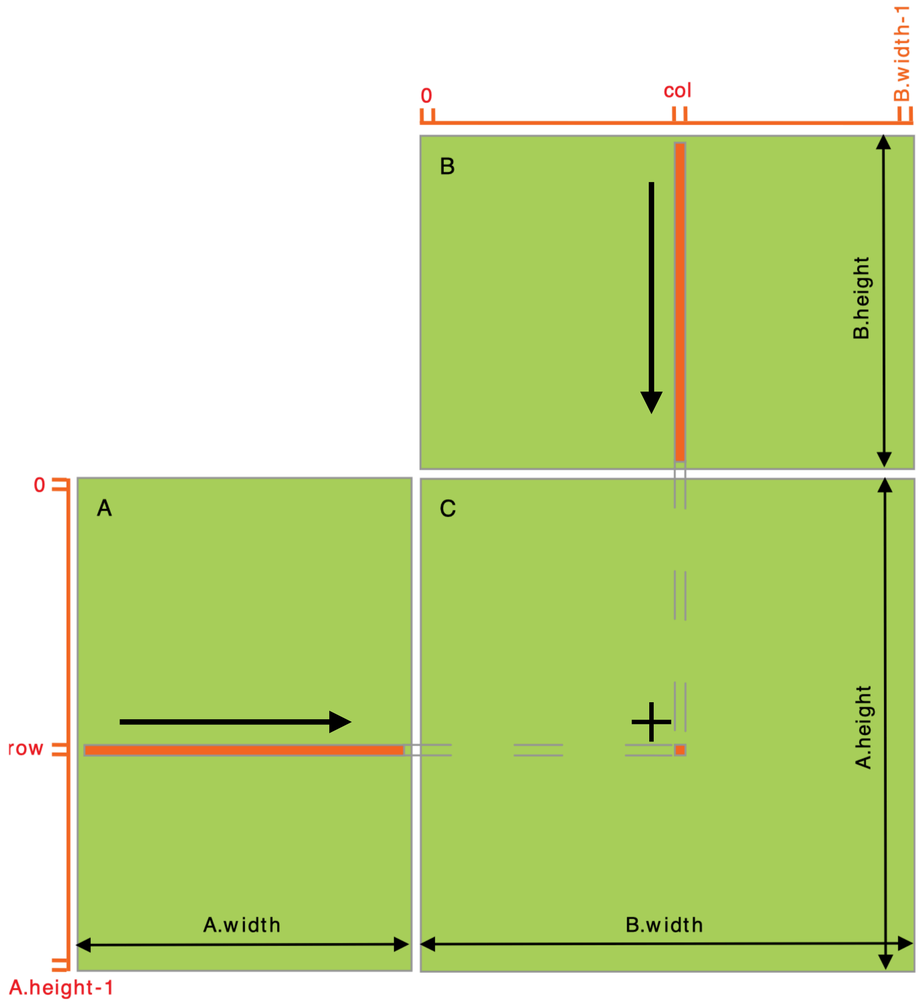
Matrix-matrix products in CUDA

Simplest algorithm

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

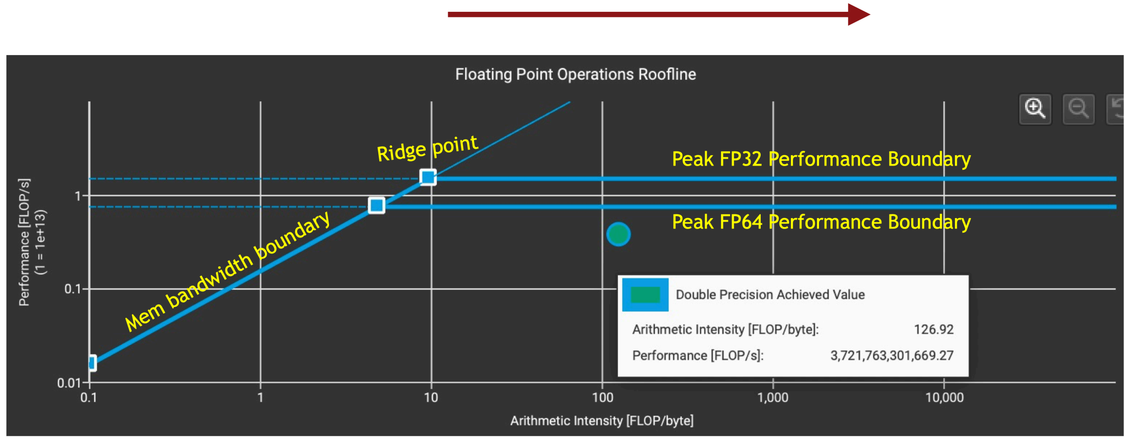
One thread per matrix entry

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

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

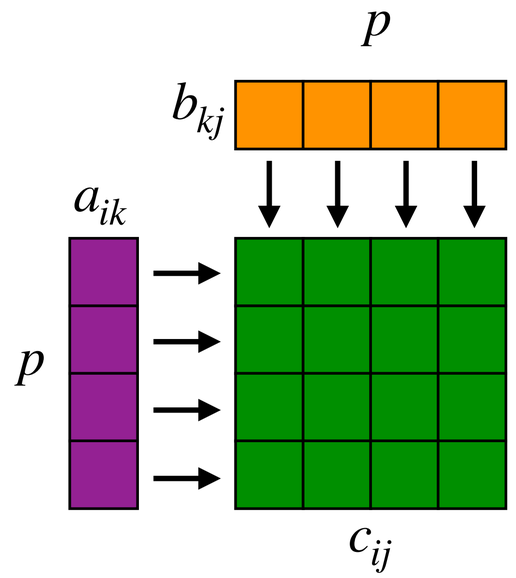
Load a_{ik} and b_{kj} for $i_0 \leq i \leq i_1$ and $j_0 \leq j \leq j_1$.

Outer product form.

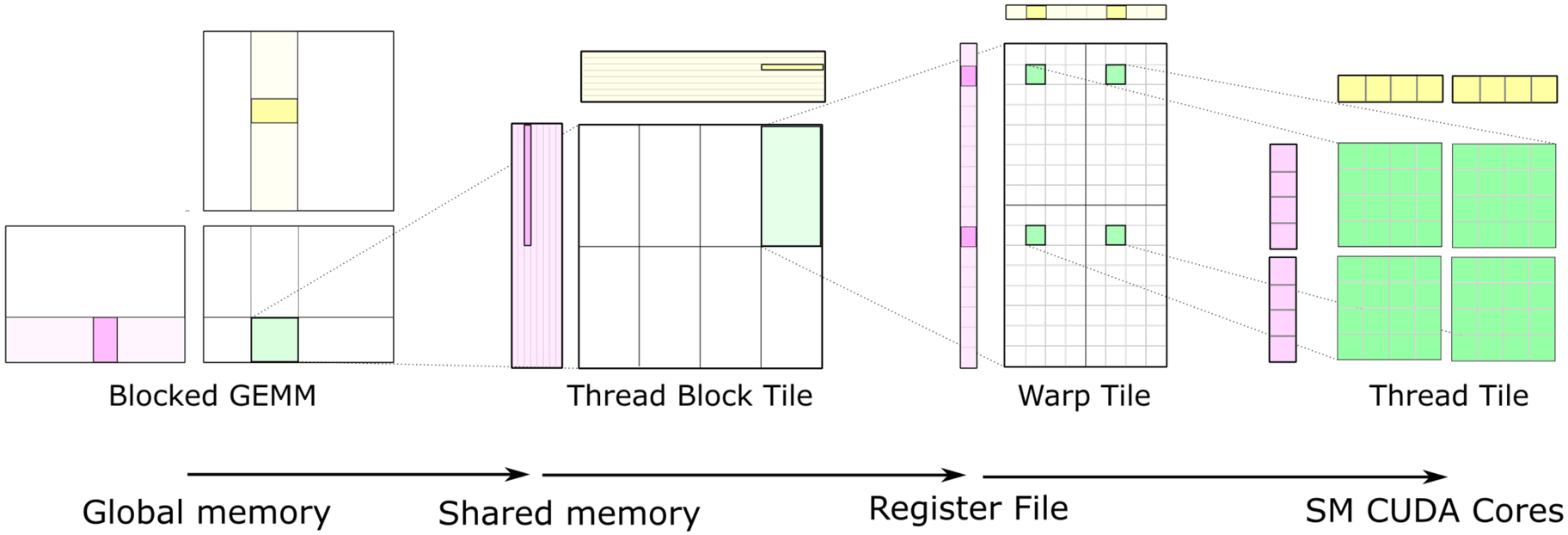
Reads: $2p$

Flops: $2p^2$

With p big enough, we can generate enough flops.



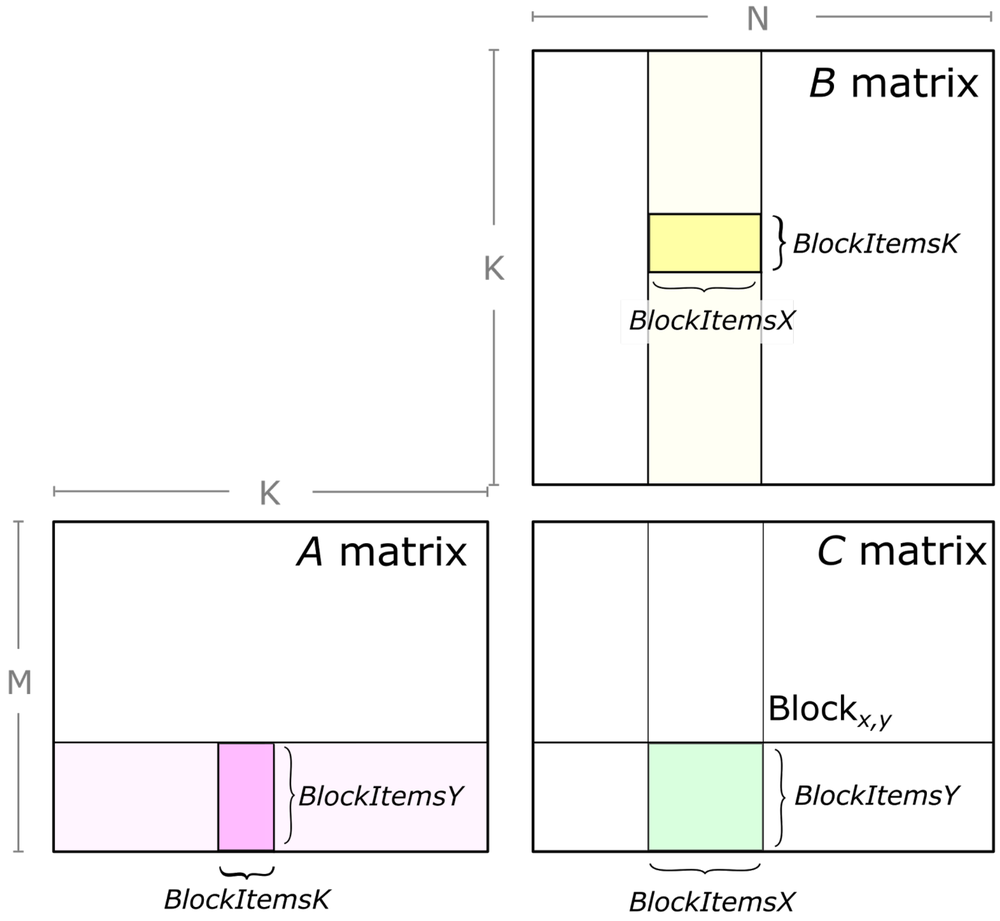
Hierarchical decomposition of the product



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

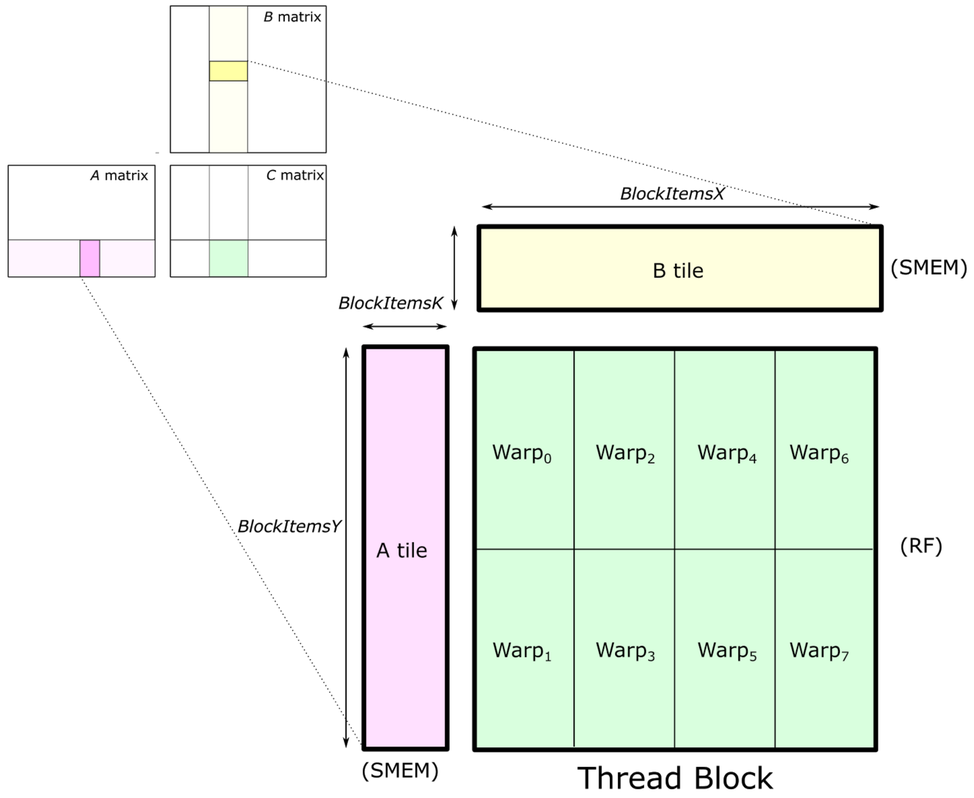
Global memory

- 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.



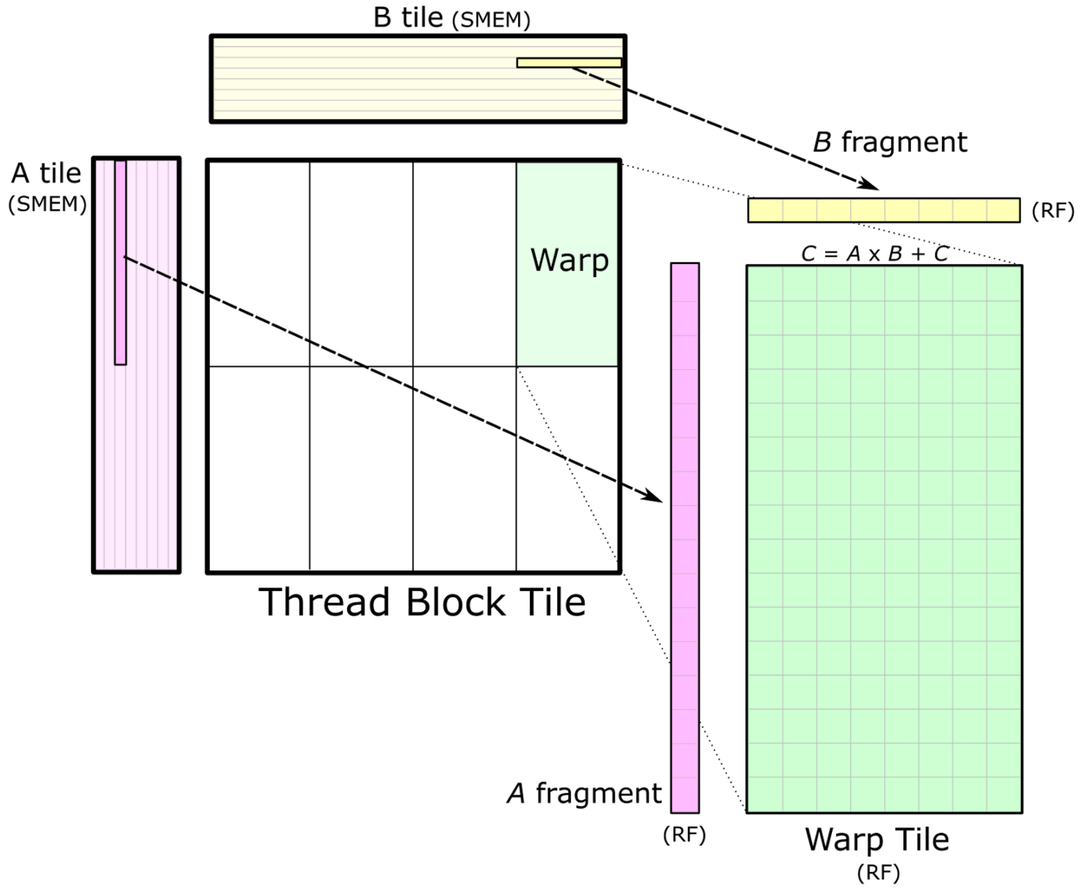
Thread block workload

- 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.



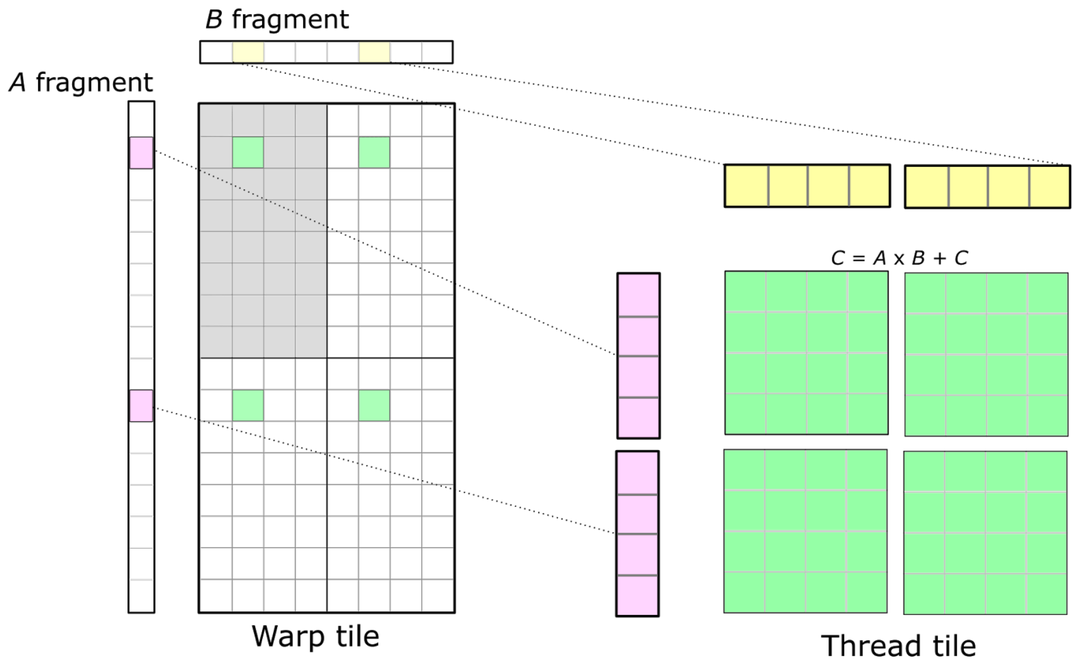
Warp level workload

- 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.



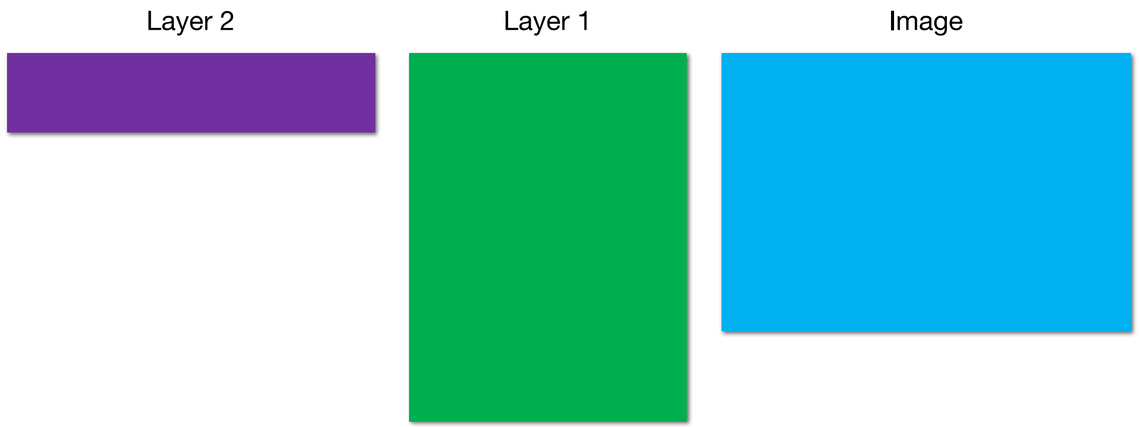
Thread level workload

- 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.

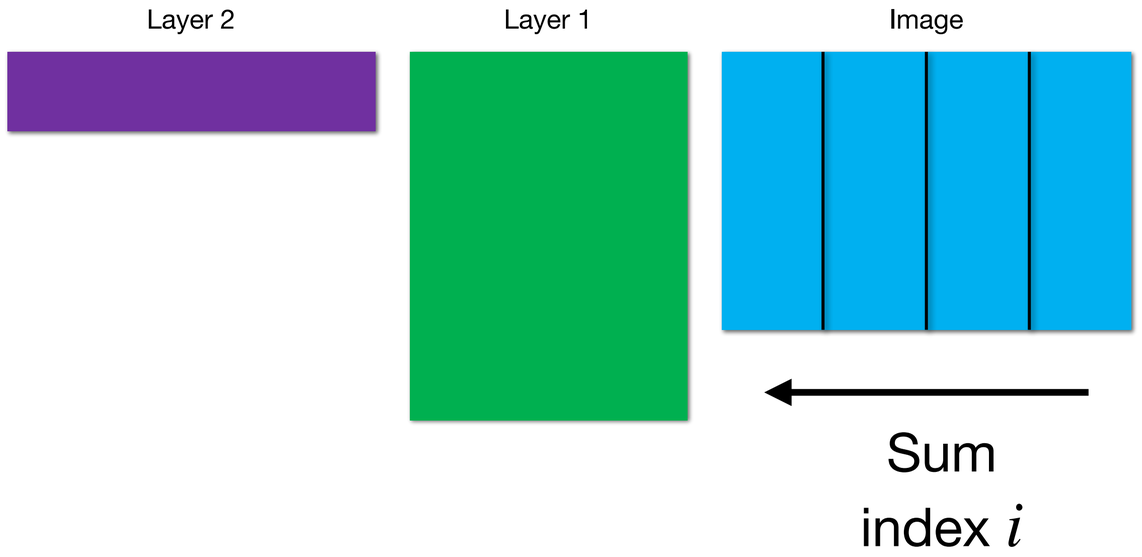


Distributed MPI

- 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

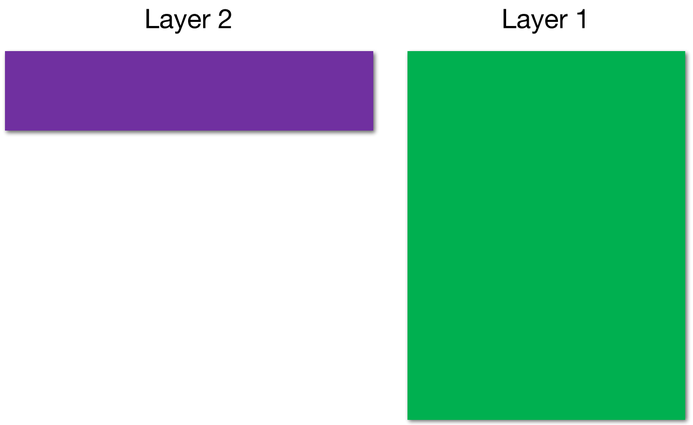


Partition the input across the columns.

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

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

Reduction

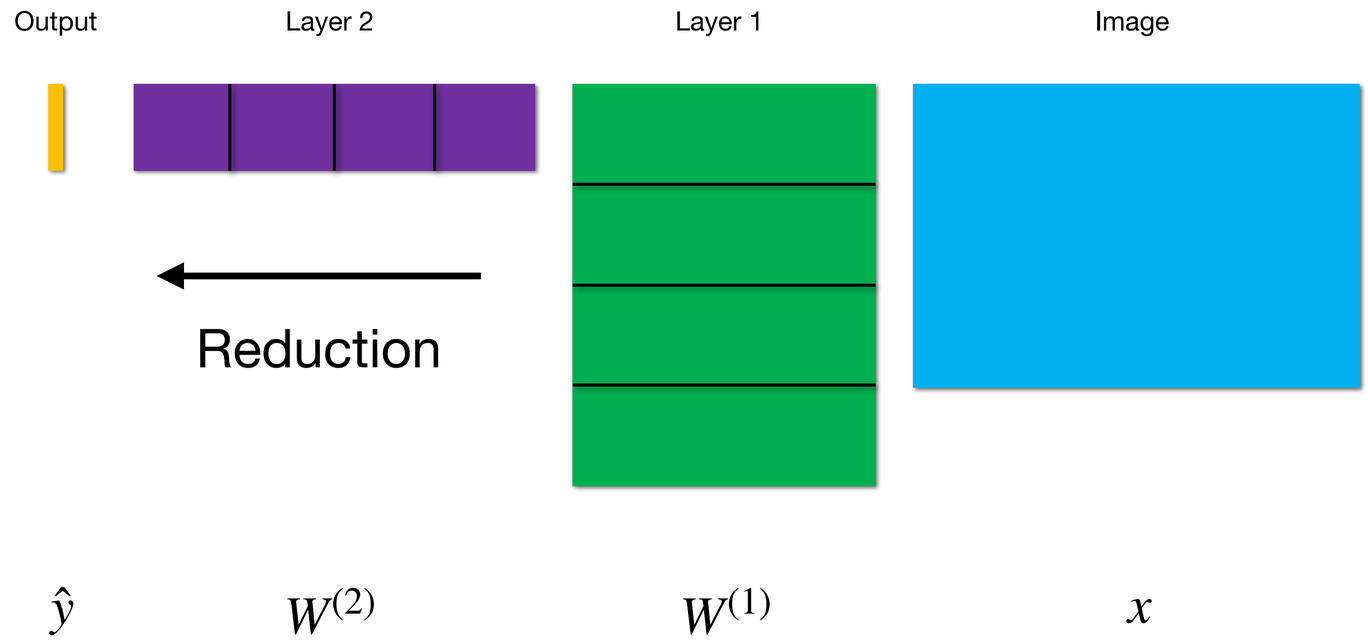


- Reduction of all weights and biases requires sending the data across the network between the different computing nodes.

- This can be time consuming even on a very fast network.

Extra Credit
Can we do better?

Better partitioning

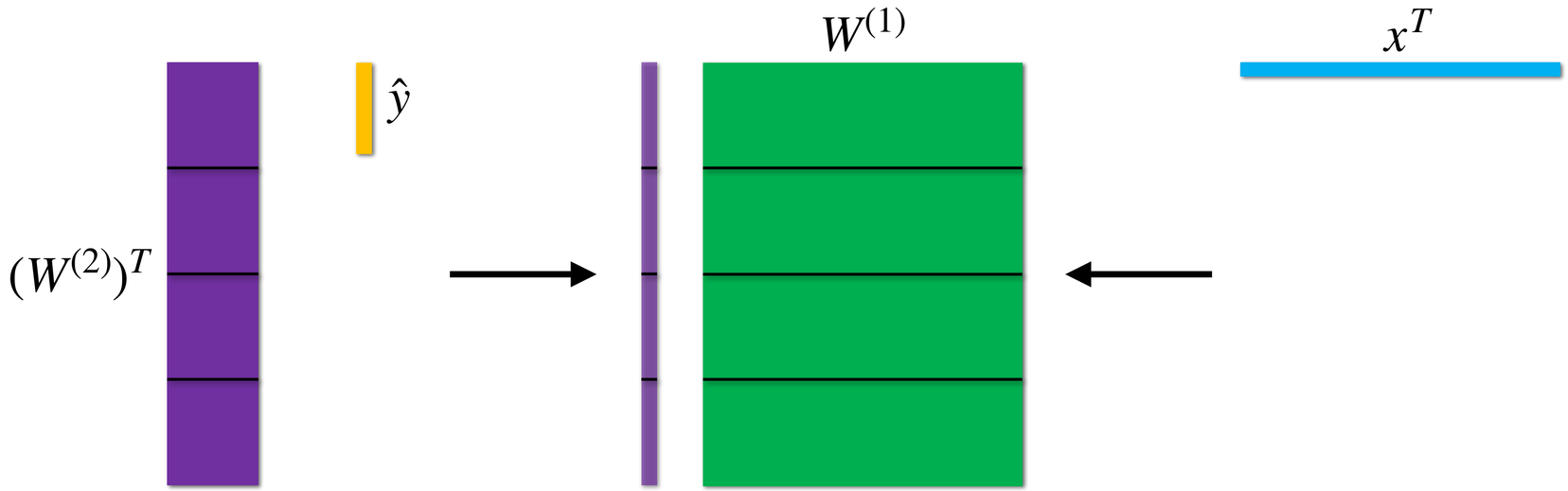


Final reduction is over a much smaller dataset.

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 required.

Final advice

- Start early
- Focus on testing and debugging **strategy**
- 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