THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE

Training Neural Networks on GPUs Using CUDA/C++

Jake Elkins, MAE PhD Candidate

Semester Project – CPE 613, Gen Purpose GPU Computing

Table of Contents

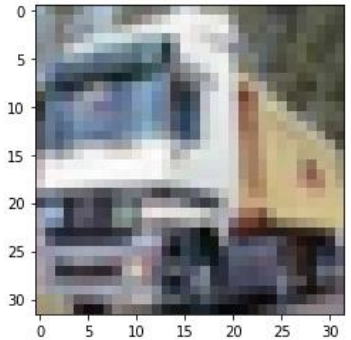
Training Neural Networks

The basics of architectures, gradient descent, and backpropagation

What is a neural network?

General nonlinear function approximators.

Suppose we want a program that can correctly tell what object is in a picture (image classification):



x

Neural Network

f(x)

'truck'
confidence: 94.0%

y

This problem becomes learning a *function* that maps given inputs *x* to given outputs *y*

$$y = f(x)$$

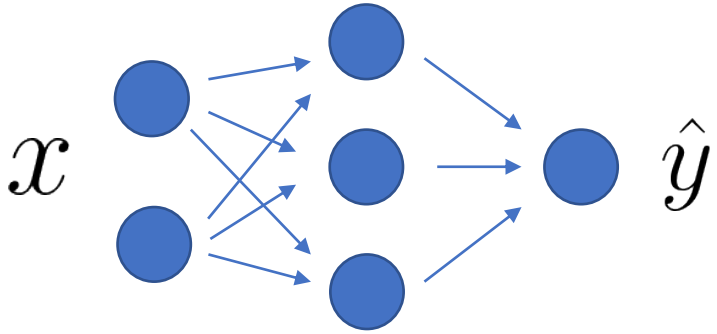
What is a neural network?

But what do we use as the function approximator for *f*?

$$y = f(x)$$

➡ Neural networks: loosely modeled after how the brain's neurons work, neural networks are data processing instruments iteratively trained with data

Neural network research began in the lates 1950, but the data and processing power was not available yet. Once data and compute drastically increased in the last 20 years, neural networks showed significant modeling capabilities in many fields. "Deep learning"



Graphical illustration of a neural network with one hidden layer.

$$\hat{f}(x) = W^3\phi(W^2\phi(W^1x))$$

Mathematically, a neural network is a recursive matrix-multiply, vector-add operation.

What is a neural network?

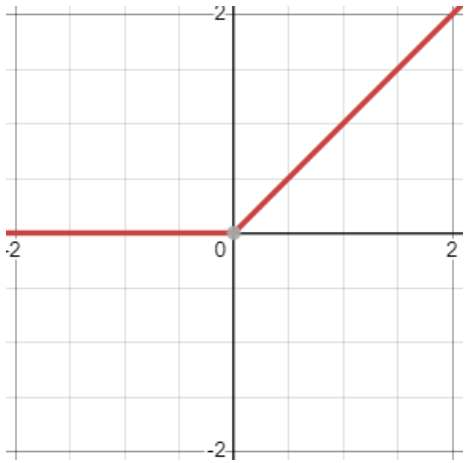
$$\hat{f}(x) = W^3\phi(W^2\phi(W^1x))$$

Weights

Activation functions

ReLU:

$$\phi(z) = \texttt{max}(0, z)$$



Each layer consists of trainable parameters: a *weight* and a *bias*

For layer *i:*

$$a_{i+1} = W^i z_i + b^i$$

$$z_{i+1} = \phi(a_{i+1})$$

Some practitioners append a 1 to the input vectors at each layer to account for a bias term, rather than explicitly training a bias.

How do we train a neural network?

We want our approximation to match the real function as close as possible:

$$\hat{f}(x) \rightarrow f(x)$$

or, equivalently

$$\hat{y} \rightarrow y$$

Our problem now formally becomes:

Given a dataset X, Y minimize $J = \frac{1}{N}\sum_{i=0}^{N}(y_i - \hat{y}_i)^2$

by updating W^i, b^i for i = 1, …, num_layers

How do we train a neural network?

At each training “epoch”:

Randomly sample a batch of size *N* from features dataset *X* and corresponding labels dataset *Y*

$$(x, y) \sim (X, Y)$$

Pass *x* through the neural network (“forward pass”)

$$\hat{y} = W^3 \phi(W^2 \phi(W^1 x))$$

Calculate error (“loss”)

$$J = \frac{1}{N} \sum_{i=0}^{N} (y_i - \hat{y}_i)^2$$

Update weights (“backward pass”) …

How do we train a neural network?

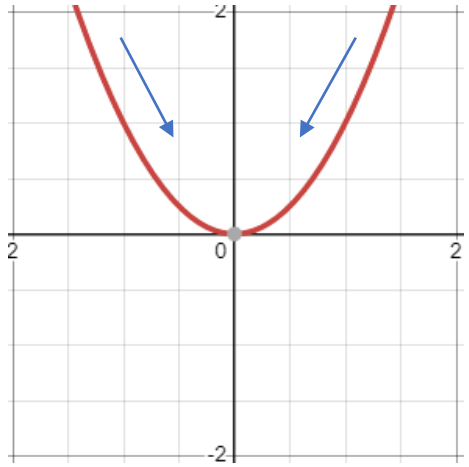
Updating the weights to minimize the loss function is done by *gradient descent:*

$$\theta(k+1) = \theta(k) - \eta \nabla_\theta J$$

where $\theta(k)$ = some parameter at timestep *k*

$\nabla_\theta J$ = gradient of loss w.r.t. parameter

η = learning rate (step size)



Edit the NN weights in the direction of decreasing loss.

How do we find the gradient of the loss with respect to our weights and biases?

→ ***Backpropagation***

How do we train a neural network?

Backpropagation: using calculus to determine $\nabla_\theta J$ for each of our weights and biases.

Example: consider W^2 in $\hat{y} = W^3\phi(W^2\phi(W^1x))$ and loss $J = \frac{1}{2}(y-\hat{y})^2$

$$\nabla_{W^2} J = \frac{\partial J}{\partial W^2} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z_2}\frac{\partial z_2}{\partial a_2}\frac{\partial a_2}{\partial W^2}$$ via chain rule

$$= [((W^3)^T(y-\hat{y})) \odot \phi'(a_2)]z_1^T$$

This is done in batches for each weight and bias in the network, at each epoch.

Backpropagation is the critical tool for training neural networks and is computationally intensive.

How do we train a neural network?

At each training “epoch”:

Randomly sample a batch of size *N* from features dataset *X* and corresponding labels dataset *Y*

$$(x, y) \sim (X, Y)$$

Pass *x* through the neural network (“forward pass”)

$$\hat{y} = W^3 \phi(W^2 \phi(W^1 x))$$

Calculate error (“loss”)

$$J = \frac{1}{N} \sum_{i=0}^{N} (y_i - \hat{y}_i)^2$$

Backpropagate (“backward pass”) and update

Calc $\nabla_\theta J, \quad \theta(k+1) = \theta(k) - \eta \nabla_\theta J$

Repeat until max epochs reached or desired convergence criterion

How do we train a neural network?

Computation graph: (series)

Neural network training is a series of large matrix multiplication and vector add operations → Perfect candidates for GPU parallelization

First, let's look at some common frameworks for implementing NN training, along with a C++ reference implementation.

Popular Deep Learning Frameworks

PyTorch

TensorFlow

Caffe2

Microsoft Cognitive Toolkit

Sonnet

ONNX

Keras

Each of these frameworks vary significantly:

- API styles
- Backends
- Extensibility
- Features
- Freedom to the user

While these libraries provide a great starting point and solution for most simple deep learning problems, building your own application gives you full control of the features

- Can lead to significant speedups in computation.

NN Training in PyTorch

In today's deep learning, PyTorch (created and maintained by Facebook/Meta) is the most popular

- Easy python API
- All of today's advanced neural network architectures
- Automatic backpropagation (autodiff engine)

For the rest of the talk, we will focus on the same data science problem: training a three-layer neural network classifier on the CiFAR-10 dataset.

```python
class SoftmaxClassfier(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()

        self.W1 = nn.Linear(input_dim, 400)
        self.W2 = nn.Linear(400, 300)
        self.W3 = nn.Linear(300, output_dim)

    def forward(self, x):

        z1 = self.W1(x)
        a1 = F.relu(z1)

        z2 = self.W2(a1)
        a2 = F.relu(z2)

        z3 = self.W3(a2)

        y_hat = F.softmax(z3, dim=1)

        return y_hat
```

Example pytorch model definition.

NN Training in PyTorch

Other modules used:
- time (for timing execution)
- Numpy (general computation)
- Pandas (loading data)

```
In [1]: import time

        import numpy as np

        import pandas as pd

        import torch
        import torch.nn as nn
        import torch.nn.functional as F
        import torch.optim as optim
```

A huge upside for python is its ease of loading data.

load data

```
In [2]: X_test = pd.read_csv('./data/cifar_x_test.csv', header=None)
        Y_test = pd.read_csv('./data/cifar_y_test.csv', header=None)
        X_train = pd.read_csv('./data/cifar_x_train.csv', header=None)
        Y_train = pd.read_csv('./data/cifar_y_train.csv', header=None)
```

NN Training in PyTorch

For demo, we will be using the CiFAR-10 dataset, which is a popular image classification dataset

- CiFAR-10 comes in as (32x32x3) RGB images, with each pixel int (0, 255)
- We reshape to (3072,) floats by subtracting by a mean of 127.5 and dividing by a stddev of 70

```
In [3]: X_train.head()
```

Out[3]:

	0	1	2	3	4	5	6	7	8	9	...
0	-0.978571	-1.207143	-1.107143	-0.850000	-0.421429	-0.121429	0.164286	0.250000	0.307143	0.307143	...
1	0.378571	-0.021429	-0.321429	-0.364286	-0.035714	0.392857	0.635714	0.750000	0.207143	-0.235714	...
2	1.821429	1.792857	1.792857	1.792857	1.792857	1.792857	1.792857	1.792857	1.792857	1.792857	...
3	-1.421429	-1.292857	-1.278571	-1.221429	-1.192857	-1.250000	-1.250000	-1.478571	-1.364286	-1.207143	...
4	0.607143	0.578571	0.707143	0.792857	0.764286	0.707143	0.764286	0.807143	0.878571	0.878571	...

```
In [4]: Y_train.head()
```

Out[4]:

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	0	0	1
3	0	0	0	0	1	0	0	0	0	0
4	0	1	0	0	0	0	0	0	0	0

The labels are one-hot encoded integers from (0, 9).

NN Training in PyTorch

We check if the GPU is available to torch (here, my local NVIDIA 2070 RTX Super)

```
In [8]: torch.cuda.is_available()
Out[8]: True
```

Define our hyperparameters and specify which device

```
In [9]: # params
        input_dim = 3072
        output_dim = 10

        batch_size = 32

        max_epochs = 100000

        device = "cuda:0"
```

Initialize our previously defined NN model onto the GPU

```
In [10]: NN = SoftmaxClassfier(input_dim, output_dim).to(device)
```

Specify cross-entropy loss (used for classification) and typical stochastic gradient descent (which is what our GPU implementation will use)

```
In [11]: ce_loss = nn.CrossEntropyLoss()
         optimizer = optim.SGD(NN.parameters(), lr=0.0001)
```

NN Training in PyTorch

This is the entire training loop, as described in pseudocode earlier (and on our computation graph)

Sample random batch

Zero out the previous gradients on the optimizer

Forward pass

Calculate loss and do backward pass

Do gradient descent to update the weights

Print statements for monitoring progress and timing

```
In [12]: ti = time.time()

# train loop
running_loss = 0.

for epoch in range(max_epochs):

    batch_idxs = np.random.choice(X_train.shape[0], size=(batch_size,), replace=False)

    x = torch.FloatTensor(X_train.loc[batch_idxs, :].to_numpy()).to(device)
    y = torch.FloatTensor(Y_train.loc[batch_idxs, :].to_numpy()).to(device)

    optimizer.zero_grad()

    y_hat = NN(x)

    loss = ce_loss(y_hat, y)

    loss.backward()

    optimizer.step()

    running_loss += loss.item()
    if epoch%1000 == 999:
        print(f'epoch: {epoch+1} \t loss: {running_loss/1000 :.3f}')
        running_loss = 0.

tf = time.time()
```

NN Training in C++

For an accurate, trusted parallel implementation, we first need a trusted reference implementation to easily port to the GPU.

While PyTorch is very effective, it hides a lot of the internal mechanics. Thus, we can use the PyTorch functions to cross-check our own C++ reference implementation.

Important C++ considerations and functions needed:

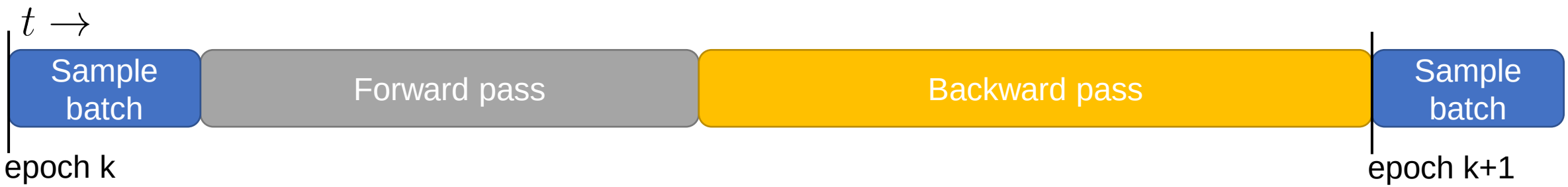
- Memory handling
- CSV reading
- Neural network class
  - Random weight initialization
  - ReLU, ReLU derivative
  - Softmax, cross entropy
  - Forward pass and backprop
- Batched gemv and gemm
- Elementwise multiply
- Mean reductions

NN Training in C++

C++ assumptions made for the reference implementation:

- All matrices and data stored in row-major order using the C++ std vector
- Try to use no external linear algebra routines (i.e. BLAS)
- Use the simple wine dataset for ease of calculation and testing (https://archive.ics.uci.edu/ml/datasets/wine)
- These functions built will be directly used in our parallel implementation, allowing us to just drop-in our parallelized kernels and function calls.

Recall:



NN Training in C++

Initialize our data dimensions and allocate memory in vector form

```cpp
int main(){

    // data dimensions: (wine dataset)
    // int train_N = 160;
    // int test_N = 18;
    // int output_dim = 3;
    // int input_dim = 13;

    // data dimensions: (cifar10 dataset)
    int train_N = 45000;
    int test_N = 5000;
    int output_dim = 10;
    int input_dim = 3072;

    // we will store the data in row-major order.
    // features
    vector<float> X_train (
        train_N*input_dim,
        0.0f
    );

    vector<float> X_test (
        test_N*input_dim,
        0.0f
    );

    // labels
    vector<float> Y_train (
        train_N*output_dim,
        0.0f
    );

    vector<float> Y_test (
        test_N*output_dim,
        0.0f
    );

```

Libraries only needed for vectors, random initialization of weights, CSV reading, and timing.

```cpp
#include <iostream>
#include <string>
#include <sstream>
#include <fstream>
#include <vector>
#include <random>
#include <chrono>
```

NN Training in C++

CSV reader into our vectors

```
read_csv("data/cifar_x_train.csv", X_train.data(), train_N, input_dim);
read_csv("data/cifar_x_test.csv", X_test.data(), test_N, input_dim);
read_csv("data/cifar_y_train.csv", Y_train.data(), train_N, output_dim);
read_csv("data/cifar_y_test.csv", Y_test.data(), test_N, output_dim);
```

CSV reader loops through each line and populates the strings to floats elementwise

```
int read_csv(string file_path, float* data, int N, int M){

    ifstream full_data;

    full_data.open(file_path);

    // be sure to handle error
    if (full_data.fail()){
        cerr << "Unable to open file "<< file_path << endl;
        return 1;
    }

    string line;
    string cell;
    unsigned int i = 0;
    while(getline(full_data, line)){   // get entire line (split by newline)
        stringstream ss(line);
        while(getline(ss, cell, ',')){  // split each line by commas
            if (i < (N*M)){
                data[i] = stof(cell);
                ++i;
            }
        }
    }

    full_data.close();

    return 0;
}
```

NN Training in C++

Initialize training hyperparameters, seed the random number generator for reproducibility

Timers for timing our wall clock computation time

Initialize the neural network class (to be shown)

Allocate for our minibatch X, Y, and loss

```
// layer dimensions:
int l1_dim = 400;
int l2_dim = 300;
float learning_rate = 0.0001;
int batch_size = 32;
int max_epochs = 100000;
mt19937 rng(5); // seed the rng with 5 here

// for timing
chrono::steady_clock::time_point ti;
chrono::steady_clock::time_point tf;

// build NN:
NeuralNetwork NN(input_dim, l1_dim, l2_dim, output_dim, learning_rate);

// get a test X and Y from the dataset:
vector<float> batch_X (
    batch_size*input_dim,
    0.0f
);

vector<float> batch_Y (
    batch_size*output_dim,
    0.0f
);

// put batch of loss here:
vector<float> batch_loss (
    batch_size*1,
    0.0f
);
```

NN Training in C++

Full training loop.

Build minibatch

Forward and backward pass (SGD is done in the backwards call)

Monitor print statements

Output simple run statistics

```
    float running_loss = 0.0f;

    ti = chrono::steady_clock::now();

    // train loop:
    for (unsigned int epoch = 0; epoch<max_epochs; ++epoch){

        // make the batch
        build_batch(rng, batch_X.data(), batch_Y.data(), X_train.data(), Y_train.data(), batch_size, input_dim, output_dim, train_N);

        NN.forward(batch_X.data(), batch_size);

        NN.backward(batch_loss.data(), batch_X.data(), batch_Y.data(), batch_size);

        float mean_loss = vector_mean_reduction(batch_loss.data(), batch_size);

        running_loss += mean_loss;
        if (epoch%1000 == 999){
            printf("--epoch: %i \t mean loss: %.4f-- \n", epoch+1, running_loss/1000);
            running_loss = 0.0f;
        }
    }

    tf = chrono::steady_clock::now();
    float time_elapsed_ms = chrono::duration_cast<chrono::milliseconds>(tf - ti).count();
    printf("elapsed time: %.4f s \n", time_elapsed_ms/1000);
    printf("avg throughput: %.4f updates/s \n", max_epochs/(time_elapsed_ms/1000));

    return 0;
}
```

NN Training in C++ - Preliminaries

In going through the forward and backward pass calculations, some important concepts to introduce:

"BLAS": **B**asic **L**inear **A**lgebra **S**ubprograms

Level I routines needed:

Single-precision ax + y, "saxpy"

$$\vec{y} \leftarrow a\vec{x} + \vec{y}$$

Level II routines needed:

Single-precision, batched general matrix-vector multiply
"batched gemv"

$$\vec{y} \leftarrow \alpha \mathtt{op}(A)\vec{x} + \beta\vec{y}$$

Level III routines needed:

Single-precision, batched general matrix-matrix multiply
"batched gemm"

$$C \leftarrow \alpha \mathtt{op}(A)\mathtt{op}(B) + \beta C$$

We will see these concepts throughout the reference and parallel implementations.

NN Training in C++

In the NN class, we maintain the interim calculations from the forward pass for efficient reuse in the backward pass.

Various parameters used throughout the network

Weights and biases of each layer

The interim calculations of the network and output

```
class NeuralNetwork {

    public:

        int input_dim;
        int l1_dim;
        int l2_dim;
        int output_dim;
        float learning_rate;

        // network params
        vector<float> W1;
        vector<float> b1;
        vector<float> W2;
        vector<float> b2;
        vector<float> W3;
        vector<float> b3;

        // calcs used for backprop
        vector<float> z1;
        vector<float> a1;
        vector<float> z2;
        vector<float> a2;
        vector<float> z3;
        vector<float> y_hat;
```

NN Training in C++

Inside the constructor, we randomly initialize the weights using a custom method.

```
_random_init(rng, W1.data(), l1_dim, input_dim);
_random_init(rng, b1.data(), l1_dim, 1);
_random_init(rng, W2.data(), l2_dim, l1_dim);
_random_init(rng, b2.data(), l2_dim, 1);
_random_init(rng, W3.data(), output_dim, l2_dim);
_random_init(rng, b3.data(), output_dim, 1);
```

This initializes like PyTorch, sampled from a uniform distribution (-σ, σ) where σ = 1/sqrt(N) and N is the leading dimension of the matrix.

```
void NeuralNetwork::_random_init(mt19937 &rng, float* W, int N, int M){
    // samples random values for weights and biases.
    // generalized for a matrix NxM.
    // init like pytorch does, uniform ()
    float stddev = 1/sqrt((float)N);
    uniform_real_distribution<float> uni(-stddev, stddev);

    for (unsigned int i = 0; i<N; ++i){
        for (unsigned int j = 0; j<M; ++j){
            auto rand_val = uni(rng);
            W[i*M + j] = rand_val;
        }
    }
}
```

NN Training in C++

Our forward pass. Note that this is simply running through the computations of each layer

I have batched the saxpy into the custom gemv function

Recall: ReLU is the activation function we use. Softmax is an output function that normalizes raw NN output into class probabilities.

Into these functions…

```
        // run the forward pass with our reference gemv: (note: X is already a pointer)
        //z1 = W1 x + b1
        _batched_gemv(z1.data(), W1.data(), X, b1.data(), l1_dim, input_dim, batch_size);
        // a1 = relu(z1)
        _relu(a1.data(), z1.data(), l1_dim, batch_size);
        // z2 = W2 a1 + b2
        _batched_gemv(z2.data(), W2.data(), a1.data(), b2.data(), l2_dim, l1_dim, batch_size);
        // a2 = relu(z2)
        _relu(a2.data(), z2.data(), l2_dim, batch_size);
        // z3 = W3 a2 + b3
        _batched_gemv(z3.data(), W3.data(), a2.data(), b3.data(), output_dim, l2_dim, batch_size);
        // y = softmax(z3)
        _softmax(y_hat.data(), z3.data(), output_dim, batch_size);

    }
```

NN Training in C++

Custom batched gemv with saxpy:
y = Ax + b

Row-major order

```cpp
void NeuralNetwork::_batched_gemv(float* y, float* A, float* x, float* b, int N, int M, int batch_size){
    // y = Ax + b, where A is NxM.
    // this is in batches, so the batch dimension is unchanged (first dim)

    // loop over batch
    for (unsigned int batch = 0; batch<batch_size; ++batch){
        for (unsigned int row = 0; row < N; ++row){
            float sum_ = 0.0f;
            for (unsigned int col = 0; col < M; ++col){
                sum_ += A[row*M + col]*x[batch*M + col];
            }
            y[batch*N + row] = sum_ + b[row];
        }
    }
}
```

Each of these functions was hand-checked against a python implementation for accuracy:

batched rowmajor gemv

```
In [129]: batch_size = 4
          rows = 3
          cols = 32

          A = np.random.uniform(size=(rows,cols))
          x = np.random.uniform(size=(batch_size, cols, 1))
          b = np.random.uniform(size=(rows,1))

In [130]: y_ref = A@x + b

In [131]: y = np.zeros((batch_size, rows, 1))

In [132]: A = A.reshape((cols*rows,))
          x = x.reshape((batch_size*cols,))
          y = y.reshape((batch_size*rows,))

          for batch in range(batch_size):
              for row in range(rows):
                  sum_ = 0
                  for col in range(cols):
                      sum_ += A[row*cols + col]*x[batch*cols + col]
                  y[batch*rows + row] = sum_ + b[row]
```

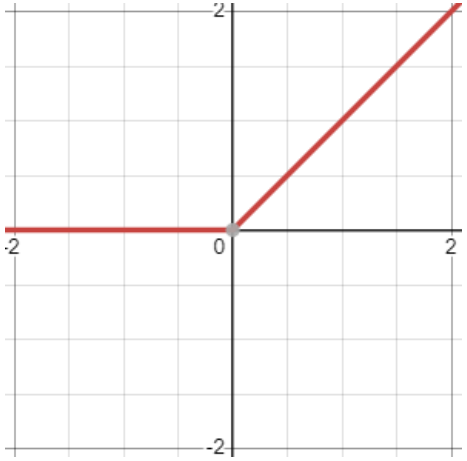
```
In [135]: y.reshape((batch_size, rows, 1)) - y_ref

Out[135]: array([[[ 8.88178420e-16],
                  [ 0.00000000e+00],
                  [-8.88178420e-16]],

                 [[ 0.00000000e+00],
                  [ 1.77635684e-15],
                  [ 0.00000000e+00]],

                 [[ 1.77635684e-15],
                  [ 8.88178420e-16],
                  [ 0.00000000e+00]],

                 [[ 1.77635684e-15],
                  [-3.55271368e-15],
                  [ 0.00000000e+00]]])
```

NN Training in C++

Simple ReLU function
(elementwise)

ReLU:

$$\phi(z) = \mathtt{max}(0, z)$$




```
711  void NeuralNetwork::_relu(float* y, float* x, int N, int batch_size){
712      // elementwise relu activation
713      // input x, output y, N is length of x.
714      for (unsigned int batch = 0; batch<batch_size; ++batch){
715          for (unsigned int i = 0; i<N; ++i){
716              float in_;
717              float out_;
718              in_ = x[batch*N + i];
719
720              if (in_ > 0.0f){
721                  out_ = in_;
722              } else {
723                  out_ = 0.0f;
724              }
725
726              y[batch*N + i] = out_;
727          }
728      }
729  }
```

NN Training in C++

Softmax output
(converts logits output from last NN layer to probabilities)

$$\hat{\vec{y}} \leftarrow \frac{1}{\sum_{i=0}^{n} e^{\hat{y}_i}} \begin{bmatrix} e^{\hat{y}_0} \\ \vdots \\ e^{\hat{y}_n} \end{bmatrix}$$

```cpp
void NeuralNetwork::_softmax(float* y, float* x, int N, int batch_size){
    // TODO: numerically stable softmax.
    // this will need parallelized? (the sum reduction)
    // softmax for output
    // input x, output y, N is length of x

    for (unsigned int batch = 0; batch<batch_size; ++batch){
        // first: convert y to exp(x)
        for (unsigned int i = 0; i<N; ++i){
            y[batch*N + i] = exp(x[batch*N + i]);
        }

        // sum each row of exp(x) (now y)
        float sum_exp = 0.0f;
        for (unsigned int i = 0; i<N; ++i){
            sum_exp += y[batch*N + i];
        }

        // now go back and divide each element of y with this value.
        for (unsigned int i = 0; i<N; ++i){
            y[batch*N + i] = y[batch*N + i]/sum_exp;
        }
    }
}
```

Note that softmax can be prone to overflow in the exponential, which can lead to NaNs via the division.

NN Training in C++

Into our backward pass. First, we calculate the current loss on the batch, just for training monitoring (for backprop, we don't need the actual loss value, just its derivative)

```cpp
void backward(float* loss, float* X, float* Y, int batch_size){
    // backward pass
    // loss is pointer to where to output the batch of loss
    // Y is the batch of the true labels
    // we have everything else we need saved in memory already.

    // edit the loss in-place for monitoring training
    _cross_entropy(loss, Y, y_hat.data(), output_dim, batch_size);

```

Cross entropy is an information-theory based loss function for calculating the difference between two probability distributions. It is a common loss function in classification, assuming each y label is a binomial distribution.

$$\texttt{ce\_loss}(y, \hat{y}) = -\sum_{i=0}^{n} y_i \log\left(\hat{y}_i\right)$$

```cpp
void NeuralNetwork::_cross_entropy(float* loss, float* y, float* y_hat, int N, int batch_size){
    // cross entropy loss for batch
    // put output in loss vector.

    for (unsigned int batch = 0; batch<batch_size; ++batch){
        float sum_of_logs = 0.0f;
        for (unsigned int i = 0; i<N; ++i){
            sum_of_logs -= y[batch*N + i]*log(y_hat[batch*N + i]);
        }
        loss[batch] = sum_of_logs;
    }

}
```

NN Training in C++

Back to our backward pass.

Derivatives of output layer

Derivatives of hidden layer

```
// do y_hat - y (cross entropy with softmax's derivative)
_batched_elementwise_subtract(dL_dz3.data(), y_hat.data(), Y, output_dim, batch_size);

// get first derivative for W3. (unreduced)
// NOTE: we will have to be very careful on our reduce eventually
// instead of transposing, in batched gemm, I just send it 1 as the internal dimension in the matmul,
// since the transpose of the vector is the same vector, just reordered.
_batched_gemm(dL_dW3.data(), dL_dz3.data(), a2.data(), output_dim, 1, l2_dim, batch_size);

// --- next layer ---

// we need W3_T batched to pass to batched gemm
_build_batch_of_transpose(batch_W3_T.data(), W3.data(), output_dim, l2_dim, batch_size);

// do W3^T dL_dz3 = dL_da2
_batched_gemm(dL_da2.data(), batch_W3_T.data(), dL_dz3.data(), l2_dim, output_dim, 1, batch_size);

// need relu_prime
_relu_prime(relu_prime_z2.data(), z2.data(), l2_dim, batch_size);

// do dL_a2 da2_dz2
// this is also dL_db2
_batched_elementwise_multiply(dL_dz2.data(), dL_da2.data(), relu_prime_z2.data(), l2_dim, batch_size);

//dL_dW2 = dL_dz2 * a1^T
_batched_gemm(dL_dW2.data(), dL_dz2.data(), a1.data(), l2_dim, 1, l1_dim, batch_size);
```

NN Training in C++

Backward pass continued

Derivatives of input layer

Reduce along each batch to get the average gradient of each parameter across the batch

Gradient descent to update the parameters

Looking at each function…

```
// --- next layer ---

// we need W2_T batched to pass to batched gemm
_build_batch_of_transpose(batch_W2_T.data(), W2.data(), l2_dim, l1_dim, batch_size);

// do W2^T dL_dz2 = dL_da1
_batched_gemm(dL_da1.data(), batch_W2_T.data(), dL_dz2.data(), l1_dim, l2_dim, 1, batch_size);

// need relu_prime(z1)
_relu_prime(relu_prime_z1.data(), z1.data(), l1_dim, batch_size);

// do dL_a1 da1_dz1 = dL_dz1
// this is also dL_db1
_batched_elementwise_multiply(dL_dz1.data(), dL_da1.data(), relu_prime_z1.data(), l1_dim, batch_size);

//dL_dW1 = dL_dz1 * x^T
_batched_gemm(dL_dW1.data(), dL_dz1.data(), X, l1_dim, 1, input_dim, batch_size);

// --- now, reduce by batch and update ---
// reductions (mean)
_batched_mean_reduction(dL_dW3_mean.data(), dL_dW3.data(), output_dim, l2_dim, batch_size);
_batched_mean_reduction(dL_dW2_mean.data(), dL_dW2.data(), l2_dim, l1_dim, batch_size);
_batched_mean_reduction(dL_dW1_mean.data(), dL_dW1.data(), l1_dim, input_dim, batch_size);

_batched_mean_reduction(dL_db3_mean.data(), dL_dz3.data(), output_dim, 1, batch_size);
_batched_mean_reduction(dL_db2_mean.data(), dL_dz2.data(), l2_dim, 1, batch_size);
_batched_mean_reduction(dL_db1_mean.data(), dL_dz1.data(), l1_dim, 1, batch_size);

//printf("made it to end of backprop reductions \n");

// do updates:
_inplace_gradient_descent(W3.data(), dL_dW3_mean.data(), output_dim, l2_dim, learning_rate);
_inplace_gradient_descent(W2.data(), dL_dW2_mean.data(), l2_dim, l1_dim, learning_rate);
_inplace_gradient_descent(W1.data(), dL_dW1_mean.data(), l1_dim, input_dim, learning_rate);

_inplace_gradient_descent(b3.data(), dL_db3_mean.data(), output_dim, 1, learning_rate);
_inplace_gradient_descent(b2.data(), dL_db2_mean.data(), l2_dim, 1, learning_rate);
_inplace_gradient_descent(b1.data(), dL_db1_mean.data(), l1_dim, 1, learning_rate);
```

NN Training in C++

Simple elementwise a - b

```cpp
void NeuralNetwork::_batched_elementwise_subtract(float* y, float* a, float* b, int N, int batch_size){
    // y = a - b, where a and b Nx1, matrices stored as (batch_size, N)
    // this is in batches, so the batch dimension is unchanged (first dim)

    // loop over batch
    for (unsigned int batch = 0; batch<batch_size; ++batch){
        for (unsigned int row = 0; row < N; ++row){
            y[batch*N + row] = a[batch*N + row] - b[batch*N + row];
        }
    }
}
```

Batched gemm: we have to multiply batches of matrices with vectors in backpropagation . Since I wrote this code with parallelization in mind, I wrote a generalized function for verification, and just pass a 1 as the dimension when I need to do gemv in the backprop.

```cpp
void NeuralNetwork::_batched_gemm(float* C, float* A, float* B, int N, int P, int M, int batch_size){
    // C = AB, where C is NxM, A NxP, B PxM.
    // this is in batches, so the batch dimension is unchanged (first dim)

    // loop over batch
    for (unsigned int batch = 0; batch<batch_size; ++batch){
        for (unsigned int i = 0; i < N; ++i){
            for (unsigned int j = 0; j < M; ++j){
                float sum_ = 0.0f;
                for (unsigned int k = 0; k < P; ++k){
                    sum_ += A[batch*N*P + i*P + k]*B[batch*P*M + k*M + j];
                }
                C[batch*N*M + i*M + j] = sum_;
            }
        }
    }
}
```

NN Training in C++

Bacthed gemm, like all C++ functions developed, was test against built-in python functions:

test with how we'd use in backprop:

```
In [74]: batch_size = 4

In [75]: output_dim = 3
         l2_dim = 32

In [76]: y_hat = np.random.uniform(size=(batch_size,output_dim,1))
         y = np.random.uniform(size=(batch_size,output_dim,1))
         a2 = np.random.uniform(size=(batch_size,l2_dim,1))

In [77]: dL_dW3_ref = (y_hat - y)@a2.transpose((0, 2, 1))

In [78]: y_hat = y_hat.reshape((-1,))
         y = y.reshape((-1,))
         a2 = a2.reshape((-1,))

In [ ]: # question: how do we transpose a batch of vectors? (batch_size, l2_dim, 1) --> (batch_size, 1, l2_dim)
        # these vectors are the same. we just reverse the way we pass the dims in the gemm fxn
        # just gotta be extra careful on the reduce.

In [79]: dL_dW3 = batched_gemm(y_hat - y, a2, batch_size, output_dim, 1, l2_dim)

In [81]: dL_dW3_ref.shape

Out[81]: (4, 3, 32)

In [83]: dL_dW3_ref - dL_dW3.reshape((batch_size, output_dim, l2_dim))

Out[83]: array([[[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                  0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                  0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                  0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]],
```

batched rowmajor gemm

```
In [69]: N = 30
         M = 20
         P = 25
         batch_size = 4

         A = np.random.uniform(size=(batch_size,N,P))
         B = np.random.uniform(size=(batch_size,P,M))
         C = np.zeros((batch_size,N,M))

         C_ref = A@B

         A = A.reshape((-1, 1))
         B = B.reshape((-1, 1))
         C = C.reshape((-1, 1))

         for batch in range(batch_size):
             for i in range(N):
                 for j in range(M):
                     sum_ = 0.
                     for k in range(P):
                         sum_ += A[batch*N*P + i*P + k]*B[batch*P*M + k*M + j]
                     C[batch*N*M + i*M + j] = sum_

In [70]: C = C.reshape((batch_size,N,M))

In [72]: C - C_ref

Out[72]: array([[[ 0.00000000e+00,  0.00000000e+00, -1.77635684e-15, ...,
                   0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
                 [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00, ...,
                   1.77635684e-15,  0.00000000e+00,  0.00000000e+00],
                 [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00, ...,
                   8.88178420e-16,  0.00000000e+00, -1.77635684e-15],
                 ...,
                 [ 8.88178420e-16,  8.88178420e-16,  0.00000000e+00, ...,
                   8.88178420e-16,  0.00000000e+00,  0.00000000e+00],
                 [ 0.00000000e+00,  0.00000000e+00,  8.88178420e-16, ...,
                   0.00000000e+00, -8.88178420e-16,  0.00000000e+00],
                 [ 0.00000000e+00, -8.88178420e-16,  0.00000000e+00, ...,
                  -8.88178420e-16,  0.00000000e+00,  0.00000000e+00]],
```

NN Training in C++

In our backward pass, the interim calculations and their derivatives are still batched, but our NN weights are not. Thus, with future parallelization in mind, to use batched gemm, we turn one matrix into a batch of its transpose

```cpp
void NeuralNetwork::_build_batch_of_transpose(float* A_T_batch, float* A, int N, int M, int batch_size){
    // need this fxn for backprop
    // takes a matrix A (NxM) and makes a batch of A_T (batch_size, M, N)
    for (unsigned int batch = 0; batch<batch_size; ++batch){
        for (unsigned int i = 0; i<N; ++i){
            for (unsigned int j = 0; j<M; ++j){
                A_T_batch[batch*M*N + j*N + i] = A[i*M + j];
            }
        }
    }
}
```

batched transpose

```
In [92]: batch_size = 4
         N = 30
         M = 20
         A = np.random.uniform(size=(batch_size, N, M))
         A_T = np.zeros((batch_size*M*N,))

In [93]: batch_size*M*N
Out[93]: 2400

In [94]: A_T.shape
Out[94]: (2400,)

In [95]: A_T_ref = A.transpose((0,2,1))
         A = A.reshape((batch_size*M*N))

In [96]: for batch in range(batch_size):
             for i in range(N):
                 for j in range(M):
                     A_T[batch*N*M + j*N + i] = A[batch*N*M + i*M + j]
```

```
In [98]: A_T.reshape((batch_size, M, N)) - A_T_ref
Out[98]: array([[[0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 ...,
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.]],

                [[0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 ...,
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.]],
```

NN Training in C++

Simple elementwise multiplication for our activation derivatives

```cpp
void NeuralNetwork::_batched_elementwise_multiply(float* y, float* a, float* b, int N, int batch_size){
    // y = a - b, where a and b Nx1, matrices stored as (batch_size, N)
    // this is in batches, so the batch dimension is unchanged (first dim)

    // loop over batch
    for (unsigned int batch = 0; batch<batch_size; ++batch){
        for (unsigned int row = 0; row < N; ++row){
            y[batch*N + row] = a[batch*N + row]*b[batch*N + row];
        }
    }
}
```

Partial derivative of ReLU

$$\phi'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

```cpp
void NeuralNetwork::_relu_prime(float* y, float* x, int N, int batch_size){
    // elementwise relu activation derivative
    // input x, output y, N is length of x.
    for (unsigned int batch = 0; batch<batch_size; ++batch){
        for (unsigned int i = 0; i<N; ++i){
            float in_;
            float out_;
            in_ = x[batch*N + i];

            if (in_ > 0.0f){
                out_ = 1.0f;
            } else {
                out_ = 0.0f;
            }

            y[batch*N + i] = out_;
        }
    }
}
```

NN Training in C++

This is an important function for our case:
Finding the mean of a matrix over a batch.

```cpp
void NeuralNetwork::_batched_mean_reduction(float* A_mean, float* A, int N, int M, int batch_size){
    // takes A (batch_size,N,M) and takes mean by batch s.t. A_mean (N, M)
    for (unsigned int i = 0; i<N; ++i){
        for (unsigned int j = 0; j<M; ++j){
            float sum_ = 0.0f;
            for (unsigned int batch = 0; batch<batch_size; ++batch){
                sum_ += A[batch*N*M + i*M + j];
            }
            A_mean[i*M + j] = sum_/batch_size;
        }
    }
}
```

Python verification:

batched reduction

```
In [111]: batch_size = 4
          N = 64
          M = 32
          A = np.random.uniform(size=(batch_size, N, M))
          A_reduced = np.zeros((N*M,))

In [112]: A_reduced_ref = np.mean(A, axis=0)

In [113]: A = A.reshape((batch_size*N*M,))

In [114]: A_reduced_ref.shape

Out[114]: (64, 32)

In [115]: for i in range(N):
              for j in range(M):
                  sum_ = 0.0
                  for batch in range(batch_size):
                      sum_ += A[batch*N*M + i*M + j]
                  A_reduced[i*M + j] = sum_/batch_size
```

```
In [118]: A_reduced_ref - A_reduced.reshape((N,M))

Out[118]: array([[0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 ...,
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.]])
```

NN Training in C++

Finally, a custom gradient descent function to update the parameters concludes our neural network trainer.

$$\theta(k+1) = \theta(k) - \eta \nabla_\theta J$$

```cpp
void NeuralNetwork::_inplace_gradient_descent(float* A, float* dA, int N, int M, float learning_rate){
    // A and dA are (N x M)
    // A = A - learning_rate*dA
    for (unsigned int i = 0; i<N; ++i){
        for (unsigned int j = 0; j<M; ++j){
            A[i*M + j] = A[i*M + j] - learning_rate*dA[i*M + j];
        }
    }
}
```

This concludes our code walkthrough of the reference implementation in C++! We will use these helper functions directly when applicable, and parallelize the math itself, using the reference to verify calculations.

NN Training in C++

A quick build with Microsoft VS and run on the wine dataset shows great results. Note the monotonically decreasing loss.

```
C:\Users\jakee\OneDrive - uah.edu\d_drive\CPE613\local_project_dev>cl /EHsc nn_dev.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.32.31332 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

nn_dev.cpp
Microsoft (R) Incremental Linker Version 14.32.31332.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:nn_dev.exe
nn_dev.obj
```

```
C:\Users\jakee\OneDrive - uah.edu\d_drive\CPE613\local_project_dev>nn_dev
mean loss: 1.0100
epoch: 0
mean loss: 0.8660
epoch: 1
mean loss: 0.7359
epoch: 2
mean loss: 0.6305
epoch: 3
mean loss: 0.5391
epoch: 4
mean loss: 0.4577
epoch: 5
mean loss: 0.3892
epoch: 6
mean loss: 0.3313
epoch: 7
mean loss: 0.2822
epoch: 8
mean loss: 0.2425
epoch: 9
mean loss: 0.2094
epoch: 10
mean loss: 0.1821
epoch: 11
mean loss: 0.1595
epoch: 12
mean loss: 0.1406
epoch: 13
mean loss: 0.1249
epoch: 14
mean loss: 0.1117
epoch: 15
mean loss: 0.1003
epoch: 16
mean loss: 0.0904
epoch: 17
mean loss: 0.0819
epoch: 18
mean loss: 0.0747
epoch: 19
mean loss: 0.0684
epoch: 20
mean loss: 0.0629
epoch: 21
mean loss: 0.0581
epoch: 22
mean loss: 0.0539
epoch: 23
mean loss: 0.0501
epoch: 24
mean loss: 0.0468
epoch: 25
mean loss: 0.0438
```

Parallelizing Our NN Trainer

Taking serial C++ code to the GPU

GPU Preliminaries

Vocabulary:
- ***Host*** = CPU
- ***Device*** = GPU
- ***Kernel*** = Function defined for execution on the GPU
- ***CUDA*** = NVIDIA GPU's C++ API
- ***Thread*** = a single unit of execution
- ***Block*** = collection of threads
- ***Grid*** = collection of blocks

Recall: we built many of our custom functions as if it was a BLAS operation. We will use the CUDA version of BLAS, called cuBLAS.

We must move data to GPU memory to perform computation on that data using GPU resources. Thus, the efficiency of the parallel implementation relies on efficiency of data transfers (or memcopies) with respect to the kernels themselves (and the algorithms/strategies used in the kernel).

GPU-Specific Considerations

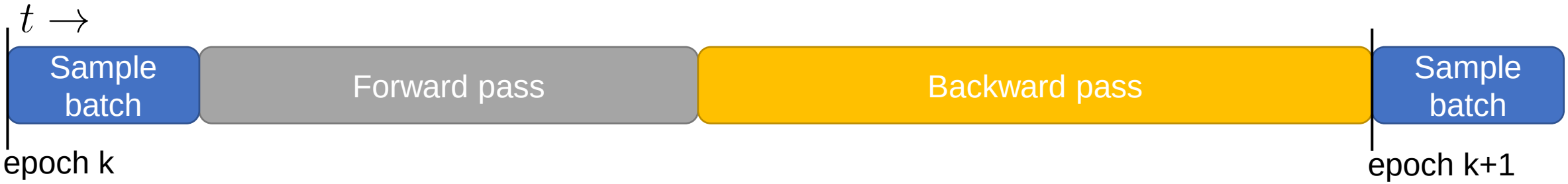
Parallelization strategy: **A**sses, **P**arallelize, **O**ptimize, **D**eploy

- Use the library cuBLAS and its kernels whenever possible.
  - cuBLAS calls highly optimized kernels to maximize occupancy (and is easily used on tensor cores with a math-mode change)
- Write custom kernels specific to my implementation when needed
  - For functions like softmax, batched reduce, etc.
  - Many of these routines are implemented in cuDNN, but cuDNN is highly specific tolarger architectures (like convolutional neural networks and transformers). Writing my own kernels allows for optimization in those portions of the code.

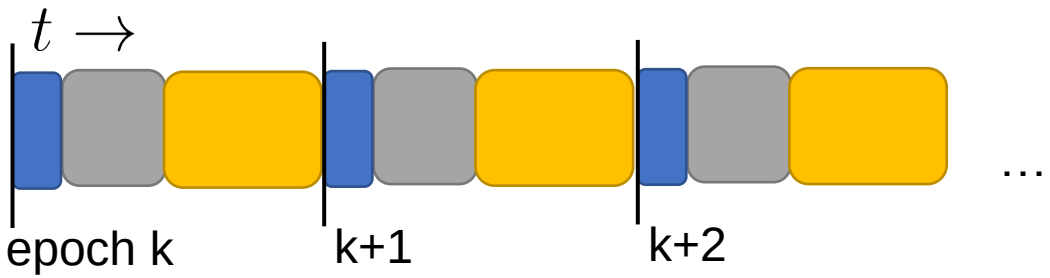
- Minimize memory transfers from host to device
  - Leave all NN parameters and interim backpropagation calculations in GPU memory
  - Only have to move around each minibatch of data from host to device and loss from device to host for monitoring

Parallelizing the NN Trainer

Recall: on the host, we had



For basic parallelization, we want

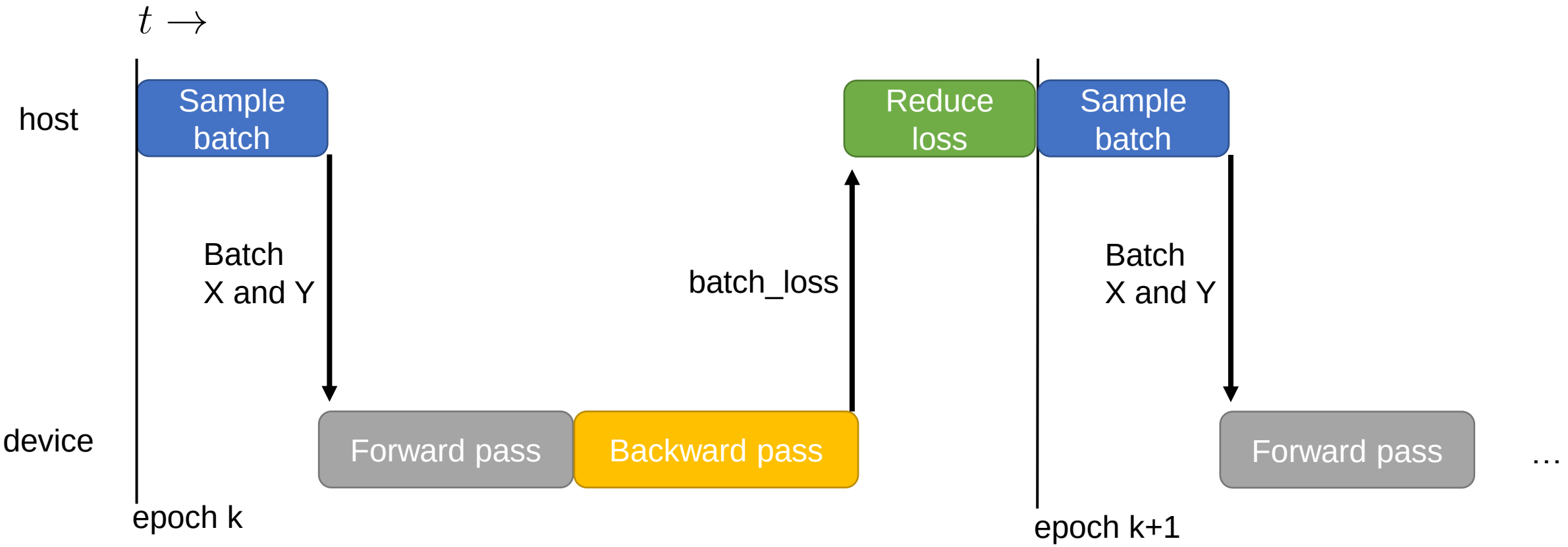


(note that these rectangles are not to scale of actual computation time)

Initial parallelization efforts will just focus on speeding up the computations as fast as possible.
Future work would be to conduct the backward pass from the previous forward pass results concurrently.

Parallelizing the NN Trainer

Our host-device trainer will look like



Parallelizing the NN Trainer

Reference Implementation	Parallel Implementation
_batched_gemv() →	cuBLAS `cublasSgemvStridedBatched` `cublasSaxpy`
_relu() →	Custom ReLU kernel
_softmax() →	Custom elementwise exponential kernel cuBLAS `cublasSgemv` Custom batched elementwise divide kernel
_cross_entropy() →	Custom elementwise log kernel Custom elementwise multiplication kernel cuBLAS `cublasSgemv`

Parallelizing the NN Trainer

Reference Implementation		Parallel Implementation
_elementwise_subtraction()	→	Custom elementwise subtraction kernel
_batched_gemm()	→	cuBLAS `cublasSgemmStridedBatched`
_relu_prime()	→	Custom ReLU prime kernel
_batch_mean_reduce()	→	Custom batch mean reduce kernel

Note that, for some functions (softmax, cross entropy), I use a vector of 1's in gemv to take advantage of cuBLAs's speed in general sum reductions.

Each of these kernels were tested against reference in separate test programs, shown in the appendix slides.

Parallelizing the NN Trainer

In our parallel implementation, we include our custom kernels through the headerfile jake_kernels.h, referencing the .cu file where our kernels are defined.

We include the various cuda routines libraries needed for math and debug, include our custom Timer class from the course, and our file readers for reading the data from a CSV.

```
#include <jake_kernels.h>
#include <Timer.hpp>

#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <helper_cuda.h>

#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <vector>
#include <random>
#include <chrono>

// file readers
#include <iostream>
#include <string>
#include <sstream>
#include <fstream>
```

Parallelizing the NN Trainer

The training loop for our parallel implementation.

Notice the only change here: copying down the batch_loss from device memory for monitoring.

```
float running_loss = 0.0f;

ti = chrono::steady_clock::now();

//train loop test:
for (unsigned int epoch = 0; epoch<max_epochs; ++epoch){
    //tb_i = chrono::steady_clock::now();

    // make the batch
    build_batch(rng, batch_X.data(), batch_Y.data(), X_train.data(), Y_train.data(), batch_size, input_dim, output_dim, train_N);

    NN.forward(batch_X.data());

    NN.backward(batch_loss.data(), batch_X.data(), batch_Y.data());

    // get result (dev)
    checkCudaErrors(cudaMemcpy(y_hat_host.data(), NN.y_hat, NN.byteSize_y_hat, cudaMemcpyDeviceToHost));

    float mean_loss = vector_mean_reduction(batch_loss.data(), batch_size);

    running_loss += mean_loss;
    if (epoch%1000 == 999){
        printf("--epoch: %i \t mean loss: %.4f-- \n", epoch+1, running_loss/1000);
        running_loss = 0.0f;
    }

}
```

Parallelizing the NN Trainer

Into the neural network class, the initializations are the same, so I have skipped them for brevity.

The big change is allocating all the memory on the device for the required values we need in our forward and backward passes.

I will skip the rest of the intializations to the forward pass:

```
// ----allocate device memory for all the parameters, batch input----
// input
X_device = nullptr;

byteSize_X = input_dim*batch_size*sizeof(float);

checkCudaErrors(cudaMalloc(&X_device, byteSize_X));

// loss calcs
loss_calcs = nullptr;
loss_device = nullptr;

byteSize_loss_device = batch_size*sizeof(float);

checkCudaErrors(cudaMalloc(&loss_calcs, batch_size*output_dim*sizeof(float)));
checkCudaErrors(cudaMalloc(&loss_device, byteSize_loss_device));

// --allocate and copy for softmax--
ones_for_softmax = nullptr;
sum_exp_y = nullptr;

checkCudaErrors(cudaMalloc(&ones_for_softmax, output_dim*sizeof(float)));
checkCudaErrors(cudaMalloc(&sum_exp_y, batch_size*sizeof(float)));

checkCudaErrors(cudaMemcpy(ones_for_softmax, ones_for_softmax_host.data(), output_dim*sizeof(float), cudaMemcpyHostToDevice));

// -- interim calcs --
z1 = nullptr;
a1 = nullptr;
z2 = nullptr;
a2 = nullptr;
z3 = nullptr;
y_hat = nullptr;
y_device = nullptr;

byteSize_z1 = batch_size*l1_dim*sizeof(float);
byteSize_a1 = batch_size*l1_dim*sizeof(float);
byteSize_z2 = batch_size*l2_dim*sizeof(float);
byteSize_a2 = batch_size*l2_dim*sizeof(float);
byteSize_z3 = batch_size*output_dim*sizeof(float);
byteSize_y_hat = batch_size*output_dim*sizeof(float);
byteSize_y_device = batch_size*output_dim*sizeof(float);
```

Parallelizing the NN Trainer

First, we copy the current batch to the device for processing

Next, we simply run through the forward pass like we did in our reference implementation, matching our custom kernels and cuBLAS calls.

A note on cuBLAS: cuBLAS expects column-major order, for Fortran compatibility. Thus, we can reorder the dimensions and pass the transpose operation to do row-major multiplications.

```
void forward(float* X){
    // forward pass.
    // X is the batch input from host.

    // copy over X to X_device (note: X is already a pointer)
    checkCudaErrors(cudaMemcpy(X_device, X, byteSize_X, cudaMemcpyHostToDevice));

    // run the forward pass
    //z1 = W1 x + b1
    stat = cublasSgemvStridedBatched(handle,
                                     CUBLAS_OP_T,
                                     input_dim, l1_dim,
                                     &alpha_default,
                                     W1_device, input_dim, l1_dim*input_dim,
                                     X_device, 1, input_dim,
                                     &beta_default,
                                     z1, 1, l1_dim,
                                     batch_size);

    stat = cublasSaxpy(handle,
                       batch_size*l1_dim,
                       &alpha_default,
                       b1_device, 1,
                       z1, 1);

    // a1 = relu(z1)
    // custom kernel
    relu(a1, z1, batch_size*l1_dim);
```

Parallelizing the NN Trainer

Continuing through the forward pass

```
// z2 = W2 a1 + b2
stat = cublasSgemvStridedBatched(handle,
                                 CUBLAS_OP_T,
                                 l1_dim, l2_dim,
                                 &alpha_default,
                                 W2_device, l1_dim, l2_dim*l1_dim,
                                 a1, 1, l1_dim,
                                 &beta_default,
                                 z2, 1, l2_dim,
                                 batch_size);

stat = cublasSaxpy(handle,
                   batch_size*l2_dim,
                   &alpha_default,
                   b2_device, 1,
                   z2, 1);

// a2 = relu(z2)
// custom kernel
relu(a2, z2, batch_size*l2_dim);

// z3 = W3 a2 + b3
stat = cublasSgemvStridedBatched(handle,
                                 CUBLAS_OP_T,
                                 l2_dim, output_dim,
                                 &alpha_default,
                                 W3_device, l2_dim, output_dim*l2_dim,
                                 a2, 1, l2_dim,
                                 &beta_default,
                                 z3, 1, output_dim,
                                 batch_size);

stat = cublasSaxpy(handle,
                   batch_size*output_dim,
                   &alpha_default,
                   b3_device, 1,
                   z3, 1);
```

Notice that we are calling the same cuBLAS operations multiple times.

A trick I should've used during dev: define my own function interfaces to these functions that better match my reference implementation. That way, it is much easier to get the dimensions in all the arguments correct.

Parallelizing the NN Trainer

Final step in forward pass: softmax

I use the custom exponential kernel to raise y to an exponent. Then, I use a predefined vector of ones in gemv to calculate the sum for each batch. Finally, an elementwise divide kernel using those sums of exponents.

```
        // y = softmax(z3)
        //_softmax(y_hat.data(), z3.data(), output_dim, batch_size);
        // first, use custom kernel to raise to exp:
        custom_exp(y_hat, z3, output_dim*batch_size);
        // then use simple gemv with ones vector to get sum_exp_y
        // y_hat is batch of vectors raised to exp now.
        stat = cublasSgemv(handle,
                           CUBLAS_OP_T,
                           output_dim, batch_size,
                           &alpha_default,
                           y_hat, output_dim,
                           ones_for_softmax, 1,
                           &beta_default,
                           sum_exp_y, 1);

        // now one more custom kernel to divide by this new sum_exp_y and we are done
        // this is done inplace on y_hat.
        batched_elementwise_divide(y_hat, sum_exp_y, output_dim, batch_size);

    }
```

Parallelizing the NN Trainer

Custom ReLU kernel:
The grid-stride loop is used for when we have more data to process than threads.

```
// relu
__global__ void relu_kernel (float* A_out, float* A_in, unsigned int N) {
  // A is vector of dim N
  for (unsigned int i = blockIdx.x*blockDim.x + threadIdx.x; i < N; i += blockDim.x*gridDim.x){
    float in_ = A_in[i];
    if (in_ > 0.0f){
        A_out[i] = in_;
    } else {
        A_out[i] = 0.0f;
    }
  }
}
```

Custom elementwise log, exp kernels (for softmax and cross entropy)

```
__global__ void log_kernel (float* A_out, float* A_in, unsigned int N) {
  // A is vector of dim N
  // used in cross entropy
  for (unsigned int i = blockIdx.x*blockDim.x + threadIdx.x; i < N; i += blockDim.x*gridDim.x){
    A_out[i] = log(A_in[i]);
  }
}

__global__ void exp_kernel (float* A_out, float* A_in, unsigned int N) {
  // A is vector of dim N
  // used in our softmax
  for (unsigned int i = blockIdx.x*blockDim.x + threadIdx.x; i < N; i += blockDim.x*gridDim.x){
    A_out[i] = exp(A_in[i]);
  }
}
```

Parallelizing the NN Trainer

Next, the backward pass:

Copy the given batch of labels Y to device memory

Using our custom kernels and vector of ones previously described to quickly calculate cross entropy loss for monitoring

```
void backward(float* loss, float* X, float* Y){
    // backward pass
    // loss is pointer to where to output the batch of loss
    // Y is the batch of the true labels
    // we have everything else we need saved in memory already.

    // copy Y over:
    checkCudaErrors(cudaMemcpy(y_device, Y, byteSize_y_device, cudaMemcpyHostToDevice));

    // edit the loss in-place for monitoring training
    // custom kernels here for cross entropy
    custom_log(loss_calcs, y_hat, batch_size*output_dim);
    // loss_calcs is now log(y_hat)
    // y*log(y_hat):
    inplace_elementwise_multiplication (loss_calcs, y_device, batch_size, output_dim);
    // loss = -sum(y*log(y_hat))
    stat = cublasSgemv(handle,
                       CUBLAS_OP_T,
                       output_dim, batch_size,
                       &alpha_neg_one,
                       loss_calcs, output_dim,
                       ones_for_softmax, 1,
                       &beta_default,
                       loss_device, 1);

    // copy loss down from device to host loss pointer
    checkCudaErrors(cudaMemcpy(loss, loss_device, byteSize_loss_device, cudaMemcpyDeviceToHost));
```

Parallelizing the NN Trainer

We continue through the backpropagation as described in the reference implementation, just dropping in our cuBLAS calls and custom kernels.

```
447        // ------ backprop ------
448
449        // do y_hat - y (cross entropy with softmax's derivative)
450        // custom kernel
451        elementwise_subtraction(dL_dz3, y_hat, y_device, batch_size*output_dim);
452
453        // get first derivative for W3. (unreduced)
454        // dL_dW3 = dL_dz3 a2^T
455        stat = cublasSgemmStridedBatched(handle,
456                                         CUBLAS_OP_N, CUBLAS_OP_N,
457                                         l2_dim, output_dim, 1,
458                                         &alpha_default,
459                                         a2, l2_dim, l2_dim,
460                                         dL_dz3, 1, output_dim,
461                                         &beta_default,
462                                         dL_dW3, l2_dim, output_dim*l2_dim,
463                                         batch_size);
464
465        // --- next layer ---
466        // do W3^T dL_dz3 = dL_da2
467        stat = cublasSgemmStridedBatched(handle,
468                                         CUBLAS_OP_N, CUBLAS_OP_T,
469                                         1, l2_dim, output_dim,
470                                         &alpha_default,
471                                         dL_dz3, 1, output_dim,
472                                         W3_device, l2_dim, output_dim*l2_dim,
473                                         &beta_default,
474                                         dL_da2, 1, l2_dim,
475                                         batch_size);
476
477        // need relu_prime
478        // custom relu_prime kernel
479        relu_prime(relu_prime_z2, z2, l2_dim*batch_size);
480
```

Parallelizing the NN Trainer

Derivatives of the hidden layer

```
// do dL_a2 da2_dz2
// this is also dL_db2
// custom elementwise multiply kernel
elementwise_multiplication(dL_dz2, dL_da2, relu_prime_z2, l2_dim*batch_size);

//dL_dW2 = dL_dz2 * a1^T
stat = cublasSgemmStridedBatched(handle,
                                 CUBLAS_OP_N, CUBLAS_OP_N,
                                 l1_dim, l2_dim, 1,
                                 &alpha_default,
                                 a1, l1_dim, l1_dim,
                                 dL_dz2, 1, l2_dim,
                                 &beta_default,
                                 dL_dW2, l1_dim, l2_dim*l1_dim,
                                 batch_size);
// --- next layer ---

// do W2^T dL_dz2 = dL_da1
stat = cublasSgemmStridedBatched(handle,
                                 CUBLAS_OP_N, CUBLAS_OP_T,
                                 1, l1_dim, l2_dim,
                                 &alpha_default,
                                 dL_dz2, 1, l2_dim,
                                 W2_device, l1_dim, l2_dim*l1_dim,
                                 &beta_default,
                                 dL_da1, 1, l1_dim,
                                 batch_size);

// need relu_prime(z1)
// custom relu prime
relu_prime(relu_prime_z1, z1, l1_dim*batch_size);
```

Parallelizing the NN Trainer

Derivatives of the input layer

Calling our custom batch mean reduce kernels to get average gradient of each parameter in the network

```
// do dL_a1 da1_dz1 = dL_dz1
// this is also dL_db1
// custom elementwise multiply kernel
elementwise_multiplication(dL_dz1, dL_da1, relu_prime_z1, l1_dim*batch_size);

//dL_dW1 = dL_dz1 * x^T
stat = cublasSgemmStridedBatched(handle,
                                 CUBLAS_OP_N, CUBLAS_OP_N,
                                 input_dim, l1_dim, 1,
                                 &alpha_default,
                                 X_device, input_dim, input_dim,
                                 dL_dz1, 1, l1_dim,
                                 &beta_default,
                                 dL_dW1, input_dim, l1_dim*input_dim,
                                 batch_size);

// --- now, reduce by batch and update ---
// reductions (mean)
custom_batch_mean_reduce(dL_dW3_mean, dL_dW3, output_dim, l2_dim, batch_size);
custom_batch_mean_reduce(dL_dW2_mean, dL_dW2, l2_dim, l1_dim, batch_size);
custom_batch_mean_reduce(dL_dW1_mean, dL_dW1, l1_dim, input_dim, batch_size);

custom_batch_mean_reduce(dL_db3_mean, dL_dz3, output_dim, 1, batch_size);
custom_batch_mean_reduce(dL_db2_mean, dL_dz2, l2_dim, 1, batch_size);
custom_batch_mean_reduce(dL_db1_mean, dL_dz1, l1_dim, 1, batch_size);

//printf("made it to end of backprop reductions \n");
```

Parallelizing the NN Trainer

Finally, we use gradient descent to update the weights with cuBLAS saxpy

Recall saxpy:

$$\vec{y} \leftarrow a\vec{x} + \vec{y}$$

Gradient descent:

$$\theta(k+1) = \theta(k) - \eta \nabla_\theta J$$

Thus, we can use the saxpy API to quickly update our parameters inplace.

```
// do updates:
// use saxpy with -LR as alpha to get inplace SGD.
stat = cublasSaxpy(handle,
                   batch_size*output_dim*l2_dim,
                   &negative_lr,
                   dL_dW3_mean, 1,
                   W3_device, 1);

stat = cublasSaxpy(handle,
                   batch_size*l2_dim*l1_dim,
                   &negative_lr,
                   dL_dW2_mean, 1,
                   W2_device, 1);

stat = cublasSaxpy(handle,
                   batch_size*l1_dim*input_dim,
                   &negative_lr,
                   dL_dW1_mean, 1,
                   W1_device, 1);

stat = cublasSaxpy(handle,
                   batch_size*output_dim,
                   &negative_lr,
                   dL_db3_mean, 1,
                   b3_device, 1);

stat = cublasSaxpy(handle,
                   batch_size*l2_dim,
                   &negative_lr,
                   dL_dW2_mean, 1,
                   b2_device, 1);

stat = cublasSaxpy(handle,
                   batch_size*l1_dim,
                   &negative_lr,
                   dL_db1_mean, 1,
                   b1_device, 1);
```

Parallelizing the NN Trainer

Additional custom kernels used in backprop. Notice that all make use of the grid-stride loop, for versatility scalability.

Operations are done inplace when possible to minimize memory usage.

```
__global__ void inplace_elementwise_multiplication_kernel (float* A, float* B, unsigned int N, unsigned int M) {
  // A = A .* B, A and B NxM
  // used in our cross entropy
  for (unsigned int i = blockIdx.x*blockDim.x + threadIdx.x; i < N*M; i += blockDim.x*gridDim.x){
    A[i] = A[i]*B[i];
  }
}

__global__ void elementwise_multiplication_kernel (float* A, float* B, float* C, unsigned int N) {
  // A = B .* C, all vectors of length N
  // used in backprop
  for (unsigned int i = blockIdx.x*blockDim.x + threadIdx.x; i < N; i += blockDim.x*gridDim.x){
    A[i] = B[i]*C[i];
  }
}

__global__ void elementwise_subtraction_kernel (float* x, float* y, float* z, unsigned int N) {
  // x = y - z, all dim N
  // used in our cross entropy derivative
  for (unsigned int i = blockIdx.x*blockDim.x + threadIdx.x; i < N; i += blockDim.x*gridDim.x){
    x[i] = y[i] - z[i];
  }
}
```

```
__global__ void relu_prime_kernel (float* A_out, float* A_in, unsigned int N) {
  // A is vector of dim N
  for (unsigned int i = blockIdx.x*blockDim.x + threadIdx.x; i < N; i += blockDim.x*gridDim.x){
    float in_ = A_in[i];
    if (in_ > 0.0f){
        A_out[i] = 1.0f;
    } else {
        A_out[i] = 0.0f;
    }
  }
}
```

Parallelizing the NN Trainer

This was by far the trickiest kernel I wrote by myself: reducing by the mean of the batch dimension in a threadsafe way.

I assume in the kernel launch configuration that N*M threads are launched, where A_batched is of dimension (batch_size, N, M).

To simply explain this kernel, each thread calculates its own mean and puts it in the first batch position of A_mean

Then each thread copies out its own mean into the rest of the batches (we maintain each network parameter in batch to avoid the expensive copy operation like on the reference implementation)

```
__global__ void custom_batch_mean_reduce_kernel(float* A_mean, float* A_batched, unsigned int N, unsigned int M, unsigned int batch_size) {
  // used for finding mean of gradients by batch
  // reduces along batch dimension
  // A is NxM in batches.
  // assumes num threads = N*M
  for (unsigned int i = blockIdx.x*blockDim.x + threadIdx.x; i < N*M; i += blockDim.x*gridDim.x){

    // first: put sum into first batch of A_mean.
    // should be threadsafe the way I have it
    if (i < N*M){
      for (unsigned int batch = 0; batch<batch_size; ++batch){
        A_mean[i] += A_batched[batch*N*M + i];
      }
    }
    __syncthreads();

    // now divide first batch of A_mean to get actual mean
    if (i < N*M){
      A_mean[i] = A_mean[i]/batch_size;
    }

    __syncthreads();

    // copy out first batch into rest of the batches
    if (i < N*M){
      for (unsigned int batch = 1; batch<batch_size; ++batch){
        A_mean[batch*N*M + i] = A_mean[i];
      }
    }
  }
}
```

Parallelizing the NN Trainer

We can build the program by dynamic linking our custom kernels file jake_kernels.cu to our main program, test.cpp, in a Makefile:

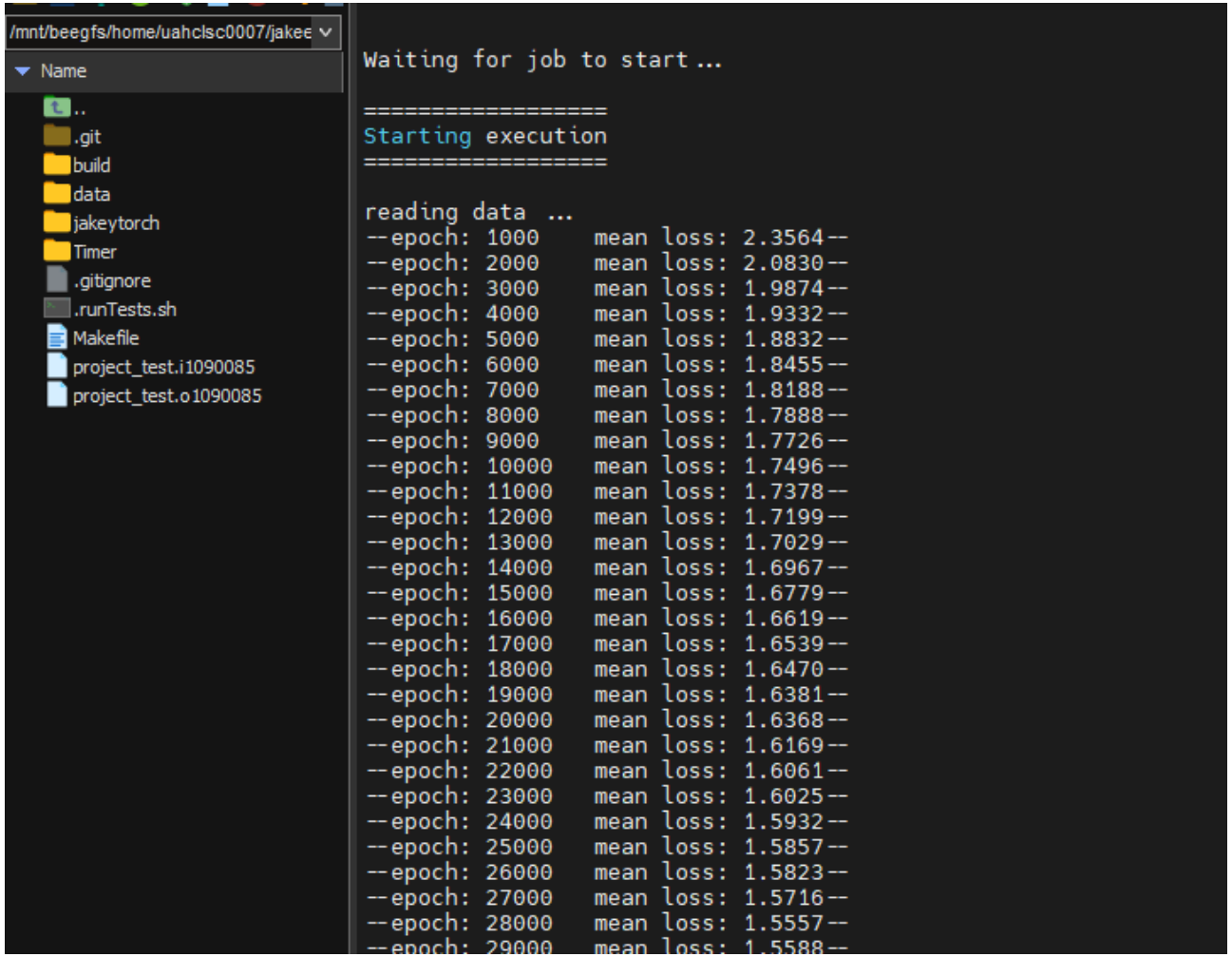
```
build/lib/libjake_kernels.so: jakeytorch/src/jake_kernels.cu
    @mkdir -p build/.objects/jake_kernels
 # dr. wise's step 1: compile with -dc
    $(NVCC) -pg $(NVCCFLAGS) $(NVCCARCHS) -Xcompiler -fPIC \
        -Ijakeytorch/include -I$(CUDAPATH)/samples/common/inc \
        -dc -o build/.objects/jake_kernels/jake_kernels.o \
        jakeytorch/src/jake_kernels.cu
  # step 2: link object files with relocatable device code with -dlink.
    $(NVCC) -pg $(NVCCFLAGS) $(NVCCARCHS) -Xcompiler -fPIC \
        -dlink -o build/.objects/jake_kernels/jake_kernels-dlink.o build/.objects/jake_kernels/jake_kernels.o
    mkdir -p build/lib
 # step 3: use gcc to make final shared library (see cuda docs: separate compilation and linking of cuda c++ device code.
    $(CC) -shared -o build/lib/libjake_kernels.so build/.objects/jake_kernels/* \
        -Wl,-rpath=$(CUDAPATH)/lib64 -L$(CUDAPATH)/lib64 -lcudart -lcublas
    @mkdir -p build/include
    @ln -sf ../../jakeytorch/include/jake_kernels.h build/include/jake_kernels.h

build/bin/project_test: build/lib/libTimer.so build/lib/libjake_kernels.so \
    jakeytorch/test/src/test.cpp
    @mkdir -p build/bin
 # now, when we build, we can link our own libraries we built.
    $(CXX) -Ibuild/include -I$(CUDAPATH)/samples/common/inc \
        -o build/bin/project_test jakeytorch/test/src/test.cpp \
        -Wl,-rpath=$(PWD)/build/lib \
        -Lbuild/lib -L$(CUDAPATH)/lib64 \
        -lTimer -ljake_kernels -lcudart -lblas -lgfortran -lcublas

run: build/bin/project_test
    @rm -f *.nsys-rep project_test.i* project_test.o* core.*
    @echo -ne "class\n1\n\n10gb\n1\nampere\nproject_test\n" | \
        run_gpu .runTests.sh > /dev/null
    @sleep 5
    @tail -f project_test.o*

clean:
    rm -rf build
    rm -f *nsys-rep
    rm -f project_test.*
```

Parallelizing the NN Trainer

We run with a simple make run command, defined from our Makefile.

I have remoted into the Alabama Supercomputer Authority DMC using MobaXterm for Windows.

As shown in our sample output, the NN trainer is working great! This run was on the CiFAR-10 dataset. I have SCP'd both the wine and CiFAR-10 dataset to the DMC for testing.



Performance Comparison

How much better is this GPU implementation?

Performance Comparison

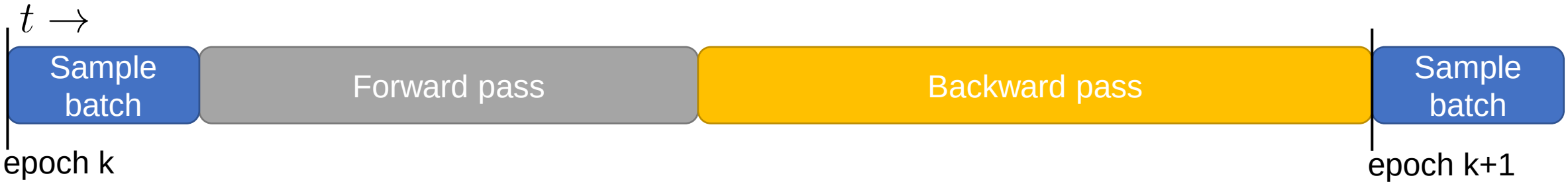
Test case: 100,00 training epochs on the CiFAR-10 dataset, learning rate 1e-4, batch size 32, neural network architecture: 2 hidden layers, 400x300 neurons, respectively.

	PyTorch, local GPU (RTX 2070 Super)	Parallelized on DMC (Ampere A100)
Execution time (s)	794.92	148.86
Average throughput (updates/s)	125.80	671.79
Final cross-entropy loss (nats)	2.23	1.31

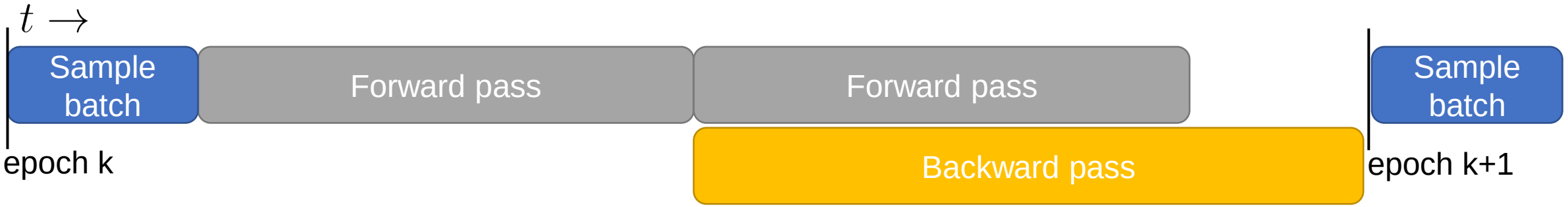
Average throughput is the important metric: we achieve over 5x more updates per unit time than the already-parallelized PyTorch implementation. Note that the A100 cards are significantly more capable than my local GPU, making these results likely a little skewed.
Accuracy is also considerably lower, which is really interesting. This is likely due to how I implemented the gradient reductions: keeping all the gradients until reducing at update time increases fidelity, rather than calculating gradients numerically with respect to the mean of the loss.

Future Steps and Further Optimizations

Recall our basic computation graph:



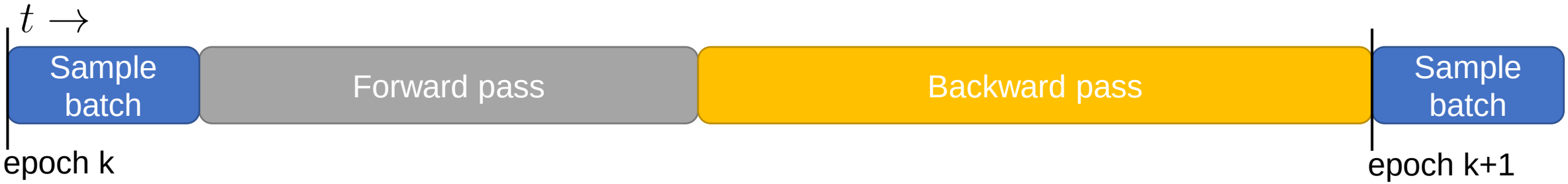
The immediate next step: run forward and backward pass asynchronously with CUDA streams and graphs



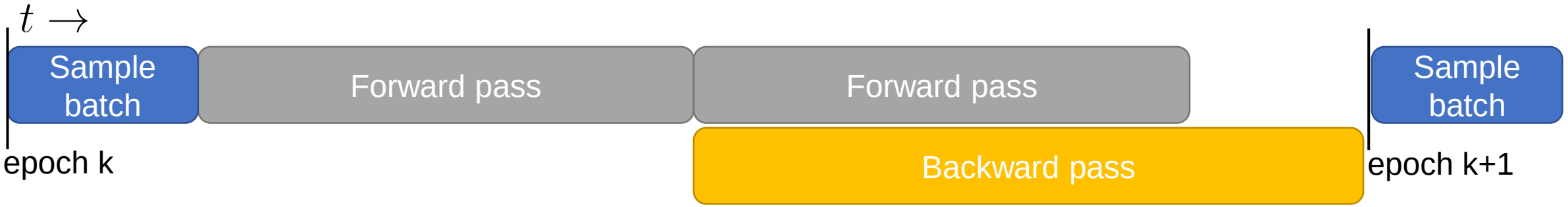
This would drastically decrease computation time and increase occupancy.

Future Steps and Further Optimizations

Recall our basic computation graph:



The immediate next step: run forward and backward pass asynchronously with CUDA streams and graphs



This would drastically decrease computation time and increase occupancy. Next, work on how to abstract this framework for varying architectures.

Conclusion

In this presentation, we walked through three neural network implementations: python using PyTorch, a C++ reference trainer, and a parallelized trainer using CUDA/C++.

The CUDA/C++ trainer was shown to drastically outperform the PyTorch implementation in both speed and accuracy. However, considerable development time was spent on parallelization. In most cases, it likely makes sense to utilize PyTorch's development efficiencies in research and development, but building you own inference and training pipelines when speed becomes a major consideration (like in production).

Throughout the development, the primary takeaway was how to make porting a C++ reference implementation over as easily as possible. Defining your own simple interfaces to the backends you use is important. Knowing which optimized tools are out there (like cuBLAS), what they do, and how they can be best used is huge. Stand on the shoulders of giants!

Epilogue

Dr. Wise – I had never written a line of C++ before this course. Going back through this presentation, I can’t tell you how proud of myself I am. This course was extremely well designed, and I enjoyed it thoroughly. It was a ton of work, but I can’t think of a course I gained more from. Thank you again, and I hope your work and teaching goes well going forward!

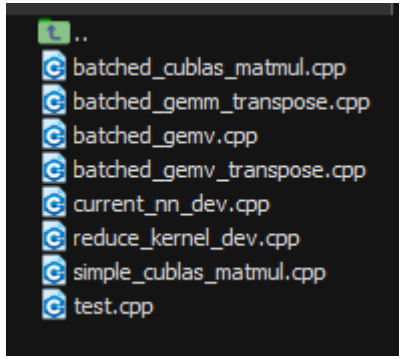
-Jake

Appendix

Additional reference implementation testing

Reference Implementation Test Code

Test cases to each cuBLAS call and kernel developed are available under jakeytorch/test/src/.




```
    // copy A,B to device as input
    checkCudaErrors(cudaMemcpy(A_device, A.data(), byteSize_A, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(B_device, B.data(), byteSize_B, cudaMemcpyHostToDevice));

    // do C = AB
    //stat = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, M, N, P, &alpha_default, B_device, M, A_device, P,
    stat = cublasSgemmStridedBatched(handle,
                                     CUBLAS_OP_N, CUBLAS_OP_N,
                                     M, N, P,
                                     &alpha_default,
                                     B_device, M, strideB,
                                     A_device, P, strideA,
                                     &beta_default,
                                     C_device, M, strideC,
                                     batch_size);

    if (stat != CUBLAS_STATUS_SUCCESS){
        printf ("error on sgemm \n");
    }

    // get C down from device
    //stat = cublasGetMatrix(N, M, N*M*sizeof(float), C_device, N, C.data(), N);

    // copy result down from device
    checkCudaErrors(cudaMemcpy(C.data(), C_device, byteSize_C, cudaMemcpyDeviceToHost));

    printf("C: \n");
    batched_print_matrix(C, N, M, batch_size);

    cudaFree(A_device);
    cudaFree(B_device);
    cudaFree(C_device);
```

For example: batched_cublas_matmul.cpp

THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE