GPU Boosted Mitosis Detection

ECE408 Applied Parallel Programming Final Project

Zibo Lin, Shihao Su, Zhenbang Wu

Directed by Prof. Steve Lumetta & Prof. Tinggang Chew

ZJU-UIUC Institute

May 2019

Outline

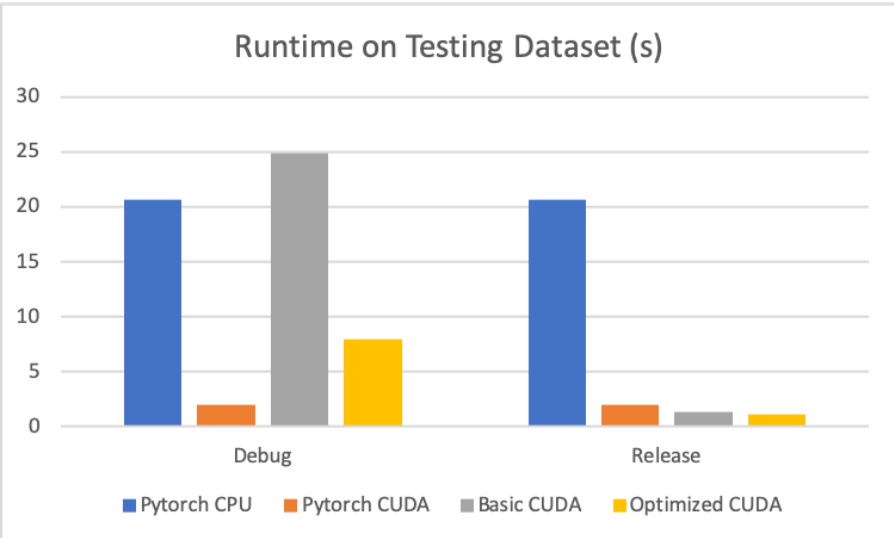
1. Overview
2. Background & Motivation
3. Before Parallelism
   - Data Preprocessing
   - CNN Training
4. Parallelism
   - Basic CUDA Approach
   - Optimization
5. Discussion

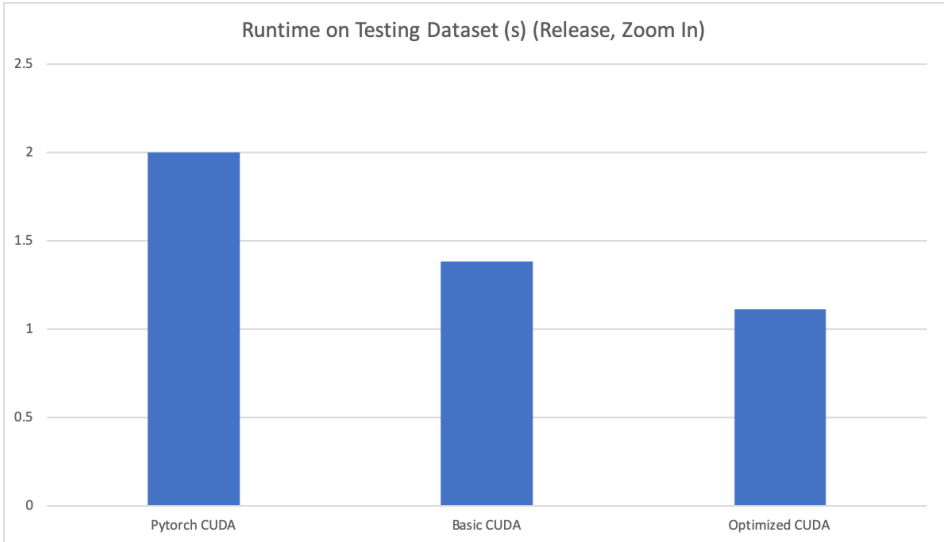
Overview

Outcome 1: Trained a CNN model to detect mitotic cells with 80.95% accuracy on testing data set.

- False Alarm Rate = 22%, Miss Rate = 10%
- True Postive Rate = 90%, True Negative Rate = 78%

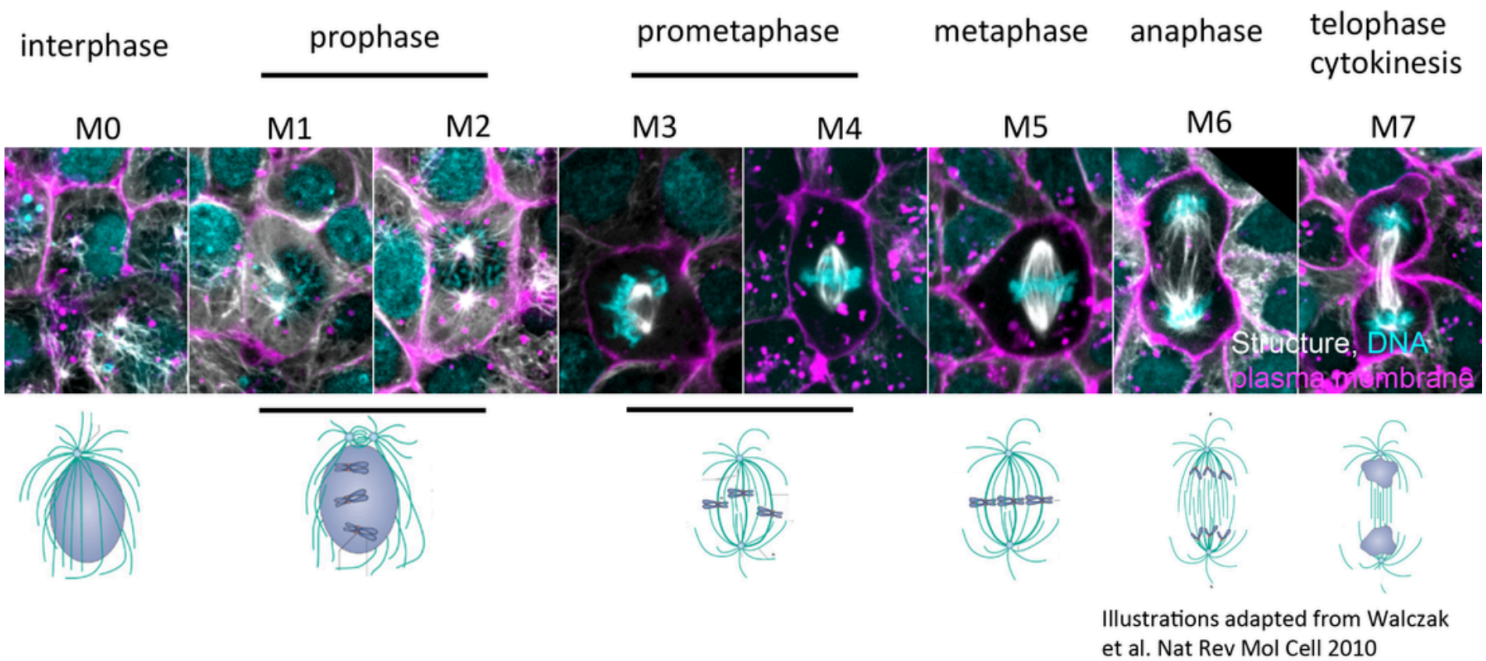
Outcome 2: Use CUDA to boost up the CNN forward propagation.





Cell Division – Cytokinesis

Cited from Prof. Chew

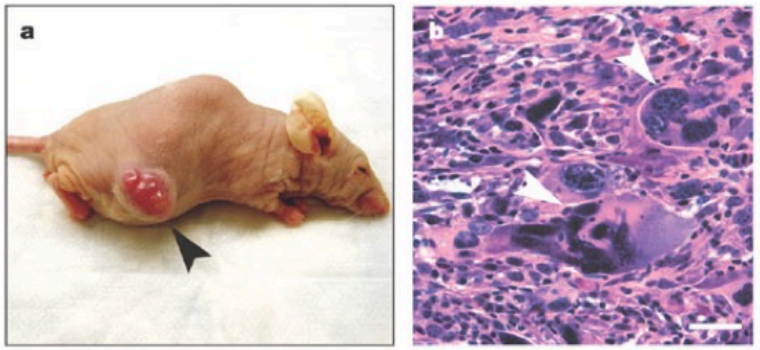


Illustrations adapted from Walczak et al. Nat Rev Mol Cell 2010

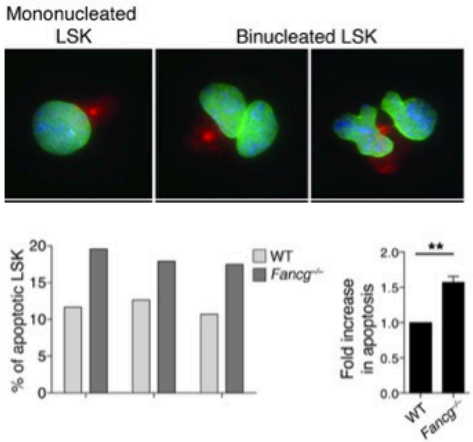
- A mechanical process to physically separate one cell into two;
- Increase cell number in a population or during embryonic development;
- Maintain tissue architectures and homeostasis.

Cytokinesis Failure and Cancer

Cited from Prof. Chew



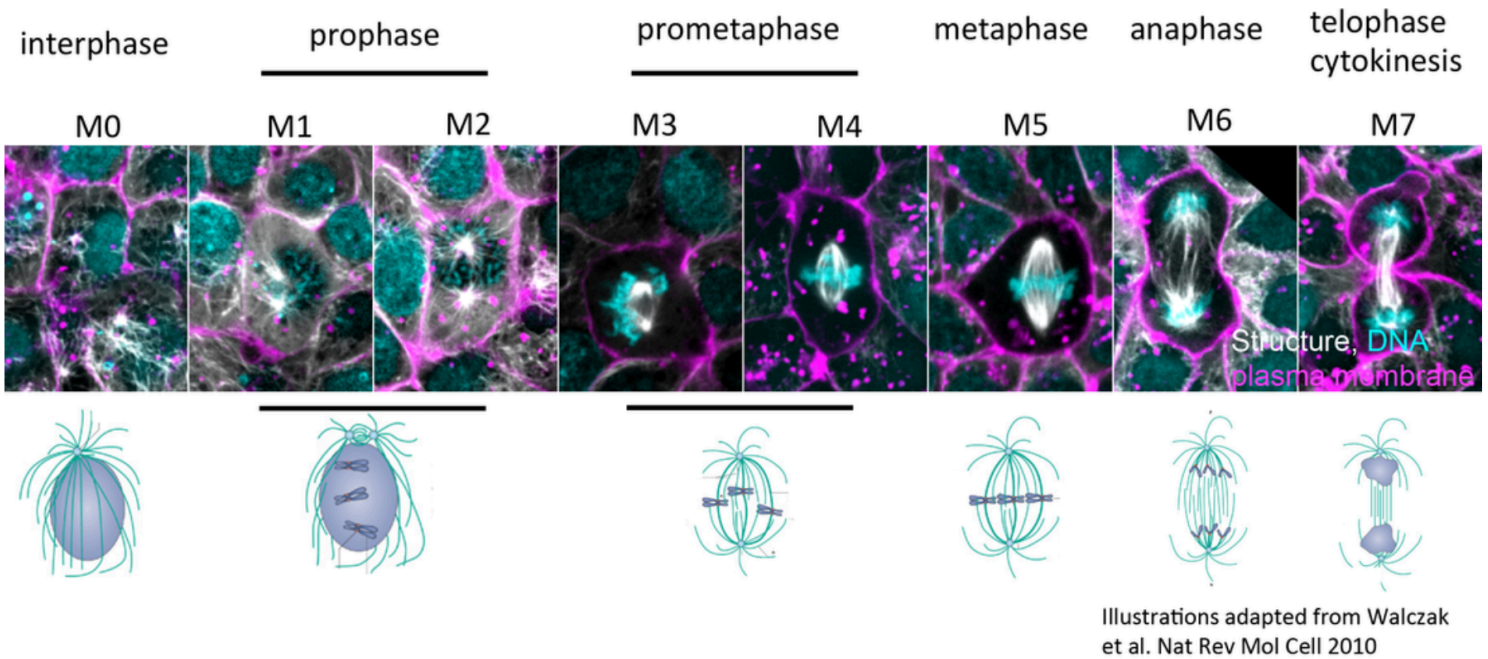
Tetraploid mammary epithelial cells
Nature, October 2005.



Fanconi anemia deficient cells
JCI, November 2010.

- Cytokinesis failure results in chromosomal instability;
- Accumulation of cells with tetraploid or aneuploidy;
- Cells fail cytokinesis form tumors in the absence of p53;
- Tetraploid cells are genetically unstable, and prone to make mistake during cell division and become aneuploidy, which is a hallmark of cancer.

Motivation



Illustrations adapted from Walczak et al. Nat Rev Mol Cell 2010

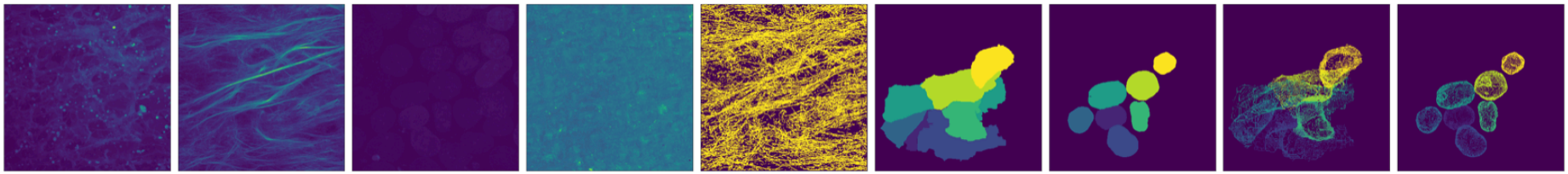
Aim: Detect mitotic cells using CNN and boost the CNN forward propagation in a parallel way.

Outline

1. Overview
2. Background & Motivation
3. Before Parallelism
   - Data Preprocessing
   - CNN Training
4. Parallelism
   - Basic CUDA Approach
   - Optimization
5. Discussion

Data Set

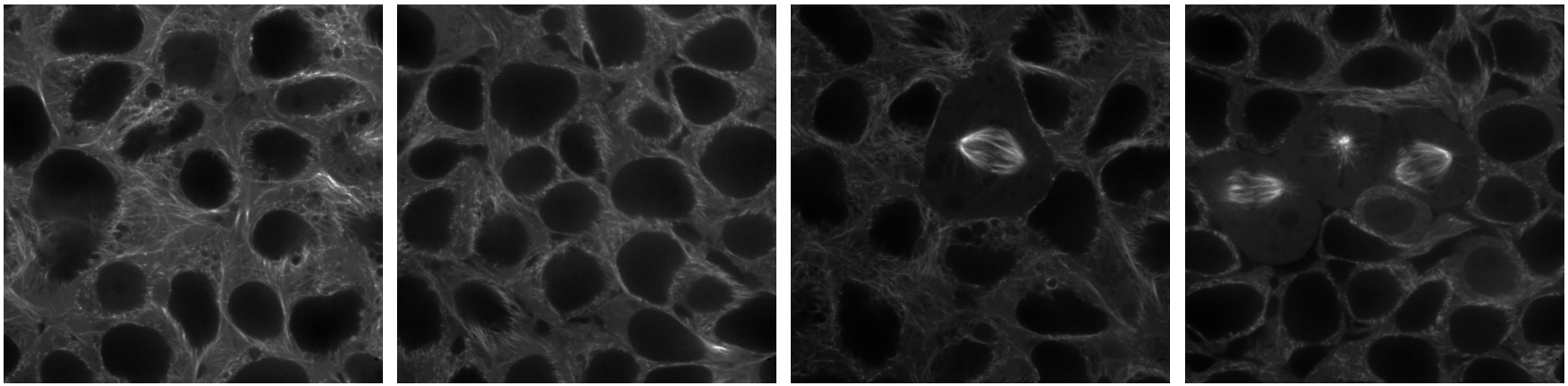
Ome-Tiff Image



0. DRAQ5 or CMDRP: membrane dye
1. EGFP: gene edited fluorescence protein
2. Hoechst 33258 or H3342: dna dye
3. TL Brightfield or Bright or Bright 100: brightfield (transmitted light) image
4. SEG_STRUCT: binary structure (EGFP) segmentation (filled volume)
5. SEG_Memb: binary membrane segmentation (filled volume). indexed by cell number. indices will not be contiguous.
6. SEG_DNA: binary nucelus segmentation (filled volume). indexed by cell number. indices will not be contiguous.
7. CON_Memb: binary membrane contour segmentation
8. CON_DNA: binary nucleus contour segmentation

Data Preprocessing

- Slicing: slice the 3D image in the middle;
- Resize: resize the sliced image into (512, 512) grayscale and normalized (min-max) images;
- CSV: convert the images into CSV files.

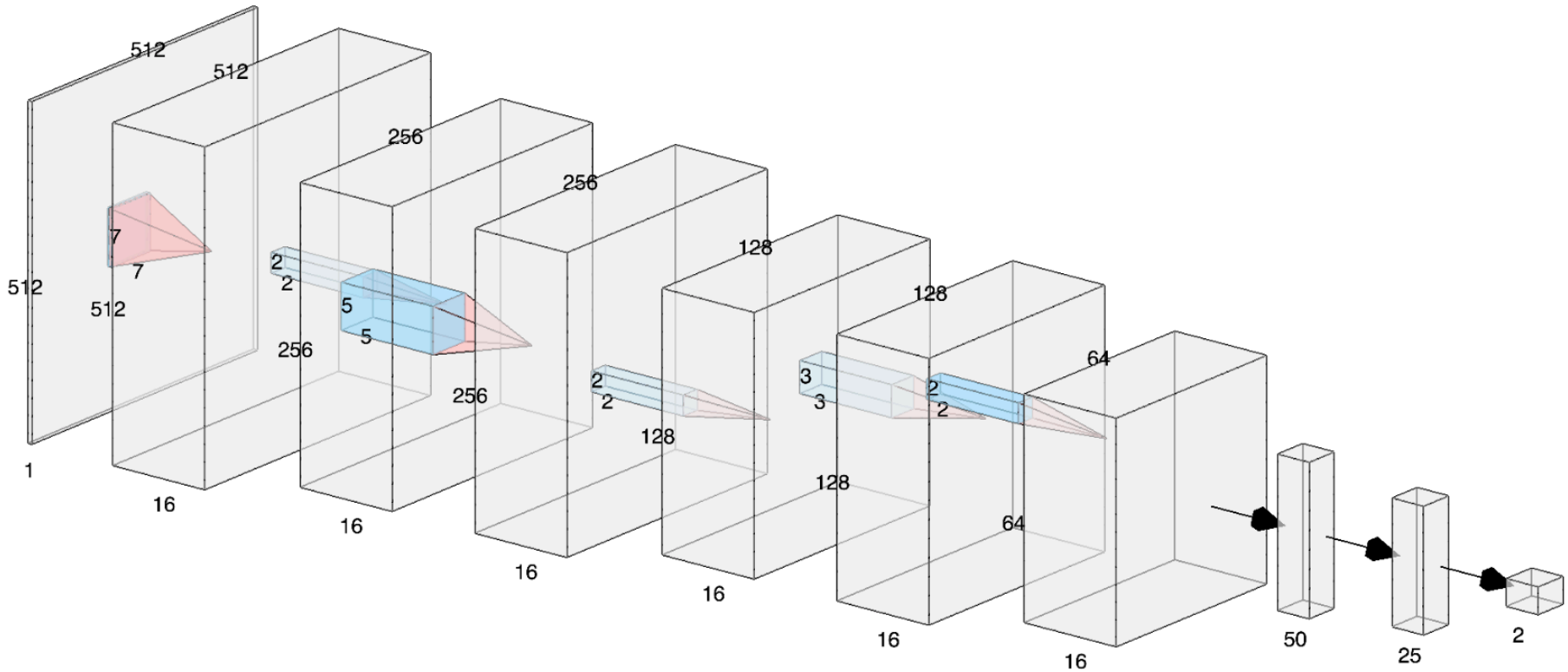


Outline

1. Overview
2. Background & Motivation
3. Before Parallelism
   - Data Preprocessing
   - CNN Training
4. Parallelism
   - Basic CUDA Approach
   - Optimization
5. Discussion

CNN Training

Model



alexlenail.me/NN-SVG/

- 3 Convolutional Layers + 3 MaxPooling Layers
- 3 Fully Connected Layers
- Adapted from Ankur Kunder & Rahul Kadyan's model

Training

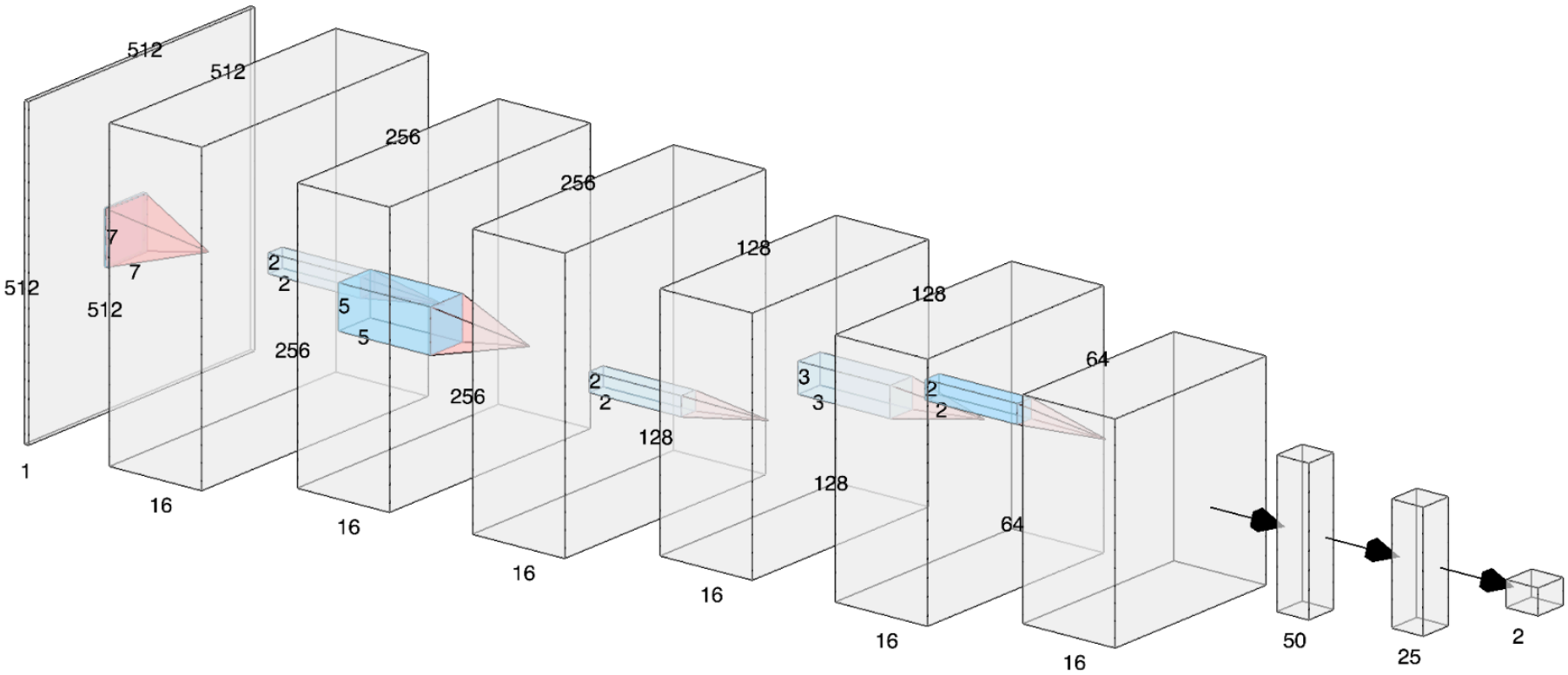
- Manually label 928 images obtained from Allen Cell Institute
- Split the images into training set (90%) and testing set (10%)
- batch size = 5, shuffle = True
- Stogastic Gradient Descent (lr = 2e-5, momentum = 0.9)
- total epochs trained = 200
- Average loss on training set = 0.117
- Accuracy on testing set = 80.95%

Outline

1. Overview
2. Background & Motivation
3. Before Parallelism
   - Data Preprocessing
   - CNN Training
4. Parallelism
   - Basic CUDA Approach
   - Optimization
5. Discussion

Basic CUDA Approach

CNN Forward Propagation



alexlenail.me/NN-SVG/

Basic CUDA Approach

CNN Forward Propagation

Convolution Layer
- Convolution
- Bias
- ReLU

Conv2d

CLASS `torch.nn.Conv2d(`*`in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros'`*`)` [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

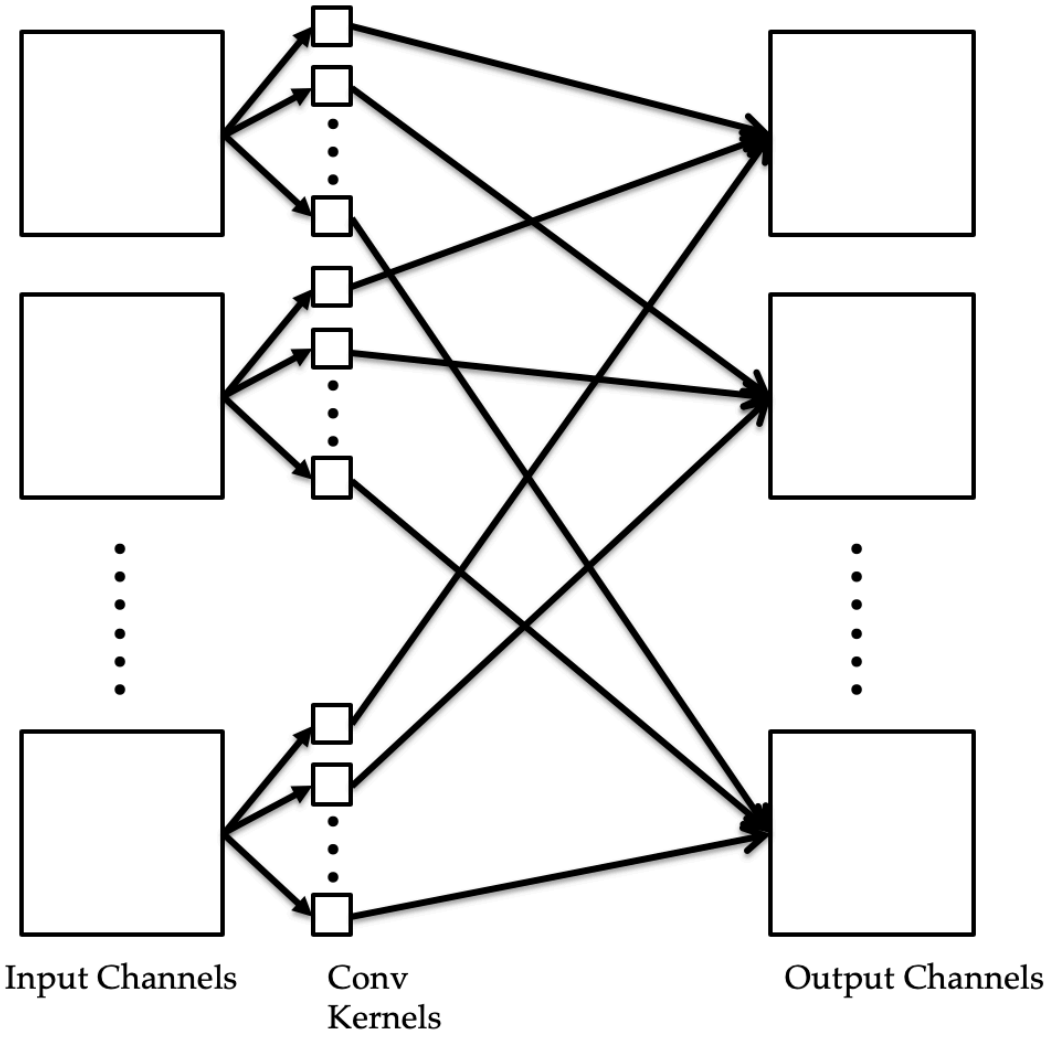
$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

pytorch.org

Basic CUDA Approach

CNN Forward Propagation



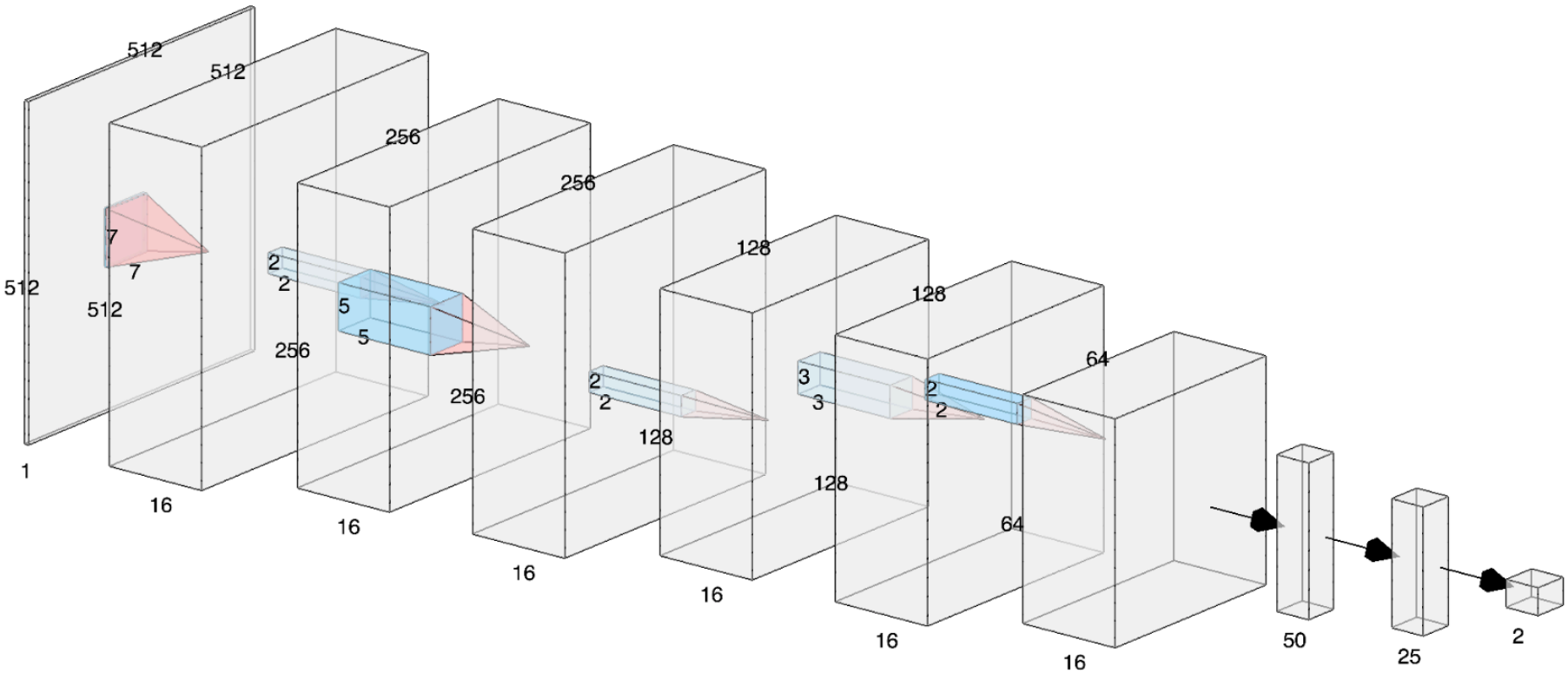
Basic CUDA Approach

CNN Forward Propagation

```
__constant__ float deviceMask[MAX_N_W*MAX_MASK_WIDTH*MAX_MASK_WIDTH];

__global__ void conv2dKernel(float* deviceInput, int height, int width, int numMask, float* deviceOutput, int maskWidth) {
 __shared__ float tile[BLOCK_WIDTH][BLOCK_WIDTH];
 int tx, ty;
 int x_o, x_i, y_o, y_i;
 float p;
 tx = threadIdx.x;
 ty = threadIdx.y;
 x_o = TILE_WIDTH*blockIdx.x + tx;
 y_o = TILE_WIDTH*blockIdx.y + ty;
 x_i = x_o - maskWidth / 2;
 y_i = y_o - maskWidth / 2;

 if (x_i >= 0 && x_i < width && y_i >= 0 && y_i < height) {
   tile[ty][tx] = deviceInput[y_i * width + x_i + blockIdx.z*height*width];
 }
 else {
   tile[ty][tx] = 0;
 }
 __syncthreads();

 if (tx < TILE_WIDTH && ty < TILE_WIDTH) {
   //for (int k = 0; k < numMask; k++) {
   p = 0;
   for (int i = 0; i < maskWidth; i++) {
    for (int j = 0; j < maskWidth; j++) {
      p += tile[ty + i][tx + j] * deviceMask[blockIdx.z * maskWidth * maskWidth + i * maskWidth + j];
    }
   }
   if (x_o >= 0 && x_o < width  && y_o >= 0 && y_o < height) {
    atomicAdd(&(deviceOutput[x_o + y_o * width]), p);
   }
   //}
 }
```

Basic CUDA Approach

CNN Forward Propagation



alexlenail.me/NN-SVG/

Fully connected layer
- Matrix Multiplication
- Bias
- ReLU

Basic CUDA Approach

CNN Forward Propagation

```
__global__ void maxPool2dKernel(float* deviceInput, int height, int width, int channels, float* deviceOutput, int poolHeight, int poolWidth) {
//////////
}

__global__ void matrixMultiply2dKernel(float* deviceInputA, float* deviceInputB, float* deviceOutput,
 int heightA, int widthA, int heightB, int widthB, int heightC, int widthC) {
 __shared__ float subA[BLOCK_SIZE][BLOCK_SIZE];
 __shared__ float subB[BLOCK_SIZE][BLOCK_SIZE];
//////////
}

__global__ void fc1matrixMultiply2dKernel(float* deviceInputA, float* deviceInputB, float* deviceOutput,
 int heightA, int widthA, int heightB, int widthB, int heightC, int widthC) {
 __shared__ float subResult[BLOCK_HEIGHT][BLOCK_WIDTH];
 __shared__ float subB[BLOCK_WIDTH];
/////////
}

__global__ void total(float *input, float *output, int len) {
 __shared__ float partial_sum[BLOCK_WIDTH];
///////
}

__global__ void conv2dKernel(float* deviceInput, int height, int width, int numMask, float* deviceOutput, int maskWidth) {
 __shared__ float tile[BLOCK_WIDTH][BLOCK_WIDTH];
//////////
}

__global__ void addBiasReLUKernel(float *c, int len, const float b){
   //////////
}
```

Basic CUDA Approach

Runtime

```
Image 87, conv3 running time:12.093376(ms)
Image 87, max3  running time:0.084320(ms)
Image 87, fc1   running time:88.324387(ms)
Image 87, fc2   running time:0.251456(ms)
Image 87, fc3   running time:0.384256(ms)
Image 87, total running time:235.467514(ms)
 -0.5043   0.3057     |     1
Image 88, conv1 running time:45.320190(ms)
Image 88, max1  running time:0.614912(ms)
Image 88, conv2 running time:89.012672(ms)
Image 88, max2  running time:0.207424(ms)
Image 88, conv3 running time:12.962464(ms)
Image 88, max3  running time:0.092416(ms)
Image 88, fc1   running time:88.346146(ms)
Image 88, fc2   running time:0.512064(ms)
Image 88, fc3   running time:0.196576(ms)
Image 88, total running time:237.264877(ms)
 -0.4521   0.2433     |     1
Image 89, conv1 running time:44.296577(ms)
Image 89, max1  running time:0.617344(ms)
Image 89, conv2 running time:89.691650(ms)
Image 89, max2  running time:0.195488(ms)
Image 89, conv3 running time:13.645856(ms)
Image 89, max3  running time:0.083136(ms)
```

Outline

1. Overview
2. Background & Motivation
3. Before Parallelism
   - Data Preprocessing
   - CNN Training
4. Parallelism
   - Basic CUDA Approach
   - Optimization
5. Discussion

Basic CUDA Approach

Performance Observation

```
Image 88, conv1 running time:45.320190(ms)
Image 88, max1  running time:0.614912(ms)
Image 88, conv2 running time:89.012672(ms)
Image 88, max2  running time:0.207424(ms)
Image 88, conv3 running time:12.962464(ms)
Image 88, max3  running time:0.092416(ms)
Image 88, fc1   running time:88.346146(ms)
Image 88, fc2   running time:0.512064(ms)
Image 88, fc3   running time:0.196576(ms)
Image 88, total running time:237.264877(ms)
 -0.4521   0.2433     |     1
```

Takes up 37.5% of the time

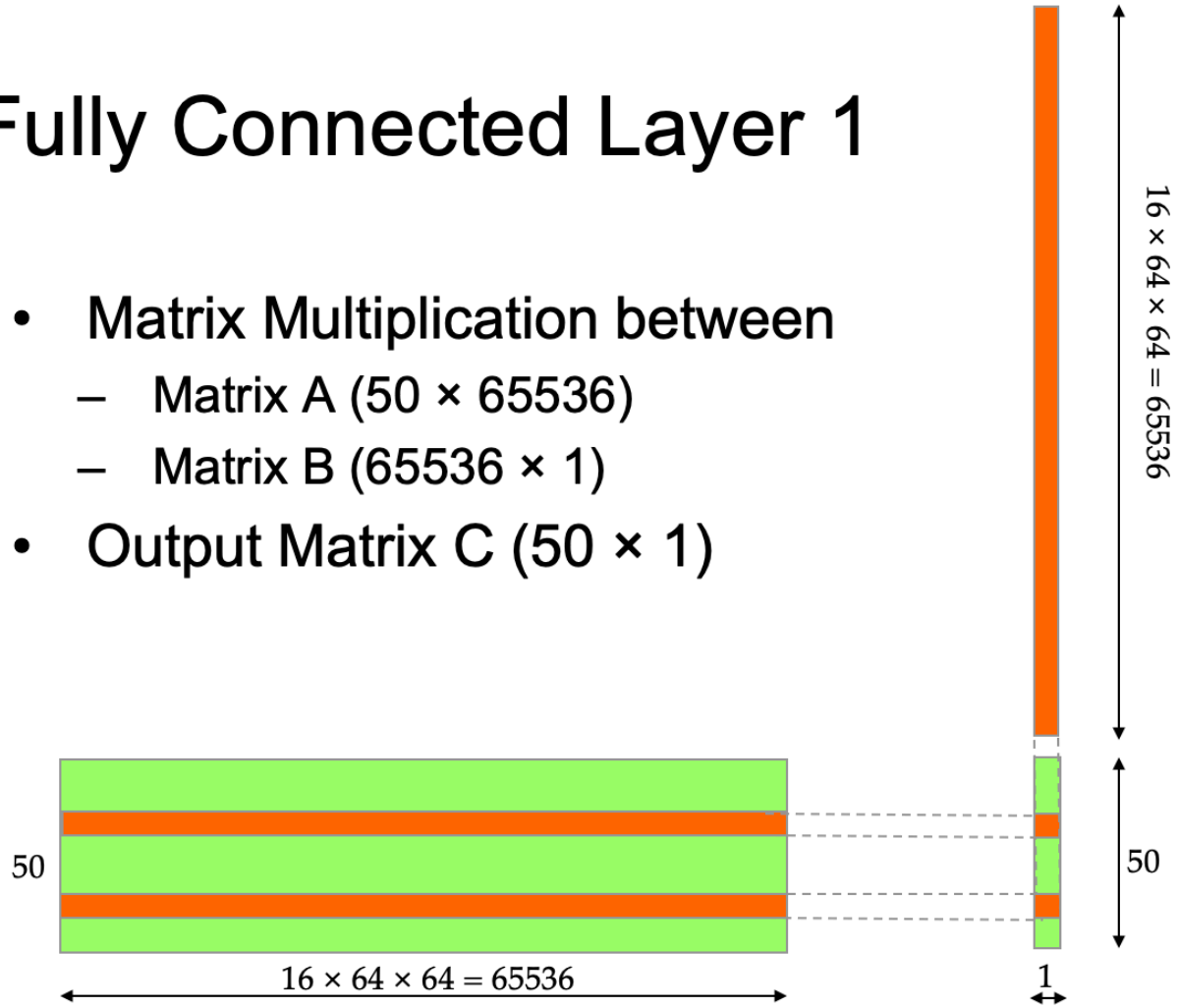
Takes up 37.2% of the time

Optimization

- Optimized fully connected layer 1 computation
- More appropriate block size
- Compiler optimization

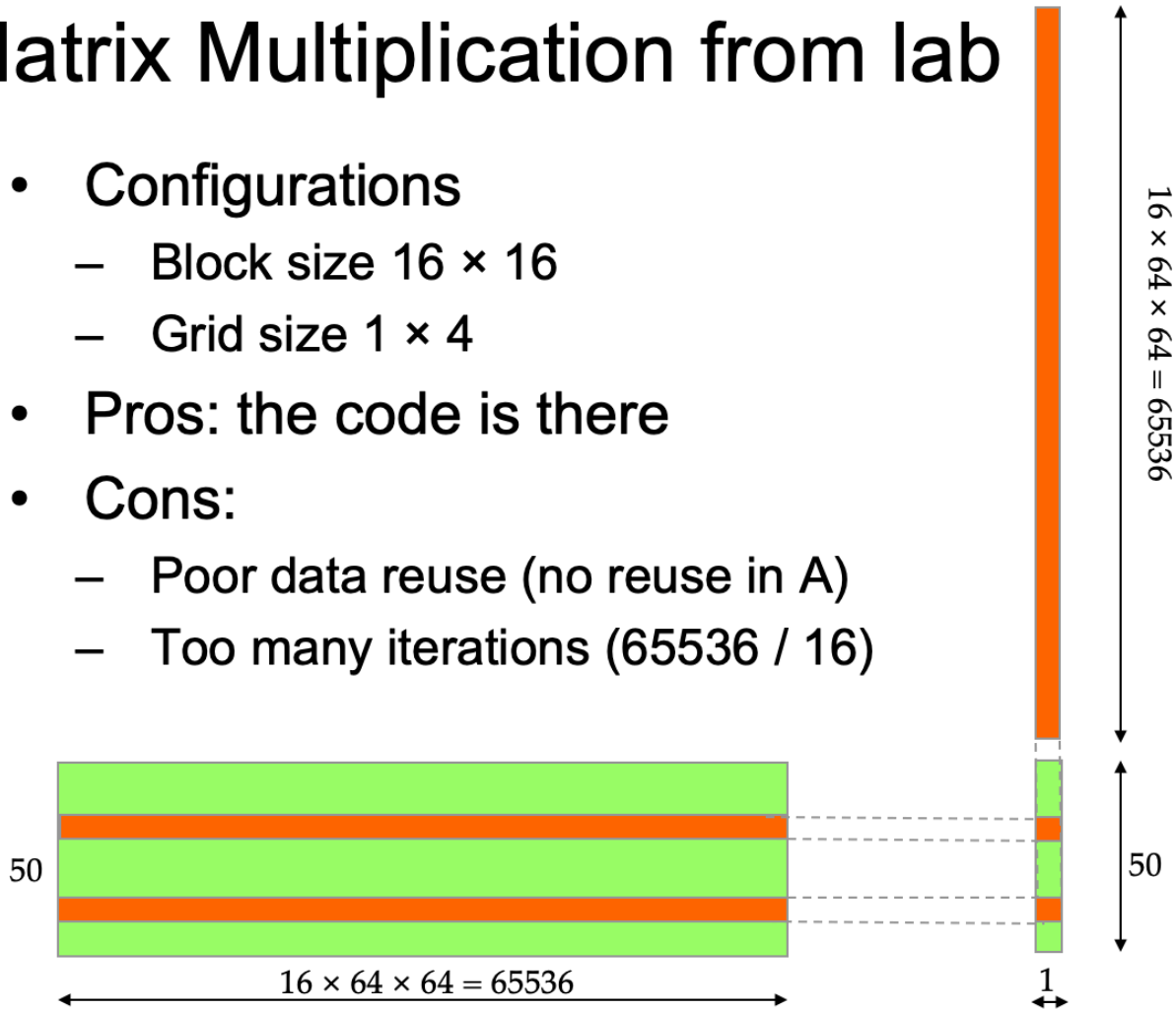
Fully Connected Layer 1

- Matrix Multiplication between
  - Matrix A (50 × 65536)
  - Matrix B (65536 × 1)
- Output Matrix C (50 × 1)



Matrix Multiplication from lab

- Configurations
  - Block size 16 × 16
  - Grid size 1 × 4
- Pros: the code is there
- Cons:
  - Poor data reuse (no reuse in A)
  - Too many iterations (65536 / 16)

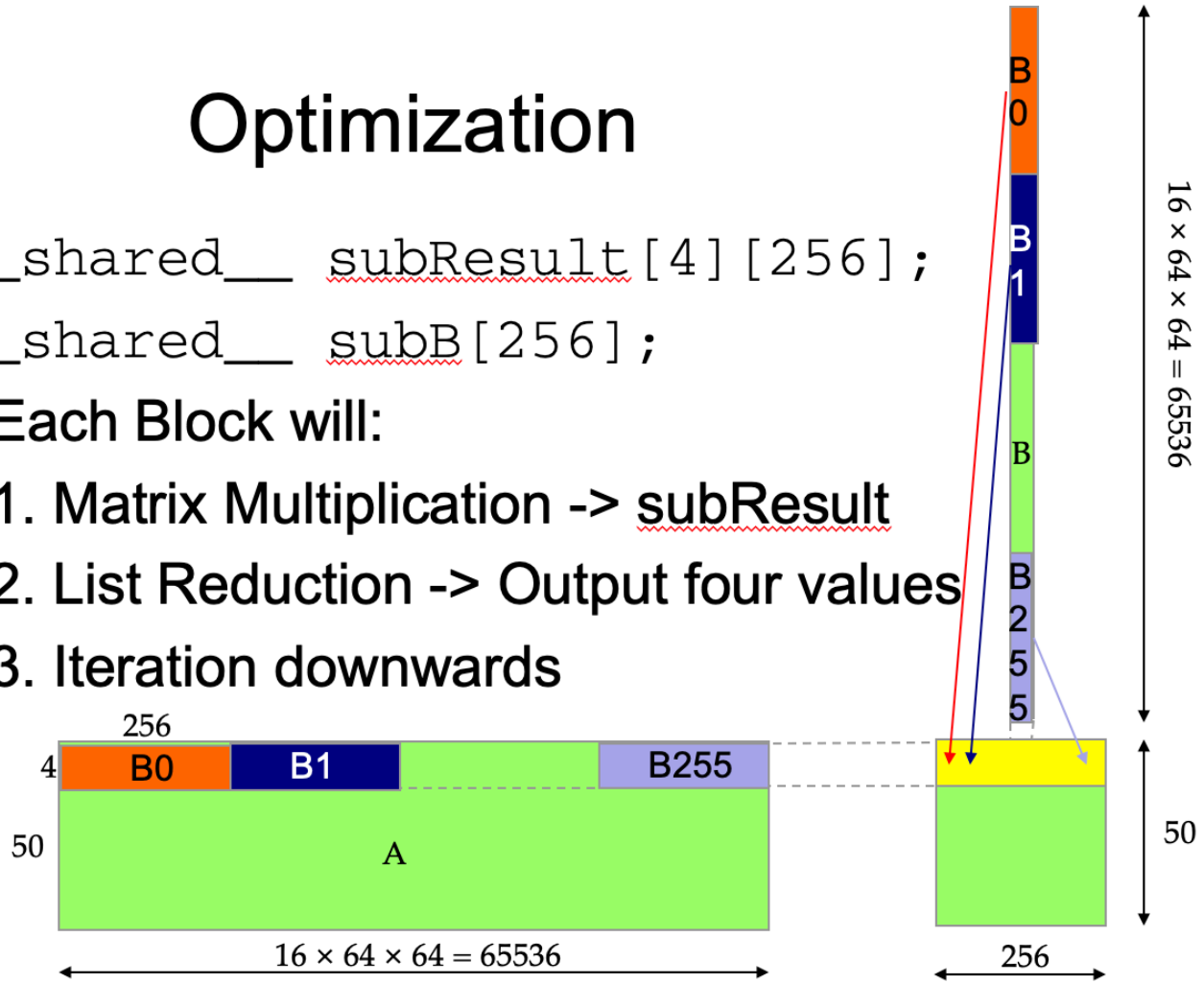


Optimization

```
__shared__ subResult[4][256];
__shared__ subB[256];
```

Each Block will:

1. Matrix Multiplication -> subResult
2. List Reduction -> Output four values
3. Iteration downwards



Optimization

Fully Connected Layer

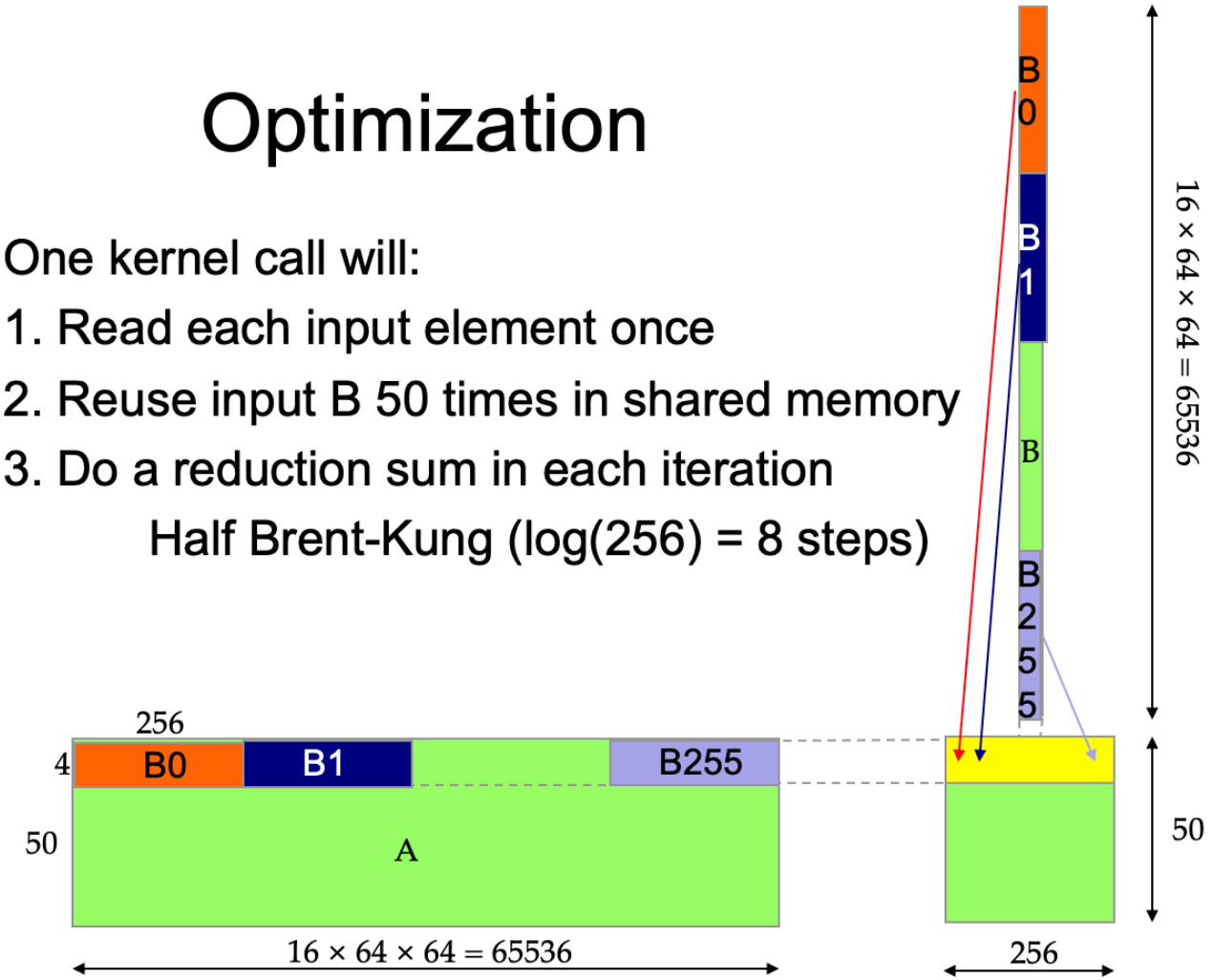
```
if (x_A < widthA && ty == 0) {
    subB[tx] = deviceInputB[x_A];
}
else if (y_A >= heightA) {
    subB[tx] = 0;
}
for (int i = 0; i < (heightA − 1) / BLOCK_HEIGHT + 1; i++) {
    __syncthreads();
    y_A = i ∗ blockDim.y + ty;
    if (y_A < heightA)
        subResult[ty][tx] = subB[tx] ∗ deviceInputA[y_A ∗ widthA + x_A];
    for (int stride = BLOCK_WIDTH / 2; stride >= 1; stride >>= 1) {
        __syncthreads();
        if (tx < stride)
            subResult[ty][tx] += subResult[ty][tx + stride];
    }
    if (tx == 0 && y_A < heightA)
        deviceOutput[y_A ∗ widthC + bx] = subResult[ty][0];
}
```

Optimization

One kernel call will:

1. Read each input element once
2. Reuse input B 50 times in shared memory
3. Do a reduction sum in each iteration

   Half Brent-Kung (log(256) = 8 steps)



Optimization

Fully Connected Layer

Image 88, conv1 running time:45.320190(ms)	Image 88, conv1 running time:23.244703(ms)
Image 88, max1 running time:0.614912(ms)	Image 88, max1 running time:0.616960(ms)
Image 88, conv2 running time:89.012672(ms)	Image 88, conv2 running time:42.267456(ms)
Image 88, max2 running time:0.207424(ms)	Image 88, max2 running time:0.193568(ms)
Image 88, conv3 running time:12.962464(ms)	Image 88, conv3 running time:8.099584(ms)
Image 88, max3 running time:0.092416(ms)	Image 88, max3 running time:0.081760(ms)
Image 88, fc1 running time:88.346146(ms)	Image 88, fc1 running time:1.398176(ms)
Image 88, fc2 running time:0.512064(ms)	Image 88, fc2 running time:0.242496(ms)
Image 88, fc3 running time:0.196576(ms)	Image 88, fc3 running time:0.190208(ms)
Image 88, total running time:237.264877(ms)	Image 88, total running time:76.334908(ms)

Optimization

More Appropriate Block Size

Convolution: bigger block size, better data reuse.

Smaller	→	Bigger
Image 88, conv1 running time:45.320190(ms)		Image 88, conv1 running time:23.244703(ms)
Image 88, max1 running time:0.614912(ms)		Image 88, max1 running time:0.616960(ms)
Image 88, conv2 running time:89.012672(ms)		Image 88, conv2 running time:42.267456(ms)
Image 88, max2 running time:0.207424(ms)		Image 88, max2 running time:0.193568(ms)
Image 88, conv3 running time:12.962464(ms)		Image 88, conv3 running time:8.099584(ms)
Image 88, max3 running time:0.092416(ms)		Image 88, max3 running time:0.081760(ms)
Image 88, fc1 running time:88.346146(ms)		Image 88, fc1 running time:1.398176(ms)
Image 88, fc2 running time:0.512064(ms)		Image 88, fc2 running time:0.242496(ms)
Image 88, fc3 running time:0.196576(ms)		Image 88, fc3 running time:0.190208(ms)
Image 88, total running time:237.264877(ms)		Image 88, total running time:76.334908(ms)

Debug	→	Release
Image 88, conv1 running time:23.244703(ms)		Image 88, conv1 running time:2.802976(ms)
Image 88, max1 running time:0.616960(ms)		Image 88, max1 running time:0.160192(ms)
Image 88, conv2 running time:42.267456(ms)		Image 88, conv2 running time:3.727840(ms)
Image 88, max2 running time:0.193568(ms)		Image 88, max2 running time:0.073024(ms)
Image 88, conv3 running time:8.099584(ms)		Image 88, conv3 running time:2.163200(ms)
Image 88, max3 running time:0.081760(ms)		Image 88, max3 running time:0.057536(ms)
Image 88, fc1 running time:1.398176(ms)		Image 88, fc1 running time:0.433472(ms)
Image 88, fc2 running time:0.242496(ms)		Image 88, fc2 running time:0.145728(ms)
Image 88, fc3 running time:0.190208(ms)		Image 88, fc3 running time:0.145248(ms)
Image 88, total running time:76.334908(ms)		Image 88, total running time:9.709215(ms)

Discussion

- CUDA optimization is really hard to generalized;
- What if the cell (spindle) is not aligned with the z-axis?
- Our model will be more robust if given more data;
- (Maybe) we can try some more advanced DL networks for this problem (e.g. YOLO, SSD).