UC San Diego

Optimizing Deep Convolutional Generative Adversarial Networks Using CUDA

ECE 277 - Final Project, Winter 2025

Kenneth Casimiro, Electrical Engineering M.S.
Nathaniel Greenberg, Electrical Engineering M.S.

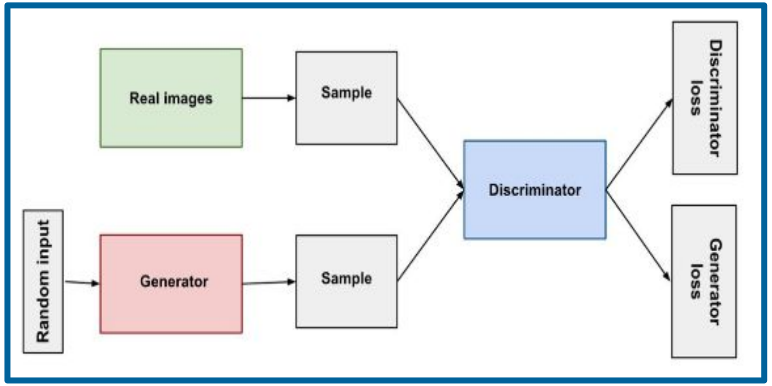
UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

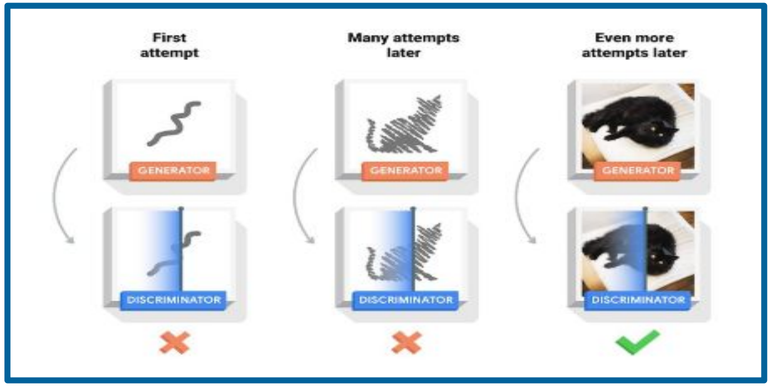
Contents

Background

Generative Adversarial Networks (GAN)

- Generator
- Discriminator
- Adversarial Process
- Creating Realistic Images





Project Overview

Project Overview - DCGAN Optimization

Objective

- This project aims to enhance the performance of DCGANs through CUDA optimizations

Scope

- Focus on lightweight architectures with a custom DCGAN trained on MNIST and Celeb-A
- Benchmark GPU performance against CPU execution for real-time applications

Datasets

- **MNIST:** A dataset of 70,000 grayscale 28×28 images of handwritten digits (0-9), widely used for image classification
- **Celeb-A:** A large-scale dataset of 200,000+ celebrity face images with 40 labeled attributes, used for facial analysis and GANs

Outcomes

- Demonstrate significant improvements in speed and efficiency through CUDA optimizations

Architecture

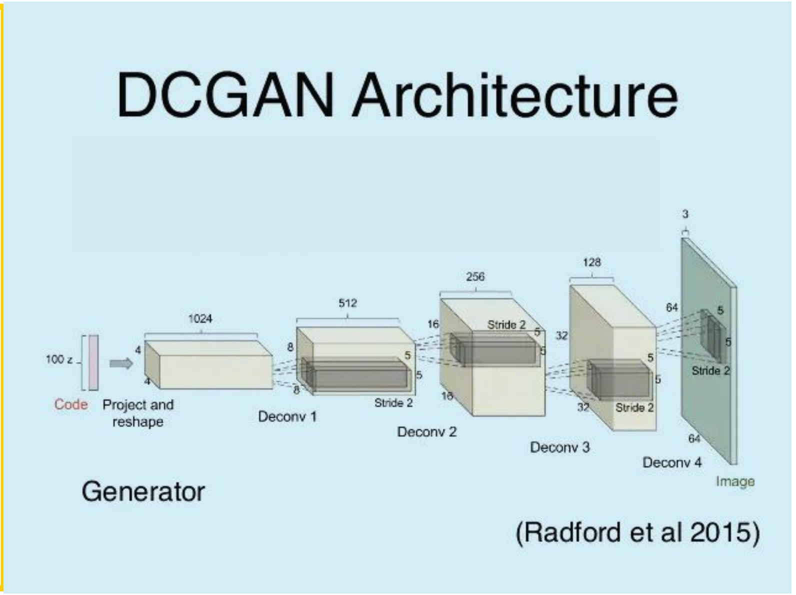
DCGAN Architecture

Generator

- Layer 1: Transposed Conv (100 → 512), 4×4, stride 1, padding 0
- Layer 2: Transposed Conv (512 → 256), 4×4, stride 2, padding 1
- Layer 3: Transposed Conv (256 → 128), 4×4, stride 2, padding 1
- Layer 4: Transposed Conv (128 → 3), 4×4, stride 2, padding 1 (RGB output)

Discriminator

- Layer 1: Conv (3 → 128), 4×4, stride 2, padding 1
- Layer 2: Conv (128 → 256), 4×4, stride 2, padding 1
- Layer 3: Conv (256 → 512), 4×4, stride 2, padding 1
- Layer 4: Conv (512 → 1), 4×4, stride 1, padding 0



DCGAN Architecture with DataHub

Baseline (CPU)

- No hardware acceleration → slowest training
- Uses Sigmoid + BCELoss (less stable)
- Adam optimizer (standard but not optimal)

FP32 (GPU - Full Precision)

- Runs on GPU with Tensor Core acceleration
- Uses BCEWithLogitsLoss (more stable) + AdamW optimizer
- Faster than CPU but high memory usage

Key Takeaway:

- CPU → Slowest, no acceleration
- FP32 → Optimized, stable, higher memory
- FP16 → Fastest, lower memory, possible instability

FP16 (GPU - Mixed Precision)

- Uses AMP for reduced memory & faster computation
- Enables larger batch size
- May cause numerical instability if not scaled properly

DCGAN Architecture DataHub Results

Image from CPU Baseline (EMNIST)

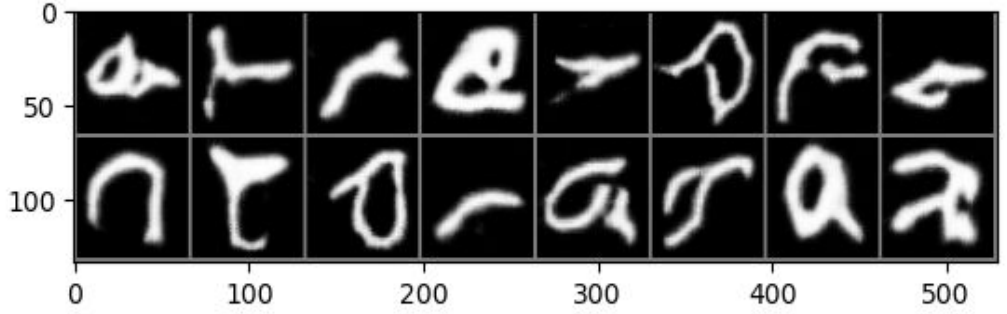
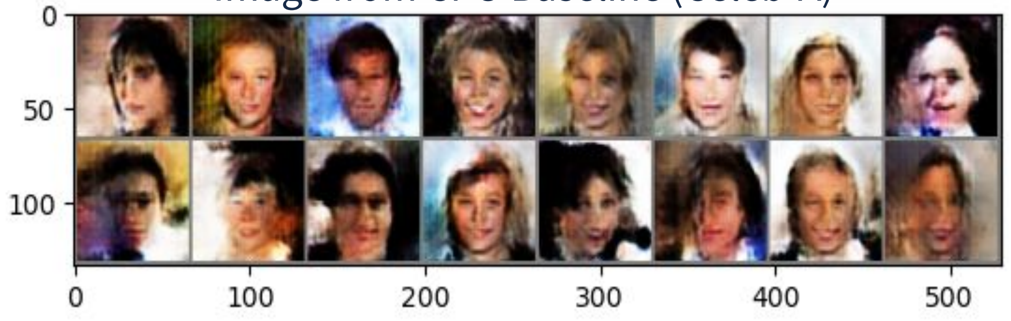


Image from CPU Baseline (Celeb-A)



Test Name	EMNIST Runtime	CELEB-A Runtime
CPU Baseline	1611.83 seconds.	1523.73 seconds.
FP32	182.20 seconds.	179.67 seconds.
FP16	85.10 seconds.	91.70 seconds.

Custom CUDA DCGAN Implementation

- **Replaced PyTorch ops** with custom CUDA kernels for matrix multiplications, convolutions, activations, etc.
- **Implemented CUDA kernels** in a separate .cu file and compiled them with nvcc.
- **Created Python wrappers** using ctypes to call CUDA functions from Python.
- **Generator & Discriminator** use these CUDA kernel functions instead of PyTorch built-in operations.
- **Optimized for GPU acceleration**, reducing latency compared to native PyTorch implementations.
- **Achieves better control over computations** and enables deeper optimizations at the kernel level.

Custom CUDA DCGAN Results

Image from CUDA (EMNIST)

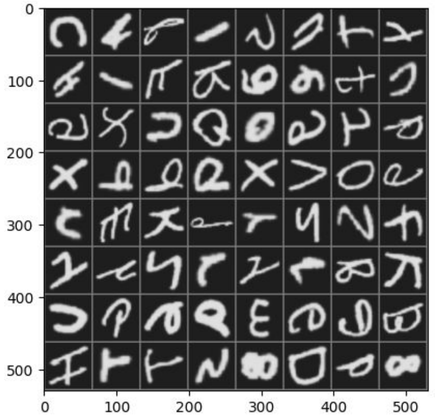
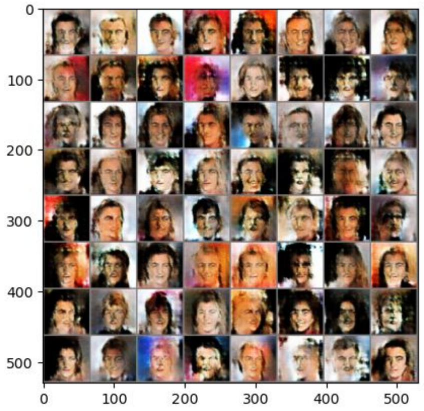


Image from CUDA (Celeb-A)

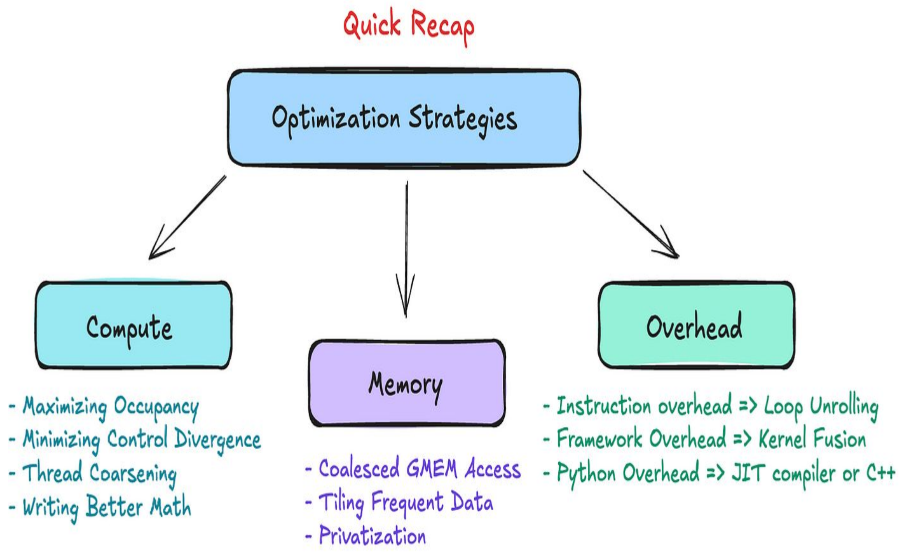


Test Name	EMNIST Runtime	CELEB-A Runtime
CUDA	54.94 seconds	113.70 seconds

CUDA Optimization Strategies

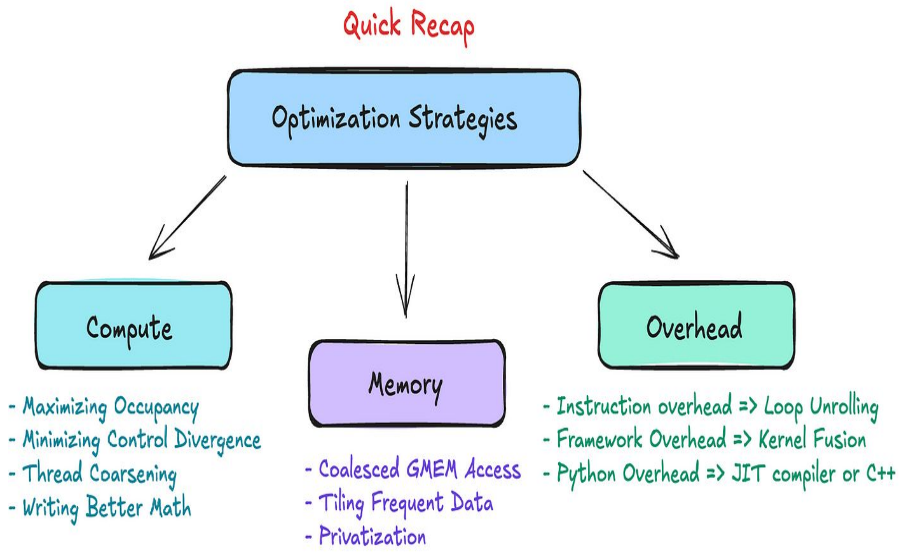
CUDA Optimization Strategies

- **Thread Parallelism:** Each thread processes one element, maximizing parallel execution (matrix multiplication)
- **Grid and Block Configuration:** Uses dim to efficiently map computations and ensure full dataset coverage
- **Memory Coalescing:** Structured memory access reduces latency, optimizing matrix operations
- **Shared Memory Usage:** Currently relies on global memory; leveraging shared memory can further improve performance, especially for matrix multiplication and convolution



CUDA Optimization Strategies

- **Optimized Conditional Logic:** Uses branchless operations (e.g., max() for ReLU) to improve efficiency
- **Maximized GPU Occupancy:** Launches 256 threads per block for element-wise ops, 16×16 blocks for 2D ops
- **Reduced Redundant Computation:** Precomputes batch norm statistics on CPU before passing to GPU
- **Efficient Strided Memory Access:** Manually handles strides in transposed convolution for proper alignment

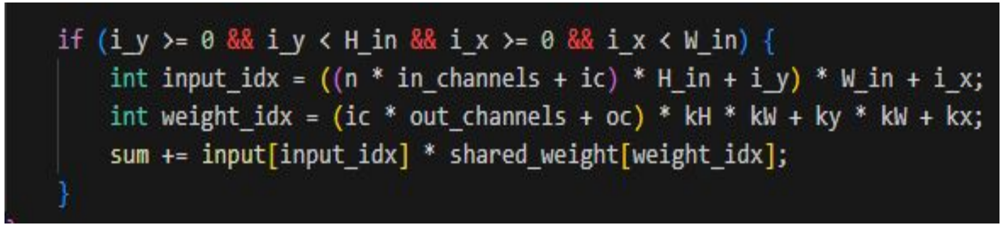


Applied CUDA Optimizations

Memory Coalesced Optimization

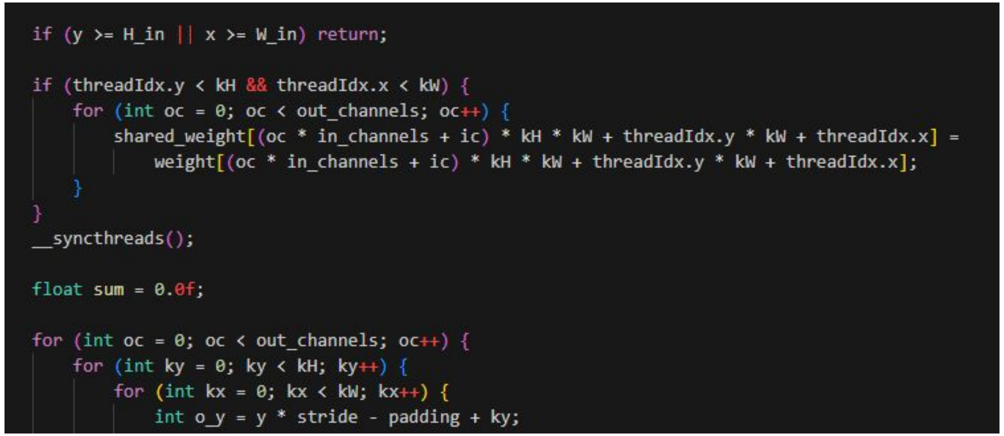
- **Uncoalesced memory access**: Scattered thread memory pattern
- **High memory latency:** Poor cache & slow reads
- **Indexing restructured:** Threads access contiguous elements
- **Coalesced execution**: Faster, efficient memory throughput

conv_transpose2d_optimized_kernel



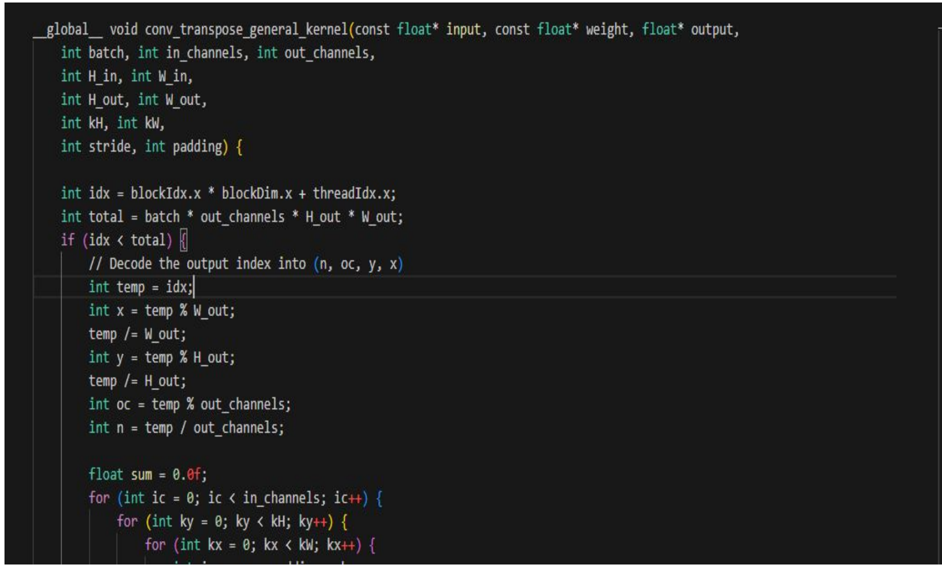
```
if (i_y >= 0 && i_y < H_in && i_x >= 0 && i_x < W_in) {
    int input_idx = ((n * in_channels + ic) * H_in + i_y) * W_in + i_x;
    int weight_idx = (ic * out_channels + oc) * kH * kW + ky * kW + kx;
    sum += input[input_idx] * shared_weight[weight_idx];
}
```

conv_transpose2d_backward_input_kernel_optimized

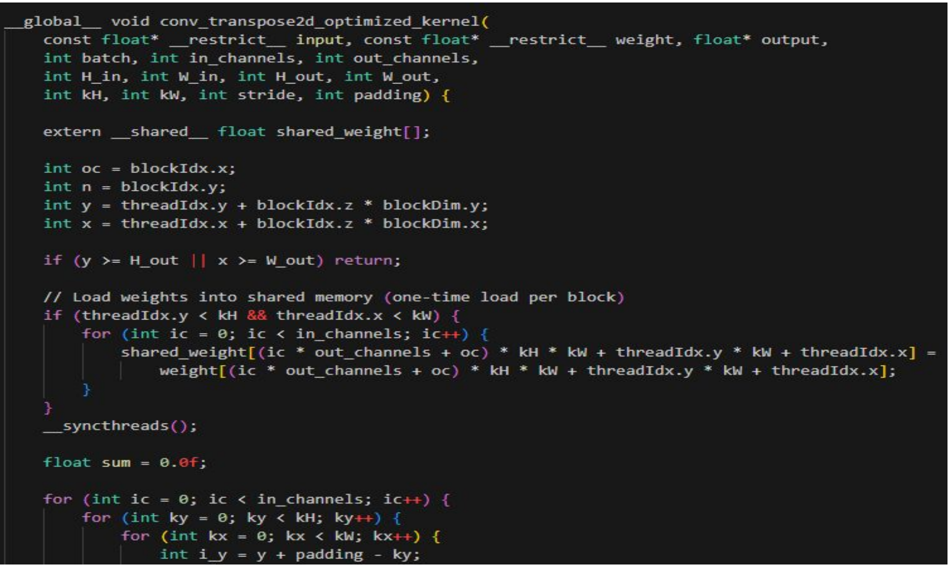


```
if (y >= H_in || x >= W_in) return;

if (threadIdx.y < kH && threadIdx.x < kW) {
    for (int oc = 0; oc < out_channels; oc++) {
        shared_weight[(oc * in_channels + ic) * kH * kW + threadIdx.y * kW + threadIdx.x] =
            weight[(oc * in_channels + ic) * kH * kW + threadIdx.y * kW + threadIdx.x];
    }
}
__syncthreads();

float sum = 0.0f;

for (int oc = 0; oc < out_channels; oc++) {
    for (int ky = 0; ky < kH; ky++) {
        for (int kx = 0; kx < kW; kx++) {
            int o_y = y * stride - padding + ky;
```

Shared Memory Optimization



```
__global__ void conv_transpose_general_kernel(const float* input, const float* weight, float* output,
    int batch, int in_channels, int out_channels,
    int H_in, int W_in,
    int H_out, int W_out,
    int kH, int kW,
    int stride, int padding) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int total = batch * out_channels * H_out * W_out;
    if (idx < total) {
        // Decode the output index into (n, oc, y, x)
        int temp = idx;
        int x = temp % W_out;
        temp /= W_out;
        int y = temp % H_out;
        temp /= H_out;
        int oc = temp % out_channels;
        int n = temp / out_channels;

        float sum = 0.0f;
        for (int ic = 0; ic < in_channels; ic++) {
            for (int ky = 0; ky < kH; ky++) {
                for (int kx = 0; kx < kW; kx++) {
```

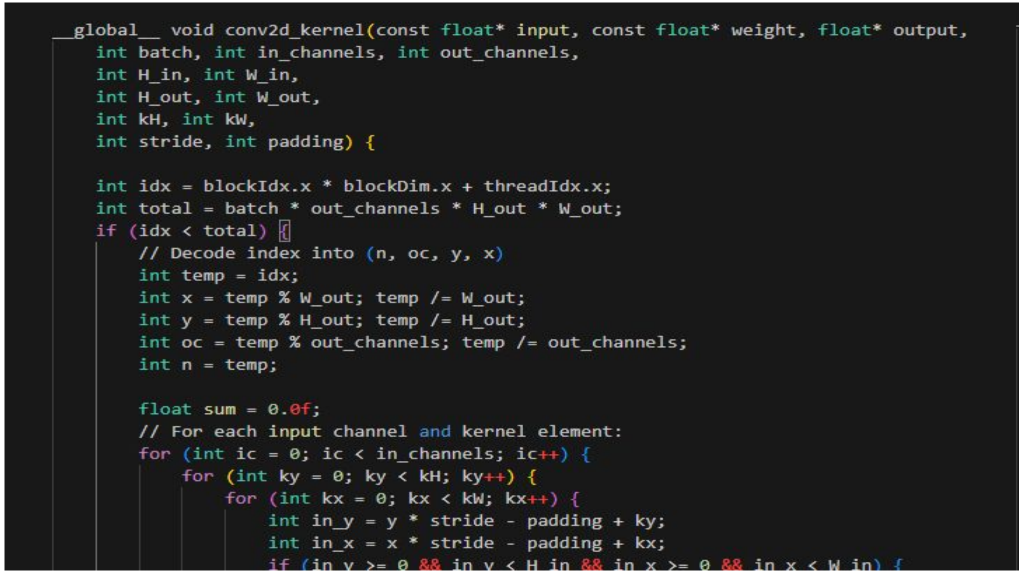


```
__global__ void conv_transpose2d_optimized_kernel(
    const float* __restrict__ input, const float* __restrict__ weight, float* output,
    int batch, int in_channels, int out_channels,
    int H_in, int W_in, int H_out, int W_out,
    int kH, int kW, int stride, int padding) {

    extern __shared__ float shared_weight[];

    int oc = blockIdx.x;
    int n = blockIdx.y;
    int y = threadIdx.y + blockIdx.z * blockDim.y;
    int x = threadIdx.x + blockIdx.z * blockDim.x;

    if (y >= H_out || x >= W_out) return;

    // Load weights into shared memory (one-time load per block)
    if (threadIdx.y < kH && threadIdx.x < kW) {
        for (int ic = 0; ic < in_channels; ic++) {
            shared_weight[(ic * out_channels + oc) * kH * kW + threadIdx.y * kW + threadIdx.x] =
                weight[(ic * out_channels + oc) * kH * kW + threadIdx.y * kW + threadIdx.x];
        }
    }
    __syncthreads();

    float sum = 0.0f;

    for (int ic = 0; ic < in_channels; ic++) {
        for (int ky = 0; ky < kH; ky++) {
            for (int kx = 0; kx < kW; kx++) {
                int i_y = y + padding - ky;
```

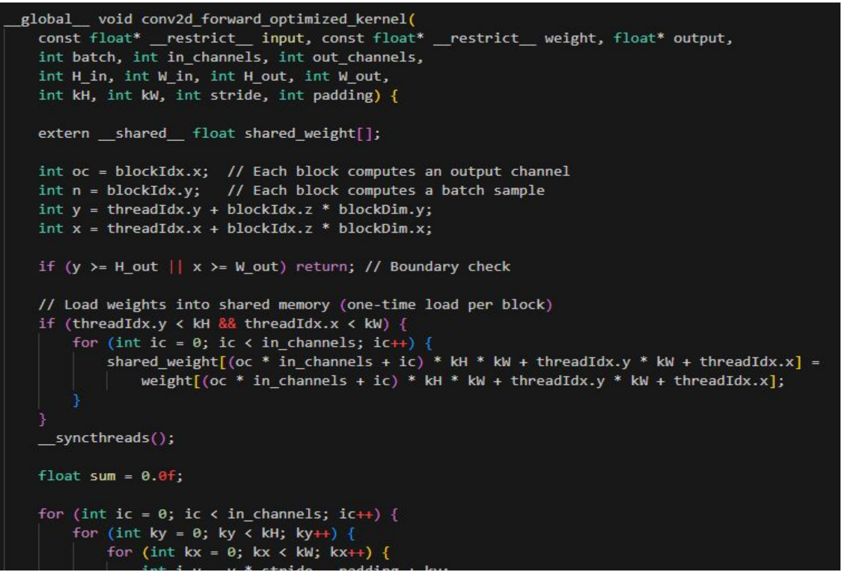
- Weights are loaded once into shared memory
- Significantly speeds up transposed convolution

- Shared memory allows for much faster data access compared to global memory
- Reduce memory accesses

Increased Parallelization



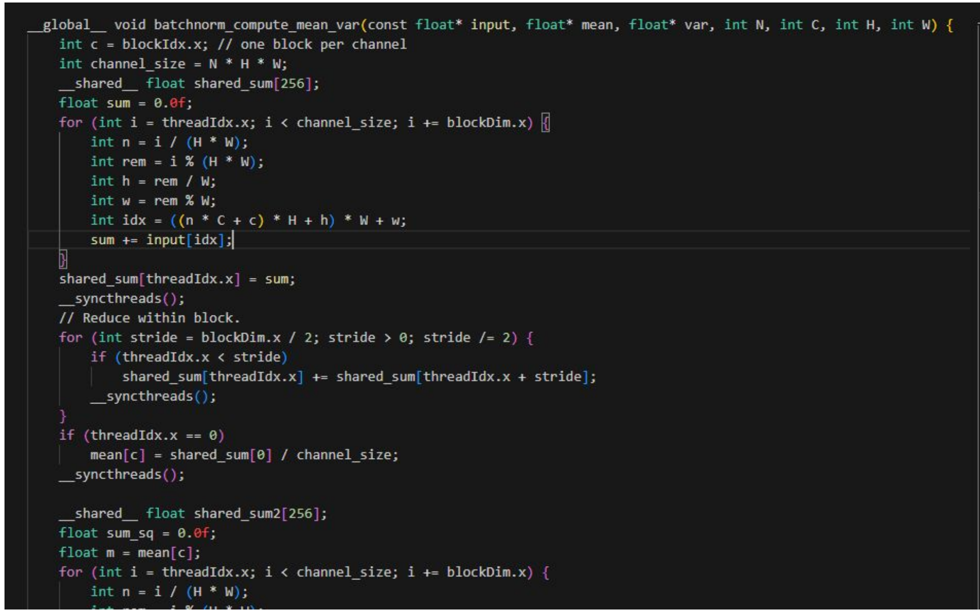
```
__global__ void conv2d_kernel(const float* input, const float* weight, float* output,
    int batch, int in_channels, int out_channels,
    int H_in, int W_in,
    int H_out, int W_out,
    int kH, int kW,
    int stride, int padding) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int total = batch * out_channels * H_out * W_out;
    if (idx < total) {
        // Decode index into (n, oc, y, x)
        int temp = idx;
        int x = temp % W_out; temp /= W_out;
        int y = temp % H_out; temp /= H_out;
        int oc = temp % out_channels; temp /= out_channels;
        int n = temp;

        float sum = 0.0f;
        // For each input channel and kernel element:
        for (int ic = 0; ic < in_channels; ic++) {
            for (int ky = 0; ky < kH; ky++) {
                for (int kx = 0; kx < kW; kx++) {
                    int in_y = y * stride - padding + ky;
                    int in_x = x * stride - padding + kx;
                    if (in_y >= 0 && in_y < H_in && in_x >= 0 && in_x < W_in) {
```



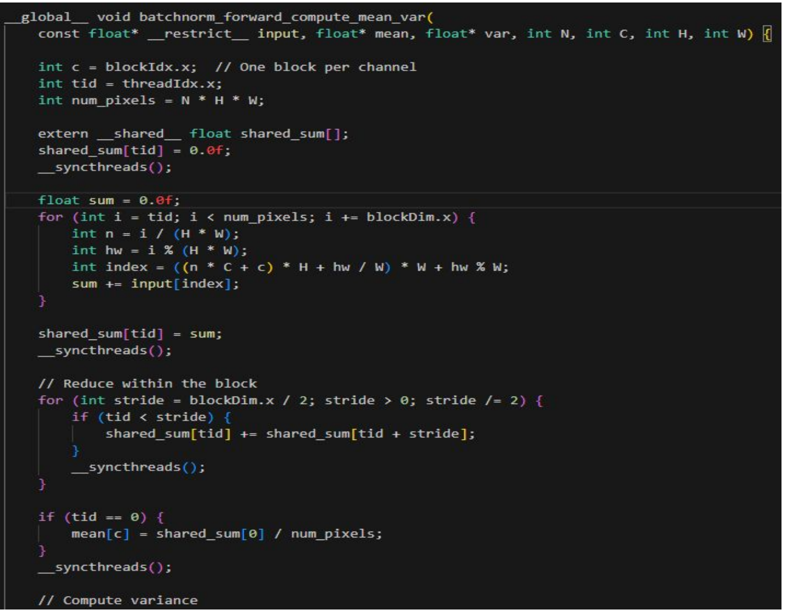
```
__global__ void conv2d_forward_optimized_kernel(
    const float* __restrict__ input, const float* __restrict__ weight, float* output,
    int batch, int in_channels, int out_channels,
    int H_in, int W_in, int H_out, int W_out,
    int kH, int kW, int stride, int padding) {

    extern __shared__ float shared_weight[];

    int oc = blockIdx.x;  // Each block computes an output channel
    int n = blockIdx.y;   // Each block computes a batch sample
    int y = threadIdx.y + blockIdx.z * blockDim.y;
    int x = threadIdx.x + blockIdx.z * blockDim.x;

    if (y >= H_out || x >= W_out) return; // Boundary check

    // Load weights into shared memory (one-time load per block)
    if (threadIdx.y < kH && threadIdx.x < kW) {
        for (int ic = 0; ic < in_channels; ic++) {
            shared_weight[(oc * in_channels + ic) * kH * kW + threadIdx.y * kW + threadIdx.x] =
                weight[(oc * in_channels + ic) * kH * kW + threadIdx.y * kW + threadIdx.x];
        }
    }
    __syncthreads();

    float sum = 0.0f;

    for (int ic = 0; ic < in_channels; ic++) {
        for (int ky = 0; ky < kH; ky++) {
            for (int kx = 0; kx < kW; kx++) {
```

● Inefficient Parrelization Fewer active CUDA cores ● Uncoalesced Memory Access Redundant global memory reads	● One thread per pixel Maximized parallel execution ● Better workload distribution Lower memory traffic & faster compute

Non-Striding Memory



```
__global__ void batchnorm_compute_mean_var(const float* input, float* mean, float* var, int N, int C, int H, int W) {
    int c = blockIdx.x; // one block per channel
    int channel_size = N * H * W;
    __shared__ float shared_sum[256];
    float sum = 0.0f;
    for (int i = threadIdx.x; i < channel_size; i += blockDim.x) {
        int n = i / (H * W);
        int rem = i % (H * W);
        int h = rem / W;
        int w = rem % W;
        int idx = ((n * C + c) * H + h) * W + w;
        sum += input[idx];
    }
    shared_sum[threadIdx.x] = sum;
    __syncthreads();
    // Reduce within block.
    for (int stride = blockDim.x / 2; stride > 0; stride /= 2) {
        if (threadIdx.x < stride)
            shared_sum[threadIdx.x] += shared_sum[threadIdx.x + stride];
        __syncthreads();
    }
    if (threadIdx.x == 0)
        mean[c] = shared_sum[0] / channel_size;
    __syncthreads();

    __shared__ float shared_sum2[256];
    float sum_sq = 0.0f;
    float m = mean[c];
    for (int i = threadIdx.x; i < channel_size; i += blockDim.x) {
        int n = i / (H * W);
```



```
__global__ void batchnorm_forward_compute_mean_var(
    const float* __restrict__ input, float* mean, float* var, int N, int C, int H, int W) {

    int c = blockIdx.x;  // One block per channel
    int tid = threadIdx.x;
    int num_pixels = N * H * W;

    extern __shared__ float shared_sum[];
    shared_sum[tid] = 0.0f;
    __syncthreads();

    float sum = 0.0f;
    for (int i = tid; i < num_pixels; i += blockDim.x) {
        int n = i / (H * W);
        int hw = i % (H * W);
        int index = ((n * C + c) * H + hw / W) * W + hw % W;
        sum += input[index];
    }

    shared_sum[tid] = sum;
    __syncthreads();

    // Reduce within the block
    for (int stride = blockDim.x / 2; stride > 0; stride /= 2) {
        if (tid < stride) {
            shared_sum[tid] += shared_sum[tid + stride];
        }
        __syncthreads();
    }

    if (tid == 0) {
        mean[c] = shared_sum[0] / num_pixels;
    }
    __syncthreads();

    // Compute variance
```

- Strided Memory Access
  Inefficient global memory transactions
- Poor memory coalescing
  Threads access scattered location

- Sequential Memory Access
  Higher global memory throughout
- Warp-level Optimization
  Reduced latency & better coalescing

Results

Benchmarks

Benchmark	Old CUDA (Forward)	Optimized CUDA (Forward)	Speedup (Forward)	Old CUDA (Backward)	Optimized CUDA (Backward)	Speedup (Backward)
Conv2d	0.000568 sec	0.000178 sec	3.19x	0.001791 sec	0.000413 sec	4.33x
ConvTranspose2d	0.000188 sec	0.0001786 sec	1.07x	0.000415 sec	0.000368 sec	1.13x
BatchNorm2d	0.001209 sec	0.000479 sec	2.53x	0.001335 sec	0.000710 sec	1.88x

CUDA Results - EMNIST

Test Name	Processing Time	Speed
EMNIST CPU	1611.83 sec	N/A
EMNIST GPU FP32	182.20 sec	8.85x
EMNIST GPU FP16	85.10 sec	18.94x
EMNIST GPU with CUDA	54.94 sec	29.34x
EMNIST with CUDA Optimized	46.95 sec	34.33x

CUDA Results - Celeb-A

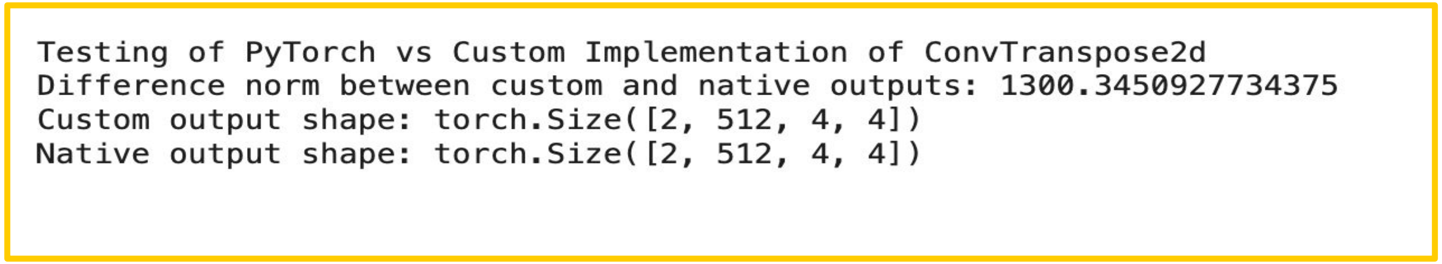
Test Name	Processing Time	Speed
Celeb-A CPU	1523.73 seconds.	N/A
Celeb-A GPU FP32	179.67 seconds.	8.48x
Celeb-A GPU FP16	91.70 seconds.	16.62x
Celeb-A GPU with CUDA	113.70 seconds	13.4x
Celeb-A with CUDA Optimized	110.69 seconds	13.76x

Future Work

Future Work

- **Optimizing our custom CUDA Kernels for PyTorch-Level Performance**: Our current implementation has **high loss and lower accuracy** compared to PyTorch's optimized kernels

- **Implement Kernel Fusion:** Combine multiple operations into a single kernel to reduce memory access overhead and improve computational efficiency.



```
Testing of PyTorch vs Custom Implementation of ConvTranspose2d
Difference norm between custom and native outputs: 1300.3450927734375
Custom output shape: torch.Size([2, 512, 4, 4])
Native output shape: torch.Size([2, 512, 4, 4])
```

Thank you!

Sources

- Radford, Alec, Luke Metz, and Soumith Chintala. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks." arXiv, 16 Nov. 2015, arxiv.org/abs/1511.06434.
- Goodfellow, Ian J., et al. "Generative Adversarial Nets." arXiv, 10 June 2014, arxiv.org/abs/1406.2661.
- Dhara, Rimika. "CUDA 3: Your Checklist for Optimizing CUDA Kernels." Medium, 17 Apr. 2020, medium.com/@rimikadhara/cuda-3-your-checklist-for-optimizing-cuda-kernels-68ef2a42332d
- Crawford, Bryce. EMNIST: Extended MNIST. Kaggle, 2017, www.kaggle.com/datasets/crawford/emnist
- Li, Jessica. CelebA Dataset. Kaggle, 2020, www.kaggle.com/datasets/jessicali9530/celeba-dataset