Introduction to GPU programming for deep learning applications

Instructor: Xin Liang (xliang@uky.edu)

Workshop on Deep Learning Systems in Advanced GPU Cyberinfrastructure

Some of the materials are from Internet. Copyrights may be reserved by NVIDIA, Paweł Luniak etc.

Agenda

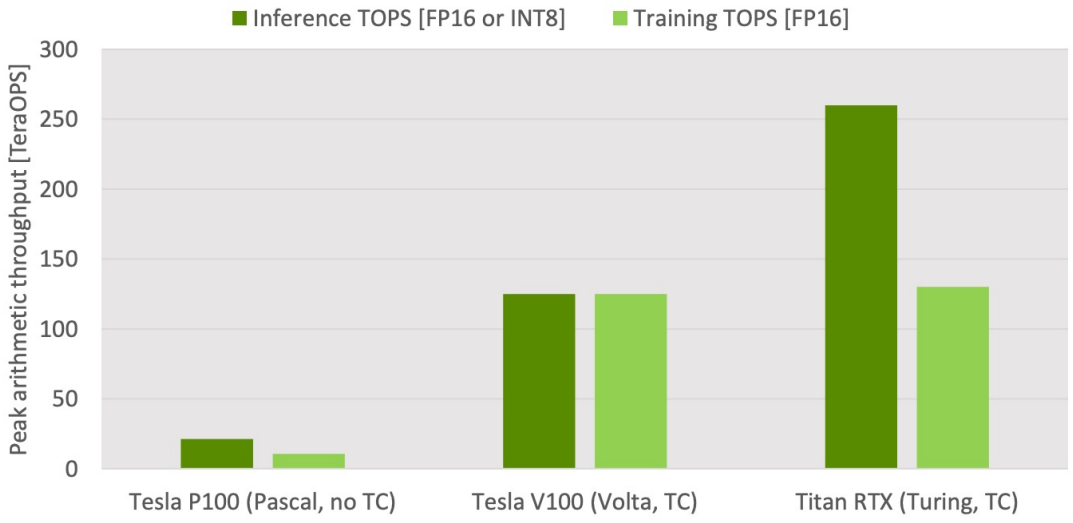
- Optimization of CUDA code
- Introduction to Tensor Core architecture and programming
- Implementing MLP using CUDA
- Advanced optimizations for DL applications with GPUs

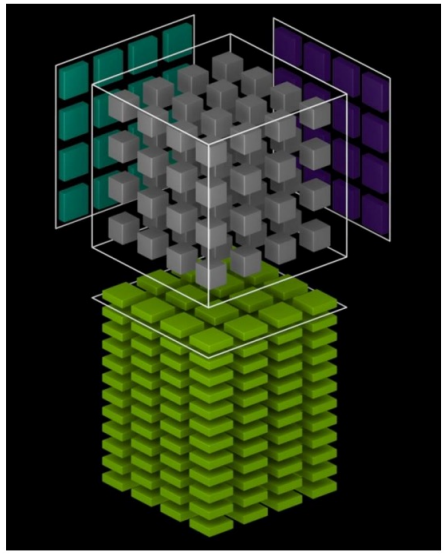
Introduction to Tensor Core Architecture and Programming

Example and materials from NVIDIA tutorials and examples
https://www.olcf.ornl.gov/wp-content/uploads/2019/11/ORNL_Tensor_Core_Training_Aug2019.pdf

What is tensor core?

Tensor Cores enable mixed-precision computing, dynamically adapting calculations to accelerate throughput while preserving accuracy. The latest generation of Tensor Cores are faster than ever on a broader array of AI and high-performance computing (HPC) tasks. From 6X speedups in transformer network training to 3X boosts in performance across all applications, NVIDIA Tensor Cores deliver new capabilities to all workloads. [1]





[1] https://www.nvidia.com/en-us/data-center/tensor-cores/.

V100 Architecture

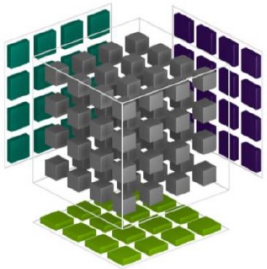
VOLTA GV100 SM

	GV100
FP32 units	64
FP64 units	32
INT32 units	64
Tensor Cores	8
Register File	256 KB
Unified L1/Shared memory	128 KB
Active Threads	2048

Tensor core

TENSOR CORE

Mixed Precision Matrix Math
4x4 matrices



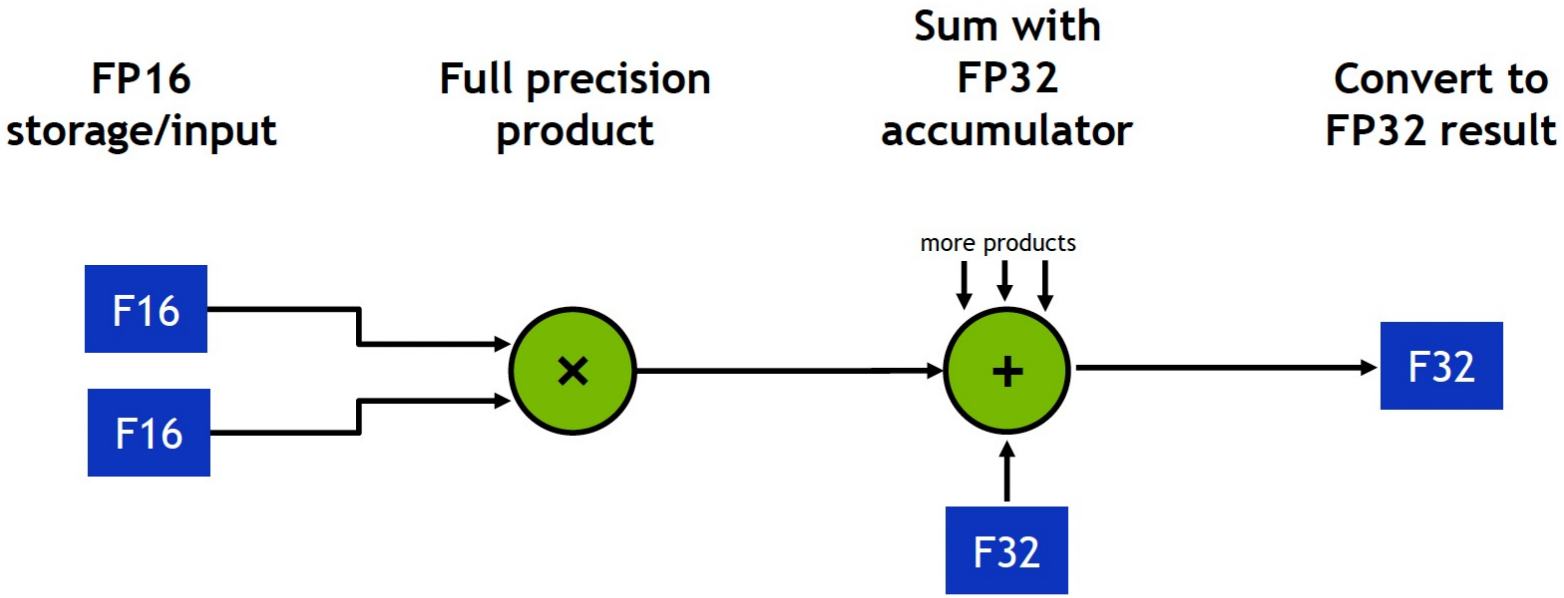
$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 FP16 FP16 FP16 or FP32

$$D = AB + C$$

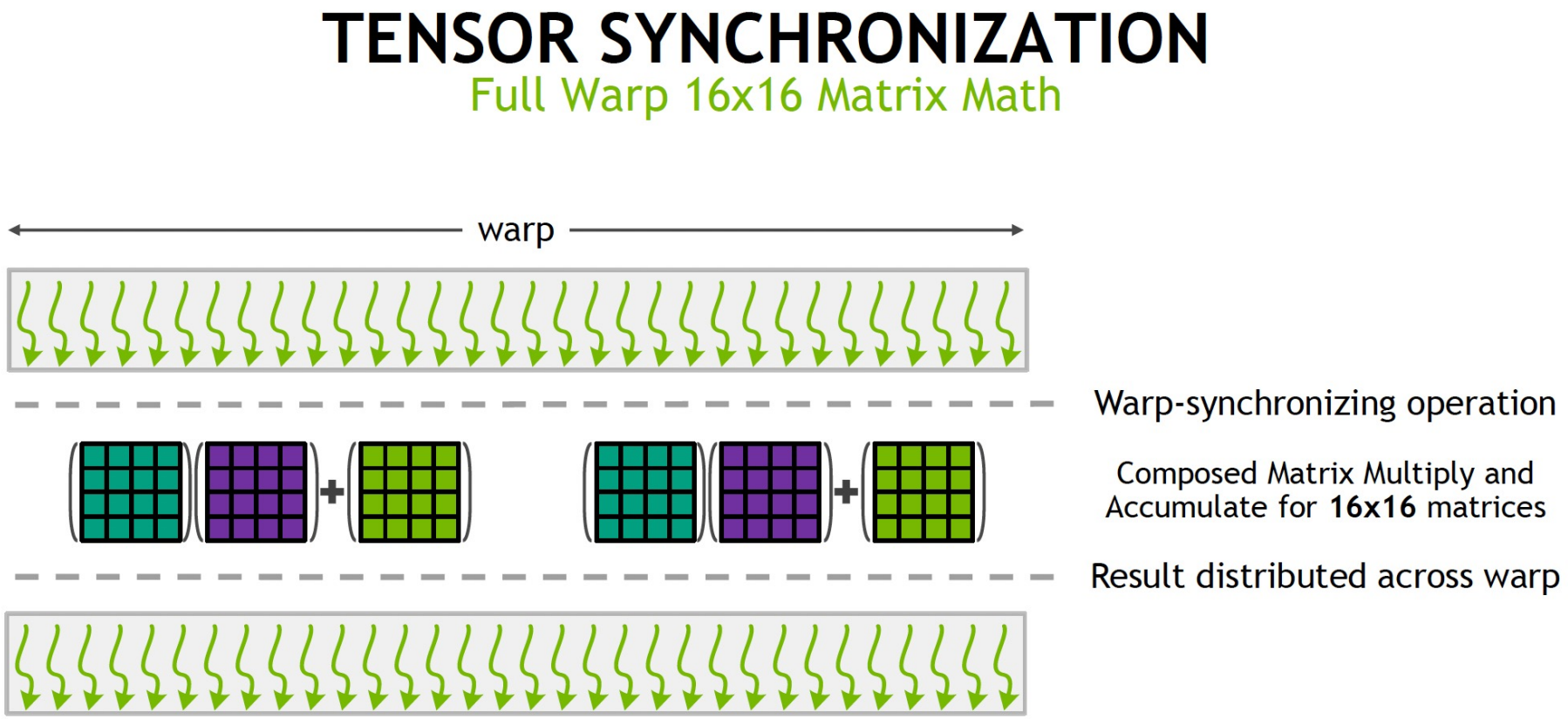
Tensor operation

VOLTA TENSOR OPERATION



Also supports FP16 accumulator mode for inferencing

Tensor synchronization

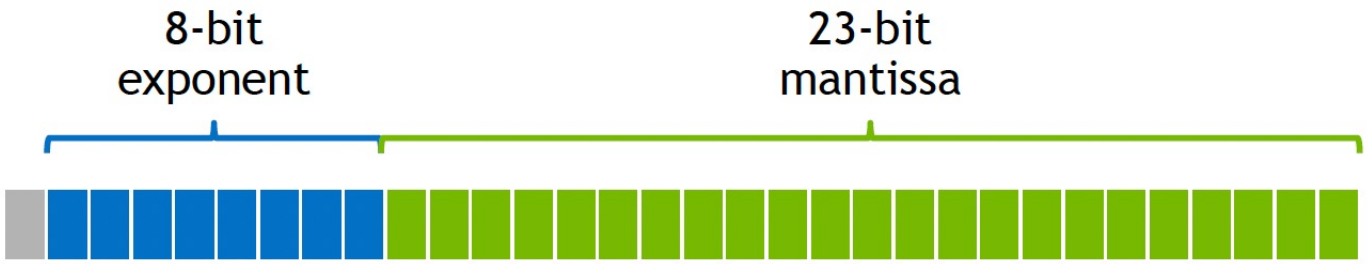


Data representation

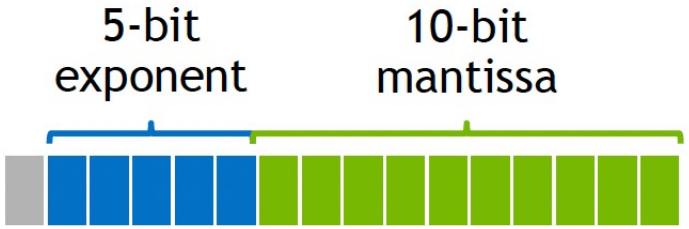
FP32 AND FP16 REPRESENTATION

Dynamic Range

1.4 x 10-45 < x < 3.4 x 1038



5.96 x 10-8 < x < 65504



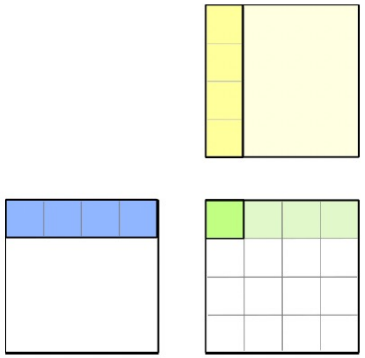
GEMM

Basic definition

General matrix product

$$\boldsymbol{C} = \alpha\ \mathrm{op}(\boldsymbol{A}) * \mathrm{op}(\boldsymbol{B}) + \beta\ \boldsymbol{C}$$

$\boldsymbol{C}$ is M-by-N, op($\boldsymbol{A}$) is M-by-K, op($\boldsymbol{B}$) is K-by-N



Compute independent dot products

```
// Independent dot products
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
       for (int k = 0; k < K; ++k)
           C[i][j] += A[i][k] * B[k][j];
```

Inefficient due to large working sets to hold parts of *A* and *B*

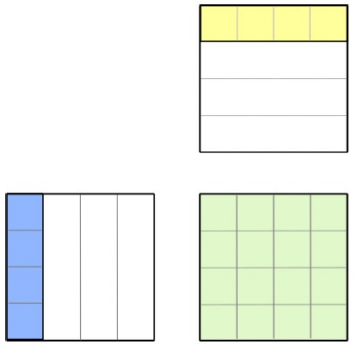
GEMM

Accumulated outer products

General matrix product

$$\boldsymbol{C} = \alpha\ \mathrm{op}(\boldsymbol{A}) * \mathrm{op}(\boldsymbol{B}) + \beta\ \boldsymbol{C}$$

$\boldsymbol{C}$ is M-by-N, $\mathrm{op}(\boldsymbol{A})$ is M-by-K, $\mathrm{op}(\boldsymbol{B})$ is K-by-N



~~Compute independent dot products~~

```
// Independent dot products
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        for (int k = 0; k < K; ++k)
            C[i][j] += A[i][k] * B[k][j];
```

Permute loop nests

```
// Accumulated outer products
for (int k = 0; k < K; ++k)
    for (int i = 0; i < M; ++i)
        for (int j = 0; j < N; ++j)
            C[i][j] += A[i][k] * B[k][j];
```

Load elements of ***A*** and ***B*** exactly once

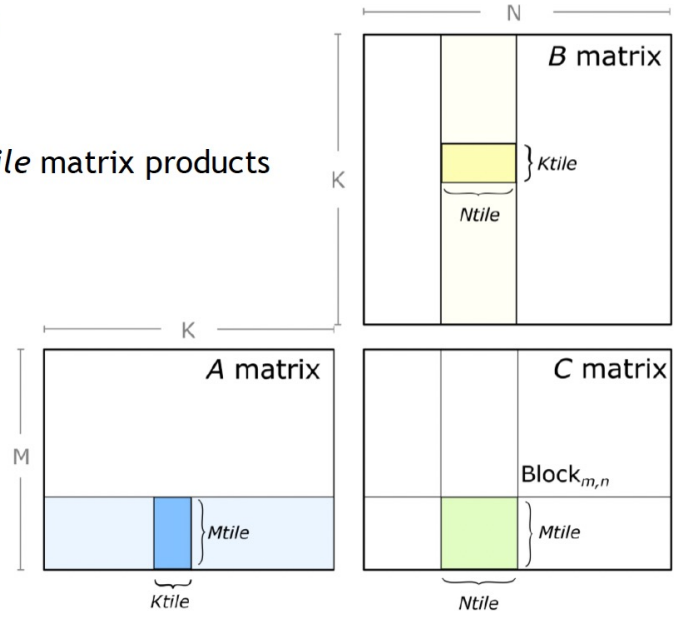
GEMM

Computing matrix product one block at a time

Partition the loop nest into *blocks* along each dimension

- Partition into *Mtile*-by-*Ntile* independent matrix products
- Compute each product by accumulating *Mtile*-by-*Ntile*-by-*Ktile* matrix products

```
for (int mb = 0; mb < M; mb += Mtile)
    for (int nb = 0; nb < N; nb += Ntile)
        for (int kb = 0; kb < K; kb += Ktile)
        {
            // compute Mtile-by-Ntile-by-Ktile matrix product
            for (int k = 0; k < Ktile; ++k)
                for (int i = 0; i < Mtile; ++i)
                    for (int j = 0; j < Ntile; ++j)
                    {
                        int row = mb + i;
                        int col = nb + j;

                        C[row][col] +=
                            A[row][kb + k] * B[kb + k][col];
                    }
        }
```



Block GEMM in CUDA

Parallelism Among CUDA Thread Blocks

Launch a CUDA kernel grid

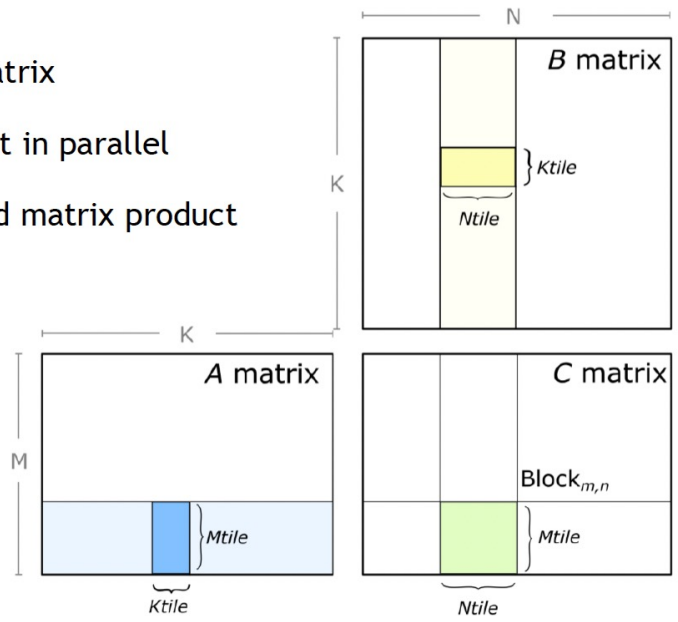
- Assign CUDA thread blocks to each partition of the output matrix

CUDA thread blocks compute *Mtile*-by-*Ntile*-by-*K* matrix product in parallel

- Iterate over *K* dimension in steps, performing an accumulated matrix product

```
for (int mb = 0; mb < M; mb += Mtile)
    for (int nb = 0; nb < N; nb += Ntile)
        for (int kb = 0; kb < K; kb += Ktile)
        {
            .. compute Mtile by Ntile by Ktile GEMM
        }
```

by each CUDA thread block



Thread block tile structure

Parallelism Within a CUDA Thread Block

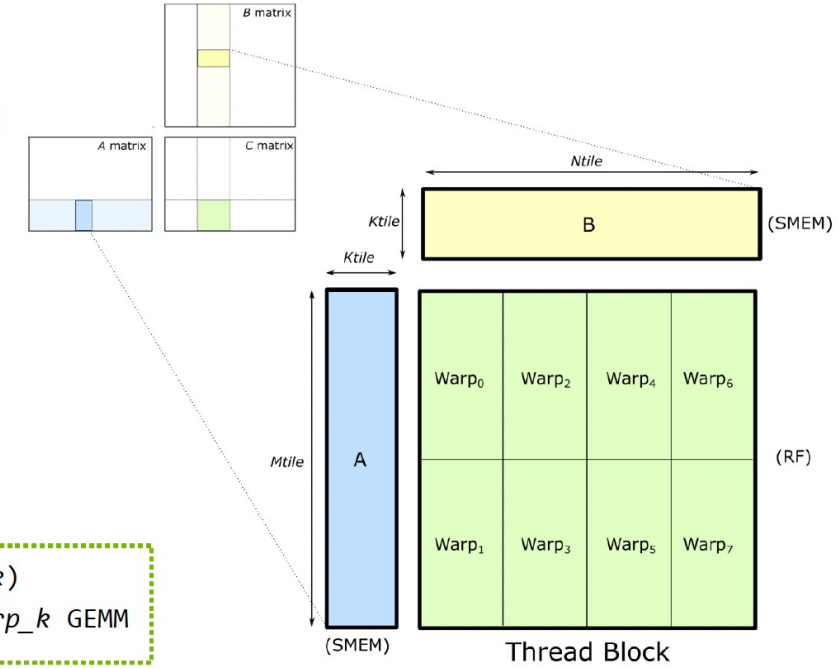
Decompose thread block into warp-level tiles

- Load ***A*** and ***B*** operands into Shared Memory (reuse)
- ***C*** matrix distributed among warps

Each warp computes an independent matrix product

```
for (int kb = 0; kb < K; kb += Ktile)
{
    .. load A and B tiles to shared memory

    for (int m = 0; m < Mtile; m += warp_m)
        for (int n = 0; n < Ntile; n += warp_n)
            for (int k = 0; k < Ktile; k += warp_k)
                .. compute warp_m by warp_n by warp_k GEMM
}
```

by each CUDA warp



Warp-level tile structure

Warp-level matrix product

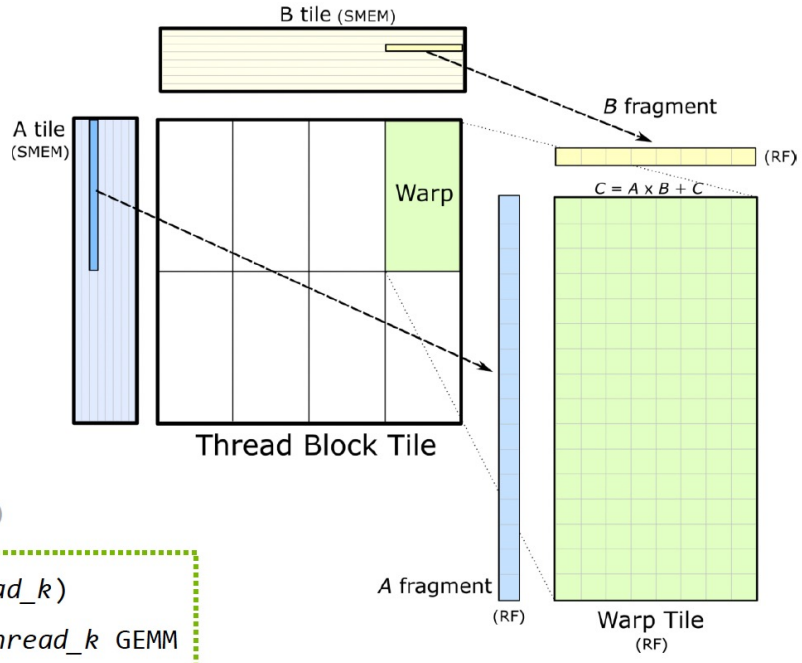
Warps perform an accumulated matrix product

- Load ***A*** and ***B*** operands from SMEM into registers
- ***C*** matrix held in registers of participating threads

Shared Memory layout is *K*-strided for efficient loads

```
for (int k = 0; k < Ktile; k += warp_k)
{
    .. load A tile from SMEM into registers
    .. load B tile from SMEM into registers

    for (int tm = 0; tm < warp_m; tm += thread_m)
        for (int tn = 0; tn < warp_n; tn += thread_n)

            for (int tk = 0; tk < warp_k; tk += thread_k)
                .. compute thread_m by thread_n by thread_k GEMM
}
```

by each CUDA thread



Thread-level tile structure

Parallelism within a thread

Threads compute accumulated matrix product

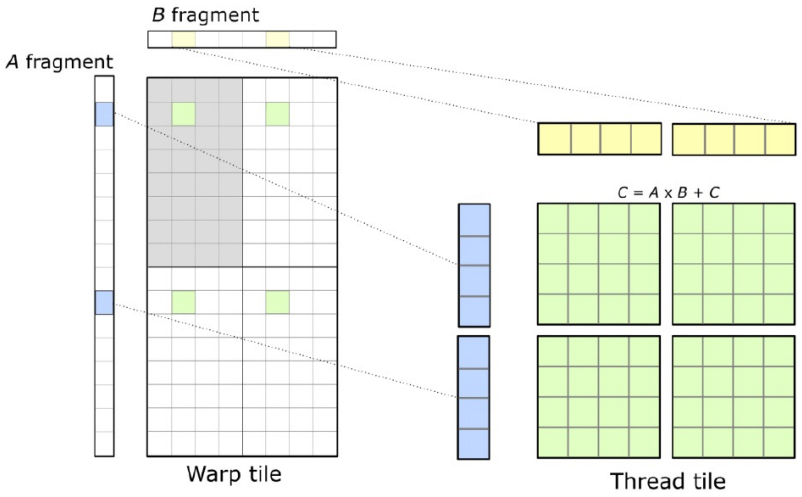
- ***A***, ***B***, and ***C*** held in registers

Opportunity for data reuse:

- *O(M*N)* computations on *O(M+N)* elements

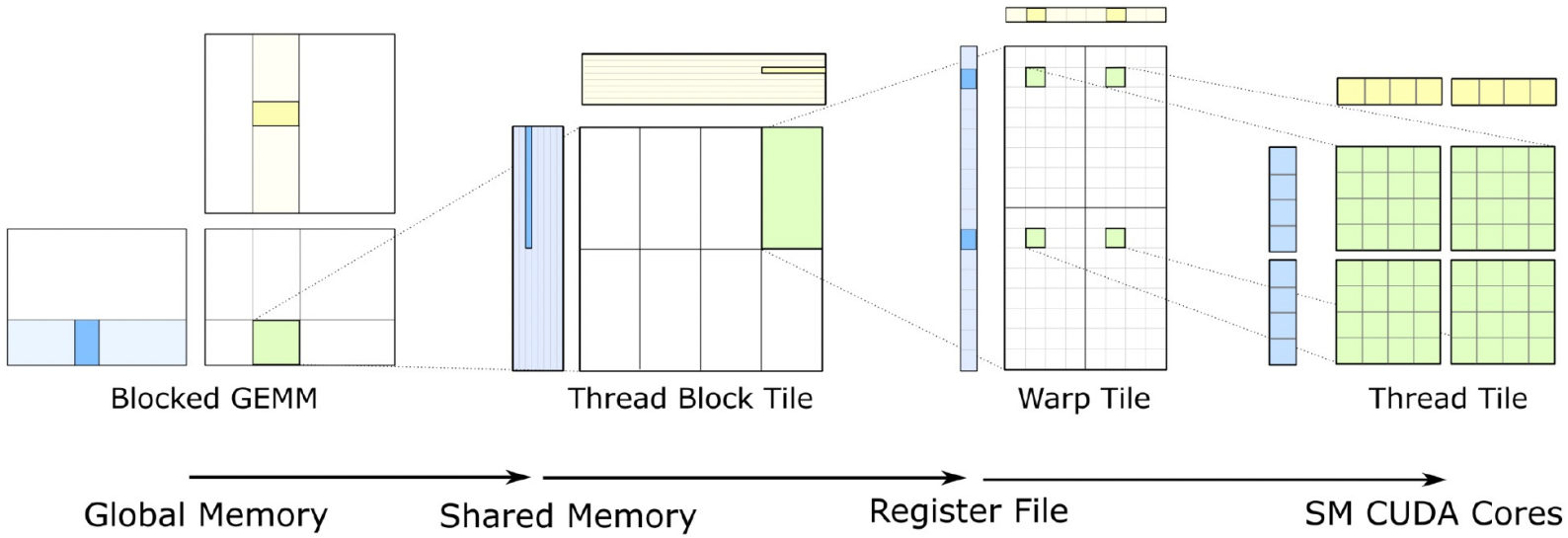
```
for (int m = 0; m < thread_m; ++m)
    for (int n = 0; n < thread_n; ++n)
        for (int k = 0; k < thread_k; ++k)
            C[m][n] += A[m][k] * B[n][k];
```

Fused multiply-accumulate instructions



Complete GEMM hierarchy

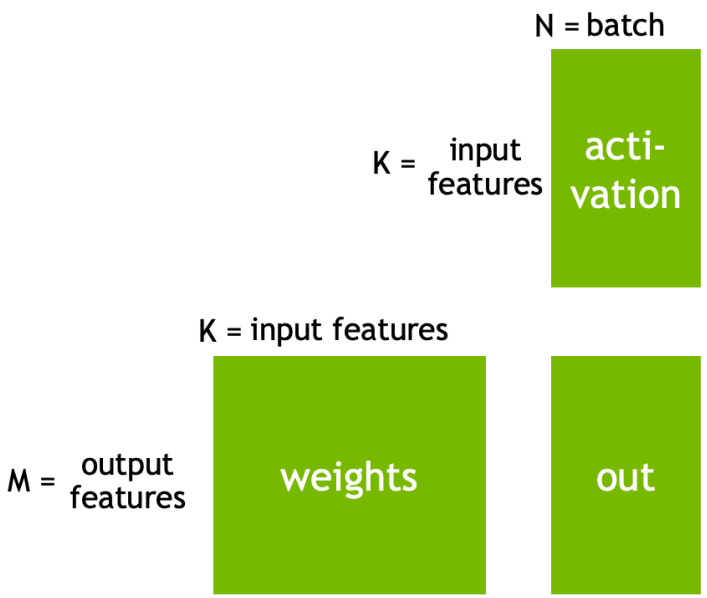
Data reuse at each level of the memory hierarchy



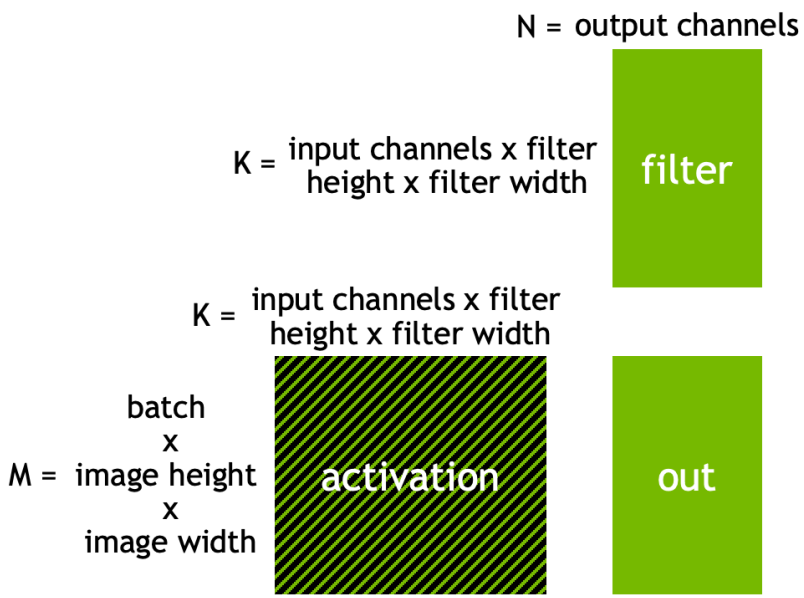
Deep Learning and GEMM

DNN OPERATION MAPPING TO GEMM

Forward pass mappings



Fully Connected / Dense / Linear
(PyTorch nn.Linear, TensorFlow swaps A and B)



Convolution
(*implicit* GEMM algorithm,
matrices are never actually created)

Deep Learning and GEMM

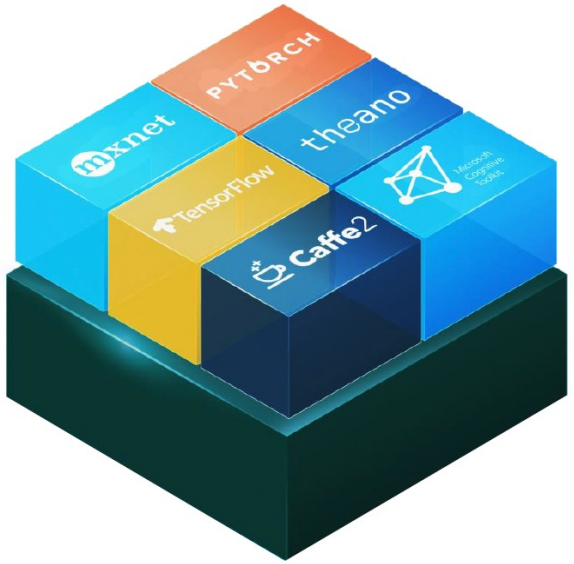
DNN OPERATION MAPPING TO GEMM

All pass mappings

Operation	Phase	GEMM "M"	GEMM "N"	GEMM "K
FC/Linear	Forward	Output features	Batch size	Input features
	Data grad	Input features	Batch size	Output features
	Weight grad	Input features	Output features	Batch size
Conv	Forward	Batch x iHeight x iWidth	Output channels	Input channels x fHeight x fWidth
	Data grad	Batch x iHeight x iWidth	Input channels	Output channels x fHeight x fWidth
	Weight grad	Input channels x fHeight x fWidth	Output channels	Batch x iHeight x iWidth

Tensor core programming

USING TENSOR CORES



NVIDIA cuDNN, cuBLAS, TensorRT

Volta Optimized Frameworks and Libraries

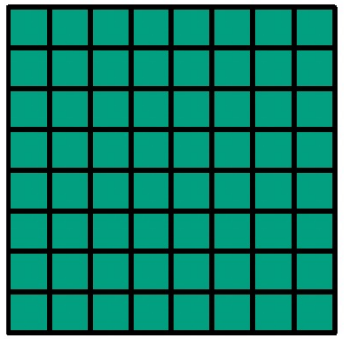
```
__device__ void tensor_op_16_16_16(
    float *d, half *a, half *b, float *c)
{
  wmma::fragment<matrix_a, …> Amat;
  wmma::fragment<matrix_b, …> Bmat;
  wmma::fragment<matrix_c, …> Cmat;

  wmma::load_matrix_sync(Amat, a, 16);
  wmma::load_matrix_sync(Bmat, b, 16);
  wmma::fill_fragment(Cmat, 0.0f);

  wmma::mma_sync(Cmat, Amat, Bmat, Cmat);

  wmma::store_matrix_sync(d, Cmat, 16,
                          wmma::row_major);
}
```

CUDA C++

Warp-Level Matrix Operations

Tensor core programming

WMMA datatypes



Per-Thread fragments to hold components of matrices for use with Tensor Cores

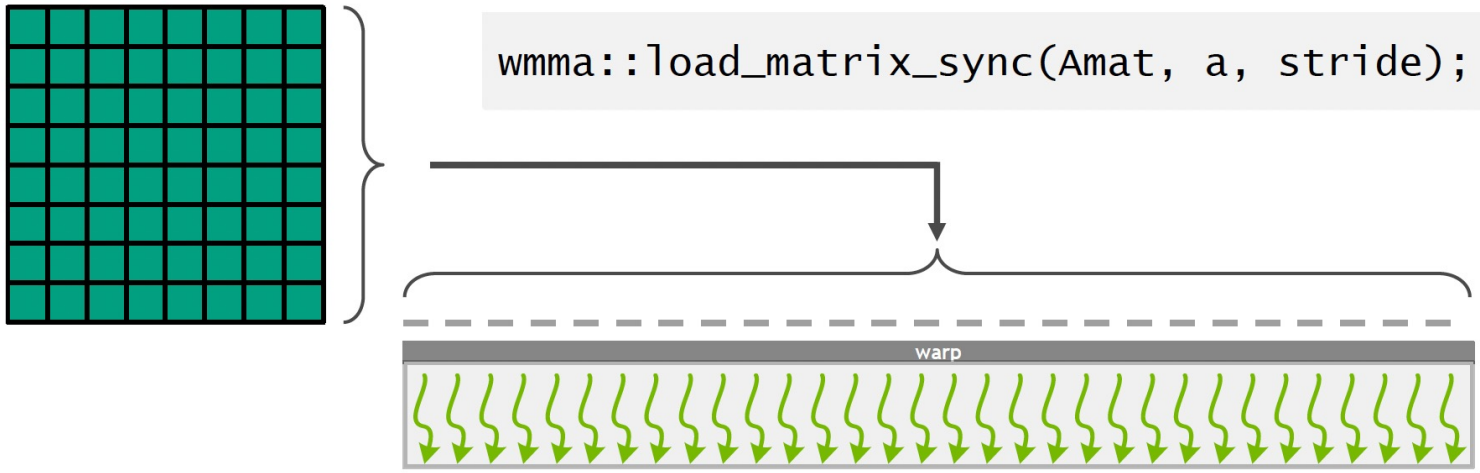
```
wmma::fragment<matrix_a, …> Amat;
```

Tensor core programming

WMMA load and store operations

Warp-level operation to fetch components of matrices into fragments

```
wmma::load_matrix_sync(Amat, a, stride);
```

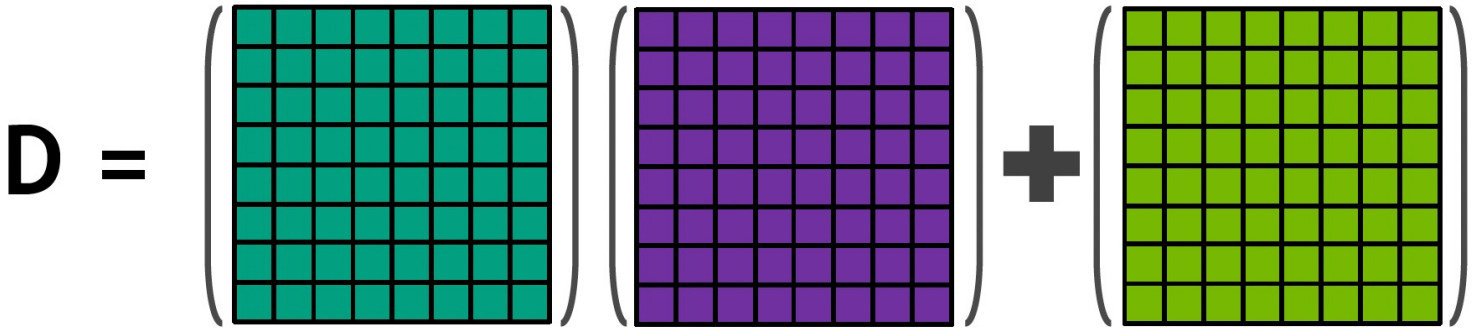


Tensor core programming

WMMA Matrix Multiply and Accumulate Operation

Warp-level operation to perform matrix multiply and accumulate

```
wmma::mma_sync(Dmat, Amat, Bmat, Cmat);
```

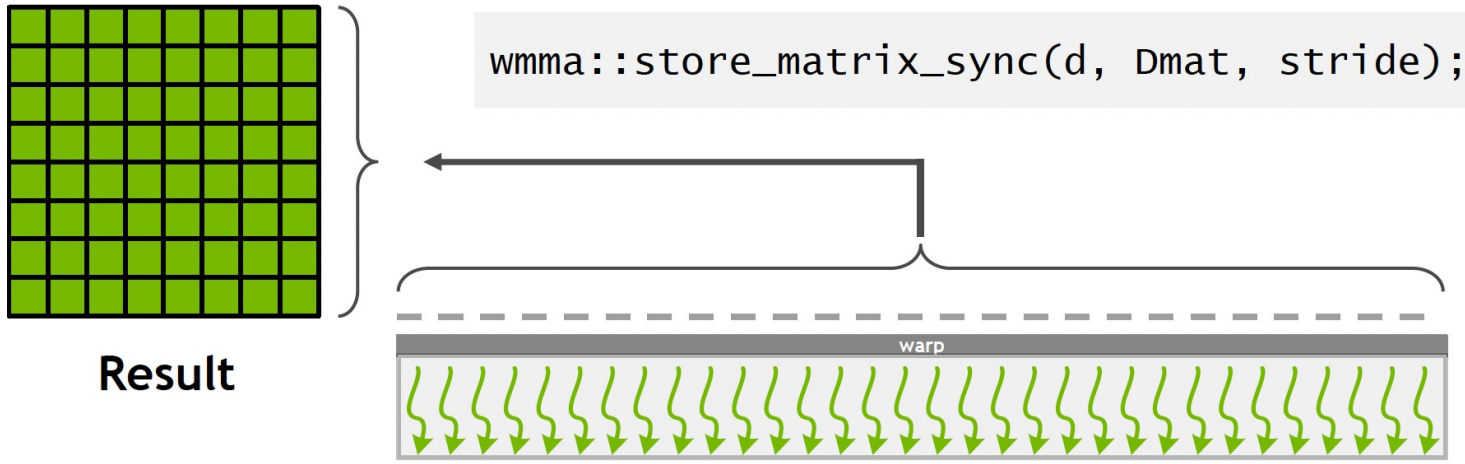


Tensor core programming

WMMA load and store operations

Warp-level operation to fetch components of matrices into fragments

```
wmma::store_matrix_sync(d, Dmat, stride);
```



Tensor core programming

TENSOR CORE EXAMPLE

Create Fragments

Initialize Fragments

Perform MatMul

Store Results

```
__device__ void tensor_op_16_16_16(
    float *d, half *a, half *b, float *c)
{
  wmma::fragment<matrix_a, …> Amat;
  wmma::fragment<matrix_b, …> Bmat;
  wmma::fragment<matrix_c, …> Cmat;

  wmma::load_matrix_sync(Amat, a, 16);
  wmma::load_matrix_sync(Bmat, b, 16);
  wmma::fill_fragment(Cmat, 0.0f);

  wmma::mma_sync(Cmat, Amat, Bmat, Cmat);

  wmma::store_matrix_sync(d, Cmat, 16,
                          wmma::row_major);
}
```

CUDA C++

Warp-Level Matrix Operations

How can we generalize it to GEMM with arbitrary size?

Hands-on exercise

GEMM:
Implement an efficient algorithm for GEMM using CUDA cores and Tensor cores.

Note: Assuming A is **row-major** and B is **column-major**.

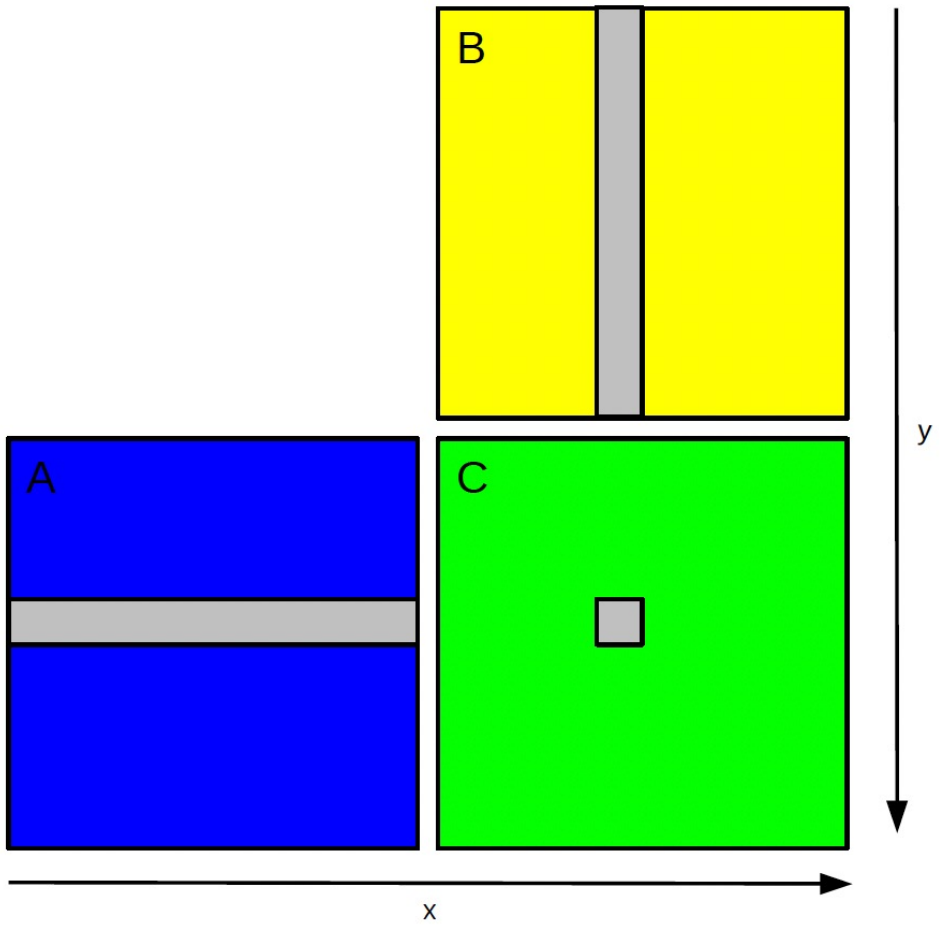
Naïve implementation (CPU)

```
__host__ void matMultiplyOnHost(half *A, half *B, float *C, float alpha,
                                float beta, int numARows, int numAColumns,
                                int numBRows, int numBColumns, int numCRows,
                                int numCColumns) {
  for (int i = 0; i < numCRows; i++) {
    for (int j = 0; j < numCColumns; j++) {
      float temp = 0.0;

      for (int k = 0; k < numAColumns; k++) {
        temp += (float)A[i * numAColumns + k] * (float)B[j * numBRows + k];
      }

      C[i * numCColumns + j] = temp * alpha + beta * C[i * numCColumns + j];
    }
  }
}
```

Naïve implementation (GPU)

```
__global__ void simple_gemm(half *a, half *b, float *c, int m_ld,
                            int n_ld, int k_ld, float alpha, float beta) {

    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    if (row < m_ld && col < n_ld) {
        float temp = 0.0;
        for (int i = 0; i < k_ld; ++i) {
            temp += (float)a[row * n_ld + i] * (float)b[col * n_ld + i];
        }
        c[row * n_ld + col] = temp * alpha + beta * c[row * n_ld + col];
    }
}
```

Tiled GPU implementation with shared memory



- Each thread computes one element of the result matrix C
- n * n threads will be needed
- Indexing of threads corresponds to 2d indexing of the matrices
- Thread(x, y) will calculate element C(x, y) using row y of A and column x of B

Tiled GPU implementation with shared memory

```
__global__ void simple_smm_gemm(half *a, half *b, float *c, int m_ld,
                                int n_ld, int k_ld, float alpha, float beta) {

    __shared__ float A[TILE_WIDTH][TILE_WIDTH];
    __shared__ float B[TILE_WIDTH][TILE_WIDTH];
    int tx = threadIdx.x, ty = threadIdx.y;
    int col = blockIdx.x * blockDim.x + tx;
    int row = blockIdx.y * blockDim.y + ty;

    if (row < m_ld && col < n_ld) {
        float temp = 0.0;
        for (int i = 0; i < k_ld; i += TILE_WIDTH){
            int aRow = row, aCol = i + tx;
            int bRow = col, bCol = i + ty;
            // collaboratively load a and b
            if (aCol < k_ld)
                A[ty][tx] = a[aRow*k_ld + aCol];
            else
                A[ty][tx] = 0;
            if (bCol < k_ld)
                B[ty][tx] = b[bRow*k_ld + bCol];
            else
                B[ty][tx] = 0;
            __syncthreads();

            for (int k = 0; k < TILE_WIDTH; ++k)
                temp += A[ty][k] * B[k][tx];
            __syncthreads();
        }
        c[row * n_ld + col] = temp * alpha + beta * c[row * n_ld + col];
    }
}
```

Tensor core implementation

```
__global__ void simple_wmma_gemm(half *a, half *b, float *c, float *d, int m_ld,
                                 int n_ld, int k_ld, float alpha, float beta) {
  // Leading dimensions. Packed with no transpositions.
  int lda = k_ld;
  int ldb = k_ld;
  int ldc = n_ld;

  // Tile using a 2D grid
  int warpM = (blockIdx.x * blockDim.x + threadIdx.x) / warpSize;
  int warpN = (blockIdx.y * blockDim.y + threadIdx.y);

  // Declare the fragments
  wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::row_major>
      a_frag;
  wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major>
      b_frag;
  wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
  wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;

  wmma::fill_fragment(acc_frag, 0.0f);
```

Tensor core implementation

```
// Loop over k
for (int i = 0; i < k_ld; i += WMMA_K) {
  int aCol = i;
  int aRow = warpM * WMMA_M;
  int bCol = warpN * N;
  int bRow = i;

  // Bounds checking
  if (aRow < m_ld && aCol < k_ld && bRow < k_ld && bCol < n_ld) {
    // Load the inputs
    wmma::load_matrix_sync(a_frag, a + aCol + aRow * lda, lda);
    wmma::load_matrix_sync(b_frag, b + bRow + bCol * ldb, ldb);

    // Perform the matrix multiplication
    wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
  }
}
```

Tensor core implementation

```
  // Load in the current value of c, scale it by beta, and add this our result
  // scaled by alpha
  int cCol = warpN * WMMA_N;
  int cRow = warpM * WMMA_M;

  if (cRow < m_ld && cCol < n_ld) {
    wmma::load_matrix_sync(c_frag, c + cCol + cRow * ldc, ldc,
                           wmma::mem_row_major);

    for (int i = 0; i < c_frag.num_elements; i++) {
      c_frag.x[i] = alpha * acc_frag.x[i] + beta * c_frag.x[i];
    }

    // Store the output
    wmma::store_matrix_sync(d + cCol + cRow * ldc, c_frag, ldc,
                            wmma::mem_row_major);
  }
}
```

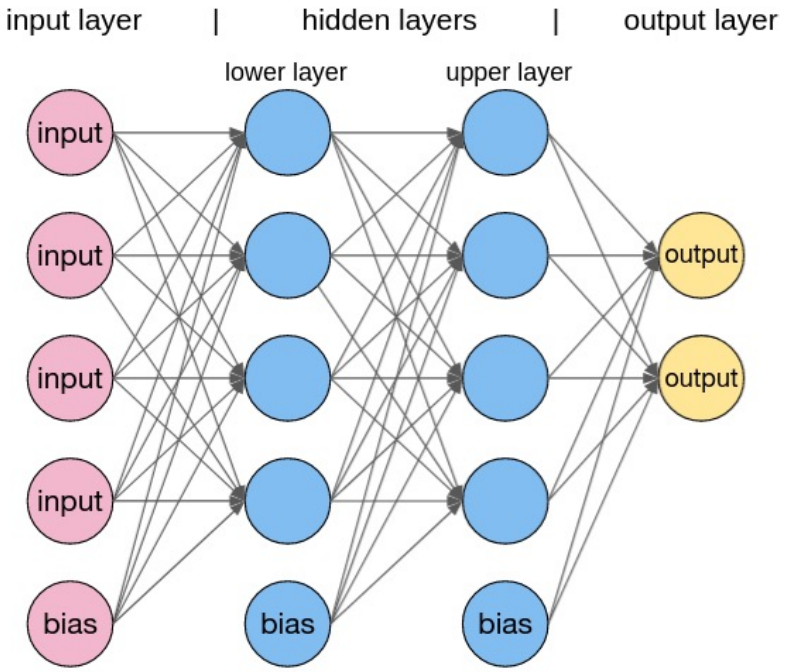
Any questions so far?

Implementing MLP using CUDA

Example and materials from Paweł Luniak
https://luniak.io/cuda-neural-network-implementation-part-1/

MLP

A Multilayer Perceptron (MLP) has input and output layers, and one or more hidden layers with many neurons stacked together. And while in the Perceptron the neuron must have an activation function that imposes a threshold, like ReLU or sigmoid, neurons in a Multilayer Perceptron can use any arbitrary activation function



Key idea: mapping MLP to GEMM operations and use CUDA for the implementation!

Basic data structure

```
class Matrix {
private:
    bool device_allocated;
    bool host_allocated;

    void allocateCudaMemory();
    void allocateHostMemory();

public:
    Shape shape;

    std::shared_ptr<float> data_device;
    std::shared_ptr<float> data_host;

    Matrix(size_t x_dim = 1, size_t y_dim = 1);
    Matrix(Shape shape);

    void allocateMemory();
    void allocateMemoryIfNotAllocated(Shape shape);

    void copyHostToDevice();
    void copyDeviceToHost();

    float& operator[](const int index);
    const float& operator[](const int index) const;
};
```

Matrix: used to represent all the numbers, including both parameters and data

NNLayer: used as abstraction for the layers in the multilayer perceptron.

```
class NNLayer {
protected:
    std::string name;

public:
    virtual ~NNLayer() = 0;

    virtual Matrix& forward(Matrix& A) = 0;
    virtual Matrix& backprop(Matrix& dZ, float learning_rate) = 0;

    std::string getName() { return this->name; };

};
```

Basic data structure

```
class NeuralNetwork {
private:
    std::vector<NNLayer*> layers;
    BCECost bce_cost;

    Matrix Y;
    Matrix dY;
    float learning_rate;

public:
    NeuralNetwork(float learning_rate = 0.01);
    ~NeuralNetwork();

    Matrix forward(Matrix X);
    void backprop(Matrix predictions, Matrix target);

    void addLayer(NNLayer *layer);
    std::vector<NNLayer*> getLayers() const;

};
```

```
Matrix NeuralNetwork::forward(Matrix X) {
    Matrix Z = X;

    for (auto layer : layers) {
        Z = layer->forward(Z);
    }

    Y = Z;
    return Y;
}
```

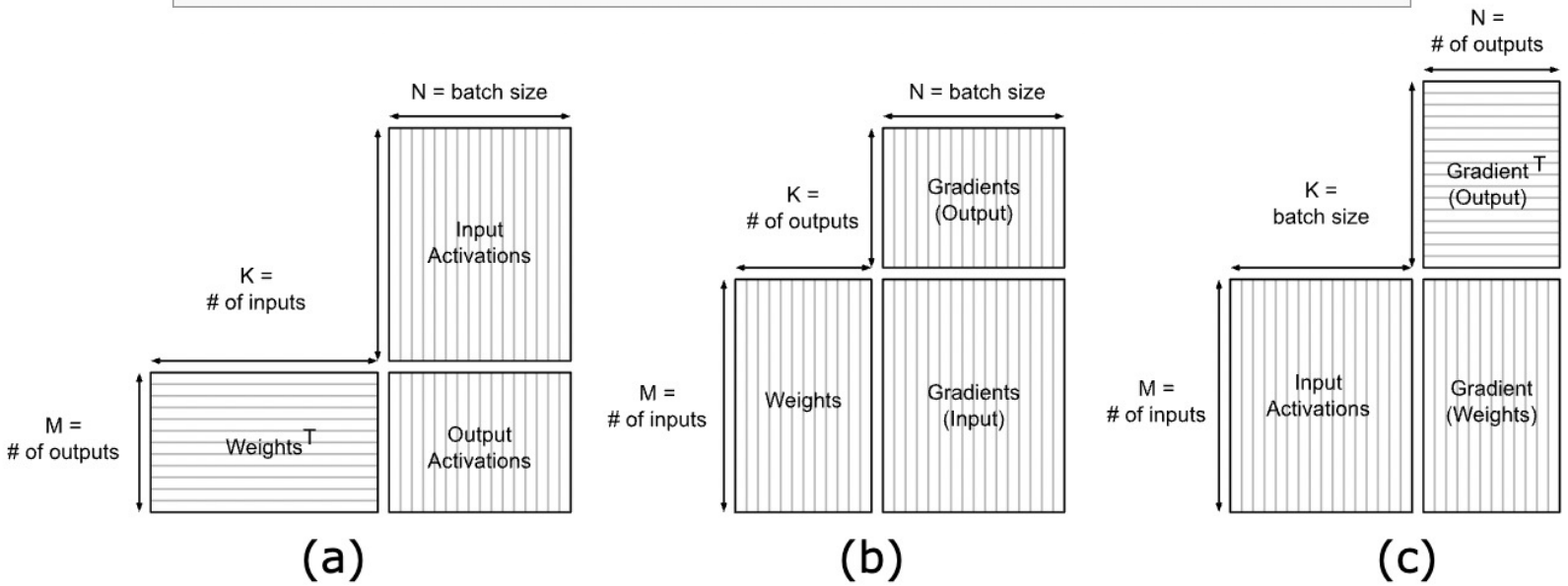
```
void NeuralNetwork::backprop(Matrix predictions, Matrix target) {
    dY.allocateMemoryIfNotAllocated(predictions.shape);
    Matrix error = bce_cost.dCost(predictions, target, dY);

    for (auto it = this->layers.rbegin(); it != this->layers.rend(); it++) {
        error = (*it)->backprop(error, learning_rate);
    }

    cudaDeviceSynchronize();
}
```

NeuralNetwork: abstraction for the MLP which contains multiple layers.

Mapping MLP operations to GEMM

Table 1. Mapping of inputs, outputs, and batch size to GEMM parameters M, N, K.

Computation Phase	M	N	K
Forward Propagation	Number of outputs	Batch size	Number of inputs
Activation Gradient	Number of inputs	Batch size	Number of outputs
Weight Gradient	Number of inputs	Number of outputs	Batch size



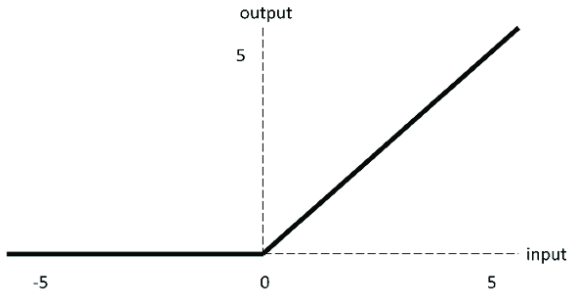
(a) Forward pass

(b) Activation gradient

(c) Weight gradient

ReLU layers

```
__global__ void reluActivationForward(float* Z, float* A,
                                      int Z_x_dim, int Z_y_dim) {

    int index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < Z_x_dim * Z_y_dim) {
        A[index] = fmaxf(Z[index], 0);
    }
}
__global__ void reluActivationBackprop(float* Z, float* dA, float* dZ,
                                       int Z_x_dim, int Z_y_dim) {

    int index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < Z_x_dim * Z_y_dim) {
        if (Z[index] > 0) {
            dZ[index] = dA[index];
        }
        else {
            dZ[index] = 0;
        }
    }
}
```



Input

-249	-91	-37
250	-134	101
27	61	-153

ReLU

0	0	0
250	0	101
27	61	0

Forward pass

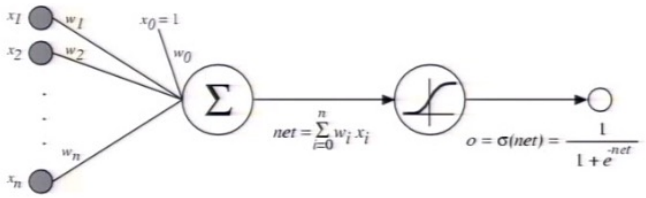
$$ReLU(x) = \begin{cases} x & \text{for } x > 0 \\ 0 & \text{otherwise} \end{cases} = max(0, x)$$

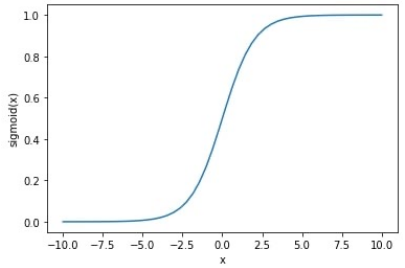
Backward propagation

$$dZ = \frac{dJ}{dReLU(x)} \frac{dReLU(x)}{dx} = \begin{cases} dA & \text{for } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Sigmoid layers

```
__device__ float sigmoid(float x) {
    return 1.0f / (1 + exp(-x));
}

__global__ void sigmoidActivationForward(float* Z, float* A,
                                         int Z_x_dim, int Z_y_dim) {

    int index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < Z_x_dim * Z_y_dim) {
        A[index] = sigmoid(Z[index]);
    }
}

__global__ void sigmoidActivationBackprop(float* Z, float* dA, float* dZ,
                                          int Z_x_dim, int Z_y_dim) {

    int index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < Z_x_dim * Z_y_dim) {
        dZ[index] = dA[index] * sigmoid(Z[index]) * (1 - sigmoid(Z[index]));
    }
}
```





Forward pass

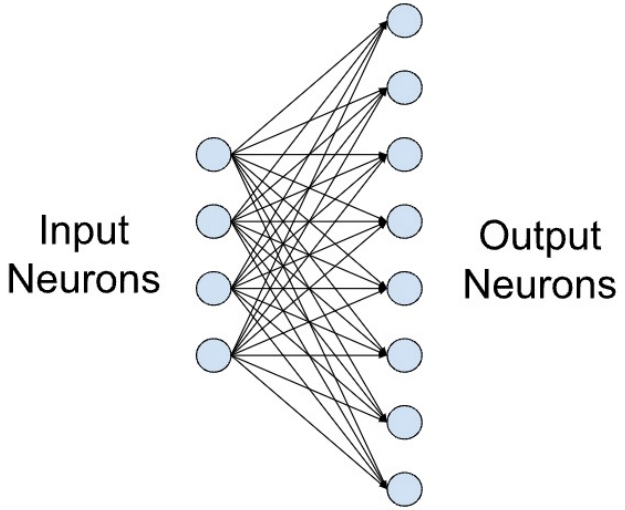
$$\sigma(x) = \frac{e^x}{1 + e^x}$$

Backward propagation

$$dZ = \frac{dJ}{d\sigma(x)} \frac{d\sigma(x)}{dx} = dA \cdot \sigma(x) \cdot \left(1 - \sigma(x)\right)$$

Linear layers

```
__global__ void linearLayerForward( float* W, float* A, float* Z, float* b,
                                    int W_x_dim, int W_y_dim,
                                    int A_x_dim, int A_y_dim) {

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    int Z_x_dim = A_x_dim;
    int Z_y_dim = W_y_dim;

    float Z_value = 0;

    if (row < Z_y_dim && col < Z_x_dim) {
        for (int i = 0; i < W_x_dim; i++) {
            Z_value += W[row * W_x_dim + i] * A[i * A_x_dim + col];
        }
        Z[row * Z_x_dim + col] = Z_value + b[row];
    }
}
```



Forward pass

$$Z = WA + b$$

Backward propagation

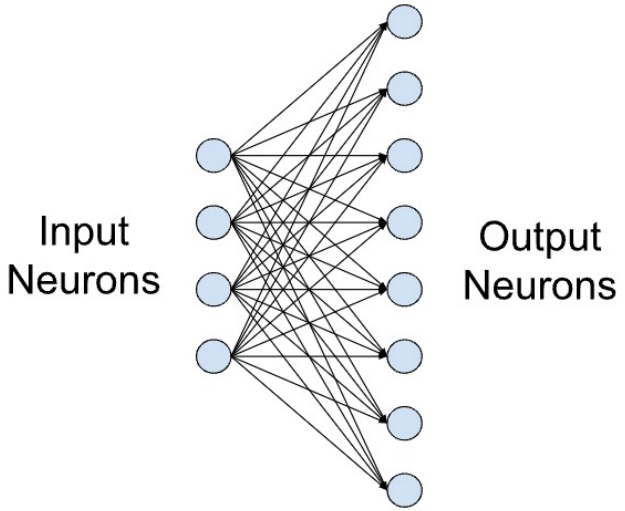
$$dA = \frac{dJ}{dZ}\frac{dZ}{dA} = W^T \cdot dZ$$

$$dW = \frac{dJ}{dZ}\frac{dZ}{dW} = \frac{1}{m} \cdot dZ \cdot A^T$$

$$db = \frac{dJ}{dZ}\frac{dZ}{db} = \frac{1}{m}\sum_{i=0}^{m} dZ^{(i)}$$

Linear layers

```
__global__ void linearLayerBackprop(float* W, float* dZ, float *dA,
                                    int W_x_dim, int W_y_dim,
                                    int dZ_x_dim, int dZ_y_dim) {

    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    // W is treated as transposed
    int dA_x_dim = dZ_x_dim;
    int dA_y_dim = W_x_dim;

    float dA_value = 0.0f;

    if (row < dA_y_dim && col < dA_x_dim) {
        for (int i = 0; i < W_y_dim; i++) {
            dA_value += W[i * W_x_dim + row] * dZ[i * dZ_x_dim + col];
        }
        dA[row * dA_x_dim + col] = dA_value;
    }
}
```



Forward pass

$$Z = WA + b$$

Backward propagation

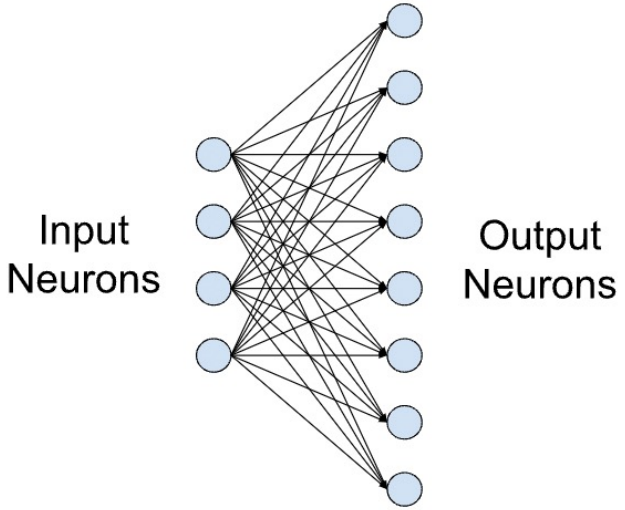
$$dA = \frac{dJ}{dZ}\frac{dZ}{dA} = W^T \cdot dZ$$

$$dW = \frac{dJ}{dZ}\frac{dZ}{dW} = \frac{1}{m} \cdot dZ \cdot A^T$$

$$db = \frac{dJ}{dZ}\frac{dZ}{db} = \frac{1}{m}\sum_{i=0}^{m} dZ^{(i)}$$

Linear layers

```
__global__ void linearLayerUpdateWeights(  float* dZ, float* A, float* W,
                                           int dZ_x_dim, int dZ_y_dim,
                                           int A_x_dim, int A_y_dim,
                                           float learning_rate) {

    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    // A is treated as transposed
    int W_x_dim = A_y_dim;
    int W_y_dim = dZ_y_dim;

    float dW_value = 0.0f;

    if (row < W_y_dim && col < W_x_dim) {
        for (int i = 0; i < dZ_x_dim; i++) {
            dW_value += dZ[row * dZ_x_dim + i] * A[col * A_x_dim + i];
        }
        W[row * W_x_dim + col] = W[row * W_x_dim + col] -
                                 learning_rate * (dW_value / A_x_dim);
    }
}
```



Forward pass

$$Z = WA + b$$

Backward propagation

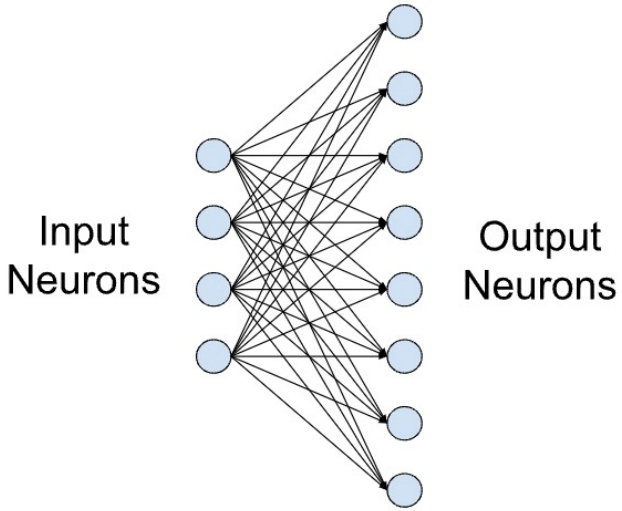
$$dA = \frac{dJ}{dZ}\frac{dZ}{dA} = W^T \cdot dZ$$

$$dW = \frac{dJ}{dZ}\frac{dZ}{dW} = \frac{1}{m} \cdot dZ \cdot A^T$$

$$db = \frac{dJ}{dZ}\frac{dZ}{db} = \frac{1}{m}\sum_{i=0}^{m} dZ^{(i)}$$

Linear layers

```
__global__ void linearLayerUpdateBias(  float* dZ, float* b,
                                        int dZ_x_dim, int dZ_y_dim,
                                        int b_x_dim,
                                        float learning_rate) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < dZ_x_dim * dZ_y_dim) {
        int dZ_x = index % dZ_x_dim;
        int dZ_y = index / dZ_x_dim;
        atomicAdd(&b[dZ_y], - learning_rate *
                (dZ[dZ_y * dZ_x_dim + dZ_x] / dZ_x_dim));
    }
}
```



Forward pass

$$Z = WA + b$$

Backward propagation

$$dA = \frac{dJ}{dZ}\frac{dZ}{dA} = W^T \cdot dZ$$

$$dW = \frac{dJ}{dZ}\frac{dZ}{dW} = \frac{1}{m} \cdot dZ \cdot A^T$$

$$db = \frac{dJ}{dZ}\frac{dZ}{db} = \frac{1}{m}\sum_{i=0}^{m} dZ^{(i)}$$

Build MLP

```
int main() {

    srand( time(NULL) );

    CoordinatesDataset dataset(100, 21);
    BCECost bce_cost;

    NeuralNetwork nn;
    nn.addLayer(new LinearLayer("linear_1", Shape(2, 30)));
    nn.addLayer(new ReLUActivation("relu_1"));
    nn.addLayer(new LinearLayer("linear_2", Shape(30, 1)));
    nn.addLayer(new SigmoidActivation("sigmoid_output"));
```

Training

```
// network training
Matrix Y;
for (int epoch = 0; epoch < 1001; epoch++) {
    float cost = 0.0;

    for (int batch = 0; batch < dataset.getNumOfBatches() - 1; batch++) {
        Y = nn.forward(dataset.getBatches().at(batch));
        nn.backprop(Y, dataset.getTargets().at(batch));
        cost += bce_cost.cost(Y, dataset.getTargets().at(batch));
    }

    if (epoch % 100 == 0) {
        std::cout   << "Epoch: " << epoch
                    << ", Cost: " << cost / dataset.getNumOfBatches()
                    << std::endl;
    }
}
```

Inference and evaluate

```
    // compute accuracy
    Y = nn.forward(dataset.getBatches().at(dataset.getNumOfBatches() - 1));
    Y.copyDeviceToHost();

    float accuracy = computeAccuracy(
            Y, dataset.getTargets().at(dataset.getNumOfBatches() - 1));
    std::cout   << "Accuracy: " << accuracy << std::endl;

    return 0;
}

float computeAccuracy(const Matrix& predictions, const Matrix& targets) {
    int m = predictions.shape.x;
    int correct_predictions = 0;

    for (int i = 0; i < m; i++) {
        float prediction = predictions[i] > 0.5 ? 1 : 0;
        if (prediction == targets[i]) {
            correct_predictions++;
        }
    }

    return static_cast<float>(correct_predictions) / m;
}
```

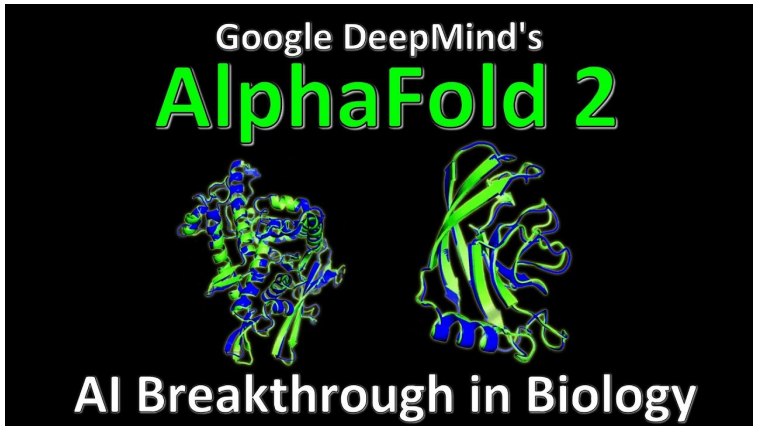
Question:

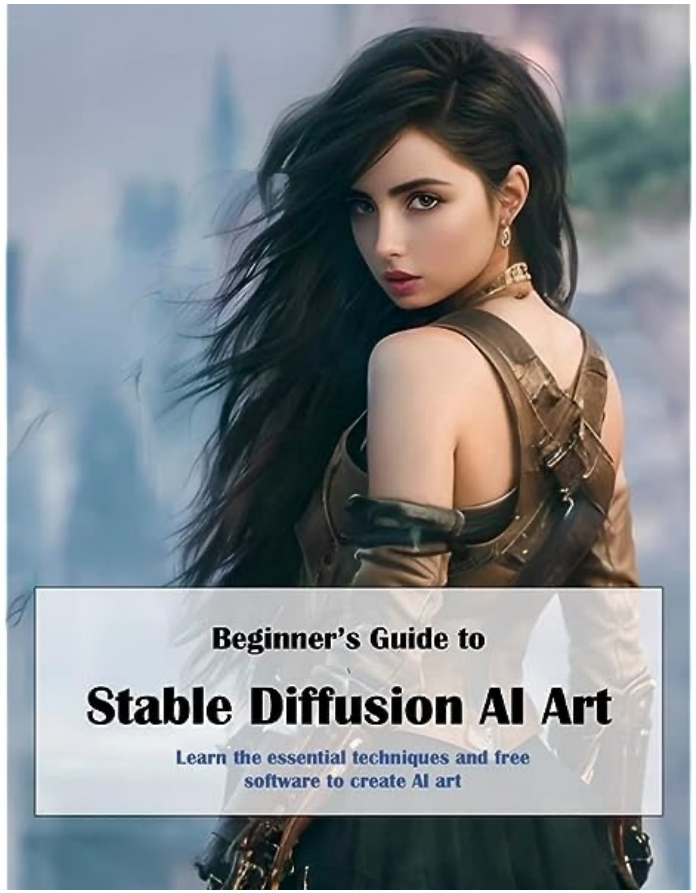
How can we potentially improve the performance of our implementation?

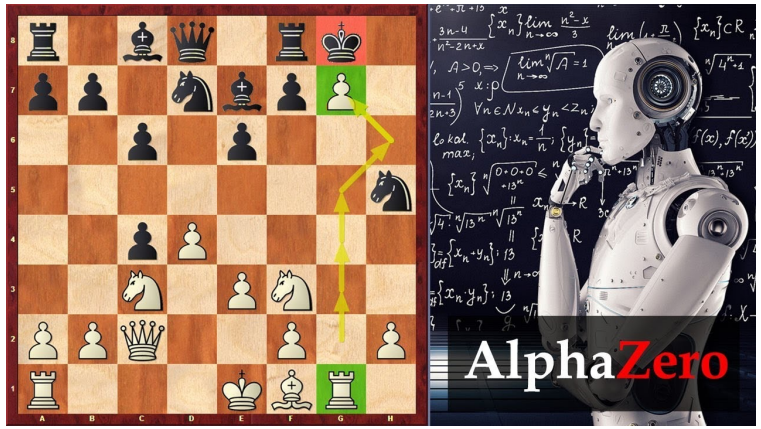
Any questions so far?

Advanced optimizations for DL applications with GPUs

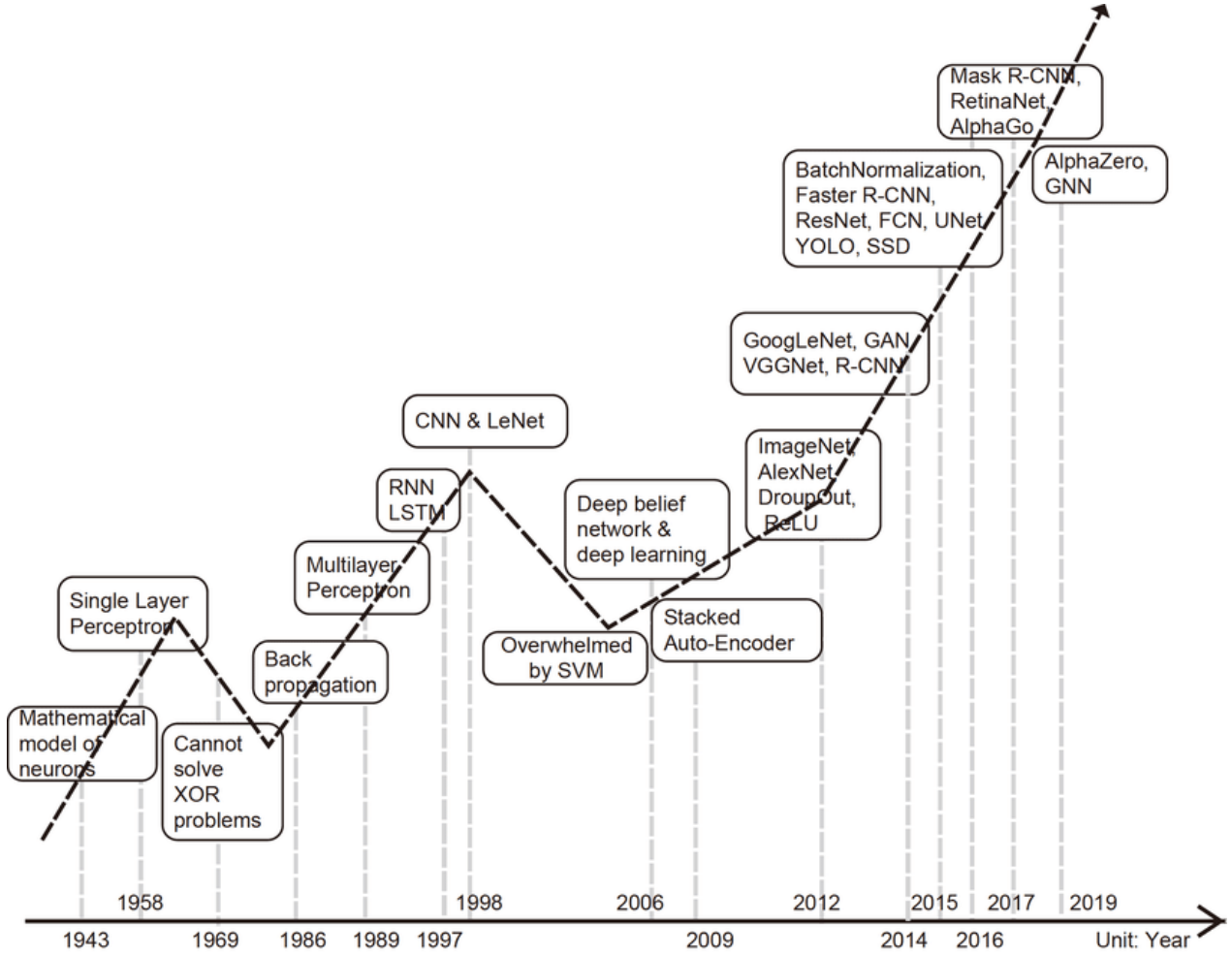
AI revolution





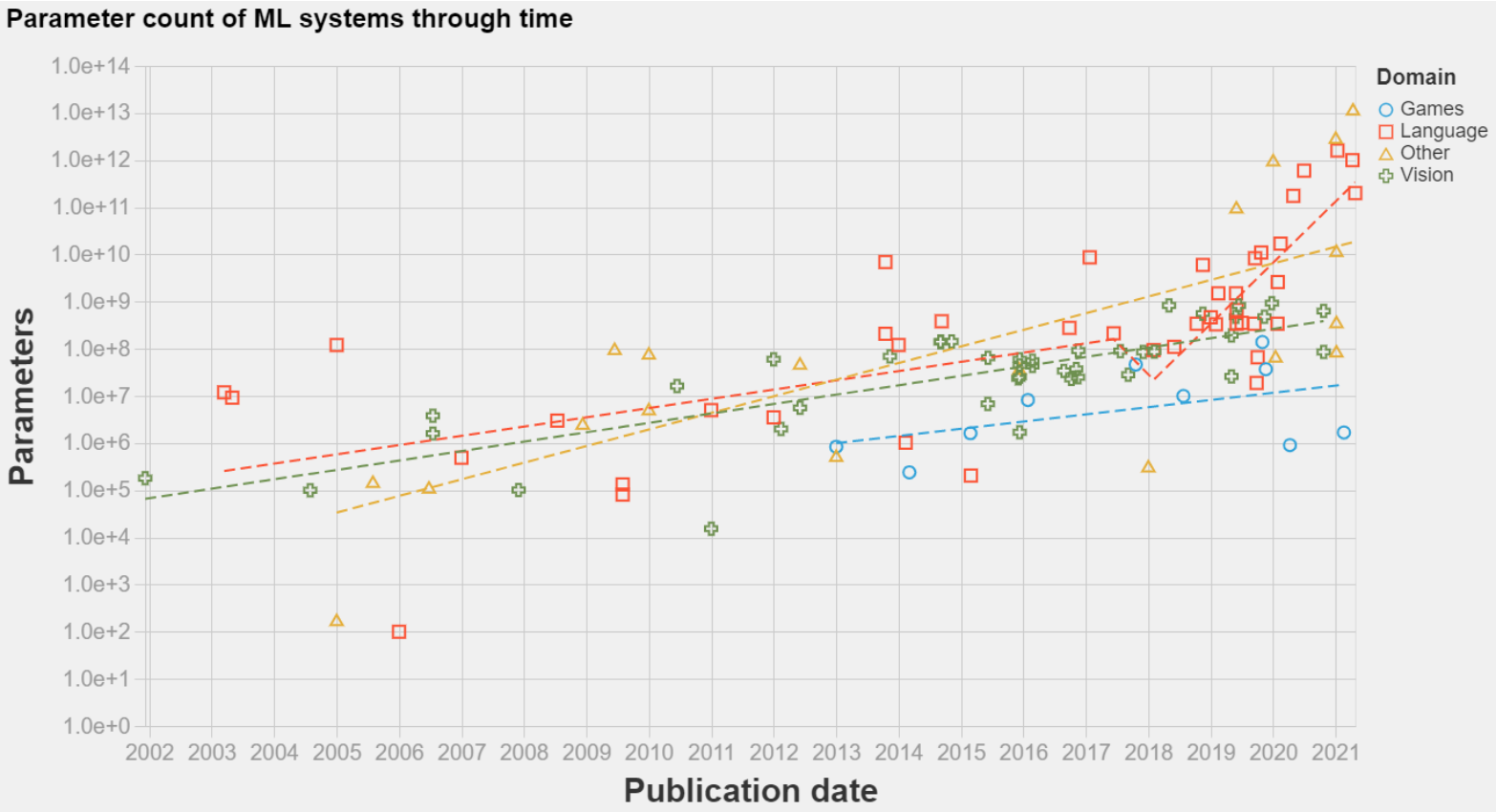


AI revolution

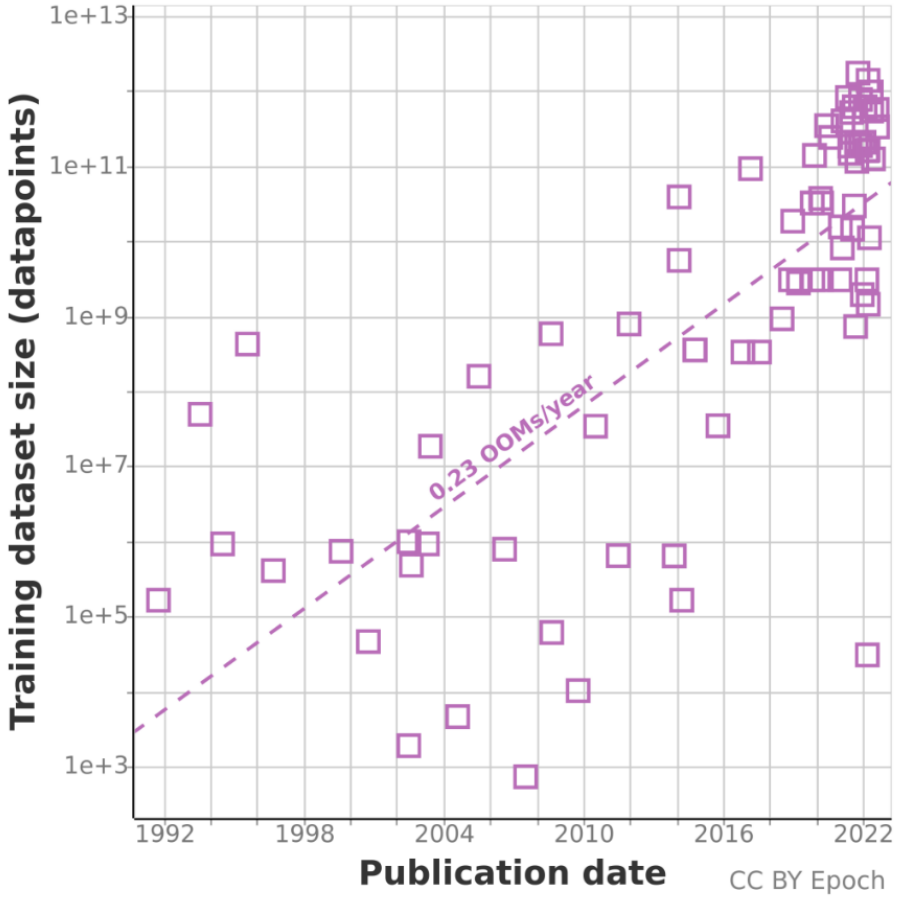


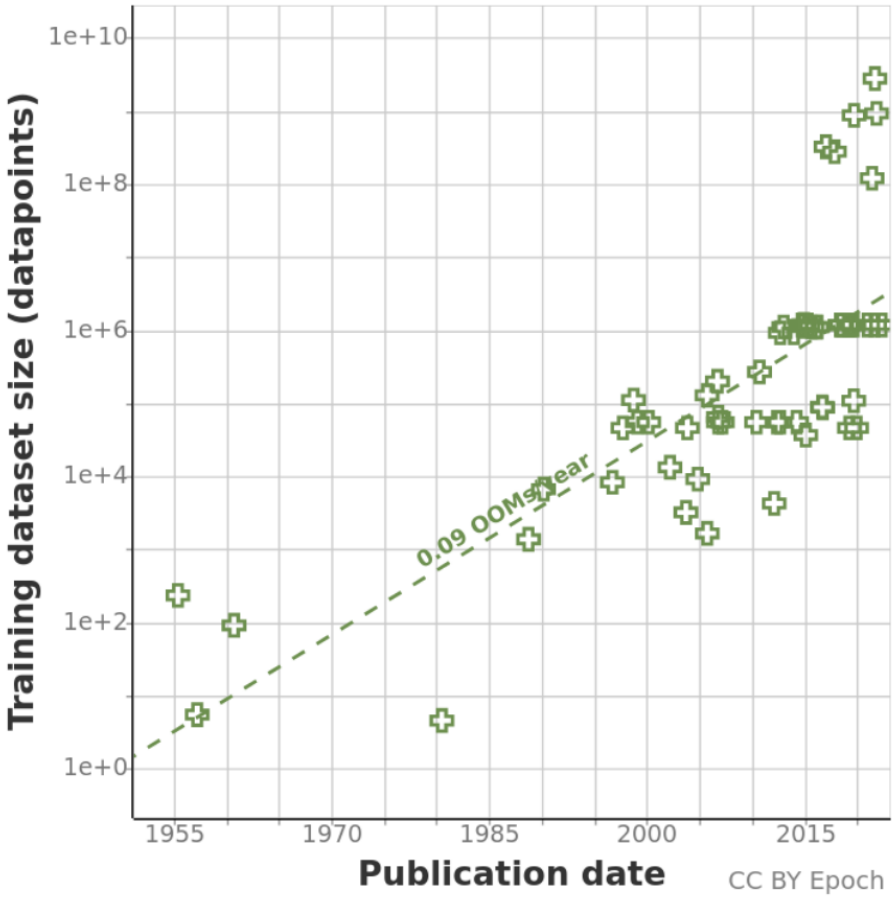
What are the most notable trends in DL models?

Trend of ML systems

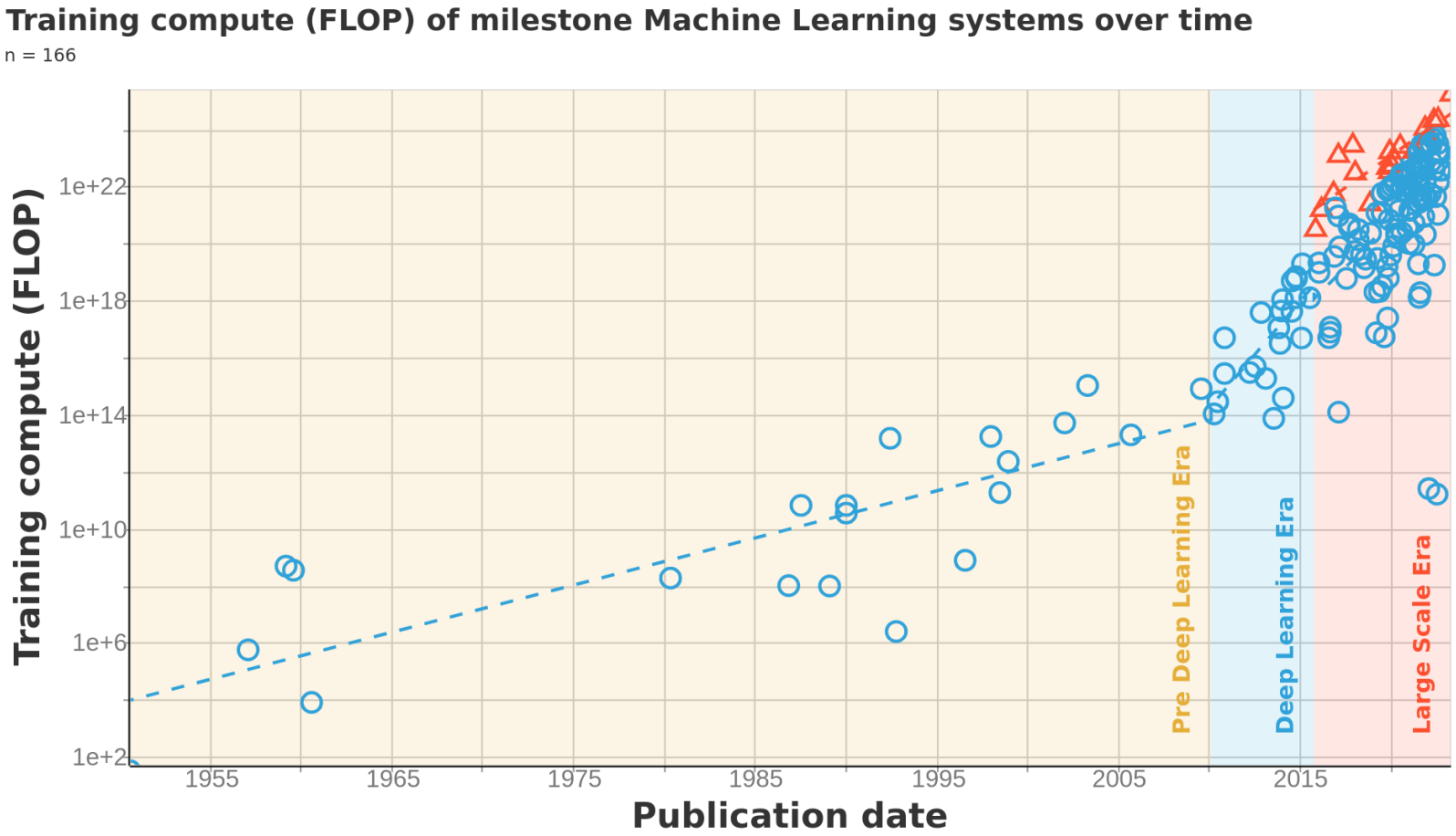


Trend of data size





Trend of computing power



This is where GPU may help!

GPU for DL

Question:

How can we unleash the power for GPU for DL? What kind of optimizations do we need for GPU to better serve DL applications?

GPU for DL

- Optimizing single-GPU performance
  - General optimizations (e.g., memory management)
  - Leverage specific hardware (e.g., Tensor core)
  - Tailored optimizations (e.g., pruning)

GPU for DL

- Optimizing single-GPU performance
  - General optimizations (e.g., memory management)
  - Leverage specific hardware (e.g., Tensor core)
  - Tailored optimizations (e.g., pruning)
- Leveraging multiple GPUs for distributed training
  - Improved network bandwidth (e.g., NV-link)
  - Optimized communication library (e.g., NCCL)
  - Reduced communication (e.g., DeepCompress)

GPU for DL

- Optimizing single-GPU performance
  - General optimizations (e.g., memory management)
  - Leverage specific hardware (e.g., Tensor core)
  - Tailored optimizations (e.g., pruning)
- Leveraging multiple GPUs for distributed training
  - Improved network bandwidth (e.g., NV-link)
  - Optimized communication library (e.g., NCCL)
  - Reduced communication (e.g., DeepCompress)
- Fitting large models to GPUs
  - Model parallelization
  - Computation offloading (e.g., ZeRO-Offload)

Optimizing GPU for DL

TABLE 1
A classification based on context and evaluation

Category	References
NN architecture	
RNN	[7, 8]
CNN	nearly all others
Machine learning phase evaluated/optimized	
Inference	[5, 8–22]
Training	[4, 5, 7, 9–12, 16, 19, 21, 23–51, 51–58]
Framework/library used for evaluation or comparison	
cuDNN	[4, 8, 9, 11, 17–20, 23, 24, 27, 31–34, 36, 39, 42, 44, 54, 59]
cuBLAS	[14, 18, 27, 54, 59]
Caffe	[7, 11, 13, 14, 18, 19, 21–23, 25, 28, 29, 40, 41, 47, 49, 52, 55, 57, 58]
TensorFlow	[8, 19, 21, 28, 30, 35, 38, 41, 43, 51, 53]
PyTorch	[28, 37, 38]
MPI	[37, 43, 56]
NCCL	[37, 43]
Others	MAGMA [5, 12, 13, 60], TensorRT [8], NERVANA [18], cuSPARSE [14], clBLAS [33], MXNet [21]
GPU make	
AMD	Radeon Fury X [29], FirePro W7100 [29], W8000 [33]
NVIDIA	nearly all others
GPU model used	
Fermi	GTX 480 [10, 16], M2050 [52]
Maxwell	GTX Titan X [4, 10, 17, 18, 24, 44, 52], GTX 750Ti [41], GTX 970m [18], GTX 980 [9, 20], M40 [35, 46]
Kepler	GTX 680 [16], K20 [16, 48], K20C [7, 18, 49, 52], K20m [59], K40 [9, 25, 27–29], K40c [23], K40m [12, 22, 26, 36, 45], K80 [19, 25, 30, 41, 46, 47, 53], GTX Titan Black [11]
Pascal	Titan X (Pascal) [13, 31], GTX 1080 [27], GTX 1080Ti [14, 34], P40 [38, 43], P100 [14, 25, 30–32, 37–41, 50, 55, 58, 60]
Volta	Titan V [8], V100 [5, 21, 30, 31, 37, 40–42, 44, 56, 61, 62]
Comparative and collaborative evaluation	
Comparison of GPU architectures	[21, 27]
Comparison of GPU with	CPU [17, 19, 22, 25, 60, 63], Xeon Phi [33, 46]
CPU-GPU heterogeneous computing	[17, 22, 39]

“A survey of techniques for optimizing deep learning on GPUs”. S. Mittal et al.

Optimizing GPU for DL

TABLE 2
A classification of compute-related challenges and optimizations

Compute-related bottlenecks	Synchronization barrier between kernels [8], data-dependencies prohibiting parallelization [11], degree of parallelization being low due to small batch size [11]
Pipelining	[7, 9, 15, 17, 24, 26, 28, 37, 37, 39, 40, 43–47, 49, 56]
Padding	[16, 33]
Loop unrolling	[10, 16]
Creating scope for increasing batch size	by performing gradient aggregation on the host to reduce memory consumption on GPU [39], by transferring intermediate data from GPU memory to CPU memory [28], by using low-precision tensors [37], by dividing the image into patches [32], by using "layer-wise adaptive rate scaling" (LARS) scheme [43]
Avoiding branch-divergence	Avoiding condition-check by performing variable increment of program-counter [10], Grouping threads that perform multiplication on the same input fmap into the same warp so that all threads in a warp can skip zero-multiplication [10], avoiding irregular computations by computing the multiplications separately for each non-zero weight and then adding it [14]
Avoiding thread-idling	by using same thread-block size for different tile sizes [5]
Optimizing multiplications/divisions	Skipping multiplications [10], performing SIMD multiplication (which are more efficient than scalar multiplications) [4], achieving division operations using multiply and shift-operations [9]
Kernel-level optimizations	kernel-fusion [11, 15, 27], prioritizing/delaying kernel execution [15], adjusting GPU resources allocated to a kernel [15], compilers for deep-learning [66]
Optimization algorithms/heuristics	Dynamic programming [41, 44], integer linear programming [41], random forest [5], greedy algorithm [24]

"A survey of techniques for optimizing deep learning on GPUs". S. Mittal et al.

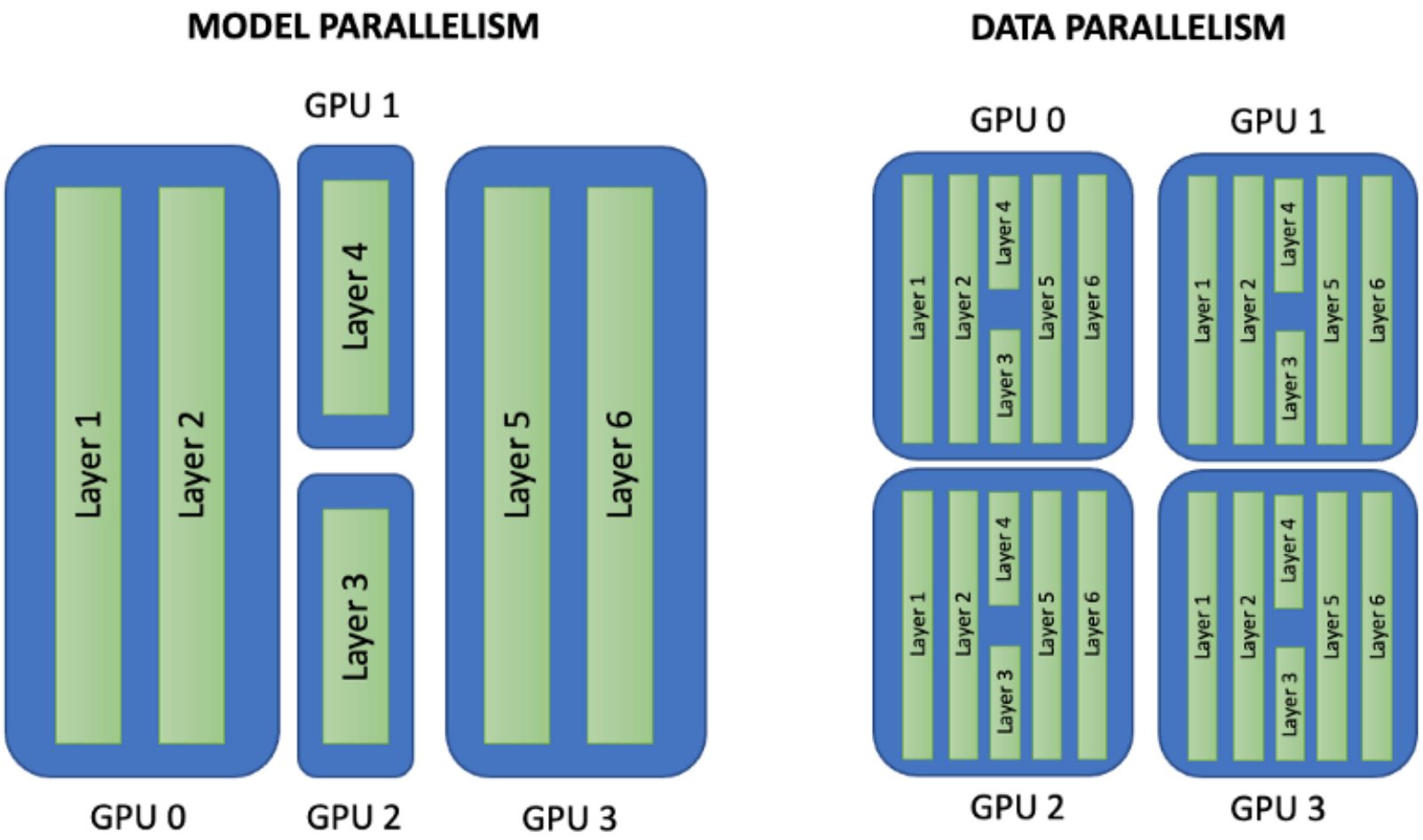
Optimizing GPU for DL

TABLE 3
A classification of memory-related bottlenecks and optimizations

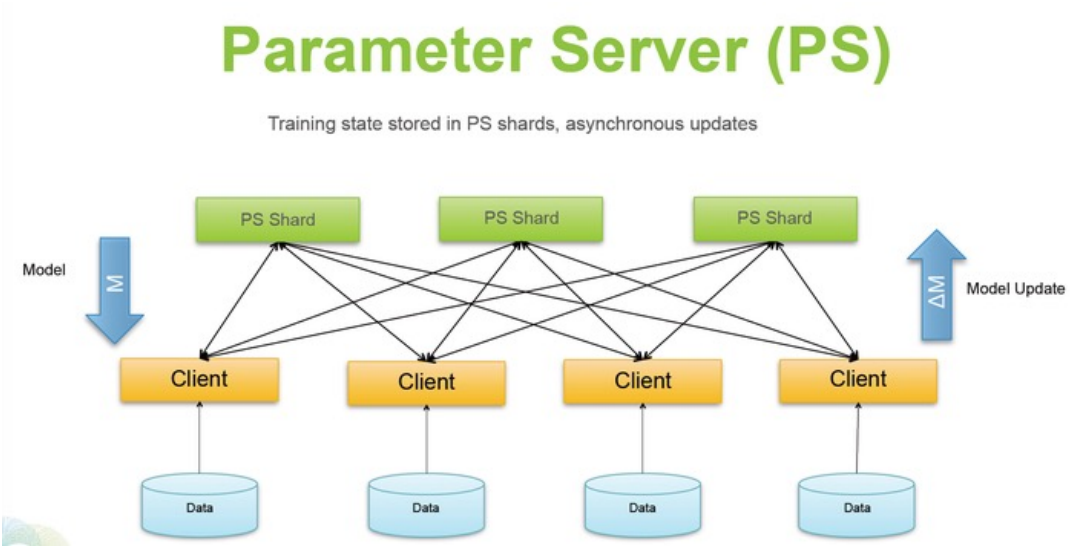
Memory-related bottleneck reasons	Uncoalesced accesses in GlM [11, 20], redundant accesses to GlM [11], bank-conflicts in ShM [11]
Tiling (blocking)	[5, 9, 11, 12, 16, 20, 31, 33, 36, 59, 60]
Use of on-chip memory (ShM) for	performing reduction operation [11], generating lowered matrix [9]
Use of constant memory for storing	input data [14], small filters [12, 33]
Use of texture memory	[27]
Avoiding ShM bank-conflicts	by using broadcast scheme when all threads access the same address [12], by using `float2` datatype [11, 12]
Use of different memories	Storing the data in constant memory in case of single input channel and in GlM in case of multiple input channels [12]
Avoiding fetching	zero-padding data [59], zero weights [10]
L1 cache bypassing	[27]
Increasing data-reuse	by reordering instructions or computations [10, 59], by increasing the number of elements computed by each thread [11, 12]
Pruning	[4, 13, 14]
Prefetching	[12, 34]
Reducing memory requirement	by image-splitting [32], reducing batch size [26, 41]
Avoiding the overhead of `cudaMalloc/cudaFree`	by allocating a large memory pool only once and then acquiring/releasing memory from this pool [24, 28]

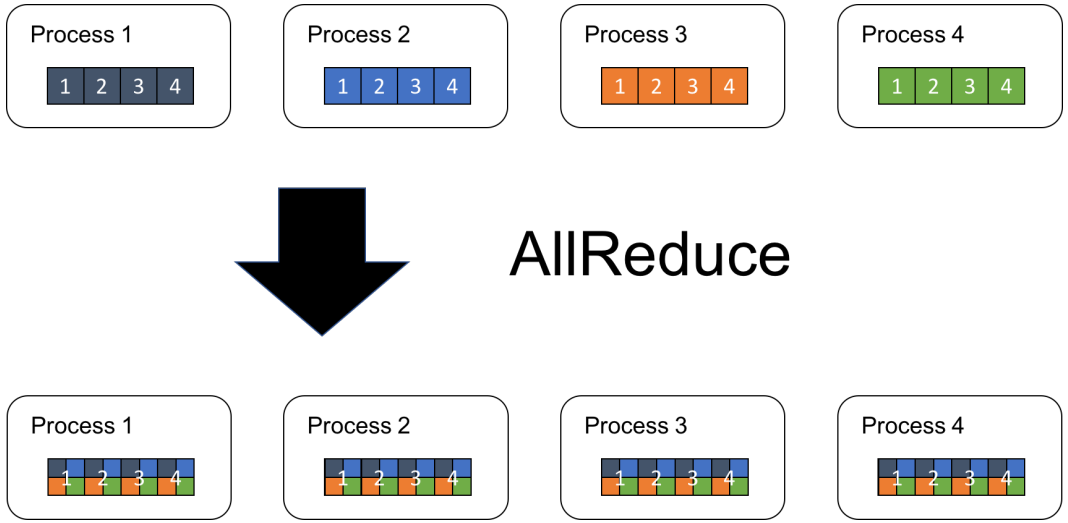
“A survey of techniques for optimizing deep learning on GPUs”. S. Mittal et al.

Distributed training

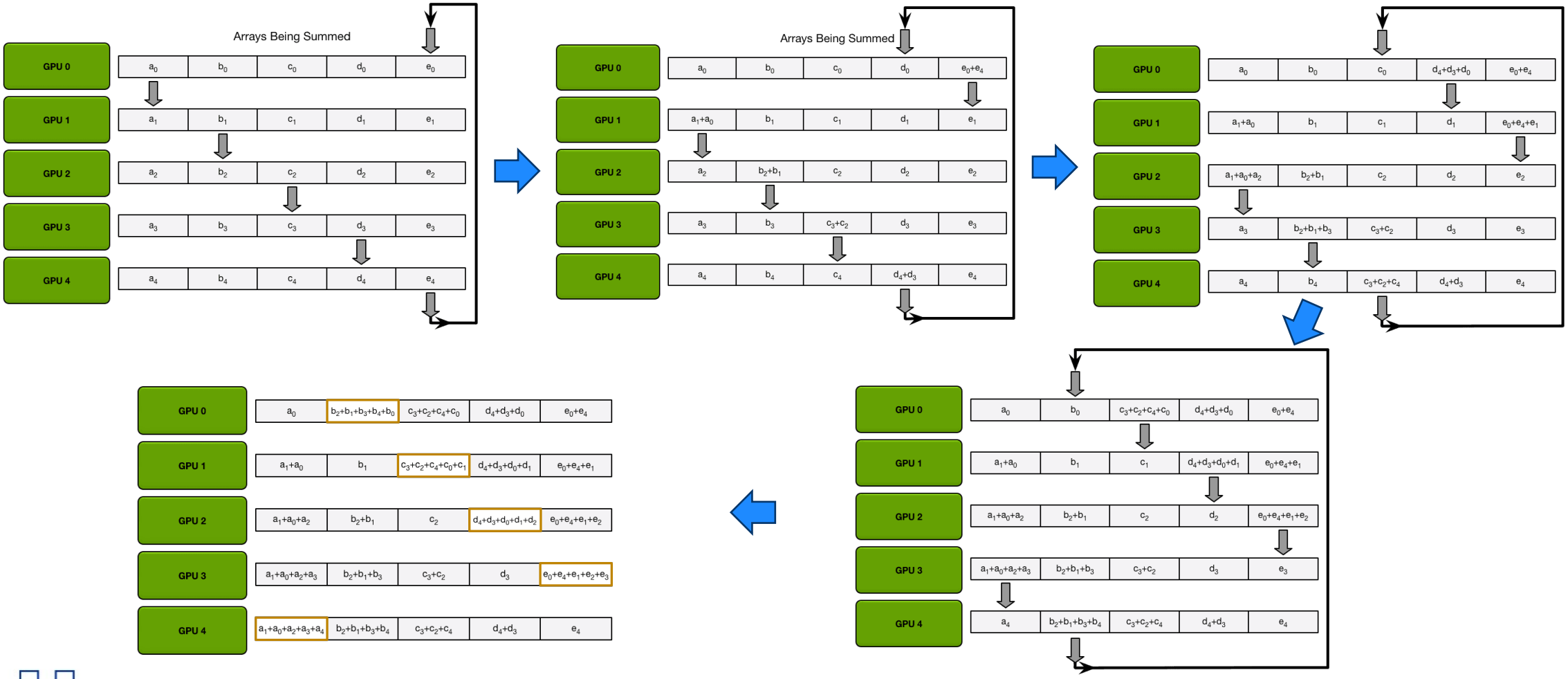


Communication patterns in data parallel models

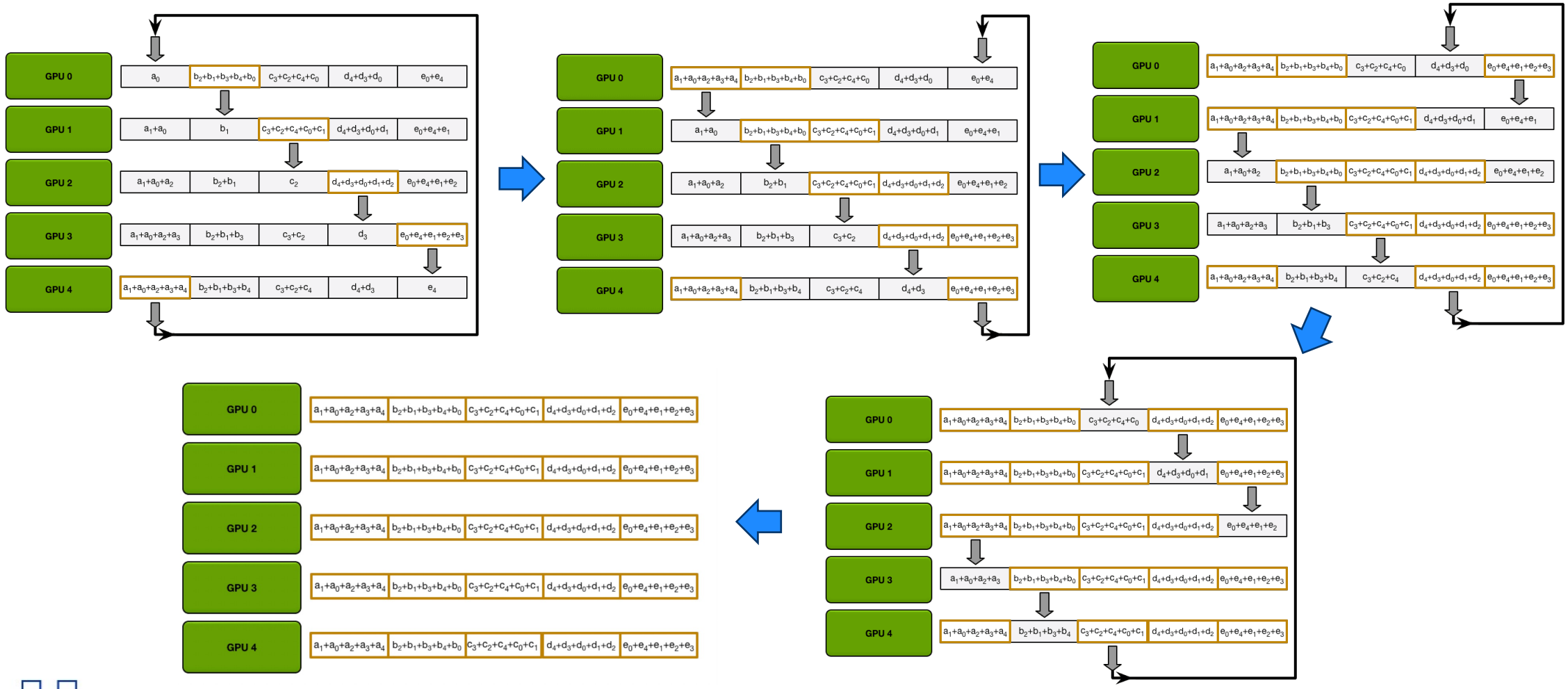




Ring-based Allreduce



Ring-based Allreduce



Pruning (model size)

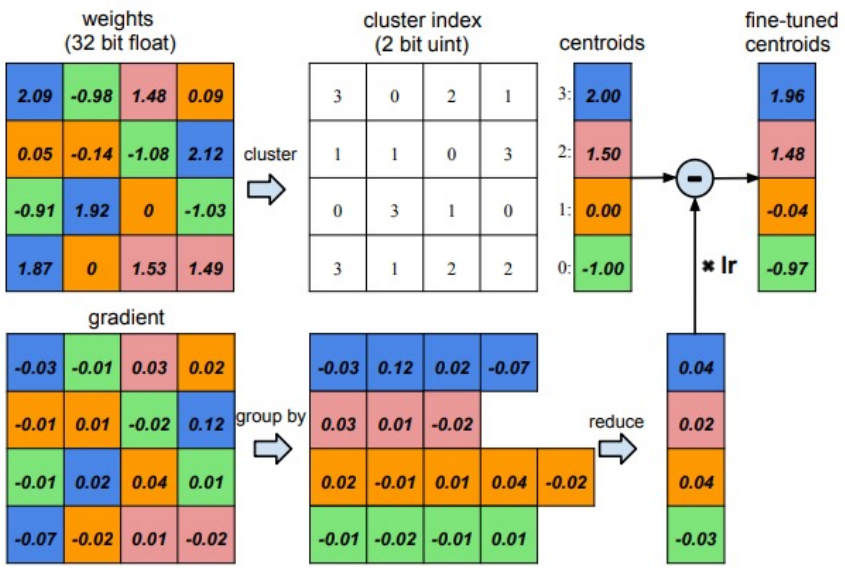


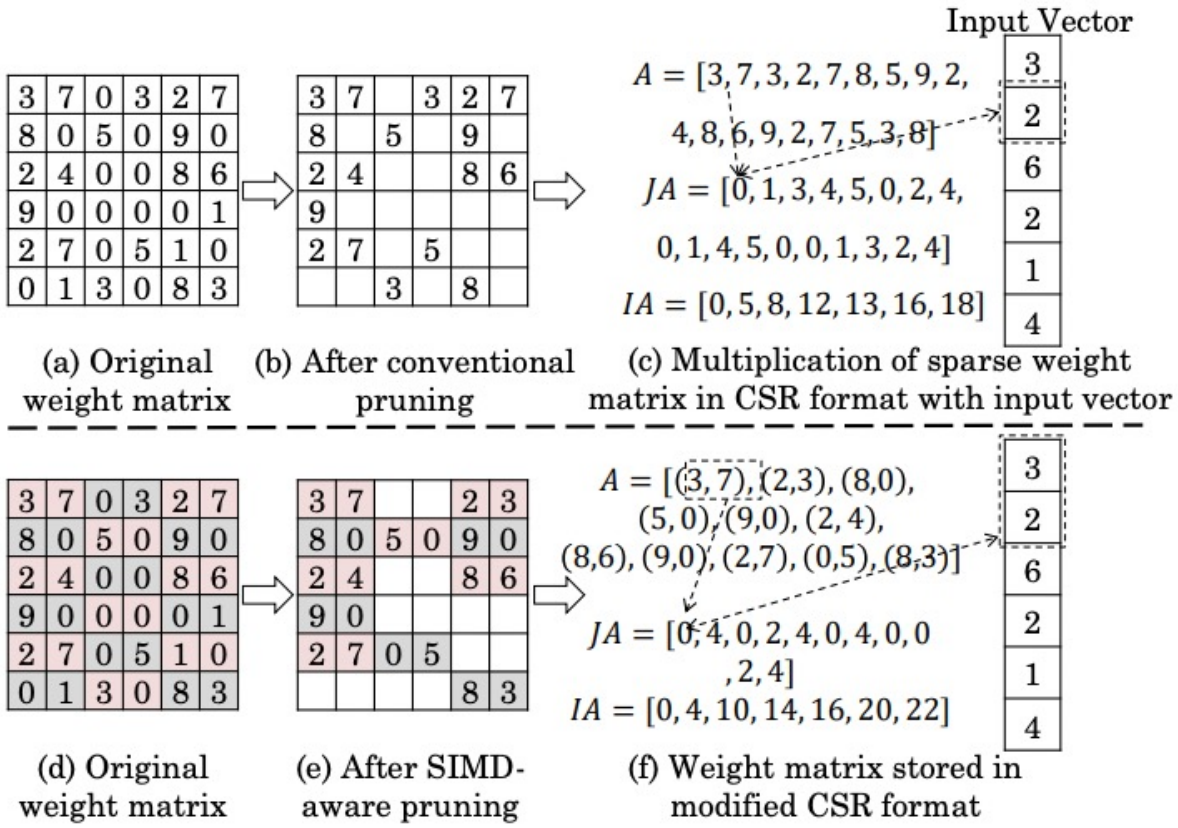
Figure 3: Weight sharing by scalar quantization (top) and centroids fine-tuning (bottom).

Table 1: The compression pipeline can save 35× to 49× parameter storage with no loss of accuracy.

Network	Top-1 Error	Top-5 Error	Parameters	Compress Rate
LeNet-300-100 Ref	1.64%	-	1070 KB	
LeNet-300-100 Compressed	1.58%	-	**27 KB**	**40×**
LeNet-5 Ref	0.80%	-	1720 KB	
LeNet-5 Compressed	0.74%	-	**44 KB**	**39×**
AlexNet Ref	42.78%	19.73%	240 MB	
AlexNet Compressed	42.78%	19.70%	**6.9 MB**	**35×**
VGG-16 Ref	31.50%	11.32%	552 MB	
VGG-16 Compressed	31.17%	10.91%	**11.3 MB**	**49×**

"Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding". S. Han et al.

Pruning (computation)



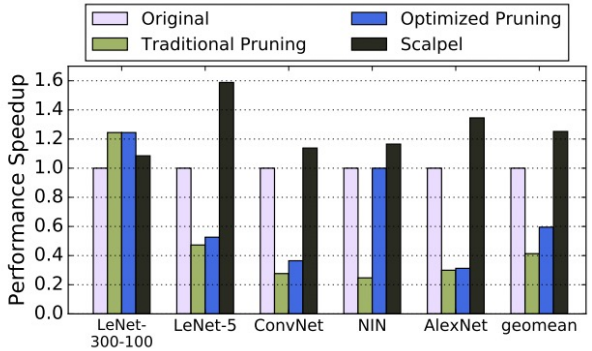


Figure 20: Relative performance speedups of the original models, traditional pruning, optimized pruning and Scalpel on NVIDIA GTX Titan X GPU.

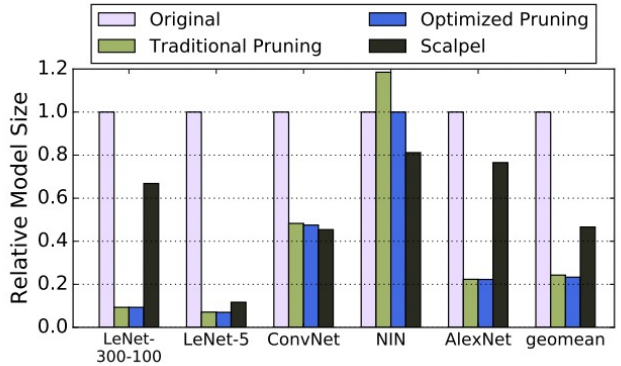


Figure 21: Relative model sizes of the original models, traditional pruning, optimized pruning and Scalpel for NVIDIA GTX Titan X GPU.

"Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism". J. Yu et al.

Pruning (communication)

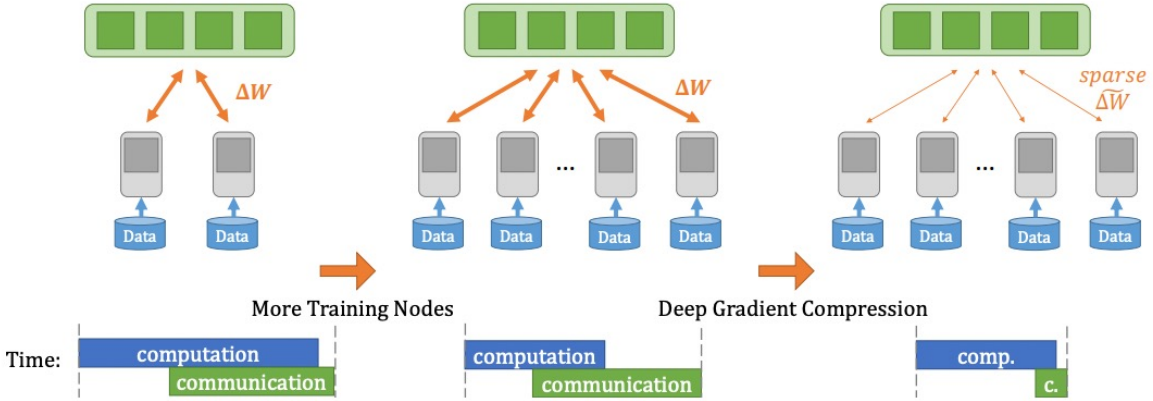


Figure 1: Deep Gradient Compression can reduce the communication time, improve the scalability, and speed up distributed training.

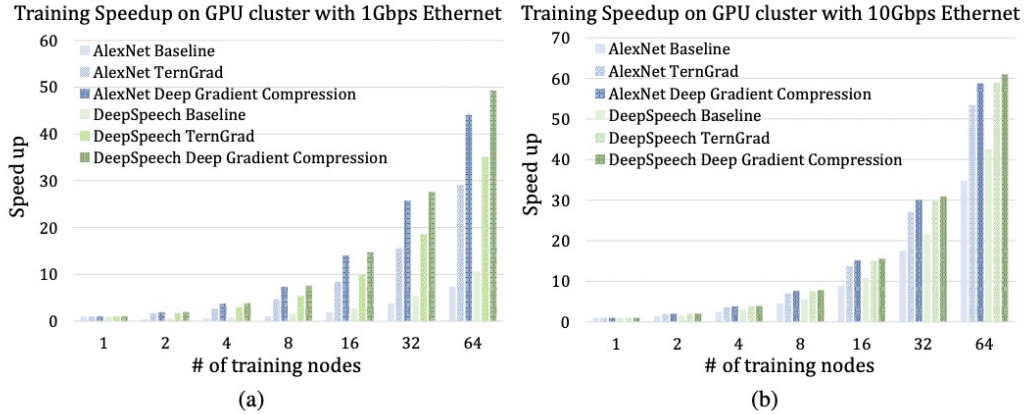


Figure 6: Deep Gradient Compression improves the speedup and scalability of distributed training. Each training node has 4 NVIDIA Titan XP GPUs and one PCI switch.

Table 2: ResNet-110 trained on Cifar10 Dataset

# GPUs in total	Batchsize in total per iteration	Training Method	Top 1 Accuracy	
4	128	Baseline	93.75%	
		Gradient Dropping (Aji & Heafield, 2017)	92.75%	-1.00%
		Deep Gradient Compression	**93.87%**	**+0.12%**
8	256	Baseline	92.92%	
		Gradient Dropping (Aji & Heafield, 2017)	93.02%	+0.10%
		Deep Gradient Compression	**93.28%**	**+0.37%**
16	512	Baseline	93.14%	
		Gradient Dropping (Aji & Heafield, 2017)	92.93%	-0.21%
		Deep Gradient Compression	**93.20%**	**+0.06%**
32	1024	Baseline	93.10%	
		Gradient Dropping (Aji & Heafield, 2017)	92.10%	-1.00%
		Deep Gradient Compression	**93.18%**	**+0.08%**

Table 3: Comparison of gradient compression ratio on ImageNet Dataset

Model	Training Method	Top-1 Accuracy	Top-5 Accuracy	Gradient Size	Compression Ratio
AlexNet	Baseline	58.17%	80.19%	232.56 MB	1 ×
	TernGrad (Wen et al., 2017)	57.28% (-0.89%)	80.23% (+0.04%)	29.18 MB [1]	8 ×
	Deep Gradient Compression	**58.20% (+0.03%)**	**80.20% (+0.01%)**	**0.39 MB** [2]	**597 ×**
ResNet-50	Baseline	75.96	92.91%	97.49 MB	1 ×
	Deep Gradient Compression	**76.15 (+0.19%)**	**92.97% (+0.06%)**	**0.35 MB**	**277 ×**

"Deep gradient compression: Reducing the communication bandwidth for distributed training". Y. Lin et al.

That's all, thanks!

Any questions?