CUDA and CNN

Ulrich Finkler – Wei Zhang

CSCI-GA.3033-022 HPML

Performance Factors

Algorithms Performance

- Algorithm choice

Hyperparameters Performance

- Hyperparameters choice

Implementation Performance

- Implementation of the algorithms on top of a framework

Framework Performance

- Python, Python, Cpython

Libraries Performance

- **CUDA, cuDNN**, Communication Libraries (MPI, GLU)

Hardware Performance

- CPU, DRAM, **GPU, HBM,** Tensor Units, Disk/Filesystem, Network

CUDA Memory Access

CUDA Memory Types

- **Global Memory:**
  - DRAM or HBM
  - Lower Bandwidth than other device memories (but higher than host memory BW)
  - Higher latency than other device memories
- **Shared Memory**
  - **User-managed cache**
  - High Bandwidth
  - Low latency
- **Caches**
  - L1 and L2
  - Texture cache
  - Constant cache

Global memory – Memory Alignment

- The device (GPU) can read 1b, 2b, 4b, 8b, or 16b aligned words from global memory into registers in a single instruction if the memory access is aligned to the size of the type (ex: char, float, etc.)
- Aligned access – will compile in a single instruction:

```
double array[16];
array[5] = 0.0; // access aligned to its type size (8 bytes)
```

- Misaligned access – will NOT compile in a single instruction:

```
struct {
        char a;
        double b;
} misaligned;
misaligned.b = 0.0 // access not aligned to its type size (8 bytes)
```

- See http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory-accesses

Global memory – Memory Alignment

- Built-in types like float2 or float4 are aligned automatically:

Data Type	Size	Alignment
float2	8 byte	8
float3	12 byte	4
float4	16 byte	16

- Data structures can be explicitly aligned with the *__align__(x)* keyword:

```
struct __align__(8){
float a;
float b;
};
```

```
struct __align__(16){
float a;
float b;
float c;
};
```

```
struct __align__(16) {
float a;
float b;
float c;
float d;
float e;
};
```

Global memory – Memory Alignment

- Alignment using CUDA primitives:
  - cudaMallocPitch()
  - cudaMalloc3D()
  - cudaMemcpy2D()
  - cudaMemcpy3D()
- **Pitch**: stride to use for the **row**

```
// Host code
int width = 64, height = 64;
float* devPtr;
size_t pitch;
cudaMallocPitch(&devPtr, &pitch,
                width * sizeof(float), height);
MyKernel<<<100, 512>>>(devPtr, pitch, width, height);

// Device code
__global__ void MyKernel(float* devPtr,
                         size_t pitch, int width, int height)
{
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
    }
}
```

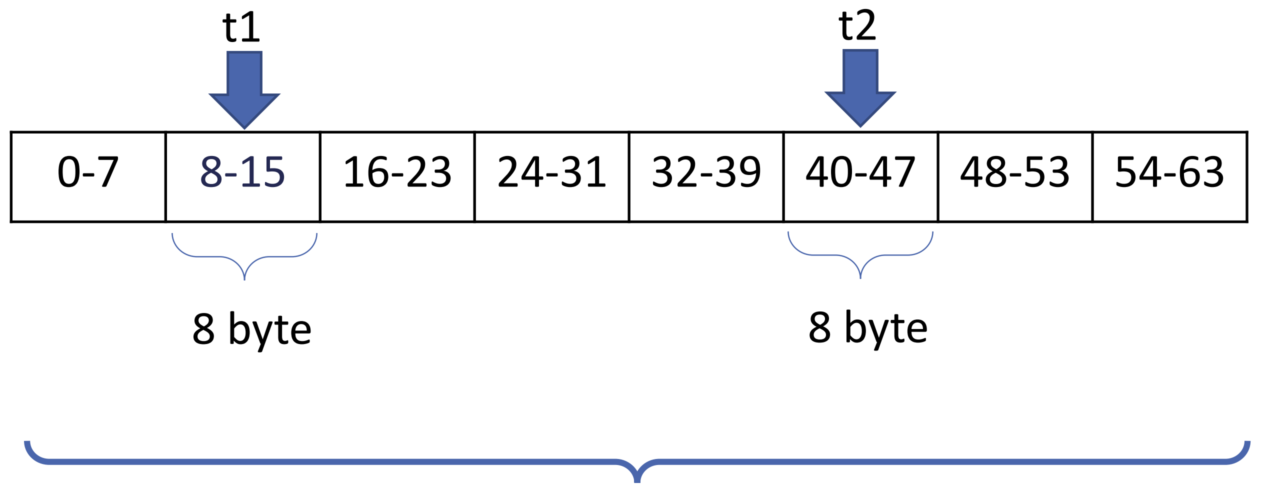
Global memory – Memory Coalescing

- Global memory bandwidth is used most efficiently by simultaneous memory accesses by many threads (32 is a warp)
- Accesses by fewer threads is not enough to tolerate access latencies
- Example of contiguous accesses:
  - 64 bytes accesses: each thread reads a word: int, float, etc
  - 128 bytes accesses: each thread reads a double-word: int2, float2, etc
  - 256 bytes accesses: each thread reads a quad-word: int4, float4, etc

Global memory – Memory Coalescing

- Accesses to memory only happen for 32b, 64b, 128b transaction granularity
- Accesses must be coalesced for best performance
- **Example:**
  - Two threads (*t1* and *t2*) do non-coalesced accesses for a total of 16 bytes
  - Need to bring in 64 bytes

1) Each thread does a non-coalesced access



2) Memory Transaction brings 64 bytes

3) Effective Bandwidth for 16 bytes is 1/4

Shared Memory

- Shared memory:
  - Shared accessed by **all threads in a block**
  - Faster access than global memory: high bandwidth / low-latency
- Usage:
  - *__shared__* keyword for arrays (pointers)
- Static shared memory:

```
__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

Dynamic Shared Memory:

```
extern __shared__ int s[];
int *integerData = s;                       // nI ints
float *floatData = (float*)&integerData[nI]; // nF floats
char *charData = (char*)&floatData[nF];     // nC chars

myKernel<<<gridSize, blockSize,
nI*sizeof(int)+nF*sizeof(float)+nC*sizeof(char)>>>(...);
```

Dimensions and thread indexing

- ***dim3*** type:
  - an **integer vector type** based on uint3 that is used to specify dimensions. When defining a variable of type dim3, any component left unspecified is initialized to 1

- 1D thread indexing example:

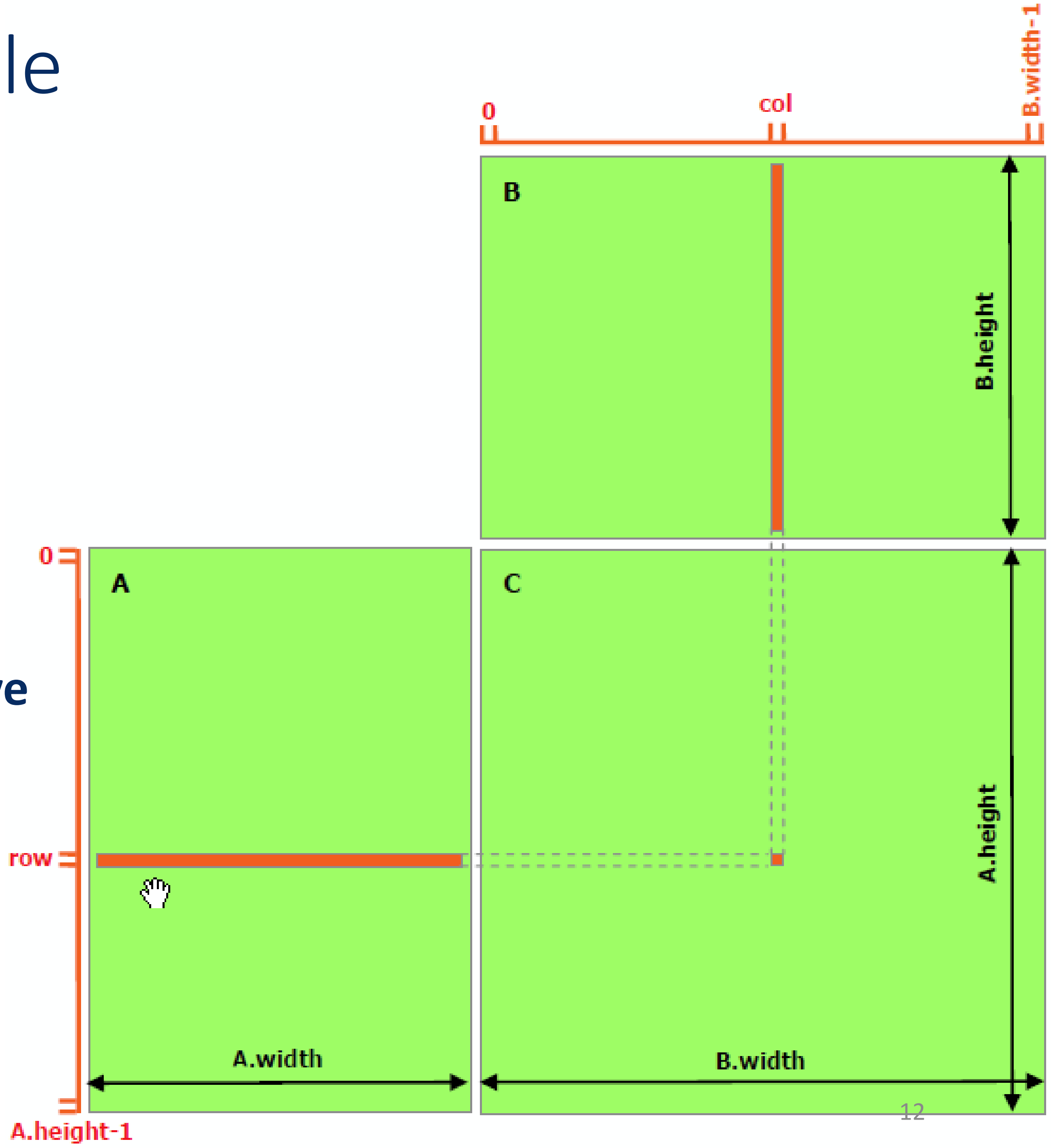
```
dim3 dimBlock(512); //  512 threads in 1D
dim3 dimGrid(1024); // 1024 blocks  in 1D
mykernel<<dimGrid, dimBlock>>();

// Equivalent to
mykernel<<1024,512>>();
```

- 2D thread indexing example:

```
dim3 dimBlock(16,32); //  512 threads in 2D
dim3 dimGrid(32,32);  // 1024 blocks  in 2D
mykernel<<dimGrid, dimBlock>>();
```

Matrix Multiplication Example

- Computing matrix multiplication
  - C = A × B
- Each thread compute one (or more) elements of C:
  - Dot product: *row* ***dot*** *column*
- Each thread needs to access global memory for the row and the column
  - Column accesses are in **non-consecutive** addresses



Matrix Multiplication Example (1)

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;

// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);
```

- Matrix in row-major order
- Computing matrix multiplication C = A × C
- Is block size optimal for every GPU/problem?

Matrix Multiplication Example (2)

- Matrix sizes assumed to be multiple of BLOCK_SIZE
- Allocating A,B and C
- Is block size optimal for every GPU?

```
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);

    // Allocate C in device memory
    Matrix d_C;
    d_C.width = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);
```

Matrix Multiplication Example (3)

- Using 2D index
- Copy to device
- Call kernel
- Copy from device
- Free memory

```
// Second part of
// void MatMul(const Matrix A, const Matrix B, Matrix C)
// {
//

// Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C.elements, Cd.elements, size,
               cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
```

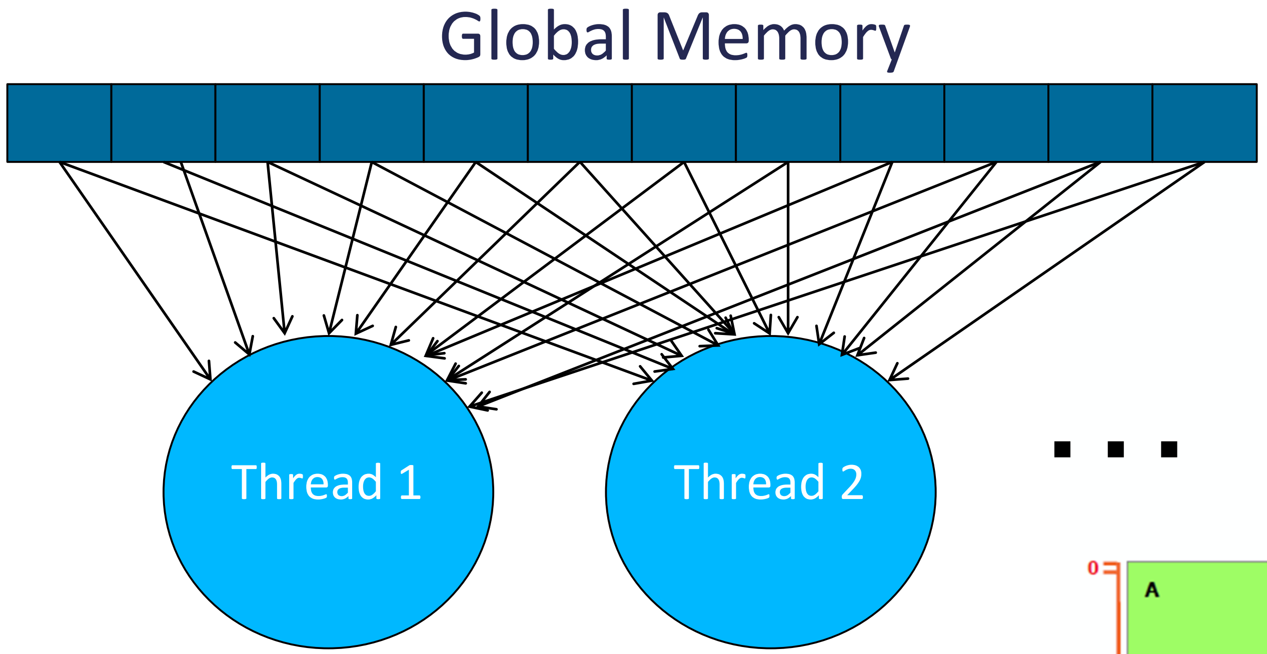
Matrix Multiplication Example (4)

- Using 2D index
- Copying to/from device
- Free memory

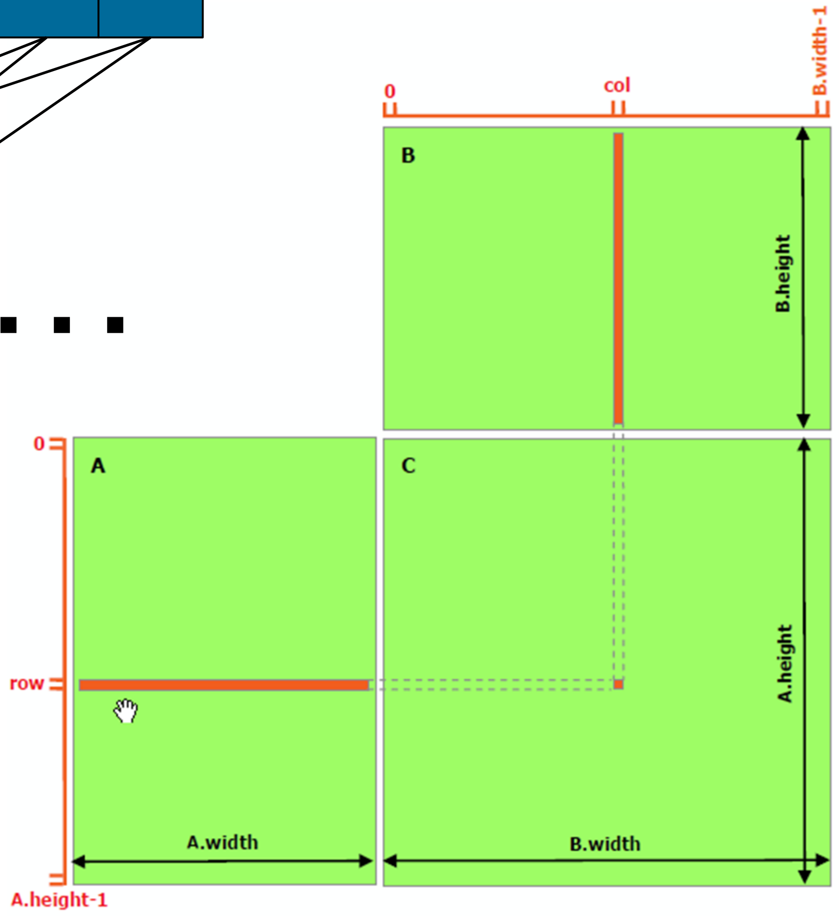
```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

Global memory access without Tiling

- Every thread access a specific address
- No reuse: accesses happen in parallel on addresses that are far apart (stride)

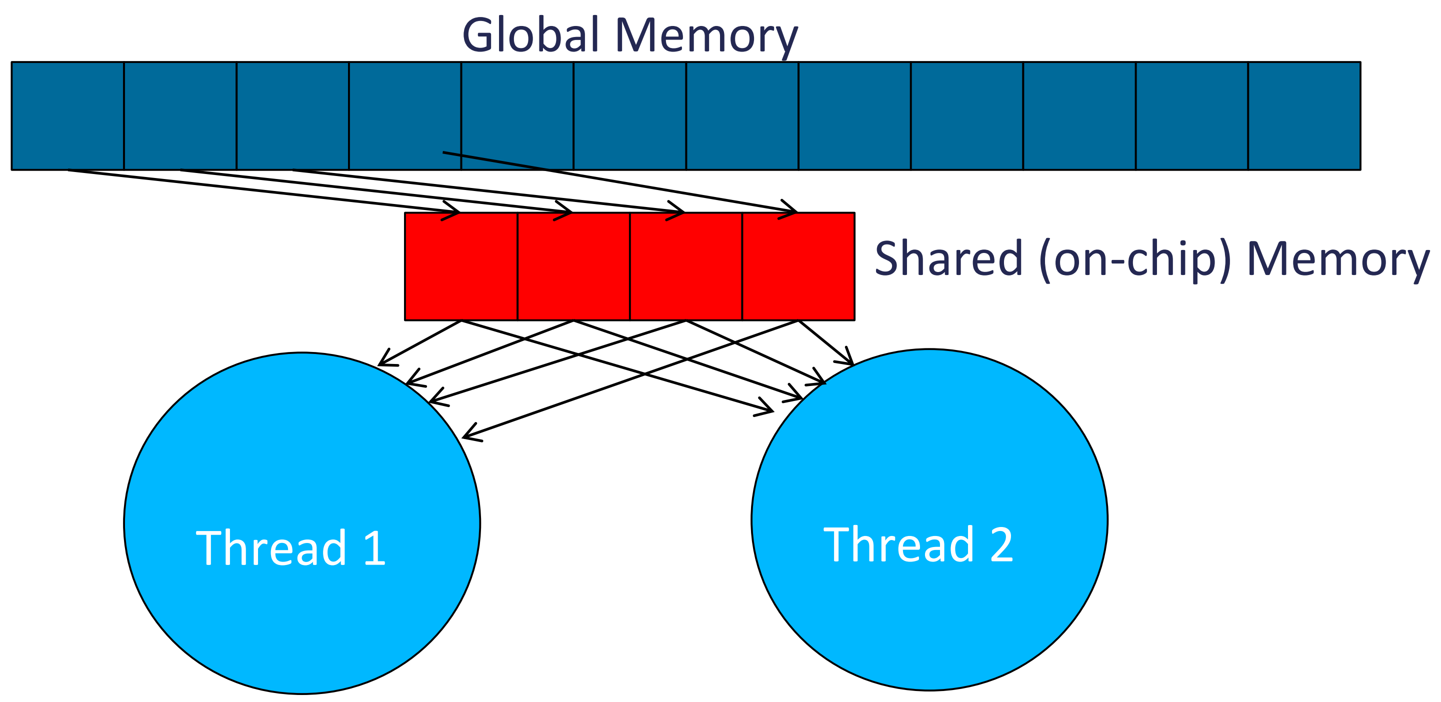


...



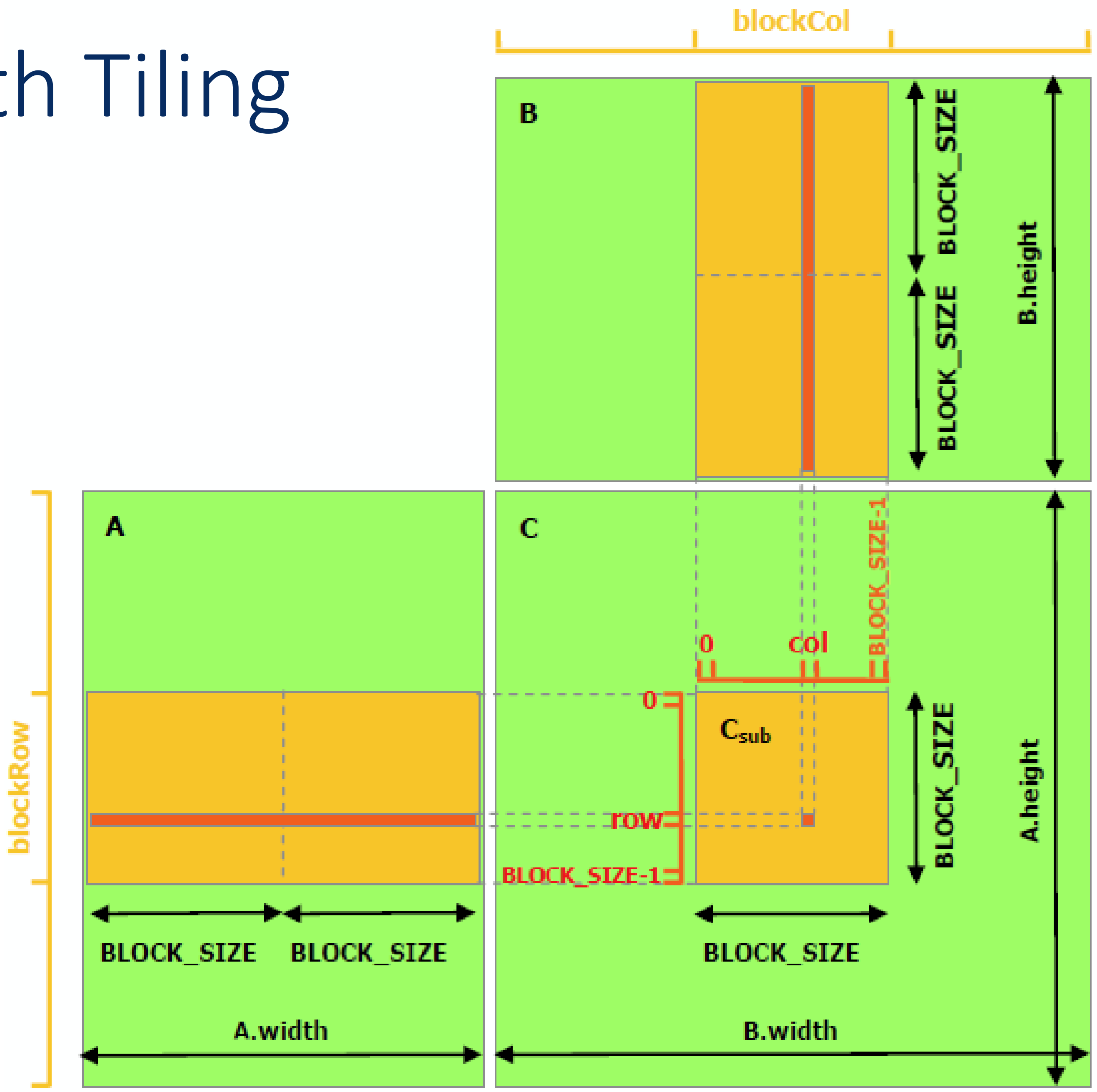
Global memory access with Tiling technique

1. Load in a tile (block)
2. **shared memory** Work on it with lower access latency
3. Store it back into global memory



Matrix Multiplication with Tiling

- Computing matrix multiplication
  - C = A × C
- Use shared memory to reduce global memory access
- Threads in a **block** work on a **tile:**
  1. Load tile in shared memory
  2. Then load elements from shared memory with lower latency
  3. Write tile back to global memory
- Each thread compute one (or more) elements of C:
  - Dot product: *row* ***dot*** *column*



Matrix Multiplication w. Tiling Example (1)

- Elements in row-major order
- Implement getter/setter for easy elements access

```
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
} Matrix;

// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col)
{
    return A.elements[row * A.stride + col];
}

// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col,
                           float value)
{
    A.elements[row * A.stride + col] = value;
}
```

Matrix Multiplication w. Tiling Example (2)

- Function to obtain the sub-matrix

```
// Get the BLOCK_SIZE x BLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices
// down from the upper-left corner of A
 __device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
    Matrix Asub;
    Asub.width    = BLOCK_SIZE;
    Asub.height   = BLOCK_SIZE;
    Asub.stride   = A.stride;
    Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
                                         + BLOCK_SIZE * col];
    return Asub;
}

// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix,
Matrix);
```

Matrix Multiplication w. Tiling Example (3)

- Load A, B into device memory
- Allocate C into device memory

```
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);

    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
    cudaMemcpyHostToDevice);

    // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);
```

Matrix Multiplication w. Tiling Example (4)

- Invoke the kernel
- Copy C to the host
- Free device memory

```
// Second part of host code function
// void MatMul(const Matrix A, const Matrix B, Matrix C)
// {

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size,
               cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
```

Matrix Multiplication w. Tiling Example (5)

- Device Code first part
- Compute dimensions

```
// Matrix multiplication kernel called by MatMul()
 __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;

    // All threads in a block compute one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

    // Each thread computes one element of Csub
    // by accumulating results into Cvalue
    float Cvalue = 0;

    // Thread row and column within Csub
    int row = threadIdx.y;
    int col = threadIdx.x;
```

Matrix Multiplication w. Tiling Example (6)

- Device code second part:

1. Loop over all the sub-matrices of A and B
2. Multiply each pair of sub-matrices together and accumulate the results
3. Write back

```
for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {

        // Get sub-matrix Asub of A
        Matrix Asub = GetSubMatrix(A, blockRow, m);

        // Get sub-matrix Bsub of B
        Matrix Bsub = GetSubMatrix(B, m, blockCol);

        // Shared memory used to store Asub and Bsub respectively
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load Asub and Bsub from device memory to shared memory
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);
        __syncthreads(); // Synch after A and B loaded

        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];
       __syncthreads(); // Synch to make computation is done
    }
    SetElement(Csub, row, col, Cvalue); // Write Csub to device mem
}
```

cuDNN and cuBLAS

NVIDIA Deep Learning SDK

- **Deep Learning Primitives (CUDA® Deep Neural Network library™ (cuDNN))**
  - High-performance building blocks for deep neural network applications including convolutions, activation functions, and tensor transformations.
- **Deep Learning Inference Engine (TensorRT™)**
  - High-performance deep learning inference runtime for production deployment.
- **Deep Learning for Video Analytics (NVIDIA DeepStream™ SDK)**
  - High-level C++ API and runtime for GPU-accelerated transcoding and deep learning inference.
- **Linear Algebra (CUDA® Basic Linear Algebra Subroutines library™ (cuBLAS))**
  - GPU-accelerated BLAS functionality that delivers 6x to 17x faster performance than CPU-only BLAS libraries,
- **Sparse Matrix Operations (NVIDIA CUDA® Sparse Matrix library™ (cuSPARSE))**
  - GPU-accelerated linear algebra subroutines for sparse matrices that deliver up to 8x faster performance than CPU BLAS (MKL), ideal for applications such as natural language processing.
- **Multi-GPU Communication (NVIDIA® Collective Communications Library ™ (NCCL))**
  - Collective communication routines, such as all-gather, reduce, and broadcast that accelerate multi-GPU deep learning training on up to eight GPUs.

cuDNN

- Nvidia's library of low-level primitives for deep learning
  - Closed source
  - Highly optimized
- Provides
  - Tensor manipulation
  - Convolution
  - Pooling
  - Softmax
  - Activation functions
    - Sigmoid, ReLU, Tanh, Clipped ReLu, ELU, identity

cuDNN context

- cuDNN API are executed on the **host** (and not on the device!)
- A *cudnnHandle_t* stores the state of the library and must be used in any subsequent library call
- Usually we need a single handle per device
- Multiple devices => multiple handles

```
#include <cudnn.h>

    /* … */
    cudnnHandle_t cudnn;
    cudnnCreate(&cudnn);
    /* … */
    CudnnDestroy(cudnn);
```

Error checking

- Calls return a *cudnnStatus_t* value:
  - CUDNN_STATUS_SUCCESS denotes... success
  - *cudnnGetErrorString(value)* returns a description of the error
- Generally used as:

```
#define CUDNN_CALL(x) do {                                          \
    cudnnStatus_t ___s = (x);                                       \
    if (___s != CUDNN_STATUS_SUCCESS) {                             \
        fprintf(stderr, "%s:%d ERROR: %s\n", __FILE__,              \
                __LINE__, cudnnGetErrorString(___s));             \
        exit(-1);                                                   \
    }                                                               \
} while (0)

/* … */
CUDNN_CALL(cudnnCreate(&handle));
```

Tensors

- Tensors are represented in memory as a contiguous sequence of elements
- For 4D tensors, the {batch, channel, height, width} coordinates of a data point are linearized in one of three ways (specified by cudnnTensorFormat_t):
  - *CUDNN_TENSOR_NCHW*: batch, channel, height, width
  - *CUDNN_TENSOR_NHWC*: batch, height, width, channel
  - *CUDNN_TENSOR_NCHW_VECT_C*: same as NCHW, but elements are vectors of features
- An arbitrary number of dimensions is supported (up to CUDNN_DIM_MAX)
- Multiple data types are supported (specified by cudnnDataType_t):
  - *CUDNN_DATA_{HALF, FLOAT, DOUBLE}*
  - *CUDNN_DATA_INT{8, 32, 8x4}*
  - *CUDNN_DATA_UINT{8, 8x4}*

Determinism

- Most cuDNN routines produce the same result across runs when:
  - executed on GPUs with the same architecture (compute capability) and the same number of SMs
- Bit-wise reproducibility is not guaranteed across cuDNN versions
- Some of them use atomic operations so results **can be different at every run on the same GPU and cuDNN version**.
- See https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html#reproducibility

Tensor cores arithmetic – Starting From Volta

- The arithmetic used on tensors is specified by *cudnnMathType_t*
  - *CUDNN_DEFAULT_MATH*: standard floating point behavior
  - *CUDNN_TENSOR_OP_MATH*: allows the use of **Tensor Cores**
- **Tensor Cores** provide improved performance at the expense of reduced precision
- Only supported in a subset of primitives and with specific parameters:
  - *cudnnConvolution{Forward,BackwardData,BackwardFilter}*: only specific algorithms
  - *CudnnRNN{ForwardInference,ForwardTraining,BackwardData,BackwardWeights}*: only specific algorithms and batch sizes

API and Descriptors

- Most entities in cuDNN are associated with a descriptor
    - The data lives in **GPU memory**
    - The **descriptor, in CPU memory**, contains information on how to access the data
- Some of the descriptors we'll encounter:
    - *cudnnTensorDescriptor_t*
    - *cudnnConvolutionDescriptor_t*
    - *cudnnFilterDescriptor_t*
    - *cudnnActivationDescriptor_t*
    - *cudnnPoolingDescriptor_t*

Descriptor lifecycle

- Descriptors in general are used according to the following pattern:

  1. Creation: with one of the cudnnCreate*${type}*Descriptor()
  2. Configuration: with one of the cudnnSet*${type}*Descriptor()
  3. Use
  4. Destruction: with one of the cudnnDestroy*${type}*Descriptor()

- Except for tensors, we're omitting the creation and destruction stages

Creating a 4D tensor

- The data has to be allocated separately, e.g., with cudaMalloc()
- The descriptor can be initialized in multiple ways, e.g.:

```
cudnnTensorDescriptor_t in_desc;
CUDNN_CALL(cudnnCreateTensorDescriptor(&in_desc));
CUDNN_CALL(cudnnSetTensor4dDescriptor(in_desc,CUDNN_TENSOR_NCHW,
                                      CUDNN_DATA_FLOAT, N, C, H, W));
```

Creating a N dimensional tensor

- The same descriptor could have been set by using cudnnSetTensorNdDescriptor and arrays of size 4 for *dimA* and *strideA*:

```
cudnnStatus_t cudnnSetTensorNdDescriptor(cudnnTensorDescriptor_t tensorDesc,
                                         cudnnDataType_t dataType, int nbDims,
                                         const int dimA[], const int strideA[])
```

More on tensor descriptors (1)

- cudnnSetTensor4dDescriptorEx() allows custom strides with the simplified interface for 4D tensors
  - Strides are necessary when padding is present in memory at the end of each dimension
  - In a 4D tensor with { N, C, H, W } coordinates the default strides (with no padding present) would be { C*H*W, H*W, W, 1 }

More on tensor descriptors (2)

- Information on a tensor descriptor can be retrieved with:

```
cudnnStatus_t cudnnGetTensorNdDescriptor(const cudnnTensorDescriptor_t tensorDesc,
                                         int nbDimsRequested, cudnnDataType_t *dataType,
                                         int *nbDims, int dimA[], int strideA[])
```

- The tensor's size in bytes can be retrieved with:

```
cudnnStatus_t cudnnGetTensorSizeInBytes(const cudnnTensorDescriptor_t tensorDesc, size_t *size)
```

More on tensor descriptors

- Remember to destroy a tensor descriptor (or any other descriptor) when not needed anymore:

```
cudnnStatus_t
cudnnDestroyTensorDescriptor(cudnnTensorDescriptor_t
tensorDesc)
```

- Also, remember that none of the resources associated with a descriptor will be freed for you
  - If tensor data was allocated with cudaMalloc() it is the programmer's responsibility to call cudaFree() on it

Filter descriptors

- Convolution coefficients are stored in a filter:
  - A tensor-like object with its own descriptor type
  - The layout is still specified in terms of { N, C, H, W }, but N refers to the output maps, and C to the input maps
- Initialize 4D filter with:

```
cudnnStatus_t cudnnSetFilter4dDescriptor(cudnnFilterDescriptor_t filterDesc,
                                         cudnnDataType_t dataType,
                                         cudnnTensorFormat_t format, int k, int c, int h, int w)
```

- Or the generic N-dim form:

```
cudnnStatus_t cudnnSetFilterNdDescriptor(cudnnFilterDescriptor_t filterDesc,
                                         cudnnDataType_t dataType,
                                         cudnnTensorFormat_t format, int nbDims, const int filterDimA[])
```

Convolution descriptors

- Specify the desired behavior for a convolution
- Initialize 2D Convolution with:

```
cudnnStatus_t cudnnSetConvolution2dDescriptor(cudnnConvolutionDescriptor_t convDesc,
                                              int pad_h, int pad_w, int u, int v,
                                              int dilation_h, int dilation_w,
                                              cudnnConvolutionMode_t mode,
                                              cudnnDataType_t computeType)
```

- Or the generic N-dim form:

```
cudnnStatus_t cudnnSetConvolutionNdDescriptor(cudnnConvolutionDescriptor_t convDesc,
                                              int arrayLength, const int padA[],
                                              const int filterStrideA[],
                                              const int dilationA[],
                                              cudnnConvolutionMode_t mode,
                                              cudnnDataType_t dataType)
```

Convolution parameters

- Padding: **zeros implicitly added around the input data** on each dimension
  - The most common case for images is usually half the size of the kernel (e.g., 1 for a 3x3 kernel)
- Stride: increment in each dimension between two consecutive filtering windows
  - To produce one output for each input value, use all ones
- Dilation: used for dilated convolutions
  - All ones, unless dilated convolution is required
- Mode: either CUDNN_CONVOLUTION or CUDNN_CROSS_CORRELATION
- Data type: the precision required for the convolution

Forward convolution

- The generic form for a N-dim convolution is:

```
cudnnStatus_t cudnnConvolutionForward(cudnnHandle_t handle, const void *alpha,
                                      const cudnnTensorDescriptor_t xDesc, const void *x,
                                      const cudnnFilterDescriptor_t wDesc, const void *w,
                                      const cudnnConvolutionDescriptor_t convDesc,
                                      cudnnConvolutionFwdAlgo_t algo,
                                      void *workSpace, size_t workSpaceSizeInBytes,
                                      const void *beta,
                                      const cudnnTensorDescriptor_t yDesc, void *y)
```

- x is the input tensor, and xDesc its descriptor, y the output
- w points to the weights of the filter and wDesc to the filter descriptor
- convDesc is the convolution descriptor
- algo is the desired algorithm (more about it in a moment)
- workSpace is a pointer to a work area of size workSpaceSizeInBytes
- **alpha** and **beta** are scaling factors for input and output (for **ResNets**):
  - dstValue = alpha[0]*result + beta[0]*priorDstValue
  - If not ResNet: alpha = 1, beta = 0

Convolution algorithms

- Multiple algorithms are available
  - Only a subset of the possible combinations is supported
  - Predicting performance of any specific parameter set is not trivial
  - Always refer to the documentation for the specific details of a cuDNN release
- Main algorithm families:
  - **GEMM-based:** transform the computation to the application of matrix multiplications to use GPUs more efficiently
  - **FFT-based:** use the FFT to reduce the computational complexity of convolution, by operating in the transformed domain
  - **Winograd:** use minimal filtering algorithms based on polynomial factorization to reduce the number of operations needed

Selecting an algorithm

- The library provides a primitive to find best algorithm based on preference:

```
cudnnStatus_t cudnnGetConvolutionForwardAlgorithm(cudnnHandle_t handle,
                                          const cudnnTensorDescriptor_t xDesc,
                                          const cudnnFilterDescriptor_t wDesc,
                                          const cudnnConvolutionDescriptor_t convDesc,
                                          const cudnnTensorDescriptor_t yDesc,
                                          cudnnConvolutionFwdPreference_t preference,
                                          size_t memoryLimitInBytes,
                                          cudnnConvolutionFwdAlgo_t *algo)
```

- Where:
  - xDesc, wDesc, convDesc and yDesc specify what kind of convolution we want to perform
  - **preference** and memoryLimitInBytes specify the constraints:
    - CUDNN_CONVOLUTION_FWD_{NO_WORKSPACE,PREFER_FASTEST,SPECIFY_WORKSPACE_LIMIT}
    - The memory limit applies in the latter case

Selection based on benchmarking

- **PyTorch** and other frameworks offer the option of benchmarking multiple algorithms for any combination of input parameters to a convolution
  - Check findAlgorithm() in aten/src/ATen/native/cudnn/Conv.cpp
  - Every time a convolution is attempted with a new input size:
    - try multiple algorithms to find the best
    - reuse best algorithm
  - Cache the results for subsequent invocations
  - Enabled with torch.backends.cudnn.benchmark=True
  - **Not convenient if convolution size changes continously**

Benchmarking through cuDNN

- Further info for algorithm selection is available through cuDNN via:

```
cudnnStatus_t cudnnFindConvolutionForwardAlgorithm(
    cudnnHandle_t                      handle,
    const cudnnTensorDescriptor_t      xDesc,
    const cudnnFilterDescriptor_t      wDesc,
    const cudnnConvolutionDescriptor_t convDesc,
    const cudnnTensorDescriptor_t      yDesc,
    const int                          requestedAlgoCount,
    int                               *returnedAlgoCount,
    cudnnConvolutionFwdAlgoPerf_t     *perfResults)
```

- **This runs multiple algorithms and returns their performance in a list**
- The caller requests at most requestedAlgoCount (making sure there is enough room in perfResults) and the count of the returned performance samples is stored in returnedAlgoCount
- A cudnnFindConvolutionForwardAlgorithmEx() is available, and it allows specifying a custom temporary workspace

Algorithm performance

- The **benchmarking results** for each algorithm are returned in a cudnnConvolutionFwdAlgo_t
- The fields are:
  - **algo**: the algorithm (enum)
  - **status**: an error code, or the return status of the convolution call
  - **time**: execution time (sorting key)
  - **memory**: workspace size
  - **determinism**: a flag specifying if the algorithm is deterministic
  - **mathType**: whether the algorithm uses TensorCores

Workspace size

- Convolution algorithms **may require GPU memory for temporary storage**
  - Availability of enough memory can **limit algorithm selection**
- To determine the memory required by al algorithm on a certain configuration:

```
cudnnStatus_t cudnnGetConvolutionForwardWorkspaceSize(cudnnHandle_t handle,
                                        const cudnnTensorDescriptor_t xDesc,
                                        const cudnnFilterDescriptor_t wDesc,
                                        const cudnnConvolutionDescriptor_t convDesc,
                                        const cudnnTensorDescriptor_t yDesc,
                                        cudnnConvolutionFwdAlgo_t algo, size_t *sizeInBytes)
```

Activation layers

- The operation is described by cudnnActivationDescriptor_t

```
cudnnStatus_t cudnnSetActivationDescriptor(cudnnActivationDescriptor_t activationDesc,
                                           cudnnActivationMode_t mode,
                                           cudnnNanPropagation_t reluNanOpt,double coef)
```

- Where:
  - mode is one of: CUDNN_ACTIVATION_{SIGMOID,RELU,TANH,CLIPPED_RELU,ELU,IDENTITY}
  - reluNanOpt specifies how NaN values are propagated
  - coef specifies the clipping for CLIPPED_RELU or the alpha coefficient for ELU

Activation layers

- The output of an activation layer is calculated with:

```
cudnnStatus_t cudnnActivationForward(cudnnHandle_t handle,
                                     cudnnActivationDescriptor_t activationDesc,
                                     const void *alpha,
                                     const cudnnTensorDescriptor_t xDesc,
                                     const void *x, const void *beta,
                                     const cudnnTensorDescriptor_t yDesc, void *y)
```

- Where:
  - activationDesc specifies the layer parameters
  - x and y are the input and output
  - xDesc and yDesc their descriptors
  - Alpha and beta are scaling factors to combine the previous value of y with the result, if desired

Pooling layers

- The pooling operations are described by a cudnnPoolingDescriptor_t object
- Configured through:

```
cudnnStatus_t cudnnSetPooling2dDescriptor(cudnnPoolingDescriptor_t poolingDesc,
                                          cudnnPoolingMode_t mode,
                                          cudnnNanPropagation_t maxpoolingNanOpt,
                                          int windowHeight, int windowWidth,
                                          int verticalPadding, int horizontalPadding,
                                          int verticalStride, int horizontalStride)
```

- Or:

```
cudnnStatus_t cudnnSetPoolingNdDescriptor(cudnnPoolingDescriptor_t poolingDesc,
                                          int nbDims, const int windowDimA[],
                                          const int paddingA[], const int strideA[])
```

Pooling parameters

- The pooling mode:
  - One of CUDNN_POOLING_{MAX,AVERAGE_COUNT_INCLUDE_PADDING, AVERAGE_COUNT_EXCLUDE_PADDING,MAX_DETERMINISTIC}
  - The average can include or exclude the zero padding from the count
  - The output of MAX can be non-deterministic if atomics are used
- The size of the window being subject to pooling
- The padding applied to the input in each dimension
- The stride for consecutive windows in each dimension

Dropout layers

- Described by a cudnnDropoutDescriptor_t, configured by:

```
cudnnStatus_t cudnnSetDropoutDescriptor(cudnnDropoutDescriptor_t dropoutDesc,
                                        cudnnHandle_t handle, float dropout,
                                        void *states, size_t stateSizeInBytes,
                                        unsigned long long seed)
```

- Where:
  - dropout specifies the probability of zeroing an input element
  - states points to GPU memory to be used to store the PRNG state
  - stateSizeInBytes is the state size
    - Use cudnnGetDropoutStateSize() to determine
  - seed the PRNG seed

Calculating dropout layers

- Performed by:

```
cudnnStatus_t cudnnDropoutForward(cudnnHandle_t handle,
                                   const cudnnDropoutDescriptor_t dropoutDesc,
                                   const cudnnTensorDescriptor_t xdesc, const void *x,
                                   const cudnnTensorDescriptor_t ydesc, void *y,
                                   void *reserveSpace, size_t reserveSpaceSizeInBytes)
```

- Where:
  - ReserveSpace points to a GPU memory area used to exchange information between forward and backward calls to this function
  - ReserveSpaceSizeInBytes is the size of the area
    - Determined by cudnnDropoutGetReserveSpaceSize()
  - Other parameters follow the usual conventions

Applying pooling

- We skip the pooling descriptor here
- Pooling is calculated with:

```
cudnnStatus_t cudnnPoolingForward(cudnnHandle_t handle,
                                  const cudnnPoolingDescriptor_t poolingDesc,
                                  const void *alpha, const cudnnTensorDescriptor_t xDesc,
                                  const void *x, const void *beta,
                                  const cudnnTensorDescriptor_t yDesc, void *y)
```

SoftMax layers

- Don't require a descriptor
- Calculated by:

```
cudnnStatus_t cudnnSoftmaxForward(cudnnHandle_t handle,
                                  cudnnSoftmaxAlgorithm_t algorithm,
                                  cudnnSoftmaxMode_t mode, const void *alpha,
                                  const cudnnTensorDescriptor_t xDesc, const void *x,
                                  const void *beta,
                                  const cudnnTensorDescriptor_t yDesc, void *y)
```

- Where:
  - The algorithm is one of CUDNN_SOFTMAX_{FAST,ACCURATE,LOG}, with FAST being the straightforward algorithm, ACCURATE an implementation avoiding overflows and LOG the logarithmic implementation
  - The mode is one of CUDNN_SOFTMAX_MODE_{INSTANCE,CHANNEL}, with INSTANCE aggregating by N over C, H, W, and CHANNEL aggregating by N, H, W over C
  - The other parameters follow the usual conventions

Lesson key points

- CUDA Memory Access
  - Global Memory
  - Shared Memory
- Matrix Multiplication
  - Simple
  - Tiled
- cuDNN
  - Create and set descriptors for input, output, kernel and algorithm
  - Run convolutions