

中国科学院大学计算机学院专业选修课（硕博通用课程）

GPU架构与编程

第九课：课程大作业讲解（一）

赵地

中科院计算所

2024年秋季学期

讲授内容

➤ 主题演讲

➤ GPU编程补充知识：Triton（二）

➤ 课程大作业讲解

主题演讲

- 《保持热爱 共赴山海》 张建中 摩尔线程智能科技（北京）有限责任公司创始人兼首席执行官
- 时间：2024年11月6日18:10—19:10
- #腾讯会议：981-917-842

讲授内容

- 主题演讲
- GPU编程补充知识：Triton（二）
- 课程大作业讲解

讲授内容：Triton（二）

I. Python API

II. Triton MLIR Dialects and Ops

III. Going Further

triton

<code>jit</code>	Decorator for JIT-compiling a function using the Triton compiler.
<code>autotune</code>	Decorator for auto-tuning a <code>triton.jit</code> 'd function.
<code>heuristics</code>	Decorator for specifying how the values of certain meta-parameters may be computed.
<code>Config</code>	An object that represents a possible kernel configuration for the auto-tuner to try.

triton.language: Programming Model

<code>tensor</code>	Represents an N-dimensional array of values or pointers.
<code>program_id</code>	Returns the id of the current program instance along the given <code>axis</code> .
<code>num_programs</code>	Returns the number of program instances launched along the given <code>axis</code> .

<https://triton-lang.org/>

triton.language: Creation Ops

<code>arange</code>	Returns contiguous values within the half-open interval <code>[start, end)</code> .
<code>cat</code>	Concatenate the given blocks
<code>full</code>	Returns a tensor filled with the scalar value for the given <code>shape</code> and <code>dtype</code> .
<code>zeros</code>	Returns a tensor filled with the scalar value 0 for the given <code>shape</code> and <code>dtype</code> .
<code>zeros_like</code>	Returns a tensor of zeros with the same shape and type as a given tensor.
<code>cast</code>	Casts a tensor to the given <code>dtype</code> .

<https://triton-lang.org/>

triton.language: Shape Manipulation Ops

<code>broadcast</code>	Tries to broadcast the two given blocks to a common compatible shape.
<code>broadcast_to</code>	Tries to broadcast the given tensor to a new <code>shape</code> .
<code>expand_dims</code>	Expand the shape of a tensor, by inserting new length-1 dimensions.
<code>interleave</code>	Interleaves the values of two tensors along their last dimension.
<code>join</code>	Join the given tensors in a new, minor dimension.
<code>permute</code>	Permutes the dimensions of a tensor.
<code>ravel</code>	Returns a contiguous flattened view of <code>x</code> .
<code>reshape</code>	Returns a tensor with the same number of elements as input but with the provided shape.
<code>split</code>	Split a tensor in two along its last dim, which must have size 2.
<code>trans</code>	Permutes the dimensions of a tensor.
<code>view</code>	Returns a tensor with the same elements as <i>input</i> but a different shape.

<https://triton-lang.org/>

triton.language: Linear Algebra Ops

<code>dot</code>	Returns the matrix product of two blocks.
<code>dot_scaled</code>	Returns the matrix product of two blocks in microscaling format.

<https://triton-lang.org/>

triton.language: Memory/Pointer Ops

<code>load</code>	Return a tensor of data whose values are loaded from memory at location defined by <i>pointer</i> :
<code>store</code>	Store a tensor of data into memory locations defined by <i>pointer</i> .
<code>make_block_ptr</code>	Returns a pointer to a block in a parent tensor
<code>advance</code>	Advance a block pointer

<https://triton-lang.org/>

triton.language: Indexing Ops

<code>flip</code>	Flips a tensor <i>x</i> along the dimension <i>dim</i> .
<code>where</code>	Returns a tensor of elements from either <code>x</code> or <code>y</code> , depending on <code>condition</code> .
<code>swizzle2d</code>	Transforms the indices of a row-major <i>size_i</i> * <i>size_j</i> matrix into the indices of a column-major matrix for each group of <i>size_g</i> rows.

<https://triton-lang.org/>

triton.language: Math Ops

<code>abs</code>	Computes the element-wise absolute value of <code>x</code> .
<code>cdiv</code>	Computes the ceiling division of <code>x</code> by <code>div</code> .
<code>ceil</code>	Computes the element-wise ceil of <code>x</code> .
<code>clamp</code>	Clamps the input tensor <code>x</code> within the range <code>[min, max]</code> .
<code>cos</code>	Computes the element-wise cosine of <code>x</code> .
<code>div_rn</code>	Computes the element-wise precise division (rounding to nearest wrt the IEEE standard) of <code>x</code> and <code>y</code> .
<code>erf</code>	Computes the element-wise error function of <code>x</code> .

<https://triton-lang.org/>

triton.language: Math Ops

<code>exp</code>	Computes the element-wise exponential of <code>x</code> .
<code>exp2</code>	Computes the element-wise exponential (base 2) of <code>x</code> .
<code>fdiv</code>	Computes the element-wise fast division of <code>x</code> and <code>y</code> .
<code>floor</code>	Computes the element-wise floor of <code>x</code> .
<code>fma</code>	Computes the element-wise fused multiply-add of <code>x</code> , <code>y</code> , and <code>z</code> .
<code>log</code>	Computes the element-wise natural logarithm of <code>x</code> .
<code>log2</code>	Computes the element-wise logarithm (base 2) of <code>x</code> .

<https://triton-lang.org/>

triton.language: Math Ops

<code>maximum</code>	Computes the element-wise maximum of <code>x</code> and <code>y</code> .
<code>minimum</code>	Computes the element-wise minimum of <code>x</code> and <code>y</code> .
<code>rsqrt</code>	Computes the element-wise inverse square root of <code>x</code> .
<code>sigmoid</code>	Computes the element-wise sigmoid of <code>x</code> .
<code>sin</code>	Computes the element-wise sine of <code>x</code> .
<code>softmax</code>	Computes the element-wise softmax of <code>x</code> .
<code>sqrt</code>	Computes the element-wise fast square root of <code>x</code> .
<code>sqrt_rn</code>	Computes the element-wise precise square root (rounding to nearest wrt the IEEE standard) of <code>x</code> .
<code>umulhi</code>	Computes the element-wise most significant N bits of the 2N-bit product of <code>x</code> and <code>y</code> .

<https://triton-lang.org/>

triton.language: Reduction Ops

<code>argmax</code>	Returns the maximum index of all elements in the <code>input</code> tensor along the provided <code>axis</code> .
<code>argmin</code>	Returns the minimum index of all elements in the <code>input</code> tensor along the provided <code>axis</code> .
<code>max</code>	Returns the maximum of all elements in the <code>input</code> tensor along the provided <code>axis</code> .
<code>min</code>	Returns the minimum of all elements in the <code>input</code> tensor along the provided <code>axis</code> .
<code>reduce</code>	Applies the <code>combine_fn</code> to all elements in <code>input</code> tensors along the provided <code>axis</code> .
<code>sum</code>	Returns the sum of all elements in the <code>input</code> tensor along the provided <code>axis</code> .
<code>xor_sum</code>	Returns the xor sum of all elements in the <code>input</code> tensor along the provided <code>axis</code> .

<https://triton-lang.org/>

triton.language: Scan/Sort Ops

<code>associative_scan</code>	Applies the <code>combine_fn</code> to each elements with a carry in <code>input</code> tensors along the provided <code>axis</code> and update the carry
<code>cumprod</code>	Returns the cumprod of all elements in the <code>input</code> tensor along the provided <code>axis</code>
<code>cumsum</code>	Returns the cumsum of all elements in the <code>input</code> tensor along the provided <code>axis</code>
<code>histogram</code>	computes an histogram based on input tensor with <code>num_bins</code> bins, the bins have a width of 1 and start at 0.
<code>sort</code>	Sorts a tensor along a specified dimension.

<https://triton-lang.org/>

triton.language: Atomic Ops

<code>atomic_add</code>	Performs an atomic add at the memory location specified by <code>pointer</code> .
<code>atomic_and</code>	Performs an atomic logical and at the memory location specified by <code>pointer</code> .
<code>atomic_cas</code>	Performs an atomic compare-and-swap at the memory location specified by <code>pointer</code> .
<code>atomic_max</code>	Performs an atomic max at the memory location specified by <code>pointer</code> .
<code>atomic_min</code>	Performs an atomic min at the memory location specified by <code>pointer</code> .
<code>atomic_or</code>	Performs an atomic logical or at the memory location specified by <code>pointer</code> .
<code>atomic_xchg</code>	Performs an atomic exchange at the memory location specified by <code>pointer</code> .
<code>atomic_xor</code>	Performs an atomic logical xor at the memory location specified by <code>pointer</code> .

<https://triton-lang.org/>

triton.language: Random Number Generation

<code>randint4x</code>	Given a <code>seed</code> scalar and an <code>offset</code> block, returns four blocks of random <code>int32</code> .
<code>randint</code>	Given a <code>seed</code> scalar and an <code>offset</code> block, returns a single block of random <code>int32</code> .
<code>rand</code>	Given a <code>seed</code> scalar and an <code>offset</code> block, returns a block of random <code>float32</code> in $U(0, 1)$.
<code>randn</code>	Given a <code>seed</code> scalar and an <code>offset</code> block, returns a block of random <code>float32</code> in $\mathcal{N}(0, 1)$.

<https://triton-lang.org/>

triton.language: Iterators

<code>range</code>	Iterator that counts upward forever.
<code>static_range</code>	Iterator that counts upward forever.

<https://triton-lang.org/>

triton.language: Inline Assembly

`inline_asm_elementwise`

Execute inline assembly over a tensor.

<https://triton-lang.org/>

triton.language: Compiler Hint Ops

`debug_barrier`

Insert a barrier to synchronize all threads in a block.

`max_constancy`

Let the compiler know that the *value* first values in `input` are constant.

`max_contiguous`

Let the compiler know that the *value* first values in `input` are contiguous.

`multiple_of`

Let the compiler know that the values in `input` are all multiples of `value`.

<https://triton-lang.org/>

triton.language: Debug Ops

<code>static_print</code>	Print the values at compile time.
<code>static_assert</code>	Assert the condition at compile time.
<code>device_print</code>	Print the values at runtime from the device.
<code>device_assert</code>	Assert the condition at runtime from the device.

<https://triton-lang.org/>

triton.testing

<code>Benchmark</code>	This class is used by the <code>perf_report</code> function to generate line plots with a concise API.
<code>do_bench</code>	Benchmark the runtime of the provided function.
<code>do_bench_cudagraph</code>	Benchmark the runtime of the provided function.
<code>perf_report</code>	Mark a function for benchmarking.
<code>assert_close</code>	Asserts that two inputs are close within a certain tolerance.

<https://triton-lang.org/>

讲授内容：Triton（二）

I. Python API

II. Triton MLIR Dialects and Ops

III. Going Further

Triton MLIR Dialects and Ops

Dialects

- 'triton_nvidia_gpu' Dialect
- 'triton_gpu' Dialect
- 'nvgpu' Dialect
- 'tt' Dialect
- 'amdgpu' Dialect

Dialect Ops

- TritonAMDGPUOps
- TritonGPUOps
- TritonOps
- TritonNvidiaGPUOps
- NVGPUOps

<https://triton-lang.org/>

Triton MLIR Dialects and Ops: Dialect Ops: TritonNvidiaGPUOps

- ✓ `triton_nvidia_gpu.async_tma_copy_global_to_local`
(`triton::nvidia_gpu::AsyncTMACopyGlobalToLocalOp`)
- ✓ `triton_nvidia_gpu.async_tma_copy_local_to_global`
(`triton::nvidia_gpu::AsyncTMACopyLocalToGlobalOp`)
- ✓ `triton_nvidia_gpu.barrier_expect`
(`triton::nvidia_gpu::BarrierExpectOp`)
- ✓ `triton_nvidia_gpu.cluster_arrive`
(`triton::nvidia_gpu::ClusterArriveOp`)

<https://triton-lang.org/>

Triton MLIR Dialects and Ops: Dialect Ops: TritonNvidiaGPUOps

- ✓ `triton_nvidia_gpu.cluster_wait`
(`triton::nvidia_gpu::ClusterWaitOp`)
- ✓ `triton_nvidia_gpu.init_barrier`
(`triton::nvidia_gpu::InitBarrierOp`)
- ✓ `triton_nvidia_gpu inval_barrier`
(`triton::nvidia_gpu::InvalBarrierOp`)
- ✓ `triton_nvidia_gpu.async_tma_store_wait`
(`triton::nvidia_gpu::TMAStoreWait`)

<https://triton-lang.org/>

Triton MLIR Dialects and Ops: Dialect Ops: TritonNvidiaGPUOps

- ✓ `triton_nvidia_gpu.wait_barrier` (`triton::nvidia_gpu::WaitBarrierOp`)
- ✓ `triton_nvidia_gpu.warp_group_dot` (`triton::nvidia_gpu::WarpGroupDotOp`)
- ✓ `triton_nvidia_gpu.warp_group_dot_wait` (`triton::nvidia_gpu::WarpGroupDotWaitOp`)

<https://triton-lang.org/>

Triton MLIR Dialects and Ops: Dialect Ops: NVGPUOps

- ✓ `nvgpu.cluster_arrive` (`triton::nvgpu::ClusterArriveOp`)
- ✓ `nvgpu.cluster_id` (`triton::nvgpu::ClusterCTAIdOp`)
- ✓ `nvgpu.cluster_wait` (`triton::nvgpu::ClusterWaitOp`)
- ✓ `nvgpu.fence_async_shared` (`triton::nvgpu::FenceAsyncSharedOp`)
- ✓ `nvgpu.stmatrix` (`triton::nvgpu::StoreMatrixOp`)

<https://triton-lang.org/>

Triton MLIR Dialects and Ops: Dialect Ops: NVGPUOps

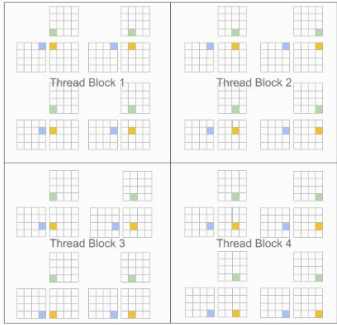
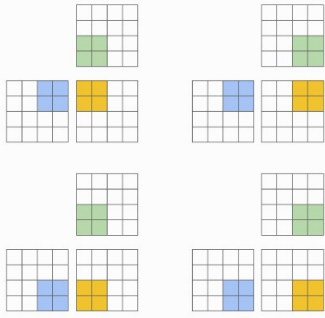
- ✓ `nvgpu.wgmma_commit_group` (`triton::nvgpu::WGMMACommitGroupOp`)
- ✓ `nvgpu.wgmma_fence` (`triton::nvgpu::WGMMAFenceOp`)
- ✓ `nvgpu.wgmma` (`triton::nvgpu::WGMMAOp`)
- ✓ `nvgpu.wgmma_wait_group` (`triton::nvgpu::WGMMAwaitGroupOp`)

<https://triton-lang.org/>

讲授内容：Triton（二）

- I. Python API
- II. Triton MLIR Dialects and Ops
- III. **Going Further**

Going Further: Introduction

CUDA Programming Model (Scalar Program, Blocked Threads)	Triton Programming Model (Blocked Program, Scalar Threads)
<pre>#pragma parallel for(int m = 0; m < M; m++){ #pragma parallel for(int n = 0; n < N; n++){ float acc = 0; for(int k = 0; k < K; k++){ acc += A[m, k] * B[k, n]; } C[m, n] = acc; } }</pre>	<pre>#pragma parallel for(int m = 0; m < M; m += MB){ #pragma parallel for(int n = 0; n < N; n += NB){ float acc[MB, NB] = 0; for(int k = 0; k < K; k += KB){ acc += A[m:m+MB, k:k+KB] * B[k:k+KB, n:n+NB]; } C[m:m+MB, n:n+NB] = acc; } }</pre>
	

<https://triton-lang.org/>

Going Further: Using Triton's Debugging Operations

- ✓ Triton includes four **debugging operators** that allow users to check and inspect tensor values:
 - ✓ **static_print** and **static_assert** are intended for compile-time debugging.
 - ✓ **device_print** and **device_assert** are used for runtime debugging.

<https://triton-lang.org/>

Going Further: Using the Interpreter

- ✓ **Print the intermediate results of each operation using the Python print function.**
 - ✓ **To inspect an entire tensor, use `print(tensor)`. To examine individual tensor values at `idx`, use `print(tensor.handle.data[idx])`.**

<https://triton-lang.org/>

Going Further: Using the Interpreter

- ✓ **Attach `pdb` for step-by-step debugging of the Triton program:**

```
TRITON_INTERPRET=1 pdb main.py  
b main.py:<line number>  
r
```

<https://triton-lang.org/>

Going Further: Using the Interpreter

✓ Import the **pdb** package and set **breakpoints** in the Triton program:

```
import triton
import triton.language as tl
import pdb

@triton.jit
def kernel(x_ptr, y_ptr, BLOCK_SIZE: tl.constexpr):
    pdb.set_trace()
    offs = tl.arange(0, BLOCK_SIZE)
    x = tl.load(x_ptr + offs)
    tl.store(y_ptr + offs, x)
```

<https://triton-lang.org/>

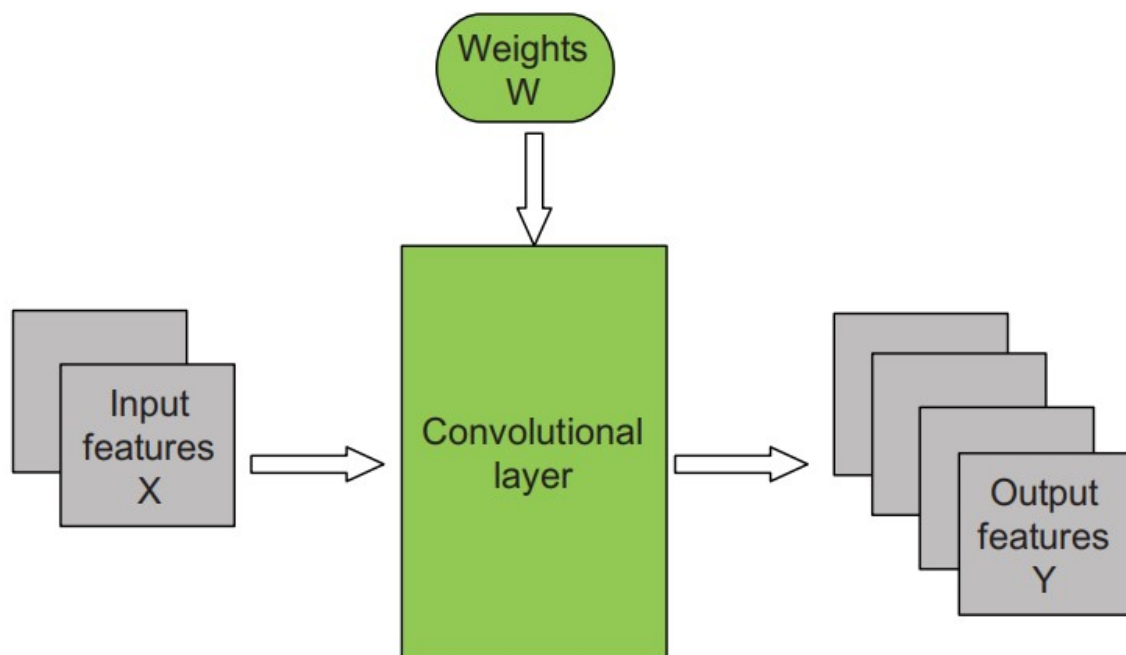
讲授内容

- 主题演讲
- 课程大作业（一）颁奖典礼
- 课程大作业讲解

讲授内容：课程大作业讲解

- **Convolutional Neural Networks: forward propagation**
- **Convolutional Neural Networks: backward propagation**
- **Convolutional Layer: A Basic CUDA Implementation of Forward Propagation**
- **Reduction of convolutional layer to matrix multiplication**

Overview of the forward propagation path of a convolution layer



David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Overview of the forward propagation path of a convolution layer

- We assume that the input feature maps are stored in a 3D array $X[C, H, W]$, where “ C ” is the number of input feature maps, “ H ” is the height of each input map image, and “ W ” is the width of each input map image.
- The highest dimension index selects one of the feature maps; and the lower two dimension indexes selects one of the pixels in a feature map.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Overview of the forward propagation path of a convolution layer

- The output feature maps of a convolutional layer is also stored in a 3D array $Y[M, H-K+1, W-K+1]$, where “ M ” is the number of output feature maps, “ H ” is the height of each input map image, “ W ” is the width of each input map image, and “ K ” is the height (and width) of each filter bank $W[C, M, K, K]$.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Overview of the forward propagation path of a convolution layer

- Two elements are used at each edge of the image, as halo cells, when generating the convolved image.
- There are $M \times C$ filter banks.
- Filter bank $W[m, c, _, _]$ is used for the input feature map $X[c, _, _]$ to calculate the output feature map $Y[m, _, _]$.
- Note that each output feature map is the sum of convolutions of all input feature maps.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Overview of the forward propagation path of a convolution layer

- Therefore, we can consider the forward propagation path of a convolutional layer as a set of M 3D-convolutions, where each 3D-convolution is specified by a 3D filter bank that is a $C \times K \times K$ submatrix of W .
- Note that W is used for both the width of the images and the name of the filter bank matrix.
- In each case, the usage should be clear from the context.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

A sequential implementation of the forward propagation path of a convolution layer

```
void convLayer_forward(int M, int C, int H, int W, int K, float* X, float* W, float* Y)
{
    int m, c, h, w, p, q;
    int H_out = H - K + 1;
    int W_out = W - K + 1;

    for(m = 0; m < M; m++)           // for each output feature maps
        for(h = 0; h < H_out; h++)   // for each output element
            for(w = 0; w < W_out; w++) {
                Y[m, h, w] = 0;
                for(c = 0; c < C; c++) // sum over all input feature maps
                    for(p = 0; p < K; p++) // KxK filter
                        for(q = 0; q < K; q++)
                            Y[m, h, w] += X[c, h + p, w + q] * W[m, c, p, q];
            }
}
```

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

A sequential implementation of the forward propagation path of a convolution layer

- Each iteration of the outermost (***m***) for-loop generates an output feature map.
- Each of the next two levels (***h*** and ***w***) of for-loops generates one pixel of the current output feature map.
- The three innermost levels perform the 3D convolution between the input feature maps and the 3D filter banks.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

A sequential C implementation of the forward propagation path of a subsampling layer

```
void poolingLayer_forward(int M, int H, int W, int K, float* Y, float* S)
{
    int m, h, w, p, q;
    for(m = 0; m < M; m++)           // for each output feature maps
        for(h = 0; h < H/K; h++)      // for each output element
            for(w = 0; w < W/K; w++) {
                S[m, h, w] = 0.;
                for(p = 0; p < K; p++) { // loop over KxK input samples
                    for(q = 0; q < K; q++)
                        S[m, h, w] = S[m, h, w] + Y[m, K*x + p, K*y + q]/(K*K);
                }
            }
    // add bias and apply non-linear activation
    S[m, h, w] = sigmoid(S[m, h, w] + b[m])
}
}
```

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

A sequential C implementation of the forward propagation path of a subsampling layer

- Each iteration of the outermost (*m*) for-loop generates an output feature map.
- The next two levels (*h*, *w*) of for-loops generate individual pixels of the current output map.
- The two innermost for-loops sum up the pixels in the neighborhood.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

A sequential C implementation of the forward propagation path of a subsampling layer

- A bias value $b[m]$ that is specific to each output feature map is then added to each output feature map, and the sum goes through a nonlinear function such as the tanh, sigmoid, or ReLU functions to provide the output pixel values a more desirable distribution.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

讲授内容：课程大作业讲解

- Convolutional Neural Networks: forward propagation
- Convolutional Neural Networks: backward propagation
- Convolutional Layer: A Basic CUDA Implementation of Forward Propagation
- Reduction of convolutional layer to matrix multiplication

Convolutional Neural Networks: backward propagation

- **Training of ConvNets is based on a procedure called gradient backpropagation.**
- **The training data set is labeled with the “correct answer”.**

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Convolutional Neural Networks: backward propagation

- **For each training image, the final stage of the network calculates the loss function or the error as the difference between the generated output vector element values and the “correct” output vector element values.**

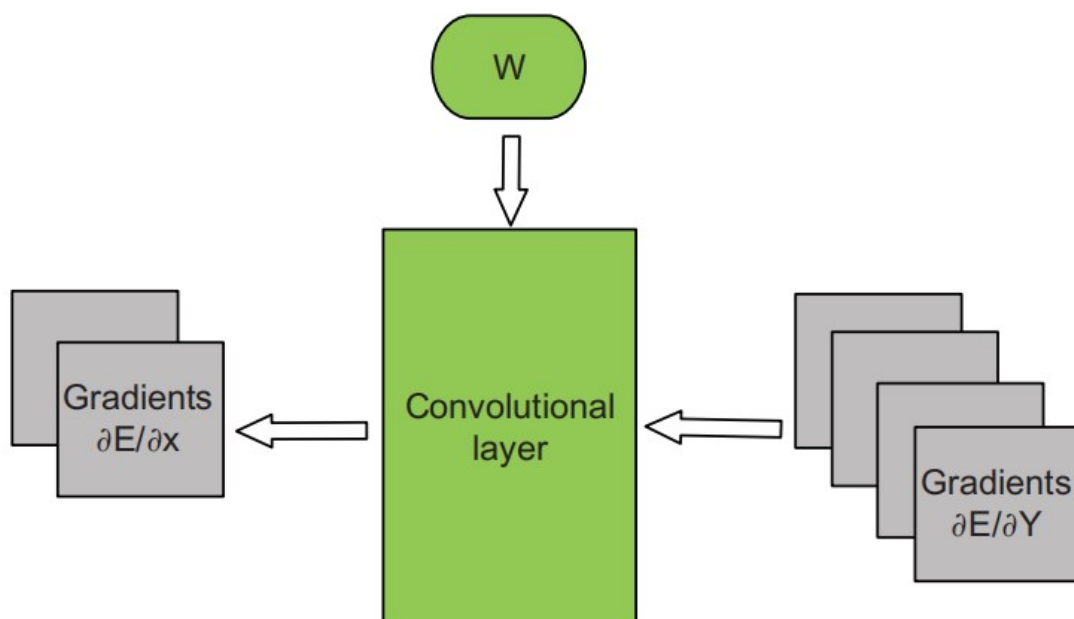
David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Convolutional Neural Networks: backward propagation

- Given a sequence of training images, we can calculate the gradient of loss function with respect to the elements of the output vector.
- Intuitively, it gives the rate at which the loss function value changes when the values of the output vector elements change.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Convolutional layer: Backpropagation of $\partial E / \partial X$



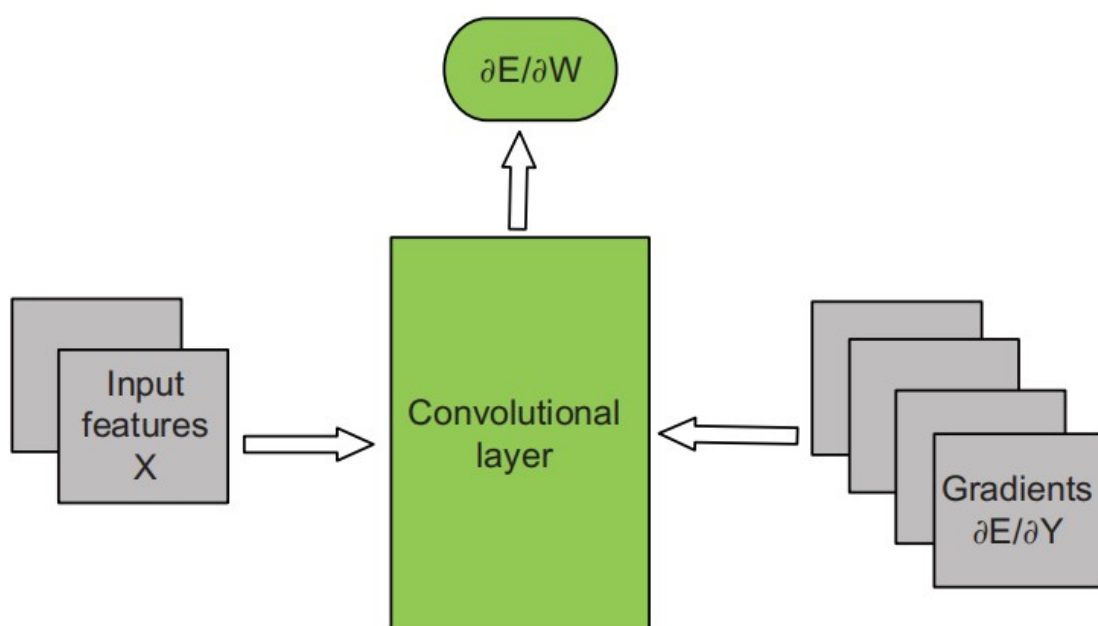
David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Convolutional layer: Backpropagation of $\partial E / \partial X$

- Backpropagation starts by calculating the gradient of loss function $\partial E / \partial Y$ for the last layer.
- This process then propagates the gradient from the last layer toward the first layer through all layers of the network.
- Each layer receives as its input $\partial E / \partial Y$ —gradient with respect to its output feature maps and calculates $\partial E / \partial X$ —gradient with respect to its input feature maps.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Convolutional layer: Backpropagation of $\partial E / \partial w$



David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Convolutional layer: Backpropagation of $\partial E / \partial w$

- If a layer has learned parameters (“weights”) W , then the layer also calculates $\partial E / \partial W$ —gradient of loss with respect to weights.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Convolutional Neural Networks: backward propagation

- For instance, the fully connected layer is given as $Y = W \times X$.
- The backpropagation of gradient $\partial E / \partial Y$ is expressed by two equations:

$$\frac{\partial E}{\partial X} = W^T * \frac{\partial E}{\partial Y} \quad \text{and} \quad \frac{\partial E}{\partial W} = \frac{\partial E}{\partial Y} * X^T$$

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

dE/dX calculation of the backward path of a convolution layer

```
void convLayer_backward_xgrad(int M, int C, int H_in, int W_in, int K,
    float* dE_dY, float* W, float* dE_dX)
{
    int m, c, h, w, p, q;
    int H_out = H_in - K + 1;
    int W_out = W_in - K + 1;
    for(c = 0; c < C; c++)
        for(h = 0; h < H_in; h++)
            for(w = 0; w < W_in; w++)
                dE_dX[c, h, w] = 0.;

    for(m = 0; m < M; m++)
        for(h = 0; h < H_out; h++)
            for(w = 0; w < W_out; w++)
                for(c = 0; c < C; c++)
                    for(p = 0; p < K; p++)
                        for(q = 0; q < K; q++)
                            dE_dX[c, h + p, w + q] += dE_dY[m, h, w] * W[m, c, p, q];
}
```

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

$\partial E/\partial X$ calculation of the backward path of a convolution layer

- We will now describe backpropagation for a convolutional layer, starting with the calculation of $\partial E/\partial X$.
- Note that the calculation of $\partial E/\partial X$ is important for propagating the gradient to the previous layer.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

$\partial E/\partial X$ calculation of the backward path of a convolution layer

- The gradient $\partial E/\partial X$ with respect to the channel c of input X is given as the sum of “backward convolution” with corresponding $WT(c, m)$ over all layer outputs m :

$$\frac{\partial E}{\partial X}(c, h, w) = \sum_{m=1}^M \sum_{p=1}^k \sum_{q=1}^k \left(W(p, q) * \frac{\partial E}{\partial Y}(h - p, w - q) \right)$$

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

$\partial E/\partial X$ calculation of the backward path of a convolution layer

- The figure in **page 29** demonstrates the calculation of the $\partial E/\partial X$ function in the form of one matrix for each input feature map.
- The code assumes that $\partial E/\partial Y$ has been calculated for all the output feature maps of the layer and passed with a pointer argument $\partial E_ \partial Y$.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

$\partial E / \partial X$ calculation of the backward path of a convolution layer

- It also assumes that the space of $\partial E / \partial X$ has been allocated in the device memory whose handle is passed as a pointer argument.
- The kernel will be generating the elements of $\partial E / \partial X$.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

$\partial E / \partial W$ calculation of the backward path of a convolutional layer

```
void convLayer_backward_wgrad(int M, int C, int H, int W, int K,
                             float* dE_dY, float* X, float* dE_dW)
{
    int m, c, h, w, p, q;
    int H_out = H - K + 1;
    int W_out = W - K + 1;
    for(m = 0; m < M; m++)
        for(c = 0; c < C; c++)
            for(p = 0; p < K; p++)
                for(q = 0; q < K; q++)
                    dE_dW[m, c, p, q] = 0.;

    for(m = 0; m < M; m++)
        for(h = 0; h < H_out; h++)
            for(w = 0; w < W_out; w++)
                for(c = 0; c < C; c++)
                    for(p = 0; p < K; p++)
                        for(q = 0; q < K; q++)
                            dE_dW[m, c, p, q] += X[c, h + p, w + q] * dE_dY[m, c, h, w];
}
```

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

$\partial E / \partial W$ calculation of the backward path of a convolutional layer

- The algorithm for calculating $\partial E / \partial W$ for a convolution layer computation is very similar to that of $\partial E / \partial X$.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

$\partial E / \partial W$ calculation of the backward path of a convolutional layer

- Since each $W(c, m)$ affects all elements of the output $Y(m)$, we should accumulate gradients over all pixels in the corresponding output feature map:

$$\frac{\partial E}{\partial W}(c, m; p, q) = \sum_{h=1}^{H_{out}} \sum_{w=1}^{W_{out}} \left(X(h + p, w + q) * \frac{\partial E}{\partial Y}(h, w) \right)$$

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

$\partial E/\partial W$ calculation of the backward path of a convolutional layer

- Note that while the calculation of $\partial E/\partial X$ is important for propagating the gradient to the previous layer, the calculation of $\partial E/\partial W$ is key to the weight value adjustments of the current layer.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

$\partial E/\partial W$ calculation of the backward path of a convolutional layer

- After the $\partial E/\partial W$ values at all positions of feature map elements are evaluated, weights are updated iteratively to minimize the expected error: $W(t+1) = W(t) - \lambda \times \partial E/\partial W$, where λ is a constant called the learning rate.
- The initial value of λ is set empirically and reduced through the iterations in accordance with the rule defined by the user.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

$\partial E / \partial W$ calculation of the backward path of a convolutional layer

- The value of λ is reduced through the iterations to ensure the convergence to a minimal error.
- The negative sign of the adjustment term makes the change opposite to the direction of the gradient so that the change will likely reduce the error.
- Recall that the weight values of the layers determine how the input is transformed through the network.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

$\partial E / \partial W$ calculation of the backward path of a convolutional layer

- This adjustment of the weight values of all the layers adapts the behavior of the network, i.e. the network “learns” from a sequence of labeled training data and adapts its behavior by adjusting weight values at all layers.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

$\partial E/\partial W$ calculation of the backward path of a convolutional layer

- The training data sets are usually large; thus, the training of ConvNets is typically accomplished using **Stochastic Gradient Descent**.
- Instead of performing a forward–backward step to determine $\partial E/\partial W$ for the whole training data set, one randomly selects a small subset (“**mini-batch**”) of N images from the training data set and computes the gradient only for this subset.
- Subsequently, one selects another subset, and so on.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

$\partial E/\partial W$ calculation of the backward path of a convolutional layer

- If we would work by the “optimization book”, we should return samples to the training set and then build a new mini-batch by randomly picking subsequent samples.
- In practice, we go sequentially over the entire training set.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

$\partial E / \partial W$ calculation of the backward path of a convolutional layer

- We then shuffle the entire training set and start the subsequent epoch.
- This procedure adds one additional dimension to all data arrays with n —the index of the sample in the mini-batch.
- It also adds another loop over samples.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Forward path of a convolutional layer with mini-batch training

```
void convLayer_forward(int N, int M, int C, int H, int W, int K, float* X, float* W, float* Y)
{
    int n, m, c, h, w, p, q;
    int H_out = H - K + 1;
    int W_out = W - K + 1;
    for(n = 0; n < N; n++)           // for each sample in the mini-batch
        for(m = 0; m < M; m++)       // for each output feature maps
            for(h = 0; h < H_out; h++) // for each output element
                for(w = 0; w < W_out; w++) {
                    Y[n, m, h, w] = 0;
                    for(c = 0; c < C; c++) // sum over all input feature maps
                        for(p = 0; p < K; p++) // KxK filter
                            for(q = 0; q < K; q++)
                                Y[n, m, h, w] += X[n, c, h + p, w + q] * W[m, c, p, q];
                }
    }
```

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Forward path of a convolutional layer with mini-batch training

- The figure in page 45 shows the revised forward path implementation of a convolutional layer.
- It generates the output feature maps for all samples of a **mini-batch**.
- During backpropagation, one first computes for the average gradient of the error with respect to the weights of the last layer over all samples in a mini-batch.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Forward path of a convolutional layer with mini-batch training

- The gradient is then propagated backward through the layers and used to adjust all the weights.
- Each iteration of the weight adjustment processes one **mini-batch**.
- The training is measured in epochs, where one epoch is a sequential pass over all the samples in the training data set.
- The training data set is typically **reshuffled** between epochs.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

讲授内容：课程大作业讲解

- **Convolutional Neural Networks: forward propagation**
- **Convolutional Neural Networks: backward propagation**
- **Convolutional Layer: A Basic CUDA Implementation of Forward Propagation**
- **Reduction of convolutional layer to matrix multiplication**

Parallelization of the forward path of a convolutional layer with mini-batch training

```
void convLayer_forward(int N, int M, int C, int H, int W, int K, float* X, float* W, float* Y)
{
    int n, m, c, h, w, p, q;
    int H_out = H - K + 1;
    int W_out = W - K + 1;
    parallel_for(n = 0; n < N; n++)
        parallel_for(m = 0; m < M; m++)
            parallel_for(h = 0; h < H_out; h++)
                parallel_for(w = 0; w < W_out; w++) {
                    Y[n, m, h, w] = 0;
                    for (c = 0; c < C; c++)
                        for (p = 0; p < K; p++)
                            for (q = 0; q < K; q++)
                                Y[n, m, h, w] += X[n, c, h + p, w + q] * W[m, c, p, q];
                }
}
```

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Parallelization of the forward path of a convolutional layer with mini-batch training

- **The computation pattern in training a convolutional network is highly similar to matrix multiplication: compute-intensive and highly parallel.**
- **We can compute in different parallel samples in a mini-batch, different output feature maps for the same sample, and different elements for each output feature map.**

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Parallelization of the forward path of a convolutional layer with mini-batch training

- **The figure in page 49 presents a conceptual parallel code for the forward path of a convolutional layer.**
- **Each `parallel_for` loop indicates that all its iterations can be executed in parallel.**

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Parallelization of the forward path of a convolutional layer with mini-batch training

- As shown in the figure in page 49, the parallelism in the forward-path convolutional layer has four levels.
- The total number of parallel iterations is the product $N \times M \times H_{out} \times W_{out}$.
- This high degree of available parallelism makes ConvNets an excellent candidate for GPU acceleration.
- To illustrate, forward path for a convolutional layer is implemented.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Parallelization of the forward path of a convolutional layer with mini-batch training

- We will refine the high-level parallel code into a kernel by making some high level design decisions.
- Assume that each thread will compute one element of one output feature map.
- We will use 2D thread blocks, with each thread block computing a tile of $TILE_WIDTH \times TILE_WIDTH$ elements in one output feature map.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Parallelization of the forward path of a convolutional layer with mini-batch training

- **Blocks will be organized into a 3D grid:**
 - The first dimension (X) of the grid corresponds to samples (N) in the batch;
 - The second dimension (Y) corresponds to the (M) output features maps;
 - The last dimension (Z) will define the location of the output tile inside the output feature map.
- The last dimension Z depends on the number of tiles in the horizontal and vertical dimensions of the output image.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Parallelization of the forward path of a convolutional layer with mini-batch training

- Assume for simplicity that **H_{out}** (height of the output image) and **W_{out}** (width of the output image) are multiples of the tile width (set to 16 below):

```
# define TILE_WIDTH 16
W_grid = W_out/TILE_WIDTH; // number of horizontal tiles per output map
H_grid = H_out/TILE_WIDTH; // number of vertical tiles per output map
Z = H_grid * W_grid;
dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
dim3 gridDim(N, M, Z);
ConvLayerForward_Kernel<<< gridDim, blockDim>>>(...);
```

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Parallelization of the forward path of a convolutional layer with mini-batch training

- As previously discussed, each thread block is responsible for computing one 16×16 tile in the output $Y(n, c, \cdot, \cdot)$, and each thread will compute one element $Y[n, m, h, w]$ where:

```
n = blockIdx.x;
m = blockIdx.y;
h = blockIdx.z / W_grid + threadIdx.y;
w = blockIdx.z % W_grid + threadIdx.x;
```

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Parallelization of the forward path of a convolutional layer with mini-batch training

- The kernel in the figure in page 49 exhibits a considerably high degree of parallelism but consumes excessive global memory bandwidth.
- Like in the convolution-based pattern, the execution speed of the kernel will be limited by the global memory bandwidth.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Parallelization of the forward path of a convolutional layer with mini-batch training with **shared memory**

```
__global__ void
ConvLayerForward_Kernel(int C, int W_grid, int K, float* X, float* W, float* Y)
{
    int n, m, h0, w0, h_base, w_base, h, w;
    int X_tile_width = TILE_WIDTH + K - 1;
    extern __shared__ float shmem[];
    float* X_shared = &shmem[0];
    float* W_shared = &shmem[X_tile_width * X_tile_width];
    n = blockIdx.x;
    m = blockIdx.y;
    h0 = threadIdx.x; // h0 and w0 used as shorthand for threadIdx.x and threadIdx.y
    w0 = threadIdx.y;
    h_base = (blockIdx.z / W_grid) * TILE_SIZE; // vertical base out data index for the block
    w_base = (blockIdx.z % W_grid) * TILE_SIZE; // horizontal base out data index for the block
    h = h_base + h0;
    w = w_base + w0;

    float acc = 0;
    int c, i, j, p, q;
```

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Parallelization of the forward path of a convolutional layer with mini-batch training with **shared memory**

```
    for (c = 0; c < C; c++) { // sum over all input channels

        if ((h0 < K) && (w0 < K))
            W_shared[h0, w0] = W[m, c, h0, w0]; // load weights for W[m, c, ..],
        __syncthreads() // h0 and w0 used as shorthand for threadIdx.x
                        // and threadIdx.y

        for (i = h; i < h_base + X_tile_width; i += TILE_WIDTH) {
            for (j = w; j < w_base + X_tile_width; j += TILE_WIDTH)
                X_shared[i - h_base, j - w_base] = X[n, c, h, w]
        } // load tile from X[n, c, ...] into shared memory

        __syncthreads();
        for (p = 0; p < K; p++) {
            for (q = 0; q < K; q++)
                acc = acc + X_shared[h + p, w + q] * W_shared[p, q];
        }
        __syncthreads();
    }
    Y[n, m, h, w] = acc;
}
```

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Parallelization of the forward path of a convolutional layer with mini-batch training with **shared memory**

- We will now modify the basic kernel to reduce traffic to global memory.
- We can use **shared memory** tiling to dramatically improve the execution speed of the kernel:
 - Load the filter $W[m, c]$ into the shared memory.
 - All threads collaborate to copy the portion of the input $X[n, c, \dots]$ that is required to compute the output tile into the shared memory array X_shared .
 - Compute for the partial sum of output $Y_shared[n, m, \dots]$.
 - Move to the next input channel c .

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Parallelization of the forward path of a convolutional layer with mini-batch training with **shared memory**

- We need to allocate shared memory for the input block $X_tile_width \times X_tile_width$, where $X_tile_width = TILE_WIDTH + K - 1$.
- We also need to allocate shared memory for $K \times K$ filter coefficients.
- Thus, the total amount of shared memory will be $(TILE_WIDTH + K - 1) \times (TILE_WIDTH + K - 1) + K \times K$.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Parallelization of the forward path of a convolutional layer with mini-batch training with **shared memory**

- Since we do not know K at compile time, we need to add it to the kernel definition as the third parameter.

```
...
size_t shmem_size = sizeof(float) * ( (TILE_WIDTH + K-1)*(TILE_
WIDTH + K-1) + K*K );
ConvLayerForward_Kernel<<< gridDim, blockDim, shmem_size>>>(...);
...
```

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Parallelization of the forward path of a convolutional layer with mini-batch training with **shared memory**

- We will divide the shared memory between the input buffer and the filter inside the kernel.
- The first $X_tile_width \times X_tile_width$ entries are allocated to the input tiles, and the remaining entries are allocated to the weight values.
- The use of shared memory tiling leads to a considerably high level of acceleration in the execution of the kernel.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

讲授内容：课程大作业讲解

- **Convolutional Neural Networks: forward propagation**
- **Convolutional Neural Networks: backward propagation**
- **Convolutional Layer: A Basic CUDA Implementation of Forward Propagation**
- **Reduction of convolutional layer to matrix multiplication**

Reduction of convolutional layer to matrix multiplication

- **We can build an even faster convolutional layer by reducing it to matrix multiplication and then using highly efficient matrix multiplication, GEneral Matrix to Matrix Multiplication (GEMM).**
- **The central idea is: unfolding and replicating the inputs to the convolutional kernel such that all elements needed to compute one output element will be stored as one sequential block.**

Reduction of convolutional layer to matrix multiplication

- The matrix version of this layer will be constructed as follows:
- First, we will rearrange all input elements.
 - Since the results of the convolutions are summed across input features, the input features can be concatenated into one large matrix.
 - Each row of this matrix contains all input values necessary to compute one element of an output feature.
 - This process means that each input element will be replicated multiple times.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Reduction of convolutional layer to matrix multiplication

- In general, the size of the expanded (unrolled) input feature map matrix can be derived by considering the number of input feature map elements required to generate each output feature map element.
- In general, the height (or the number of rows) of the expanded matrix is the number of input feature elements contributing to each output feature map element.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Reduction of convolutional layer to matrix multiplication

- The number is $C \times K \times K$: each output element is the convolution of $K \times K$ elements from each input feature map, and there are C input feature maps.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Reduction of convolutional layer to matrix multiplication

- The width, or the number column, of the expanded matrix should be the number of elements in each output feature map.
- Assuming that the output feature maps are $H_out \times W_out$ matrices, the number of columns of the expanded matrix is $H_out \times W_out$.
- The number of output feature maps M does not affect the duplication as all output feature maps share the same expanded matrix.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Reduction of convolutional layer to matrix multiplication

- The ratio of expansion for the input feature maps is the size of the expanded matrix over the total size of the original input feature maps.
- The reader should verify that the expansion ratio is $(K \times K \times H_{out} \times W_{out}) / (H_{in} \times W_{in})$, where H_{in} and W_{in} denote the height and width of each input feature map, respectively.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Reduction of convolutional layer to matrix multiplication

- The filter banks are represented as a filter-bank matrix in a fully linearized layout, where each row contains all weight values needed to produce one output feature map.
- The height of the filter-bank matrix is the number of output feature maps (M).
- The height of the filter-bank matrix allows the output feature maps to share a single expanded input matrix.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Reduction of convolutional layer to matrix multiplication

- Meanwhile, the width of the filter-bank matrix is the number of weight values needed to generate each output feature map element, which is $C \times K \times K$.
- Note that no duplication occurs when placing the weight values into the filter-banks matrix.

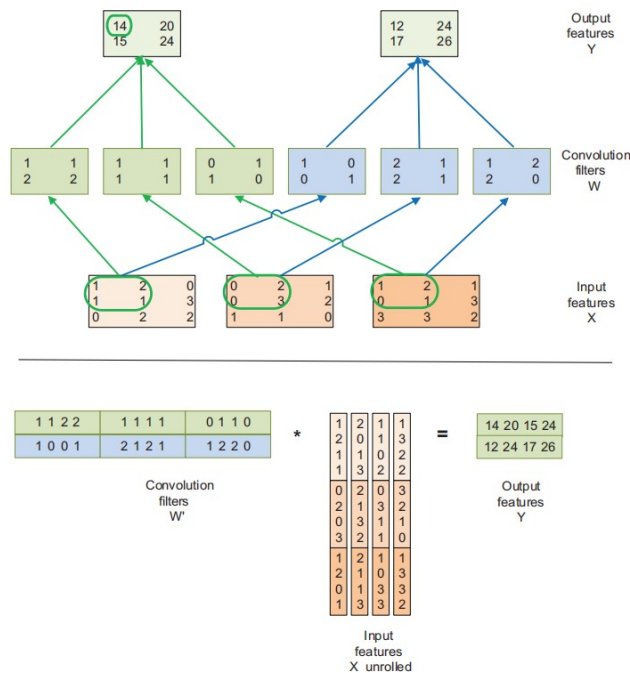
David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Reduction of convolutional layer to matrix multiplication

- When the filter-bank matrix W is multiplied by the expanded input matrix $X_{unrolled}$, the output features Y are computed as one large matrix of height M and width $H_{out} \times W_{out}$.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Reduction of convolutional layer to matrix multiplication



David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Reduction of convolutional layer to matrix multiplication

- The discussion that follows is on the method of implementing this algorithm in CUDA:
 - We assume that the input feature map samples in a mini-batch will be supplied in the same way as that for the basic CUDA kernel.
 - It is organized as an $N \times C \times H \times W$ array, where N is the number of samples in a mini-batch, C is the number of input feature maps, H is the height of each input feature map, and W is the width of each input feature map.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Reduction of convolutional layer to matrix multiplication

- The discussion that follows is on the method of implementing this algorithm in CUDA:
 - the matrix multiplication will naturally generate an output \mathbf{Y} stored as an $M \times H_{out} \times W_{out}$ array.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Reduction of convolutional layer to matrix multiplication

- The discussion that follows is on the method of implementing this algorithm in CUDA:
 - Since the filter-bank matrix does not involve duplication of weight values, we assume that it will be prepared as early and organized as an $M \times C \times (K \times K)$ array.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Reduction of convolutional layer to matrix multiplication

- The preparation of the expanded input feature map matrix ***X_{unroll}*** involves greater complexity.
- Since each expansion increases the size of the input by approximately up to $K \times K$ times, the expansion ratio can be very large for typical K values of 5 or larger.
- The memory footprint for keeping all sample input feature maps for a mini-batch can be prohibitively large.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Reduction of convolutional layer to matrix multiplication

- To reduce the memory footprint, we will allocate only one buffer for ***$X_{unrolled}$*** [$C \times K \times K \times H_{out} \times W_{out}$].
- We will reuse this buffer by adding a loop over samples in the batch.
- During each iteration, we will convert the simple input feature map from its original form into the expanded matrix.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Implementing the forward path of a convolutional layer with matrix multiplication

```
void convLayer_forward(int N, int M, int C, int H, int W, int K, float* X, float* W_unroll, float* Y)
{
    int W_out = W - K + 1;
    int H_out = H - K + 1;
    int W_unroll = C * K * K;
    int H_unroll = H_out * W_out;
    float* X_unrolled = malloc(W_unroll * H_unroll * sizeof(float));
    for (int n=0; n < N; n++) {
        unroll(C, H, W, K, n, X, X_unrolled);
        gemm(H_unroll, M, W_unroll, X_unrolled, W, Y[n]);
    }
}
```

David B. Kirk, Wen-mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, third Edition, Morgan Kaufmann;

The function that generates the unrolled X matrix.

```
void unroll(int C, int H, int W, int K, float* X, float* X_unroll)
{
    int c, h, w, p, q, w_base, w_unroll, h_unroll;
    int H_out = H - K + 1;
    int W_out = W - K + 1;
    for(c = 0; c < C; c++) {
        w_base = c * (K*K);
        for(p = 0; p < K; p++)
            for(q = 0; q < K; q++) {
                for(h = 0; h < H_out; h++)
                    for(w = 0; w < W_out; w++){
                        w_unroll = w_base + p * K + q;
                        h_unroll = h * W_out + w;
                        X_unroll(h_unroll, w_unroll) = X(c, h + p, w + q);
                    }
            }
    }
}
```

David B. Kirk, Wen-mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, third Edition, Morgan Kaufmann;

The function that generates the unrolled X matrix.

- The figure in page 79 shows a sequential function that produces the X_{unroll} array by gathering and duplicating the elements of an input feature map X .
- The function uses five levels of loops.
- The two innermost levels of the for-loop (w and h) place one element of the input feature map for each of the output feature map elements.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

The function that generates the unrolled X matrix.

- The next two levels repeat the process for each of the $K \times K$ element of the input feature map for the filtering operations.
- The outermost loop repeats the process of all input feature maps.
- This implementation is conceptually straightforward and can be quite easily parallelized since the loops do not impose dependencies among the iterations.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

The function that generates the unrolled X matrix.

- In addition, successive iterations of the innermost loop read from a localized tile of one of the input feature maps in X and write into sequential locations in the expanded matrix $X_{unrolled}$.
- This process should result in efficient usage of memory bandwidth on a CPU.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Host code for invoking the unroll kernel

```
void unroll_gpu(int C, int H, int W, int K, float* X, float* X_unroll)
{
    int H_out = H - K + 1;
    int W_out = W - K + 1;
    int num_threads = C * H_out * W_out;
    int num_blocks = ceil((C * H_out * W_out) / CUDA_MAX_NUM_THREADS);
    unroll_Kernel<<<num_blocks, CUDA_MAX_NUM_THREADS>>>();
}
```

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Host code for invoking the unroll kernel

- We are now ready to design a CUDA kernel that implements the input feature map unrolling.
- Each CUDA thread will be responsible for gathering ($K \times K$) input elements from one input feature map for one element of an output feature map.
- The total number of threads will be $(C \times H_{out} \times W_{out})$.

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

Host code for invoking the unroll kernel

- We will use one-dimensional blocks.
- If we assume that a maximum number of threads per block is `CUDA_MAX_NUM_THREADS` (e.g., 1024), the total number of blocks in the grid will be

$$\text{num_blocks} = \text{ceil}((C \times H_{out} \times W_{out}) / \text{CUDA_MAX_NUM_THREADS}) .$$

David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, third Edition, Morgan Kaufmann;

High-performance implementation of the unroll kernel

```
__global__ void unroll_Kernel(int C, int H, int W, int K, float* X, float* X_unroll)
{
    int c, s, h_out, w_out, h_unroll, w_base, p, q;
    int t = blockIdx.x * CUDA_MAX_NUM_THREADS + threadIdx.x;
    int H_out = H - K + 1;
    int W_out = W - K + 1;
    int W_unroll = H_out * W_out;

    if (t < C * W_unroll) {
        c = t / W_unroll;
        s = t % W_unroll;
        h_out = s / W_out;
        w_out = s % W_out;
        h_unroll = h_out * W_out + w_out;
        w_base = c * K * K;
        for(p = 0; p < K; p++)
            for(q = 0; q < K; q++) {
                w_unroll = w_base + p * K + q;
                X_unroll(h_unroll, w_unroll) = X(c, h_out + p, w_out + q);
            }
    }
}
```

David B. Kirk, Wen-mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, third Edition, Morgan Kaufmann;

High-performance implementation of the unroll kernel

- Figure in page 87 illustrates an implementation of the unroll kernel.
- Each thread will build a $K \times K$ section of a column, shown as a shaded box in the Input Features *X_Unrolled* array in page 72.

David B. Kirk, Wen-mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, third Edition, Morgan Kaufmann;

THANKS