



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 12: Computation in Deep Neural Networks

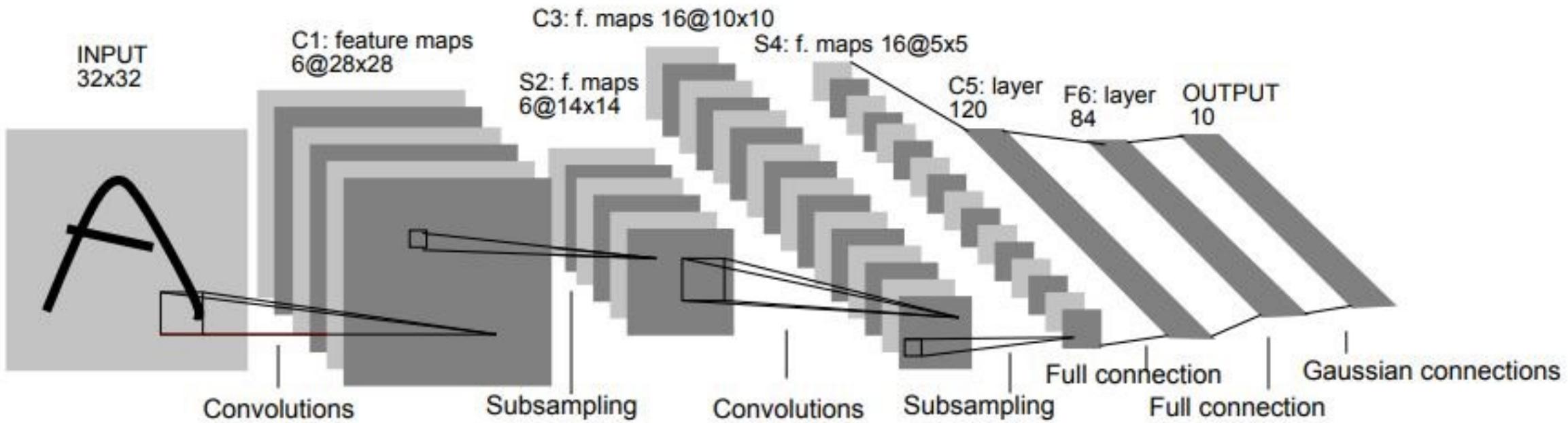
Course Reminders

- Lab 4 is due this week
- Project Milestone 1: Baseline CPU implementation is due Friday March 10th
 - Project details are posted on the wiki
- Midterm 1 is on Tuesday, March 7th
 - On-line, everybody will be taking it at the same time
 - Tuesday, March 7th 7:00pm-8:30pm US Central time
 - If you have a conflict with this time, email me by March 1st
 - Includes materials from Lecture 1 through Lecture 9
- Lecture 13 is recorded, please watch it at your own convenience
 - No in-class or on-line lecture

Objective

- To learn to implement the different types of layers in a Convolutional Neural Network (CNN)

LeNet-5:CNN for hand-written digit recognition



Anatomy of a Convolution Layer

Input features

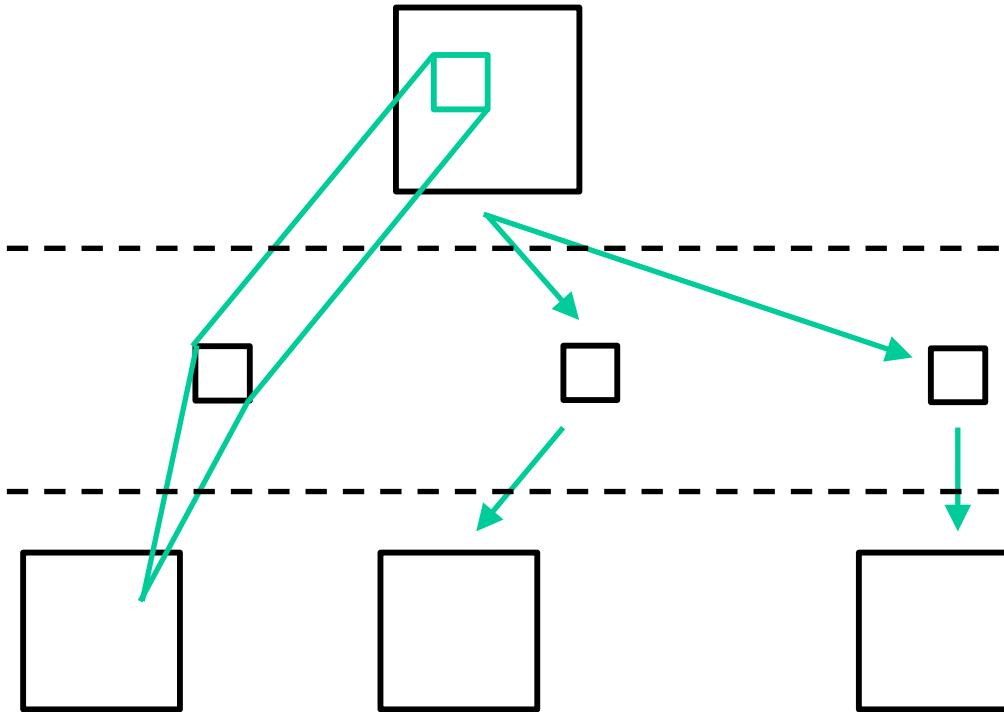
- A inputs each $N_1 \times N_2$

Convolution Layer

- B convolution kernels each $K_1 \times K_2$

Output Features (total of B)

- $A \times B$ outputs each
 $(N_1 - K_1 + 1) \times (N_2 - K_2 + 1)$



Notion of a Channel in Input Layer

Some Set of Input Features are Related

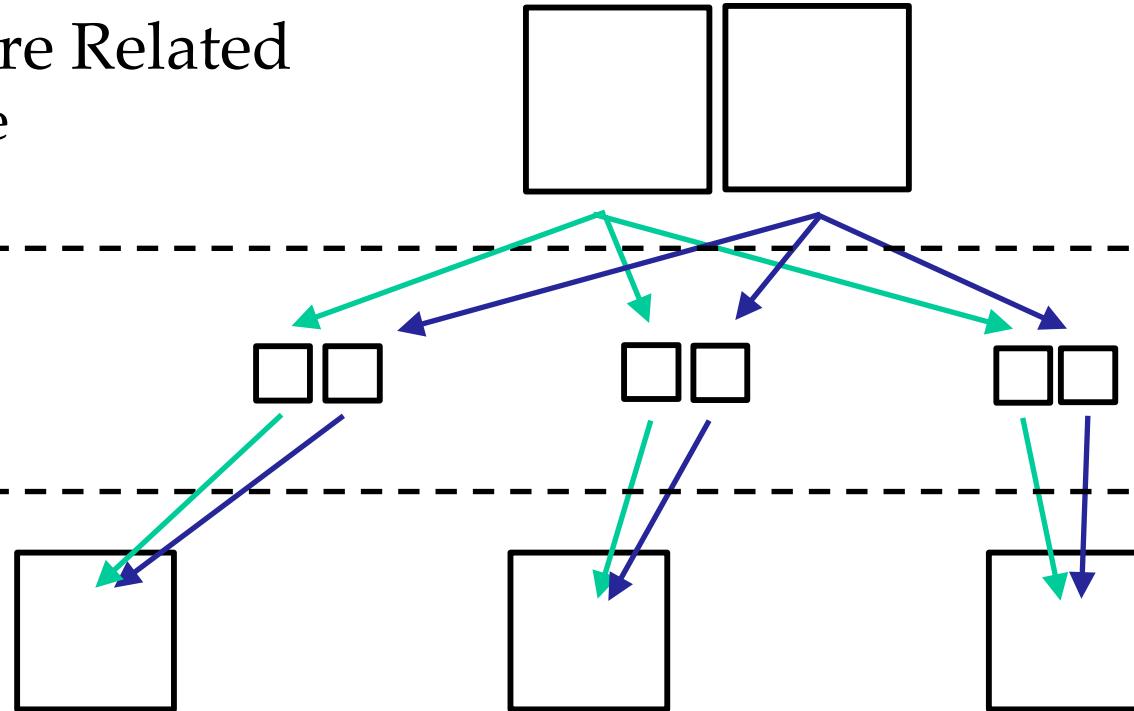
- For example: Red, Green, Blue

Convolution Layer

- Different kernels per channel

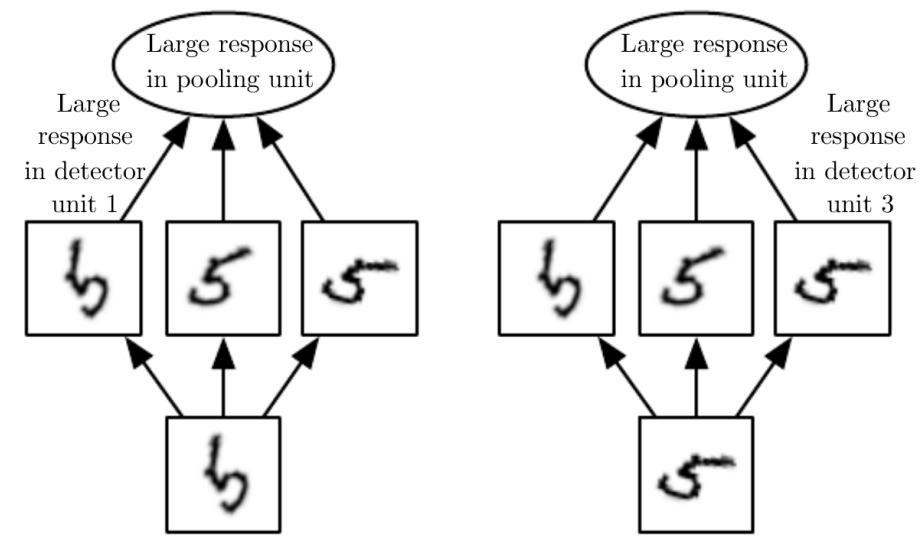
Output Features

- Channels combine per output feature

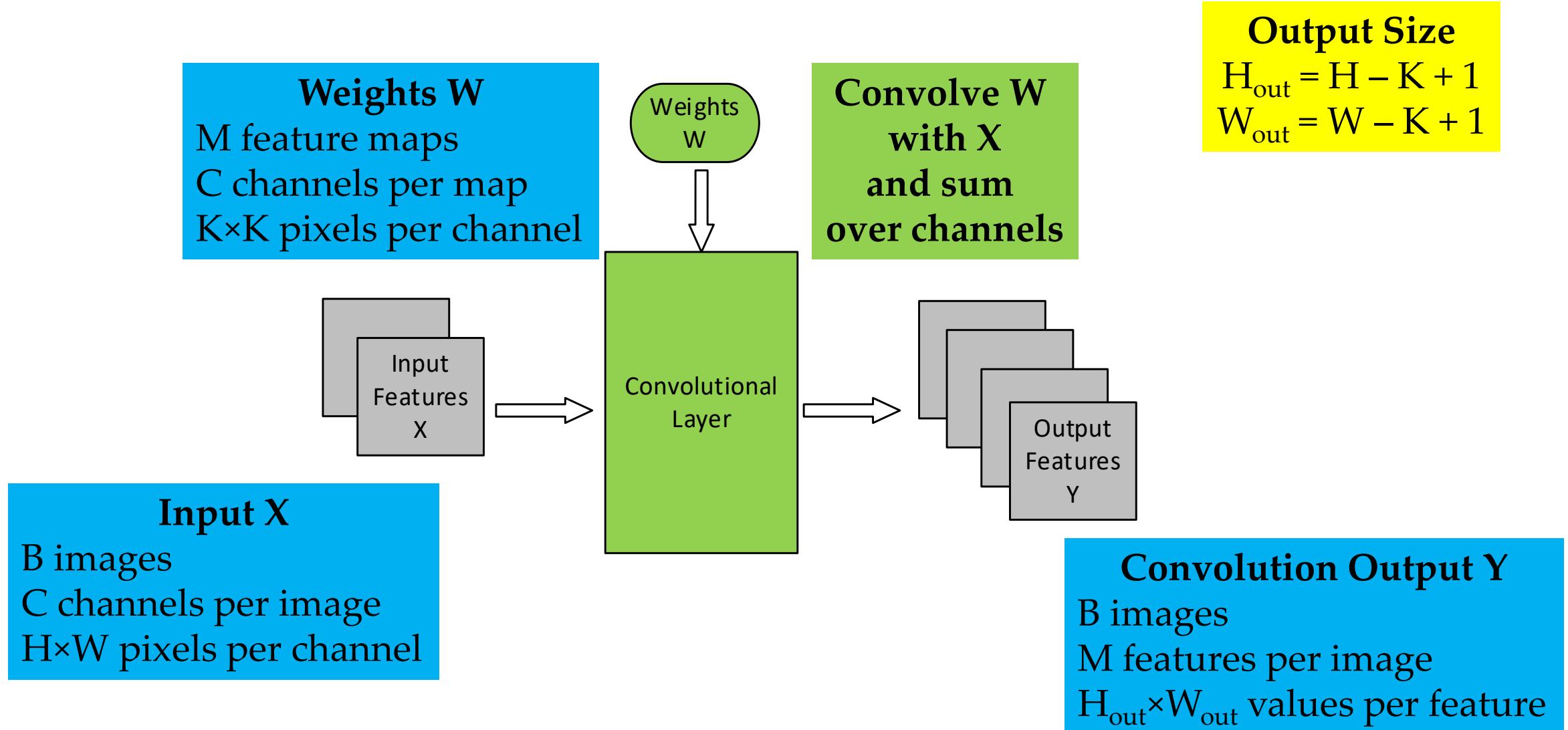


2-D Pooling (Subsampling)

- A subsampling layer
 - Sometimes with bias and non-linearity built in
- Common types
 - max, average, L^2 norm, weighted average
- Helps make representation invariant to size scaling and small translations in the input



Forward Propagation

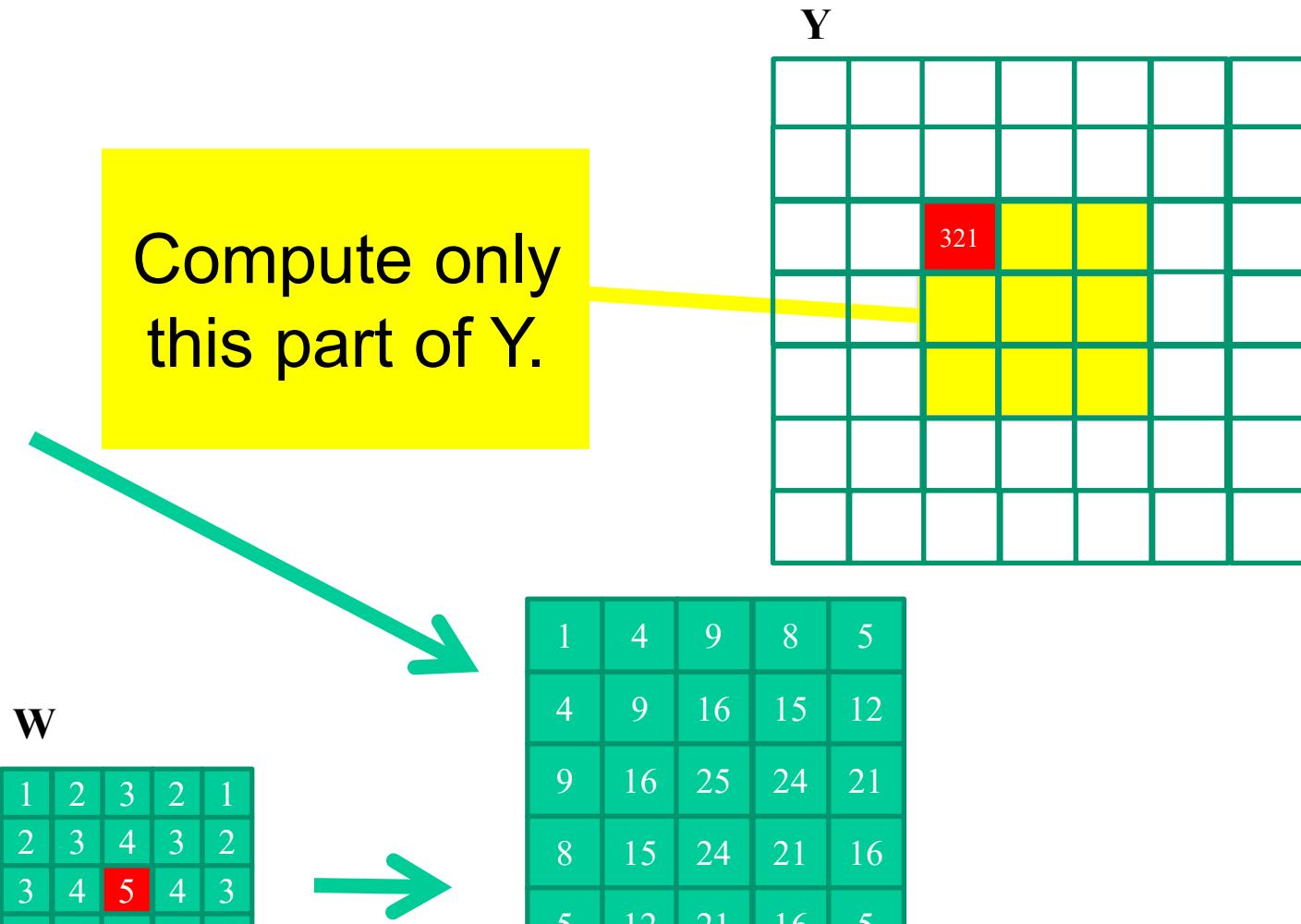


Outputs Must Use Full Mask/Kernel

X						
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	5	6
5	6	7	8	5	6	7
6	7	8	9	0	1	2
7	8	9	0	1	2	3

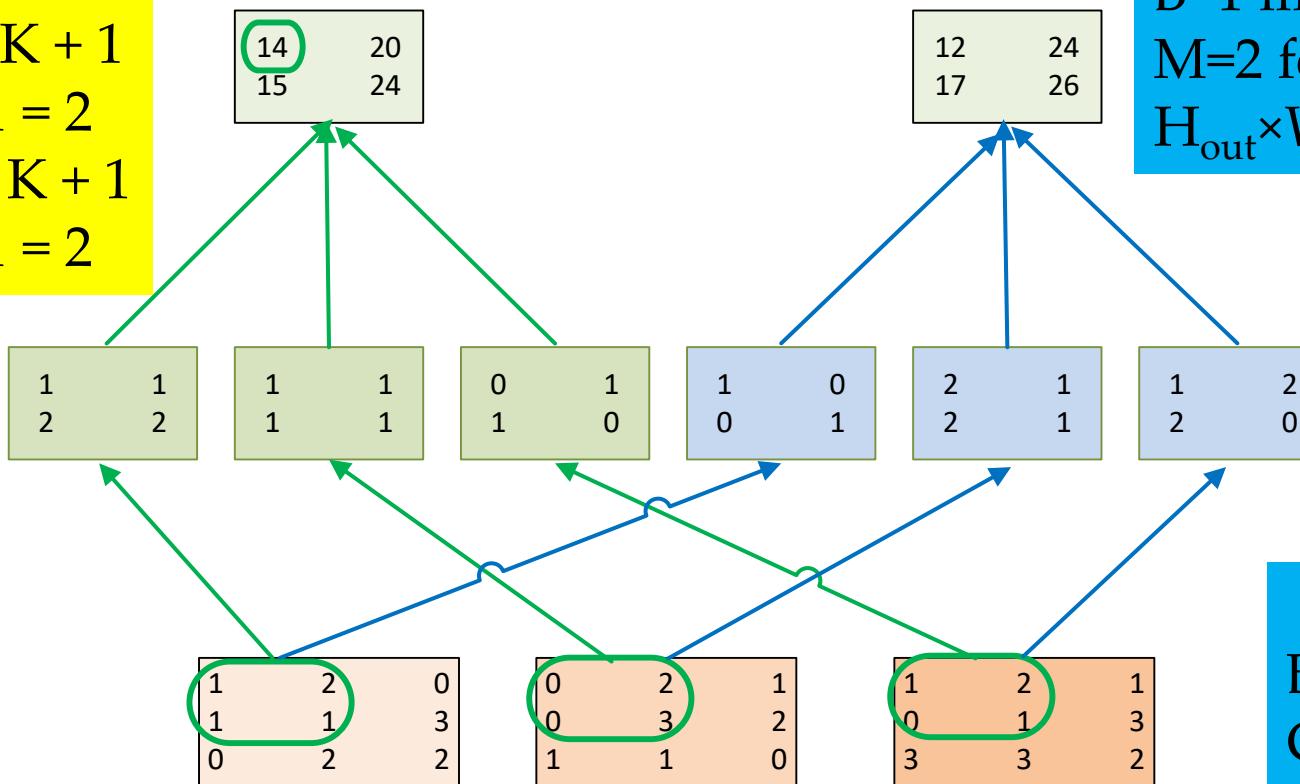
Compute only
this part of Y.

W				
1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1



Example of the Forward Path of a Convolution Layer

Output Size
 $H_{out} = H - K + 1$
 $= 3 - 2 + 1 = 2$
 $W_{out} = W - K + 1$
 $= 3 - 2 + 1 = 2$



Convolution Output Y
B=1 image
M=2 features per image
 $H_{out} \times W_{out} = 2 \times 2$ values per feature

Weights W
M=2 feature maps
C=3 channels per map
 $K \times K = 2 \times 2$ pixels per channel

Input X
B=1 image
C=3 channels
 $H \times W = 3 \times 3$ pixels per channel

Sequential Code: Forward Convolutional Layer

```
void convLayer_forward(int B, int M, int C, int H, int W, int K, float* X, float* W, float* Y) {  
    int H_out = H - K + 1;                                // calculate H_out, W_out  
    int W_out = W - K + 1;  
  
    for (int b = 0; b < B; ++b)                          // for each image  
        for(int m = 0; m < M; m++)                      // for each output feature map  
            for(int h = 0; h < H_out; h++)                // for each output value (two loops)  
                for(int w = 0; w < W_out; w++) {  
                    Y[b, m, h, w] = 0.0f;                  // initialize sum to 0  
                    for(int c = 0; c < C; c++)           // sum over all input channels  
                        for(int p = 0; p < K; p++)         // KxK filter  
                            for(int q = 0; q < K; q++)  
                                Y[b, m, h, w] += X[b, c, h + p, w + q] * W[m, c, p, q];  
                }  
}
```

A Small Convolution Layer Example

$X[b, 1, \dots]$ input channel
 $W[0,1,\dots]$
 $Y[b, 0, \dots]$ output map

Image b in mini batch
 $X[b,0,\dots]$

1	2	0	1
1	1	3	2
0	2	2	0
2	1	0	3

1	1	1
2	2	3
2	1	0

$w[0,0,\dots]$

0	2	1	0
0	3	2	1
1	1	0	2
2	1	0	3

1	2	3
1	1	0
3	0	1

$w[0,1,\dots]$

0	?
?	?
?	?

$y[b,0,\dots]$

$x[b,2,\dots]$

1	2	1	0
0	1	3	2
3	3	2	0
1	3	2	0

0	1	1
1	0	2
1	2	1

$w[0,2,\dots]$

A Small Convolution Layer Example

$c = 0$

$x[b,0,_,_]$

1	2	0	1
1	1	3	2
0	2	2	0
2	1	0	3

$x[b,1,_,_]$

0	2	1	0
0	3	2	1
1	1	0	2
2	1	0	3

$x[b,2,_,_]$

1	2	1	0
0	1	3	2
3	3	2	0
1	3	2	0

$w[0,0,_,_]$

1	1	1
2	2	3
2	1	0

$$3+13+2$$

18	?
?	?

$y[b,0,_,_]$

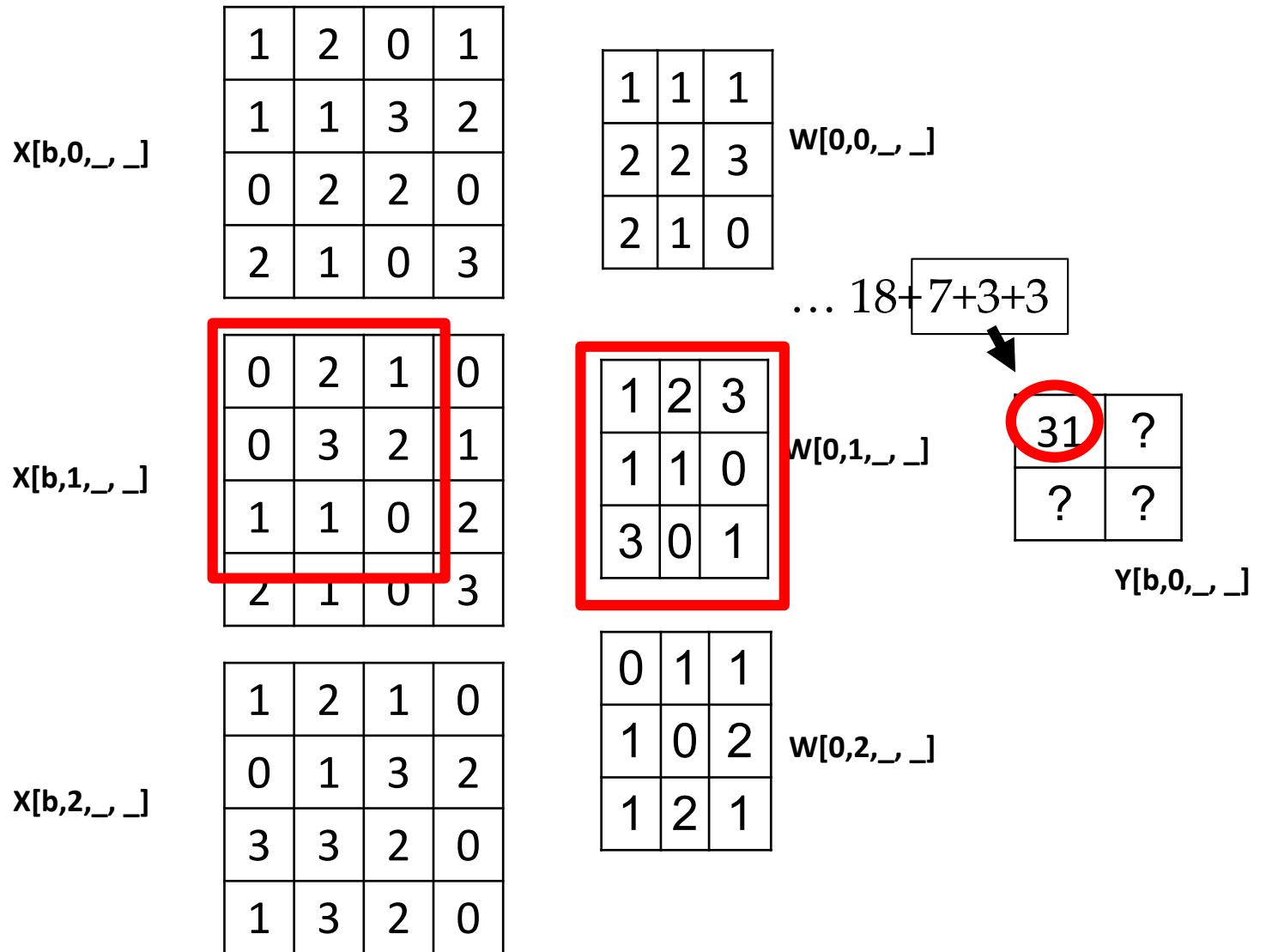
1	2	3
1	1	0
3	0	1

$w[0,1,_,_]$

0	1	1
1	0	2
1	2	1

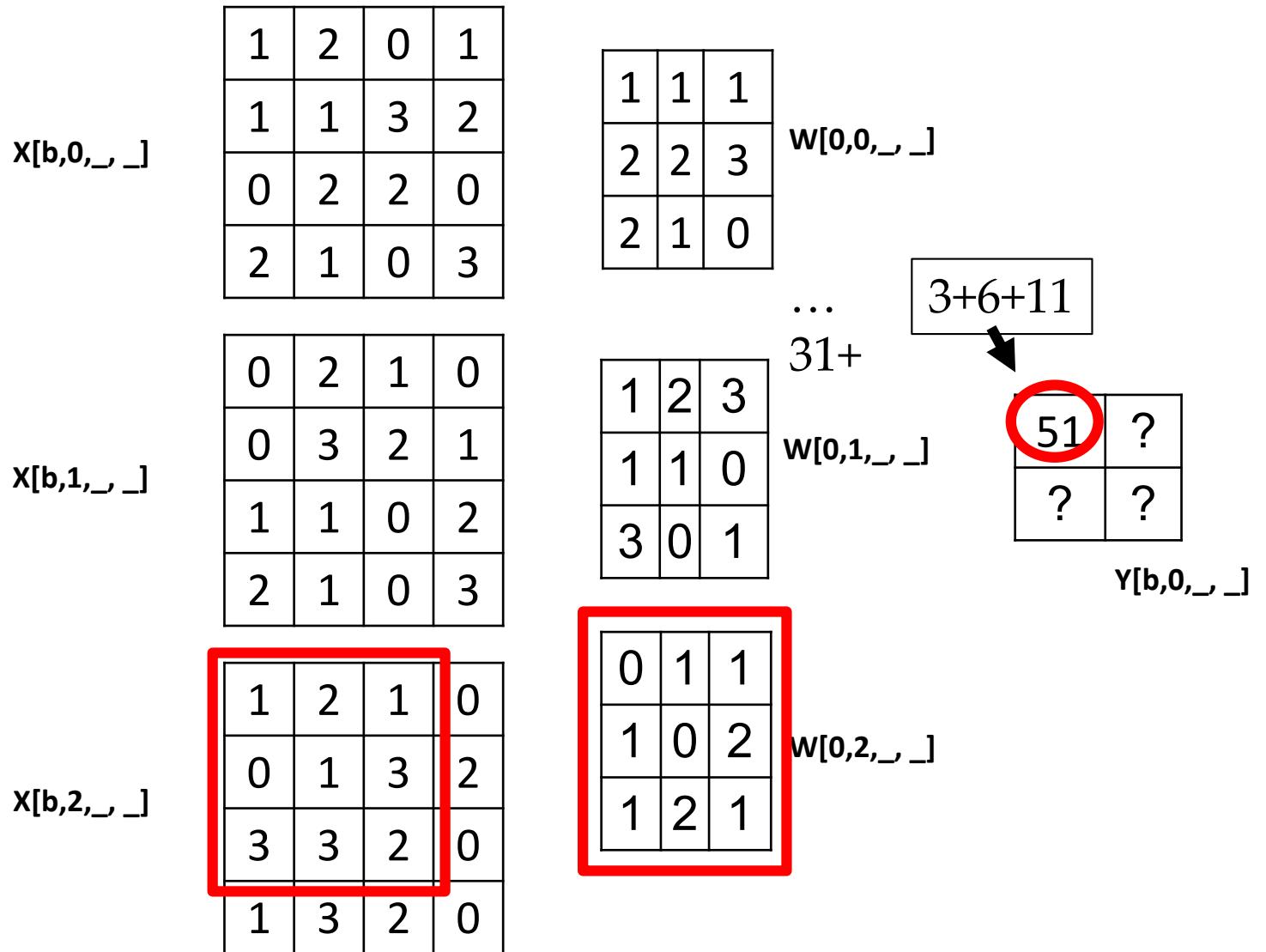
A Small Convolution Layer Example

$c = 1$



A Small Convolution Layer Example

$c = 2$



Parallelism in a Convolution Layer

Output feature maps can be calculated in parallel

- Usually a small number, not sufficient to fully utilize a GPU

All **output** feature map **pixels** can be calculated in parallel

- All rows can be done in parallel
- All pixels in each row can be done in parallel
- Large number but diminishes as we go into deeper layers

All **input feature maps** can be processed in parallel,
but need atomic operation or tree reduction (we'll learn later)

Different layers may demand different strategies.

Subsampling (Pooling) by Scale N

Convolution Output Y
B images
M features per image
 $H_{out} \times W_{out}$ values per feature

Average over $N \times N$ blocks,
then calculate sigmoid

Output Size
 $H_{S(N)} = \text{floor}(H_{out} / N)$
 $W_{S(N)} = \text{floor}(W_{out} / N)$

Subsampling/Pooling Output S
B images
M features per image
 $H_{S(N)} \times W_{S(N)}$ values per feature

Sequential Code: Forward Pooling Layer

```
void poolingLayer_forward(int B, int M, int H_out, int W_out, int N, float* Y, float* S)
{
    for (int b = 0; b < B; ++b)                      // for each image
        for (int m = 0; m < M; ++m)                  // for each output feature map
            for (int x = 0; x < H_out/N; ++x)          // for each output value (two loops)
                for (int y = 0; y < W_out/N; ++y) {
                    float acc = 0.0f                      // initialize sum to 0
                    for (int p = 0; p < N; ++p)           // loop over NxN block of Y (two loops)
                        for (int q = 0; q < N; ++q)
                            acc += Y[b, m, N*x + p, N*y + q];
                    acc /= N * N;                      // calculate average over block
                    S[b, m, x, y] = sigmoid(acc + bias[m]) // bias, non-linearity
                }
}
```

Kernel Implementation of Subsampling Layer

- Straightforward mapping from grid to subsampled output feature map pixels
- in GPU kernel,
 - need to manipulate index mapping
 - for accessing the output feature map pixels
 - of the previous convolution layer.
- Often merged into the previous convolution layer to save memory bandwidth

Design of a Basic Kernel

- Each block computes
 - a tile of output pixels for one feature
 - `TILE_WIDTH` pixels in each dimension
- Grid's X dimension maps to M output feature maps
- Grid's Y dimension maps to the tiles in the output feature maps (linearized order).
- (Grid's Z dimension is used for images in batch, which we omit from slides.)

tiles covering an output feature map, marked with linearized indices

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

A Small Example

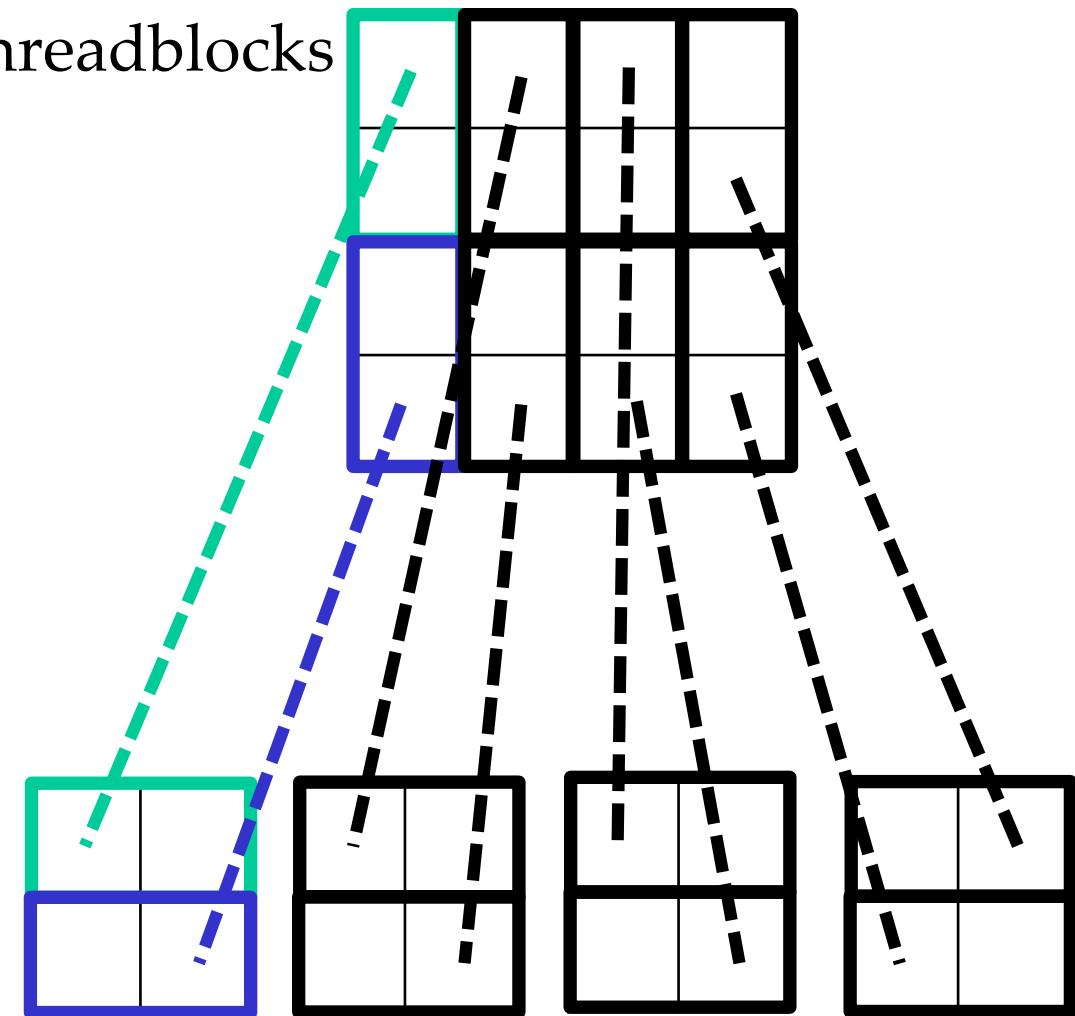
Assume

- **M = 4** (4 output feature maps),
- thus 4 blocks in the X dimension, and
- **W_out = H_out = 8** (8x8 output features).

If **TILE_WIDTH = 4**,
we also need 4 blocks in the Y
dimension:

- for each output feature,
- top two blocks in each column
calculates the top row of tiles, and
- bottom two calculate the bottom row.

CUDA Grid and
Threadblocks



Output Feature Maps and Tiles 21

Host Code for a Basic Kernel: CUDA Grid

Consider an output feature map:

- width is **W_out**, and
- height is **H_out**.
- Assume these are multiples of **TILE_WIDTH**.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

Let **X_grid** be the number of blocks needed in X dim (5 above).

Let **Y_grid** be the number of blocks needed in Y dim (4 above).

Host Code for a Basic Kernel: CUDA Grid

(Assuming W_out and H_out are multiples of TILE_WIDTH.)

```
#define TILE_WIDTH 16          // We will use 4 for small examples.  
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  
Y = H_grid * W_grid;  
  
dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1); // output tile for untiled code  
dim3 gridDim(M, Y, 1);  
  
ConvLayerForward_Kernel<<< gridDim, blockDim >>>(...);
```

Partial Kernel Code for a Convolution Layer

```
__global__ void ConvLayerForward_Basic_Kernel
    (int C, int W_grid, int K, float* X, float* W, float* Y)
{
    int m = blockIdx.x;
    int h = (blockIdx.y / W_grid) * TILE_WIDTH + threadIdx.y;
    int w = (blockIdx.y % W_grid) * TILE_WIDTH + threadIdx.x;
    float acc = 0.0f;
    for (int c = 0; c < C; c++) {                      // sum over all input channels
        for (int p = 0; p < K; p++) {                  // loop over KxK filter
            for (int q = 0; q < K; q++)
                acc += X[c, h + p, w + q] * W[m, c, p, q];
    }
    Y[m, h, w] = acc;
}
```

Some Observations

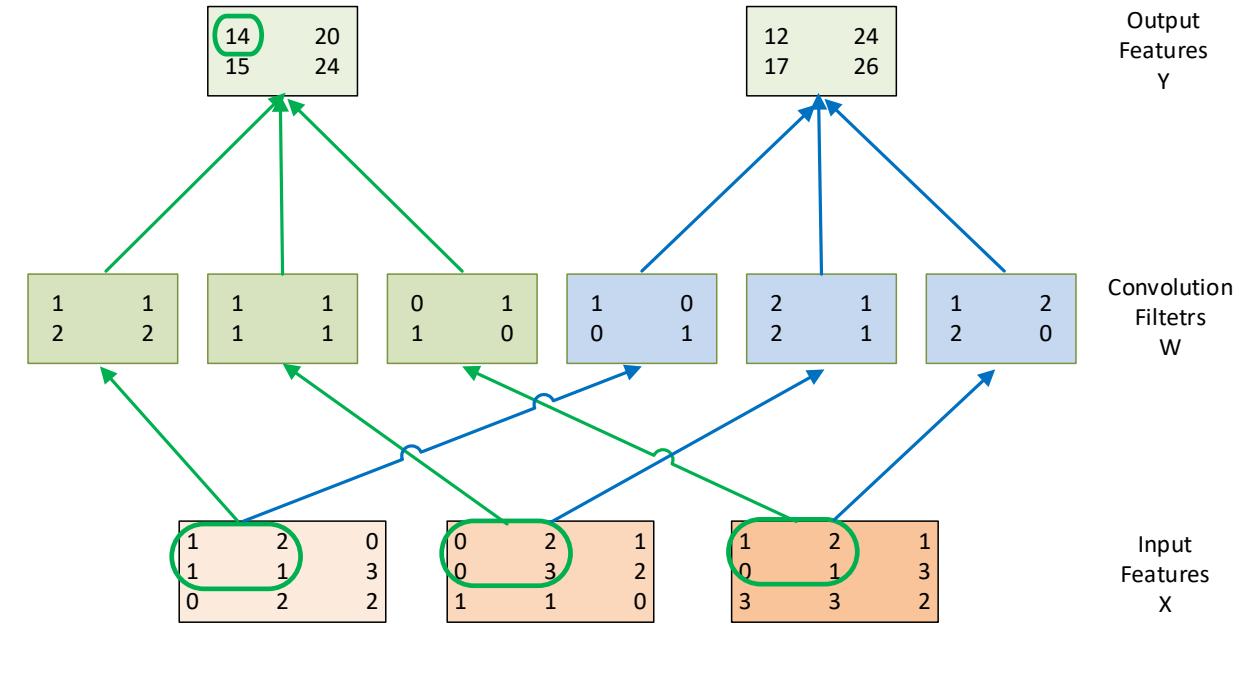
Enough parallelism

- if the total number of pixels
- across all output feature maps is large
- (often the case for CNN layers)

Each input tile

- loaded M times (number of output features), so
- **not efficient in global memory bandwidth,**
- but block scheduling in X dimension should give cache benefits.

Implementing a Convolution Layer with Matrix Multiplication



$$\begin{array}{c} \begin{array}{|c|c|c|} \hline 1 & 1 & 2 & 2 \\ \hline 1 & 0 & 0 & 1 \\ \hline \end{array} & \begin{array}{|c|c|c|} \hline 1 & 1 & 1 & 1 \\ \hline 2 & 1 & 2 & 1 \\ \hline \end{array} & \begin{array}{|c|c|c|} \hline 0 & 1 & 1 & 0 \\ \hline 1 & 2 & 2 & 0 \\ \hline \end{array} \\ \hline \end{array} * \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 1 & 1 \\ \hline 1 & 3 \\ \hline 0 & 2 \\ \hline 2 & 1 \\ \hline 0 & 3 \\ \hline 3 & 2 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 14 & 20 \\ \hline 12 & 24 \\ \hline \end{array}$$

Convolution Filters W'

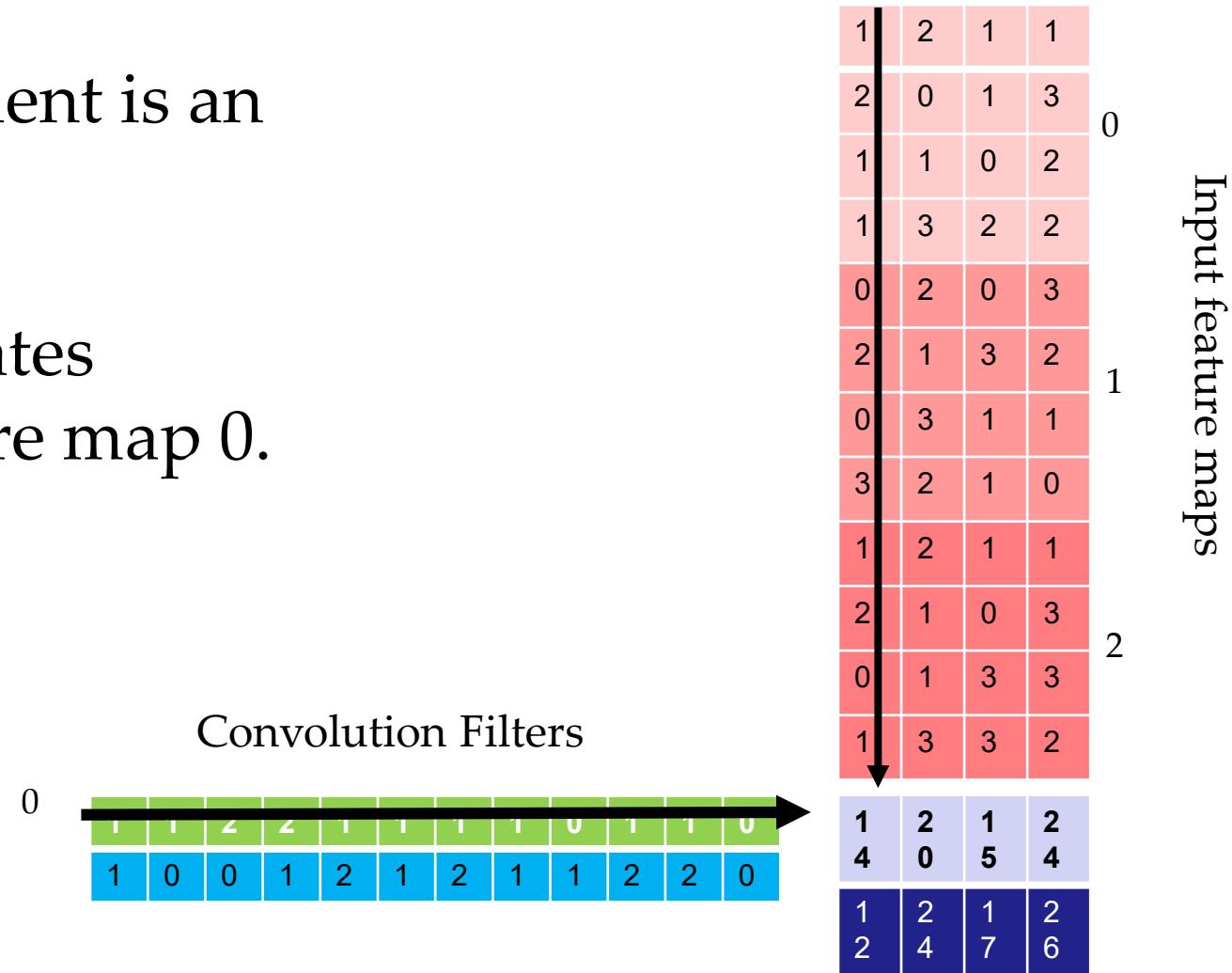
Output Features Y

Input Features X_{unrolled}

Simple Matrix Multiplication

Each product matrix element is an output feature map pixel.

This inner product generates element 0 of output feature map 0.

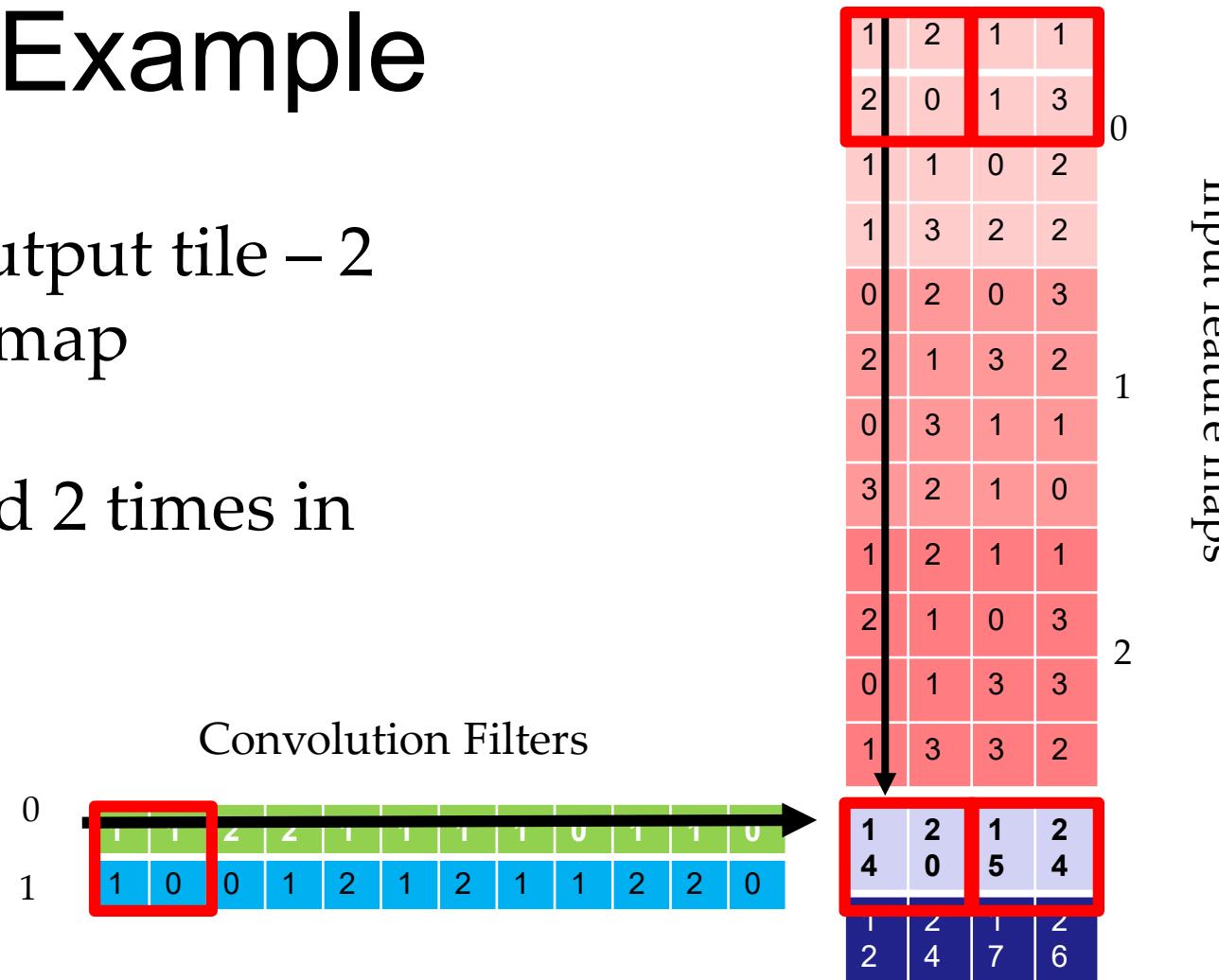


Tiled Matrix Multiplication

2x2 Example

Each block calculates one output tile – 2 elements from each output map

Each input element is reused 2 times in the shared memory

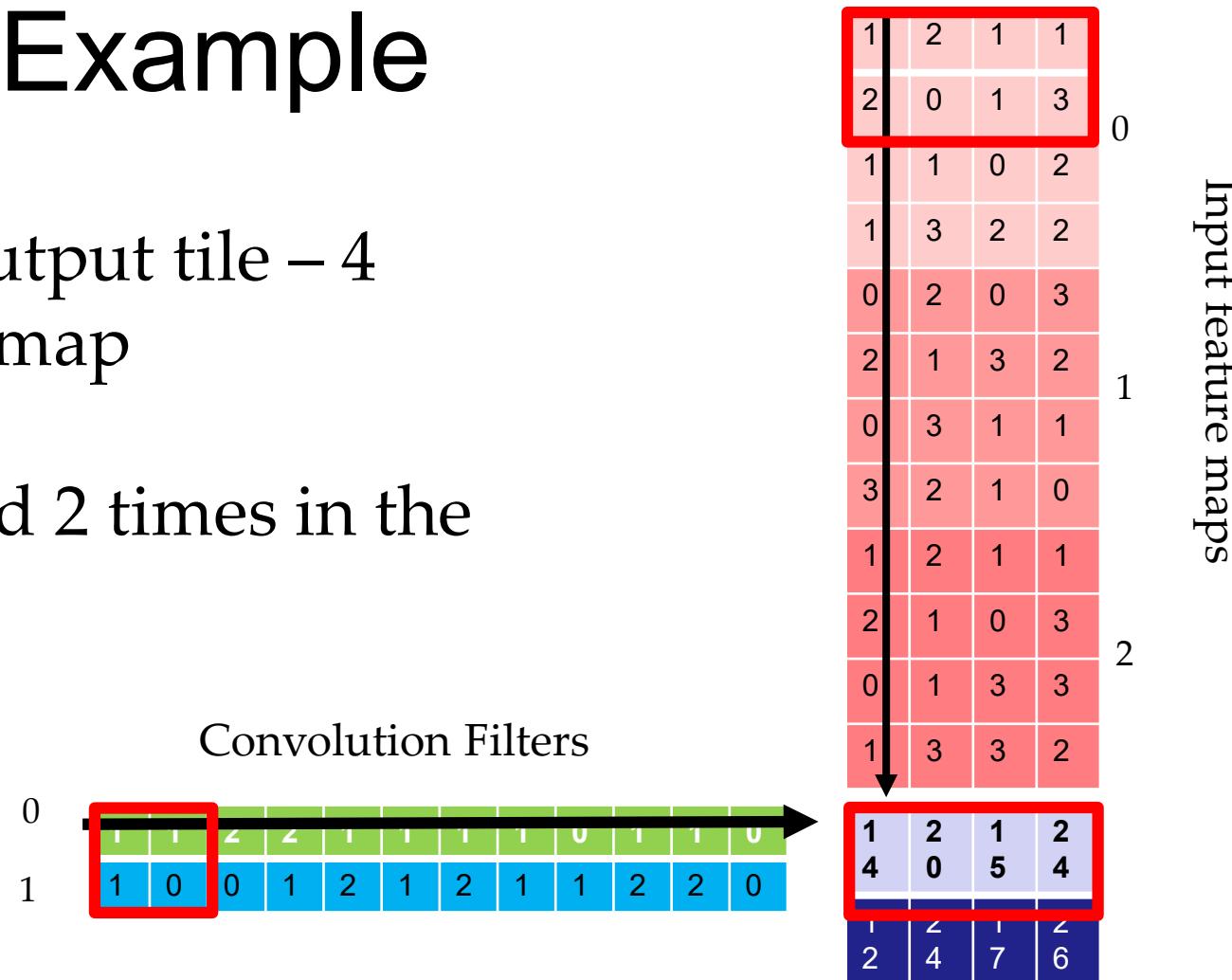


Tiled Matrix Multiplication

2x4 Example

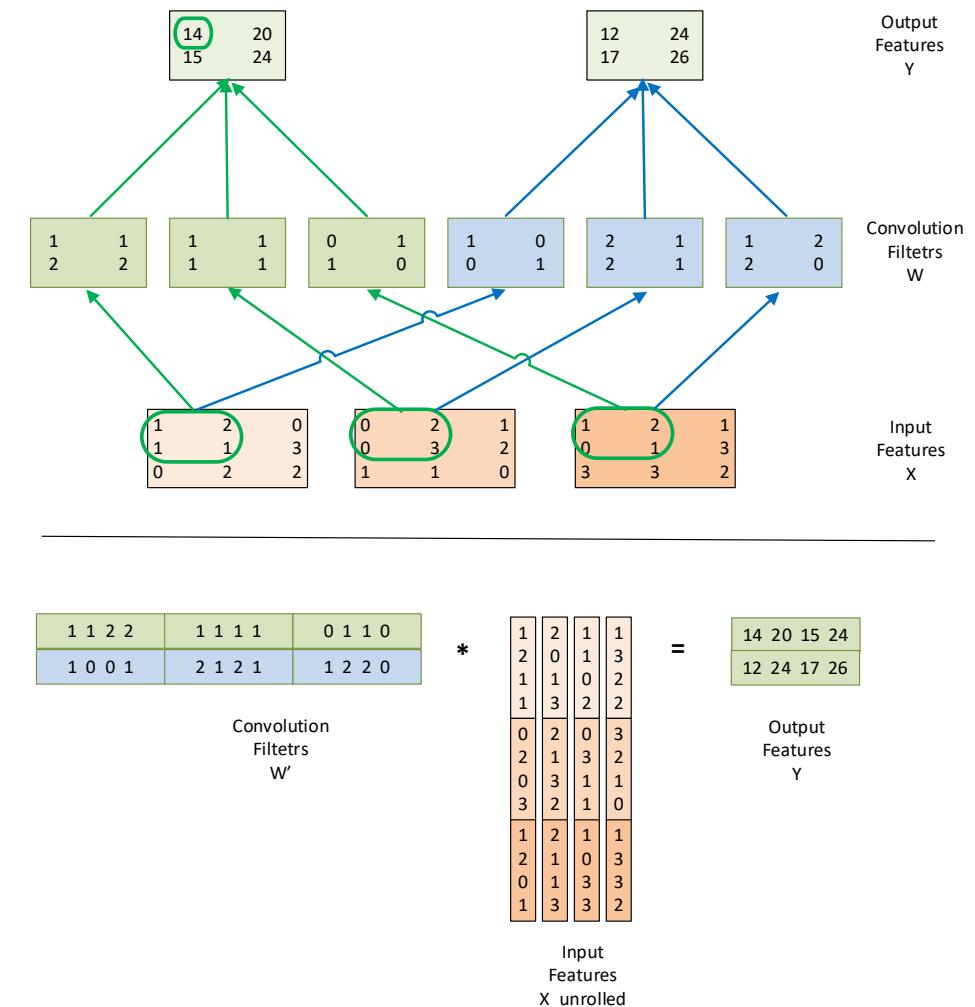
Each block calculates one output tile – 4 elements from each output map

Each input element is reused 2 times in the shared memory



Efficiency Analysis: Total Input Replication

- Replicated input features are shared among output maps
 - There are $H_{out} * W_{out}$ output feature map elements
 - Each requires $K*K$ elements from the input feature maps
 - So, the total number of input element after replication is $H_{out}*W_{out}*K*K$ times for each input feature map
 - The total number of elements in each original input feature map is $(H_{out}+K-1) * (W_{out}+K-1)$



Analysis of a Small Example

$$H_{out} = 2$$

$$W_{out} = 2$$

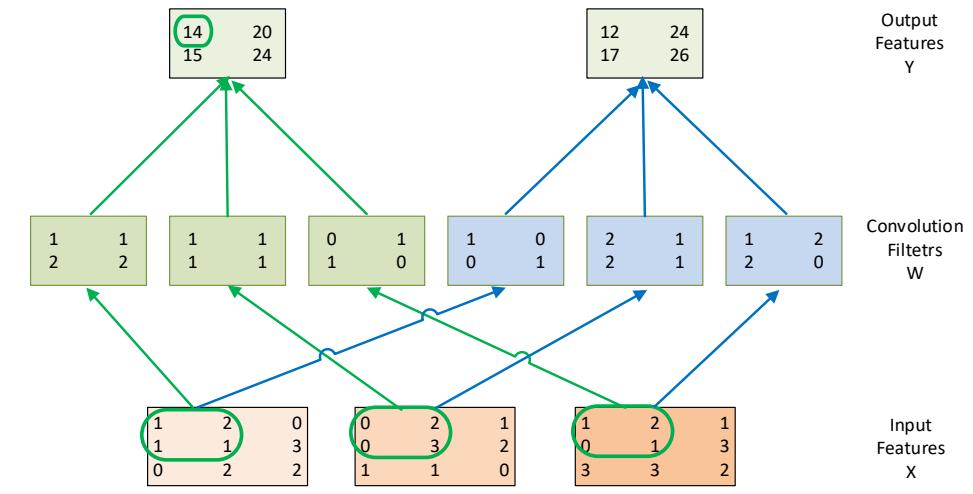
$$K = 2$$

There are 3 input maps (channels)

The total number of input elements in the replicated (“unrolled”) input matrix is $3*2*2*2*2$

The replicating factor is

$$(3*2*2*2*2)/(3*3*3) = 1.78$$



$$\begin{array}{c|c|c} 1 & 1 & 2 \\ \hline 1 & 1 & 1 \\ \hline 0 & 1 & 0 \end{array} \quad \begin{array}{c|c|c} 1 & 1 & 1 \\ \hline 2 & 1 & 2 \\ \hline 1 & 2 & 1 \end{array} \quad \begin{array}{c|c|c} 0 & 1 & 1 \\ \hline 1 & 2 & 2 \\ \hline 1 & 2 & 0 \end{array} \quad * \quad \begin{array}{c|c|c} 1 & 2 & 1 \\ \hline 2 & 1 & 3 \\ \hline 1 & 3 & 2 \\ \hline 0 & 2 & 0 \\ \hline 2 & 1 & 3 \\ \hline 0 & 3 & 1 \\ \hline 3 & 2 & 1 \\ \hline 1 & 2 & 1 \\ \hline 2 & 1 & 0 \\ \hline 0 & 1 & 3 \\ \hline 1 & 3 & 2 \end{array} = \begin{array}{c|c|c} 14 & 20 & 15 \\ \hline 12 & 24 & 17 \\ \hline 15 & 26 \end{array}$$

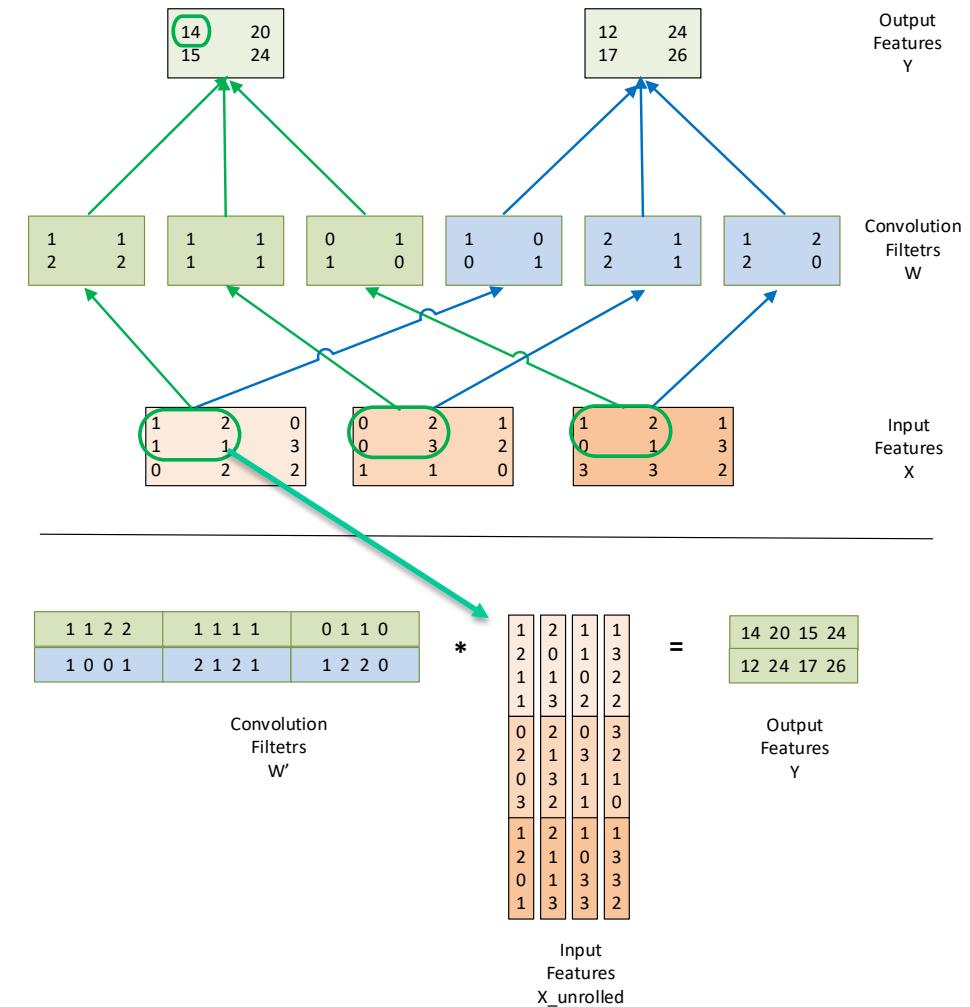
Convolution Filters W' Input Features X_unrolled Output Features Y

Memory Access Efficiency of Original Convolution Algorithm

- Assume that we use tiled 2D convolution
- For input elements
 - Each output tile has TILE_WIDTH^2 elements
 - Each input tile has $(\text{TILE_WIDTH}+K-1)^2$
 - The total number of input feature map element accesses was $\text{TILE_WIDTH}^2 \times K^2$
 - The reduction factor of the tiled algorithm is $K^2 \times \text{TILE_WIDTH}^2 / ((\text{TILE_WIDTH}+K-1)^2)$
- The convolution filter weight elements are reused within each output tile

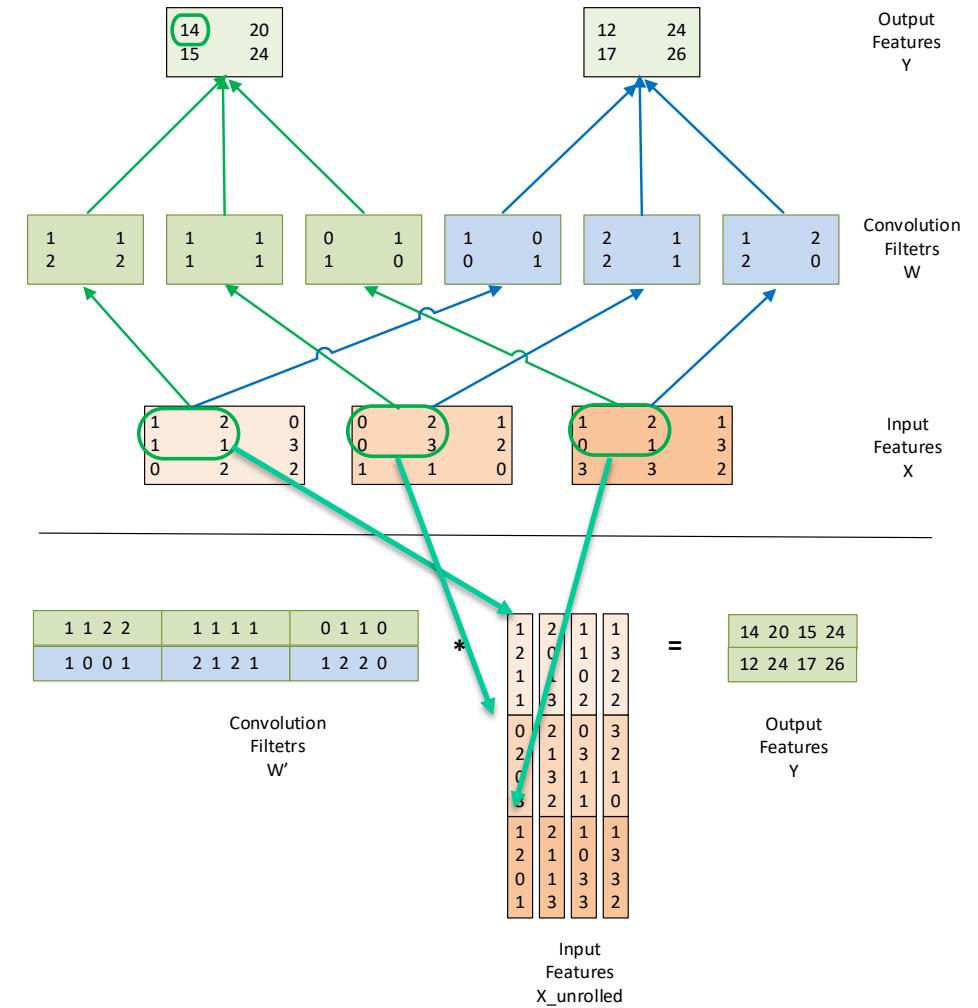
Properties of the Unrolled Matrix

- Each unrolled column corresponds to an output feature map element
- For an output feature element (h, w) , the index for the unrolled column is $h * W_{out} + w$ (linearized index of the output feature map element)



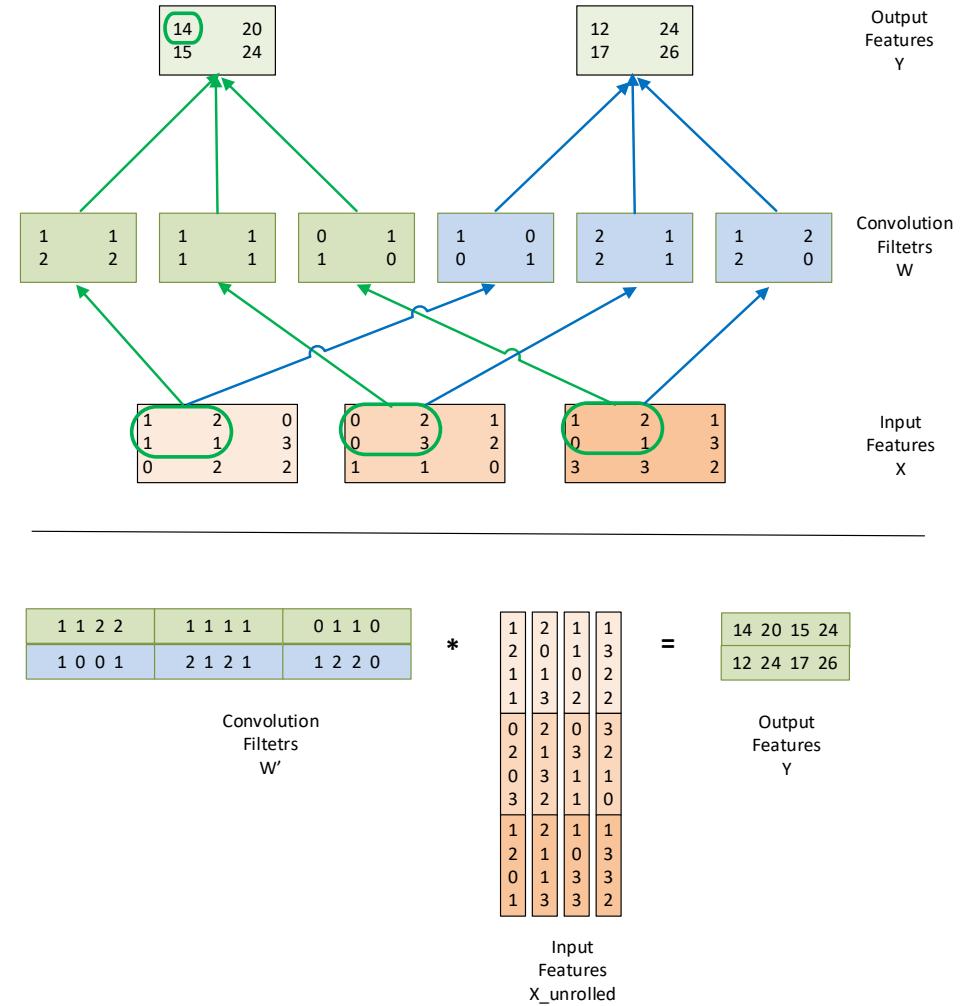
Properties of the Unrolled Matrix (cont.)

- Each section of the unrolled column corresponds to an input feature map
- Each section of the unrolled column has k^k elements (convolution mask size)
- For an input feature map c , the vertical index of its section in the unrolled column is $c \cdot k^k$ (linearized index of the output feature map element)



To Find the Input Elements

- For output element (h, w) , the base index for the upper left corner of the input feature map c is (c, h, w)
- The input element index for multiplication with the convolution mask element (p, q) is $(c, h+p, w+q)$



Input to Unrolled Matrix Mapping

Output element (h, w)

Mask element (p, q)

Input feature map c

// calculate the horizontal matrix index

```
int w_unroll = h * W_out + w;
```

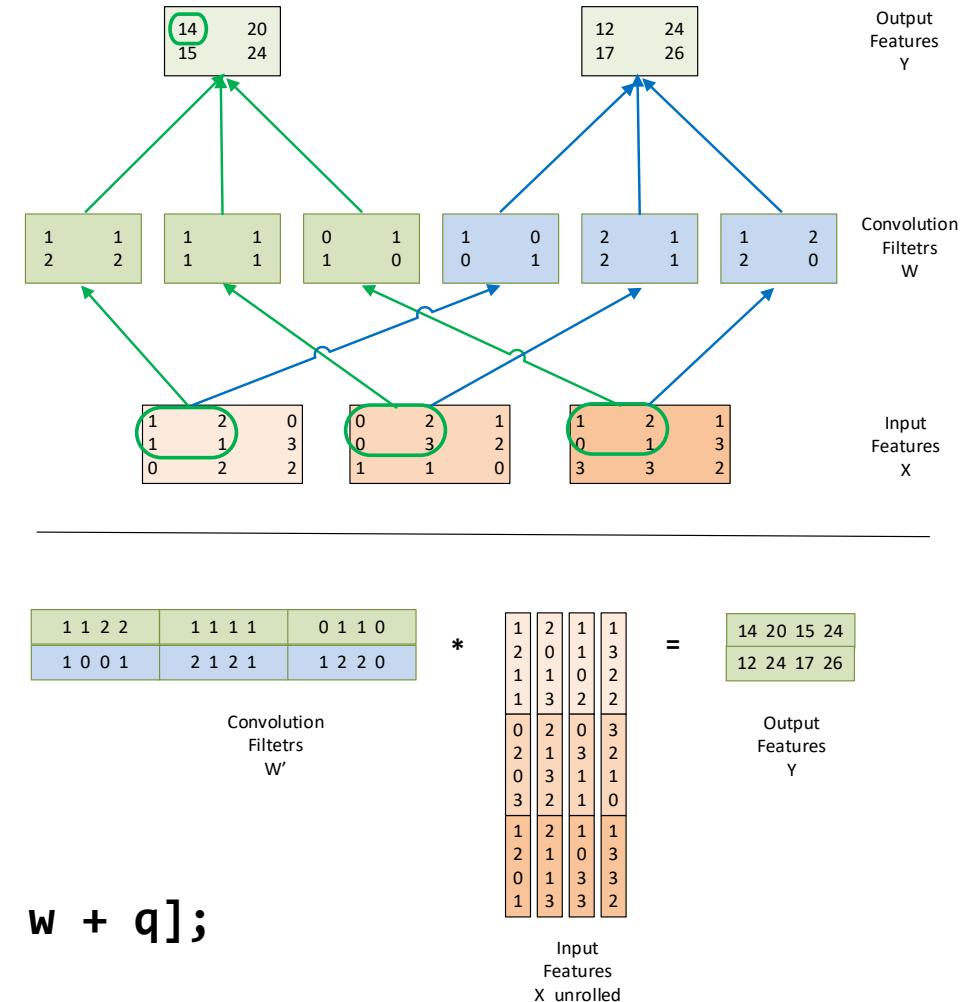
// find the beginning of the unrolled

```
int w_base = c * (K*K);
```

// calculate the vertical matrix index

```
int h_unroll = w_base + p * K + q;
```

```
X_unroll[b, h_unroll, w_unroll] = X[b, c, h + p, w + q];
```



Function to generate “unrolled” X

```
void unroll(int B, int C, int H, int W, int K, float* X, float* X_unroll)
{
    int H_out = H - K + 1;                                // calculate H_out, W_out
    int W_out = W - K + 1;
    for (int b = 0; b < B; ++b)                          // for each image
        for (int c = 0; c < C; ++c) {                    // for each input channel
            int w_base = c * (K*K);                      // per-channel offset for smallest X_unroll index
            for (int p = 0; p < K; ++p) {                // for each element of KxK filter (two loops)
                for (int q = 0; q < K; ++q) {
                    for (int h = 0; h < H_out; ++h)       // for each thread (each output value, two loops)
                        for (int w = 0; w < W_out; ++w) {
                            int h_unroll = w_base + p * K + q; // data needed by one thread
                            int w_unroll = h * W_out + w;      // smallest index--across threads (output values)
                            X_unroll[b, h_unroll, w_unroll] = X[b, c, h + p, w + q]; // copy input pixels
                        }
                }
            }
        }
}
```

Implementation Strategies for a Convolution Layer

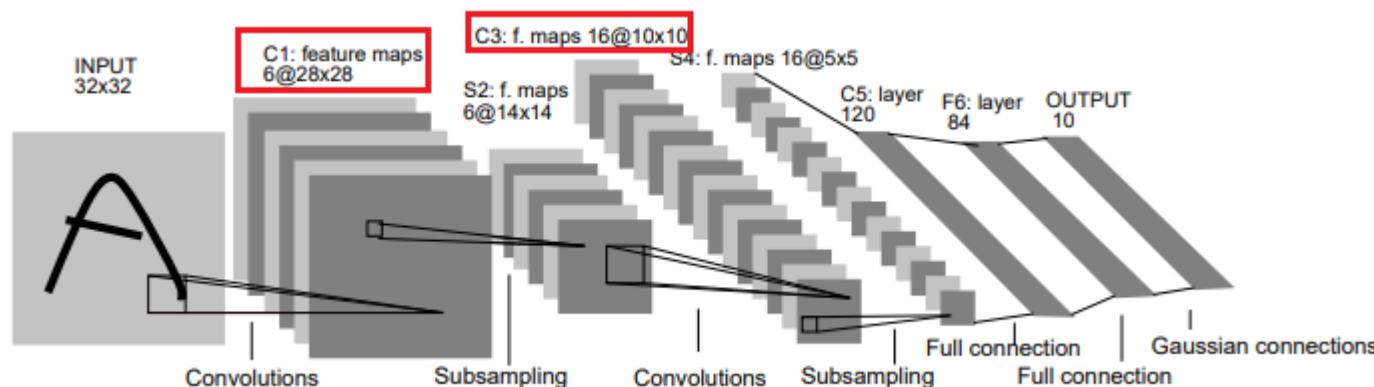
- **Baseline**
 - Tiled 2D convolution implementation, use constant memory for convolution masks
- **Matrix-Multiplication Baseline**
 - Input feature map unrolling kernel, constant memory for convolution masks as an optimization
 - Tiled matrix multiplication kernel
- **Matrix-Multiplication with built-in unrolling**
 - Perform unrolling only when loading a tile for matrix multiplication
 - The unrolled matrix is only conceptual
 - When loading a tile element of the conceptual unrolled matrix into the shared memory, use the properties in the lecture to load from the input feature map
- **More advanced Matrix-Multiplication**
 - Use joint register-shared memory tiling

Project Overview

- Optimize the forward pass of the convolutional layers in a modified LeNet-5 CNN using CUDA. (CNN implemented using Mini-DNN, a C++ framework)
- The network will be classifying Fashion MNIST dataset
- Some network parameters to be aware of
 - Input Size: 86x86 pixels, batch of 10k images
 - Input Channels: 1
 - Convolutional kernel size: 7x7
 - Number of kernels: Variable (your code should support this)



<https://github.com/zalandoresearch/fashion-mnist>



<http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf>

Project Timeline

- **All milestones are due on Fridays at 8 pm Central Time**
- Everyone must individually submit all milestones.
 - No sharing of code is allowed
- Project milestone 1:
 - CPU Convolution, profiling
- Project milestone 2:
 - Baseline GPU Convolution Kernel
- Project milestone 3:
 - GPU Convolution Kernel Optimizations

Project Release

- Project is released now (only PM1 for now)
 - Check the course wiki page for the link to the github repository
 - <https://github.com/aschuh703/ECE408/tree/main/Project>
- The readme in the repository contains all the instructions and details to complete the project.
- The github repo will be updated with additional code and instructions for PM2 & PM3



**ANY MORE QUESTIONS?
READ CHAPTER 16**

ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 11: Feed-Forward Networks and Gradient-Based Training

Course Reminders

- Lab 2 is graded – check your grade in Canvas
- Lab 4 is due this week
- Midterm 1 is on Tuesday, March 7th
 - On-line, everybody will be taking it at the same time
 - Tuesday, March 7th 7:00pm-8:30pm US Central time
 - Includes materials from Lecture 1 through Lecture 9
- Project Milestone 1: Baseline CPU implementation is due Friday March 10th
 - Project details are posted on the wiki

Objective

- To learn the basic approach to feedforward neural networks:
 - neural model
 - common functions
 - training through gradient descent

Example: Digit Recognition

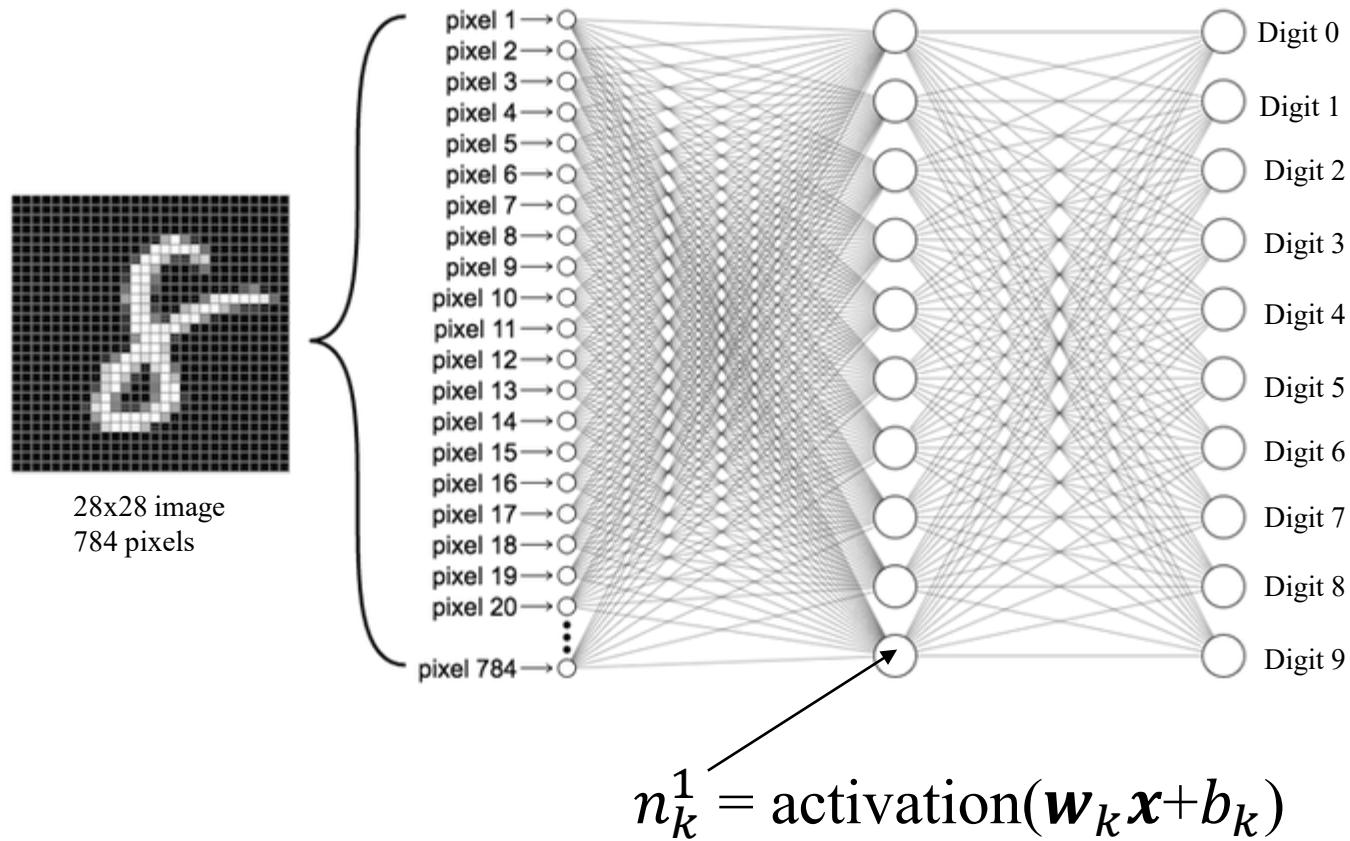
Let's consider an example.

- **handwritten digit recognition:**
- given a **28×28 grayscale image**,
- produce a **number from 0 to 9**.

Input dataset

- **60,000** images
- Each labeled by a human with correct answer.

MultiLayer Perceptron (MLP) for Digit Recognition



This network would have

- 784 nodes on input layer (L0)
- 10 nodes on hidden layer (L1)
- 10 nodes on output layer (L2)

784*10 weights + 10 biases for L1

10*10 weights + 10 biases for L2

A total of 7,960 parameters

Each node represents a function, based on a linear combination of inputs + bias

Activation function “repositions” output value.

Sigmoid, sign, ReLU are common... 5

How Do We Determine the Weights?

First layer of perceptrons

- 784 (28^2) inputs, 1024 outputs, fully connected
- $[1024 \times 784]$ weight matrix W
- $[1024 \times 1]$ bias vector b

Use labeled training data to pick weights.

Idea:

- given enough labeled input data,
- we can approximate the input-output function.

Forward and Backward Propagation

Forward (**inference**):

- given input \mathbf{x} (for example, an image),
- **use parameters Θ** (\mathbf{W} and \mathbf{b} for each layer)
- **to compute probabilities $k[i]$** (ex: for each digit i).

Backward (**training**):

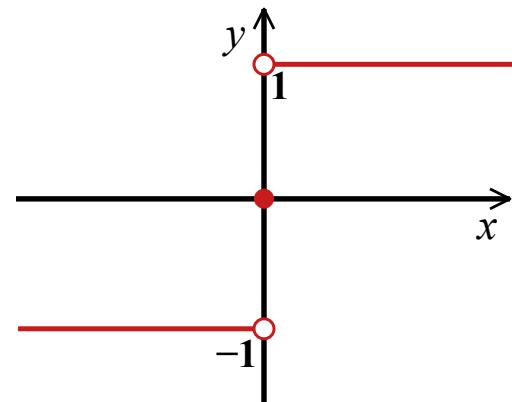
- given input \mathbf{x} , parameters Θ , and outputs $k[i]$,
- **compute error E** based on target label \mathbf{t} ,
- then **adjust Θ** proportional to E to reduce error.

Neural Functions Impact Training

Recall perceptron function: $y = \text{sign}(\mathbf{W} \cdot \mathbf{x} + b)$

To propagate error backwards,

- use chain rule from calculus.
- Smooth functions are useful.



Sign is not a smooth function.

One Choice: Sigmoid/Logistic Function

Until about 2017,

- **sigmoid / logistic function** most popular

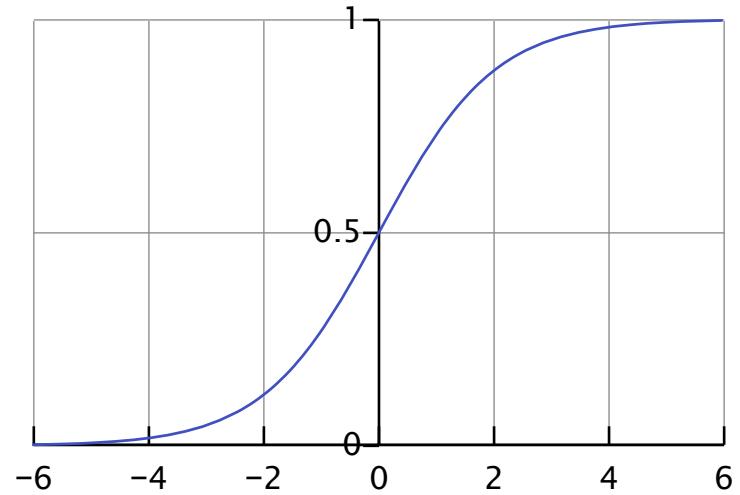
$$f(x) = \frac{1}{1+e^{-x}} \quad (\text{f: } \mathbb{R} \rightarrow (0,1))$$

for replacing sign.

- Once we have $f(x)$, finding df/dx is easy:

$$\frac{df(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = f(x) \frac{e^{-x}}{(1 + e^{-x})} = f(x)(1 - f(x))$$

(Our example used this function.)



Today's Choice: ReLU

In 2017, most common choice became

- **rectified linear unit / ReLU / ramp function**

$$f(x) = \max(0, x) \quad (\text{f: } \mathbb{R} \rightarrow \mathbb{R}^+)$$

which is much faster (no exponent required).

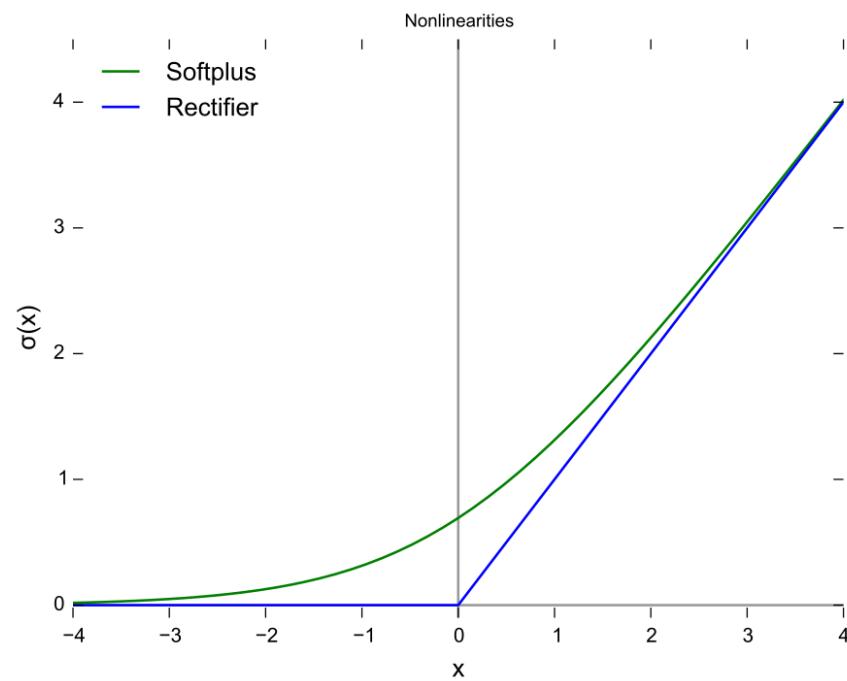
- A smooth approximation is

softplus/SmoothReLU

$$f(x) = \ln(1 + e^x) \quad (\text{f: } \mathbb{R} \rightarrow \mathbb{R}^+)$$

which is the integral of the logistic function.

- Lots of variations exist. See Wikipedia for an overview and discussion of tradeoffs.



Use Softmax to Produce Probabilities

How can sigmoid / ReLU produce probabilities?

They can't.

- Instead, given output vector $\mathbf{Z} = (z[0], \dots, z[C-1])^*$,
- we produce a second vector $\mathbf{K} = (k[0], \dots, k[C-1])$
- using the **softmax function**

$$k[i] = \frac{e^{z[i]}}{\sum_{j=0}^{C-1} e^{z[j]}}$$

Notice that the $k[i]$ sum to 1.

*Remember that we classify into one of C categories.

Softmax Derivatives Needed to Train

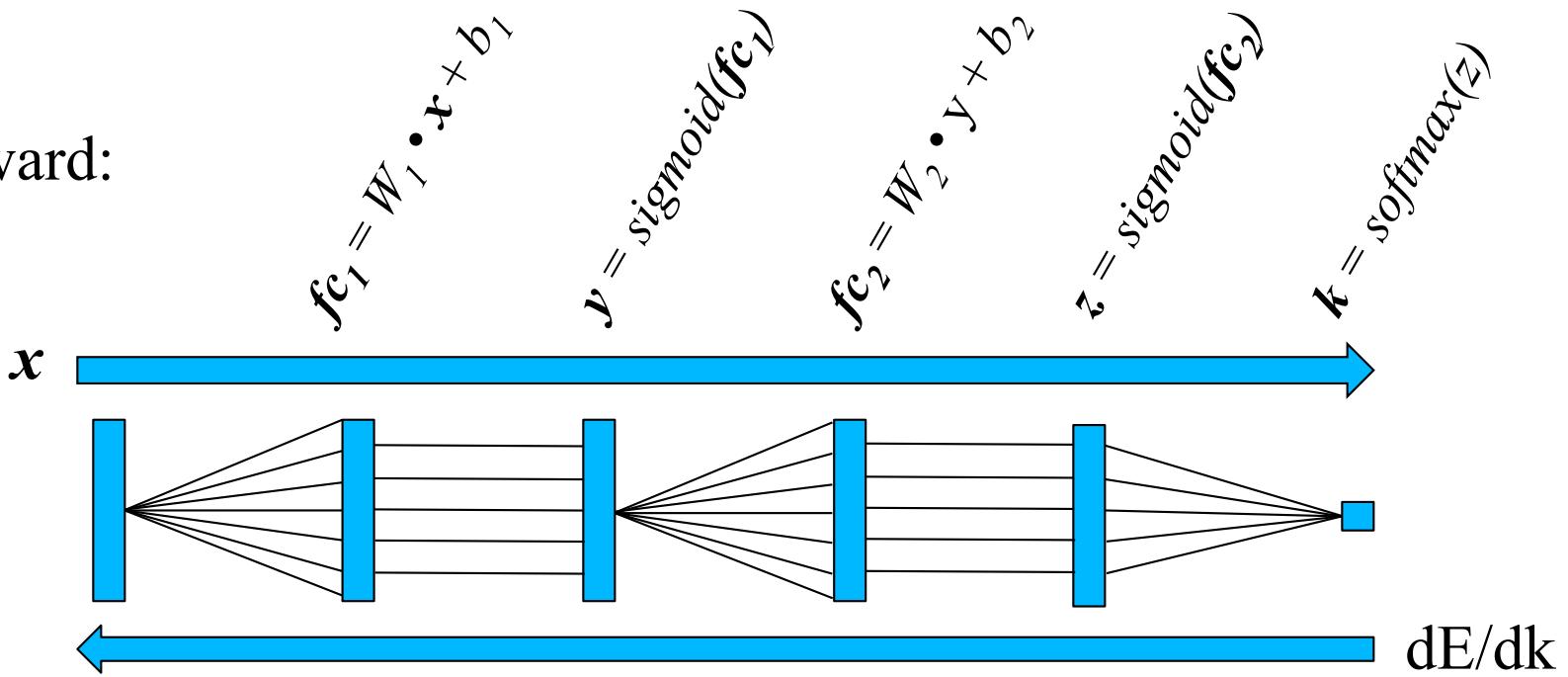
We also need the **derivatives of softmax**,

$$\frac{dk[i]}{dz[m]} = k[i](\delta_{i,m} - k[m]),$$

where $\delta_{i,m}$ is the Kronecker delta
(1 if $i = m$, and 0 otherwise).

Forward and Backward Propagation

Forward:



Backward:

$$\frac{dE}{dfc_1} = \frac{dE}{dy} \frac{dy}{dfc_1}$$
$$\frac{dE}{dy} = \frac{dE}{dfc_2} \frac{dfc_2}{dy}$$
$$\frac{dE}{dfc_2} = \frac{dE}{dz} \frac{dz}{dfc_2}$$
$$\frac{dE}{dz} = \frac{dE}{dk} \frac{dk}{dz}$$

Choosing an Error Function

Many error functions are possible.

For example, **given label T** (digit $\textcolor{green}{T}$),

- $E = 1 - k[T]$, the **probability of not classifying as t .**

Alternatively, since our categories are numeric,
we can **penalize quadratically**:

$$E = \sum_{j=0}^{C-1} k[j](j - T)^2$$

Let's **go with the latter**.

Stochastic Gradient Descent

How do we calculate the weights?

One common answer: stochastic gradient descent.

1. Calculate

- derivative of sum of error E
- over all training inputs
- for all network parameters Θ .

2. Change Θ slightly in the opposite direction (to decrease error).

3. Repeat.

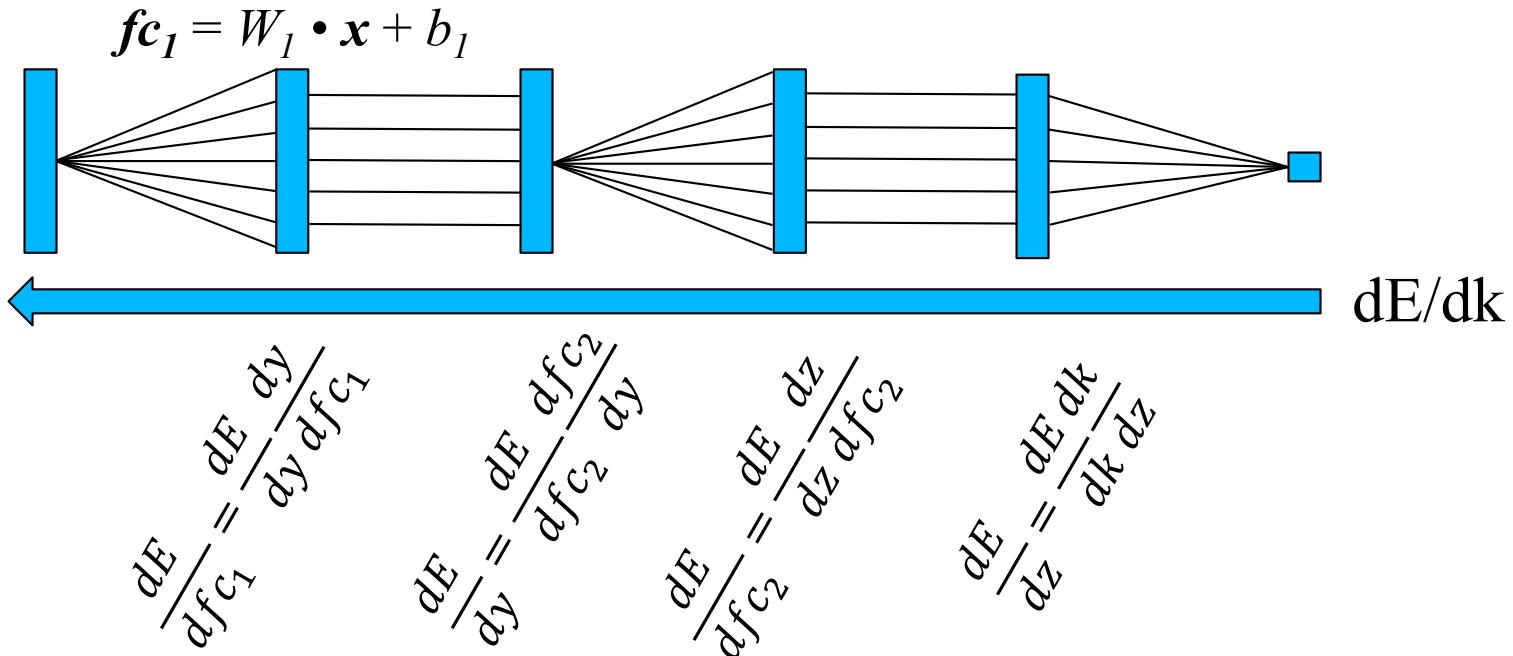
Stochastic Gradient Descent

More precisely,

1. For every input X ,
2. evaluate network to compute $k[i]$ (forward),
3. then use $k[i]$ and label T (target digit)
to compute error E .
4. Backpropagate error derivative to
find derivatives for each parameter.
5. Adjust Θ to reduce total E : $\Theta_{i+1} = \Theta_i - \epsilon \Delta \Theta$

(Update ϵ uses most accurate minima estimation.)

Parameter Updates and Propagation



Need propagated error gradient (from backward pass)

Weight update

$$\frac{dE}{dW_1} = \frac{dE}{dfc_1} \frac{dfc_1}{dW_1} = \frac{dE}{dfc_1} \chi$$

Need input (from forward pass)

Example: Gradient Update with One Layer

$$\Theta_{i+1} = \Theta_i - \varepsilon \Delta \Theta \quad W_{i+1} = W_i - \varepsilon \Delta W \quad \text{Parameter Update}$$

$$y = W \cdot x + b \quad \text{Network function}$$

$$\frac{dy}{dW} = x \quad \text{Network weight gradient}$$

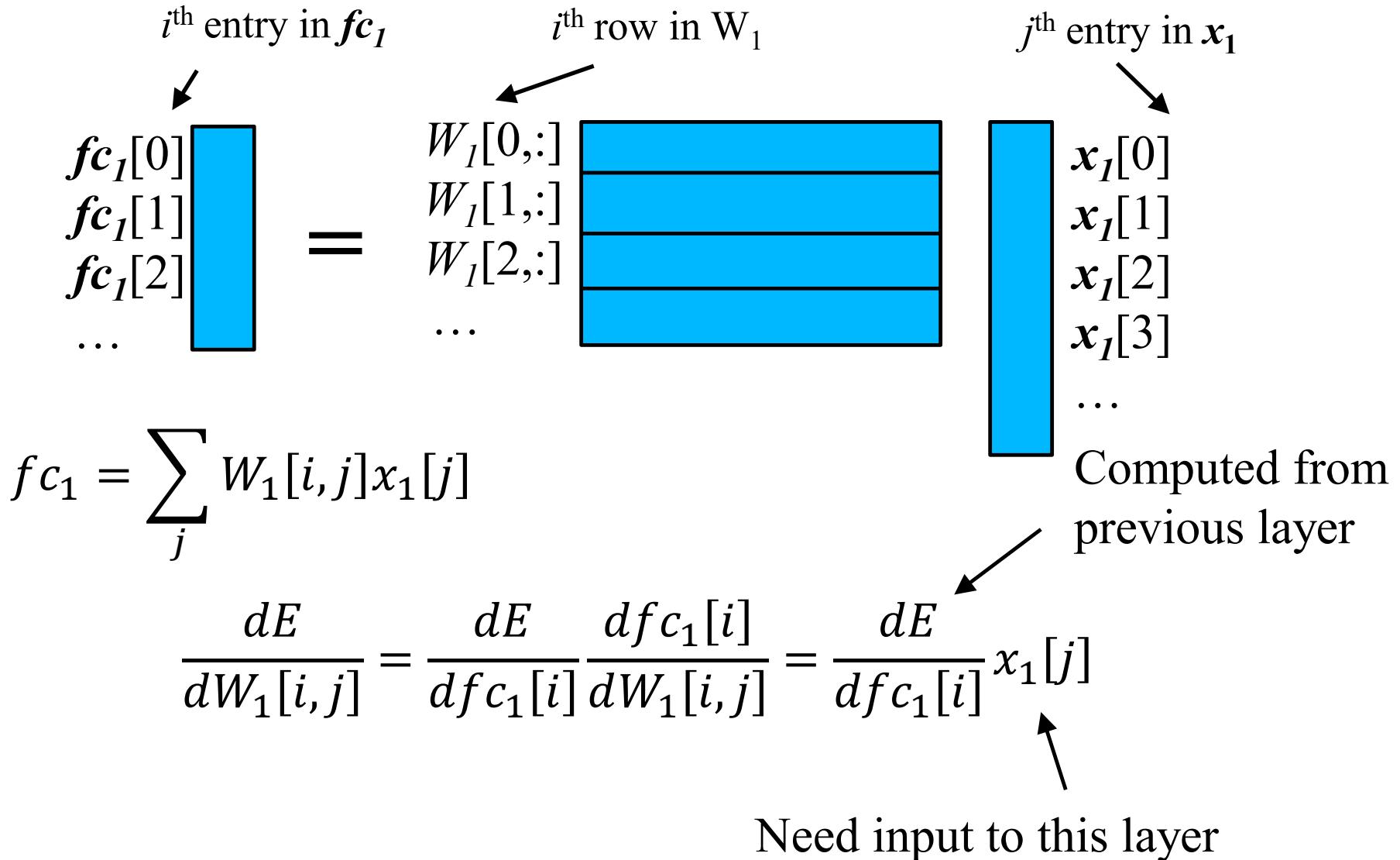
$$E = \frac{1}{2} (y - t)^2 \quad \text{Error function}$$

$$\frac{dE}{dy} = y - t = Wx + b - t \quad \text{Error function gradient}$$

$$\Delta W = \frac{dE}{dW} = \frac{dE}{dy} \frac{dy}{dW} \quad \text{Full weight update expression}$$

$$W_{i+1} = W_i - \varepsilon (Wx + b - t)x \quad \text{Full weight update term}$$

Fully-Connected Gradient Detail



Batched Stochastic Gradient Descent

- A training *epoch* (a pass through whole training set)
 - Set $\Delta\Theta = 0$
 - For each labeled image:
 - Read data to initialize input layer
 - Evaluate network to get y (forward)
 - Compare with target label t to get error E
 - Backpropagate error derivative to get parameter updates
 - Accumulate parameter updates into $\Delta\Theta$
 - $\Theta_{i+1} = \Theta_i - \varepsilon\Delta\Theta$

Aggregate gradient update most accurately reflects true gradient

Mini-batch Stochastic Gradient

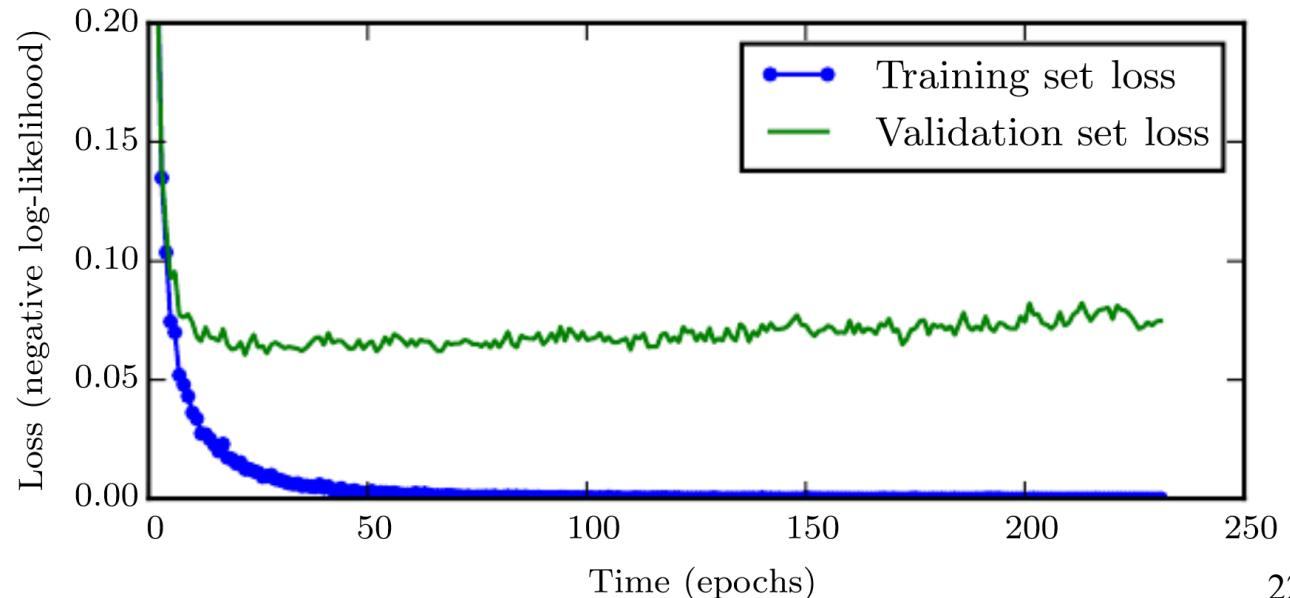
- For each batch in training set
 - For each labeled image in batch:
 - Read data to initialize input layer
 - Evaluate network to get y (forward)
 - Compare with target label t to get error E
 - Backpropagate error derivative to get parameter updates
 - Accumulate parameter updates into $\Delta\theta$
 - $\theta_{i+1} = \theta_i - \varepsilon \Delta\theta$

Balance between accuracy of gradient estimation and parallelism

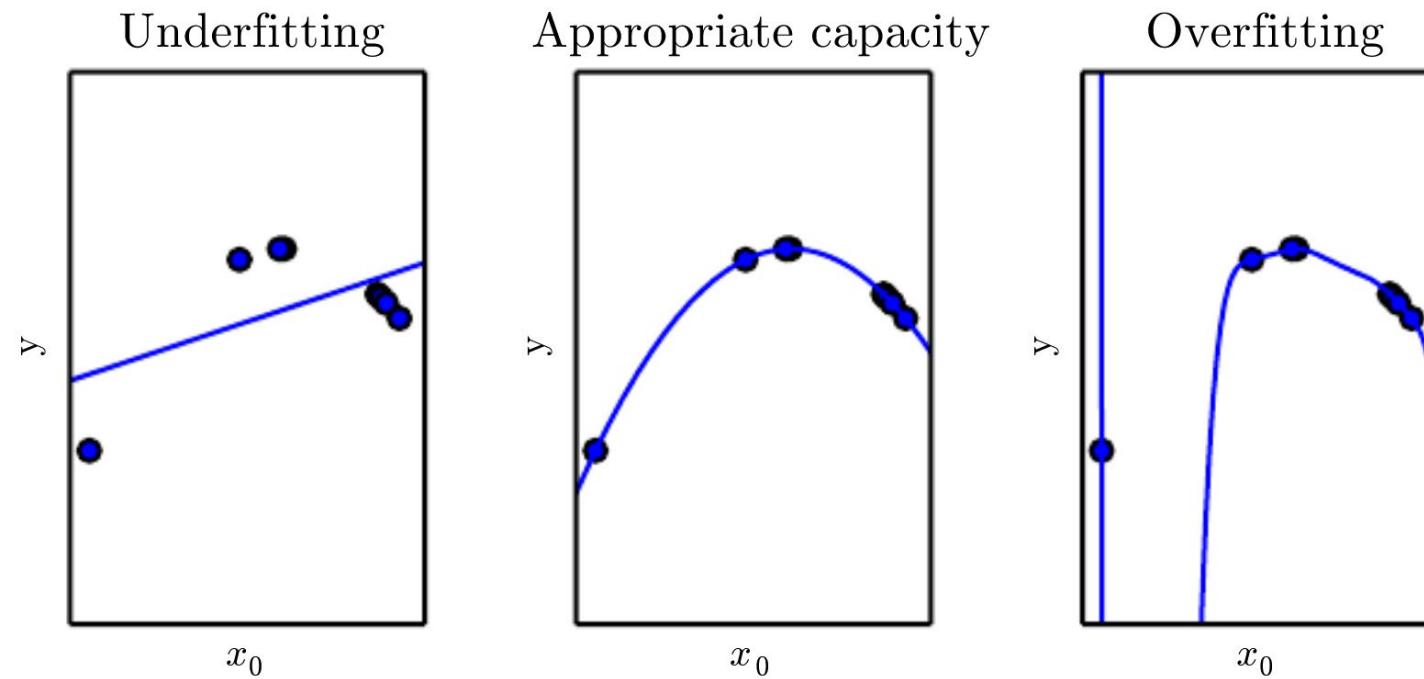
When is Training Done?

Split labeled data into *training* and *test* sets.

- Training data to compute parameter updates.
- Test data to check how model generalizes to new inputs (the ultimate goal!)
- The network can become *too good* at classifying training inputs!

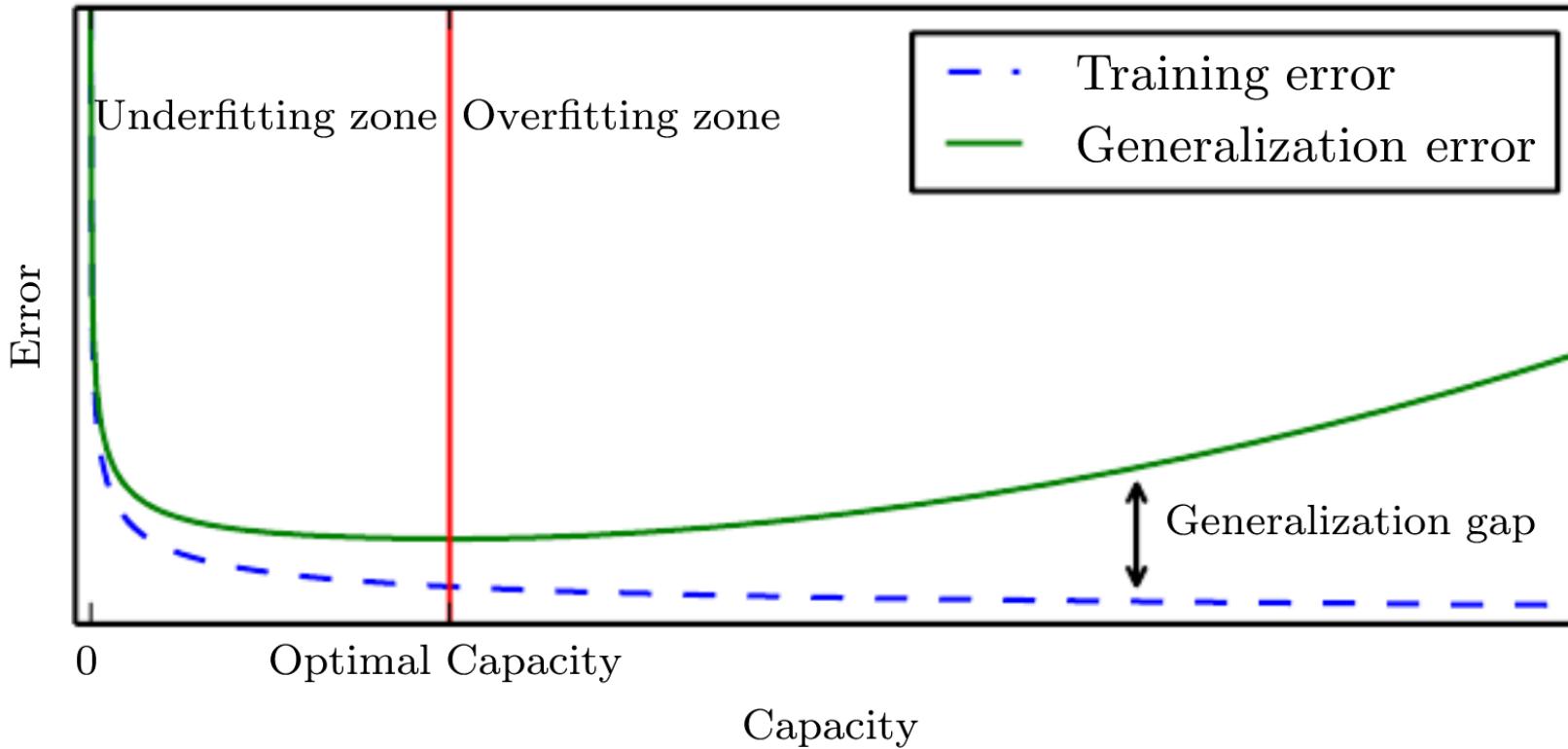


How Complicated Should a Network Be?



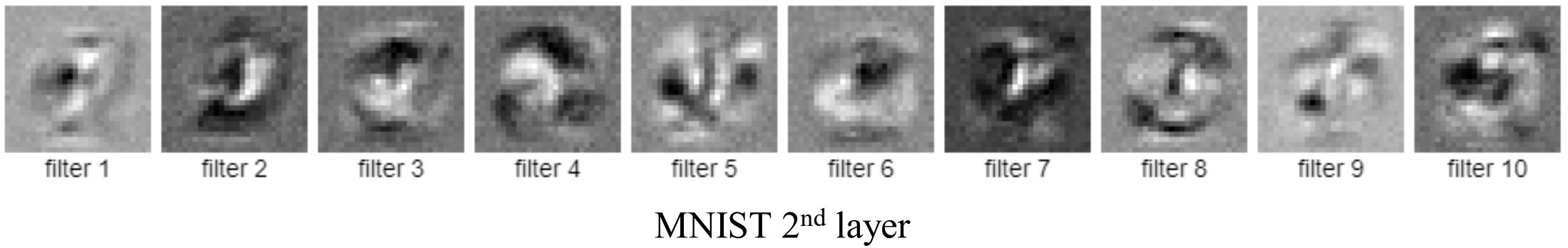
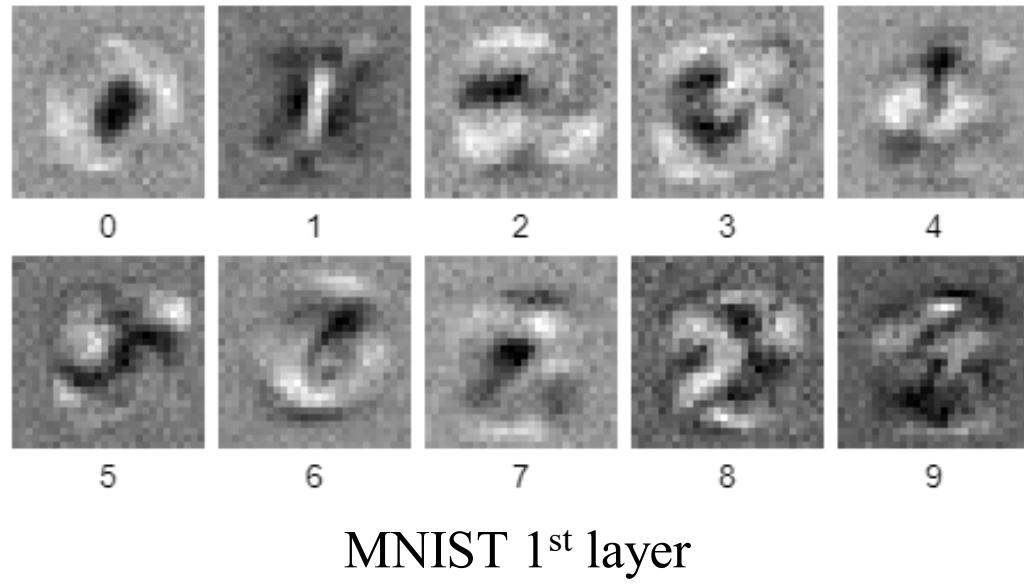
Intuition: like a polynomial fit. High-order terms improve fit, but add unpredictable swings for inputs outside the training set.

Overtraining Decreases Accuracy



If network works too well for training data,
new inputs cause big unpredictable output changes.

Visualizing Neural Network Weights



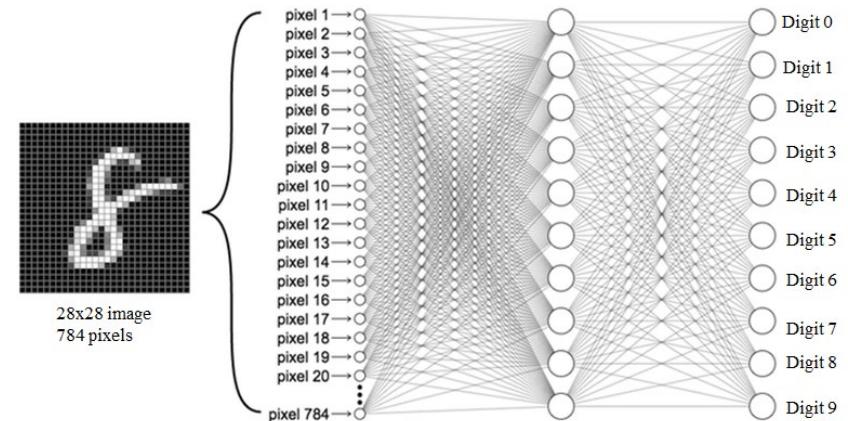
No Free Lunch Theorem

- Every classification algorithm has the same error rate when classifying previously unobserved inputs when averaged over all possible input-generating distributions.
- Neural networks must be tuned for specific tasks

Multi-Layer Perceptron (MLP) for an Image

Consider a 250×250 image...

- input: 2D image treated as 1D vector
- Fully connected layer is huge:
 - $62,500 (250^2)$ weights per node!
 - Comparable number of nodes gives $\sim 4B$ weights total!
- Need >1 hidden layer? Bigger images?
- Too much computation, and too much memory.



Traditional feature detection in image processing uses

- Filters \rightarrow Convolution kernels
- Can we use them in neural networks?

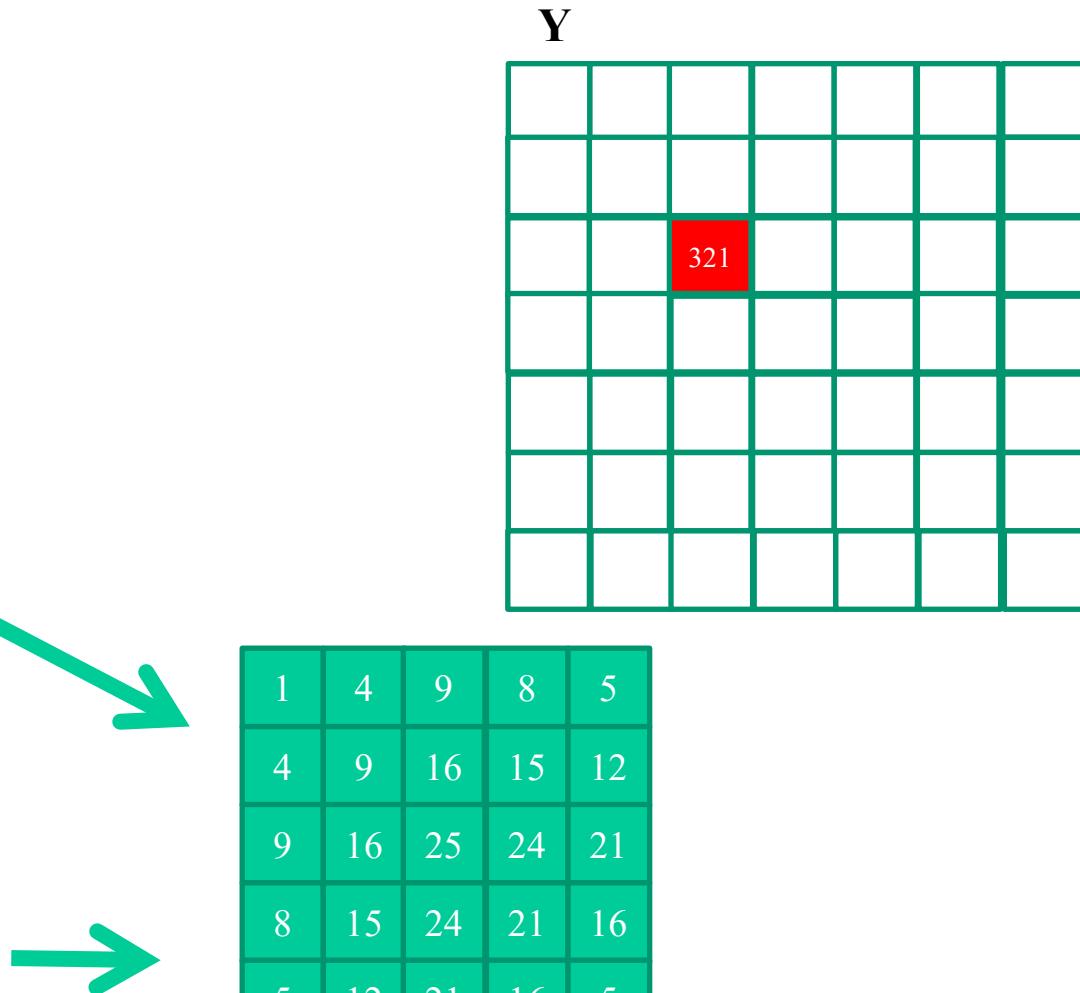
2-D Convolution

X

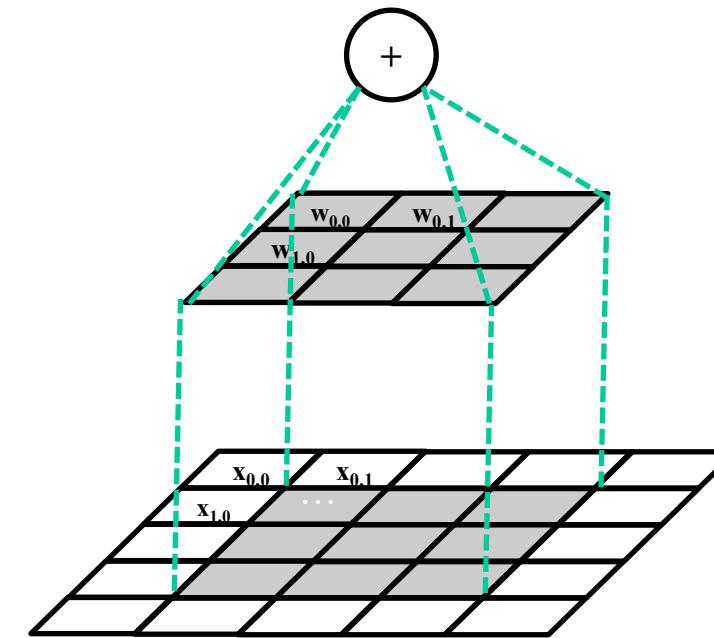
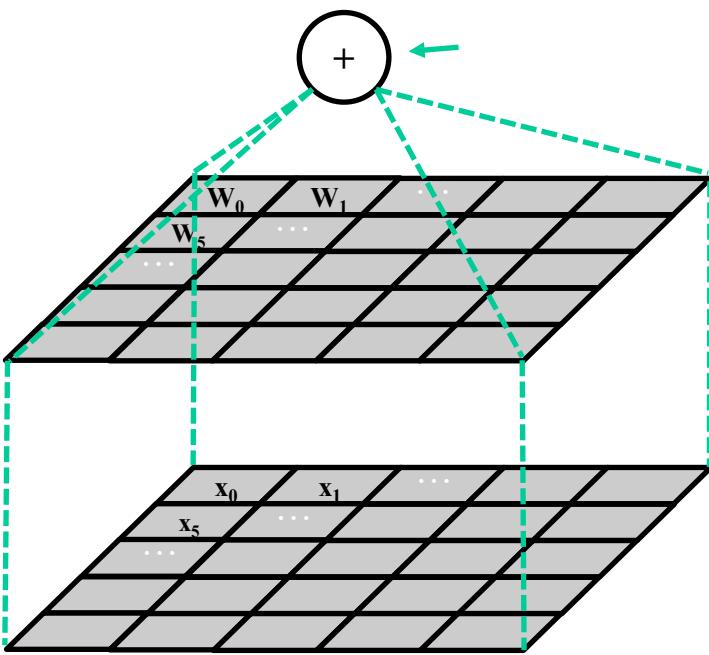
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	5	6
5	6	7	8	5	6	7
6	7	8	9	0	1	2
7	8	9	0	1	2	3

W

1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1



Convolution vs Fully-Connected (Weight Sharing)



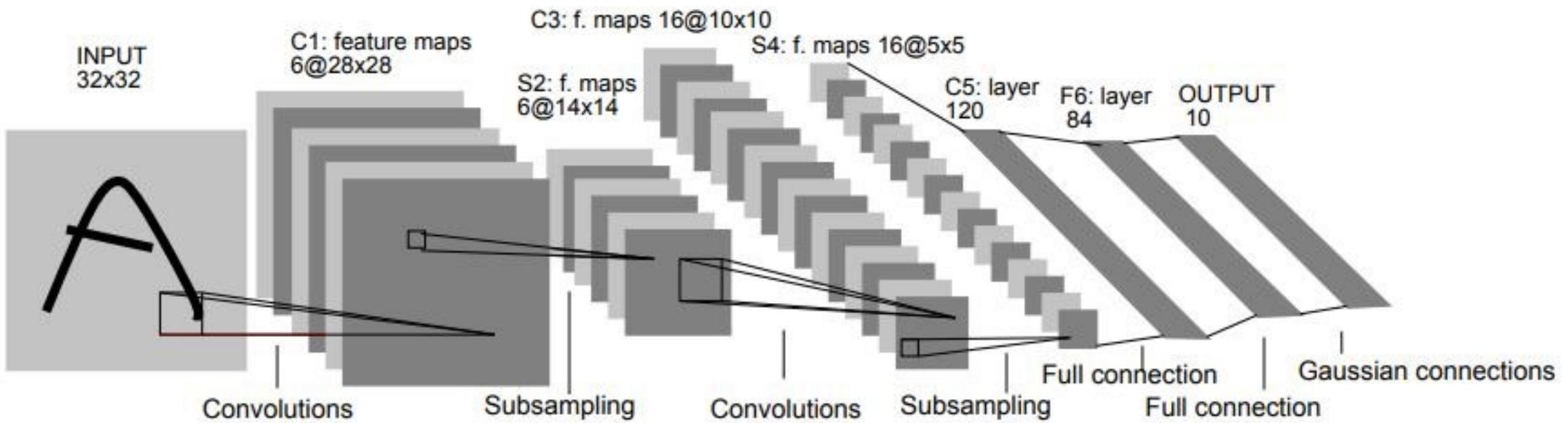
Convolution Naturally Supports Varying Input Sizes

- As discussed so far,
 - perceptron layers have fixed structure, so
 - number of inputs / outputs is fixed.
- Convolution enables variably-sized inputs (observations of the same kind of thing)
 - Audio recording of different lengths
 - Image with more/fewer pixels

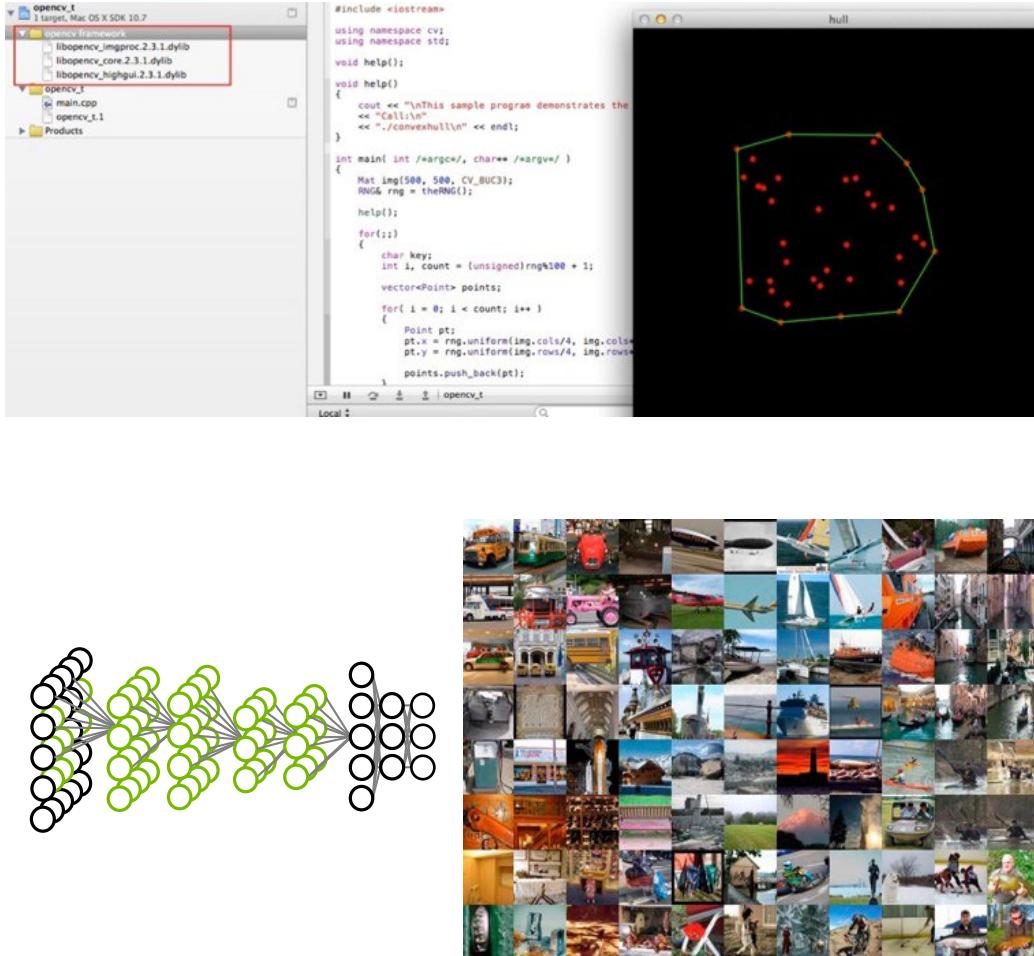
Example Convolution Inputs

	Single-channel	Multi-channel
1D	audio waveform	Skeleton animation data: 1-D joint angles for each joint
2D	Fourier-transformed audio data Convolve over frequency axis: invariant to frequency shifts Convolve over time axis: invariant to shifts in time	Color image data: 2D data for R,G,B channels
3D	Volumetric data (example: medical imaging)	Color video: 2D data across 1D time for R,G,B channels

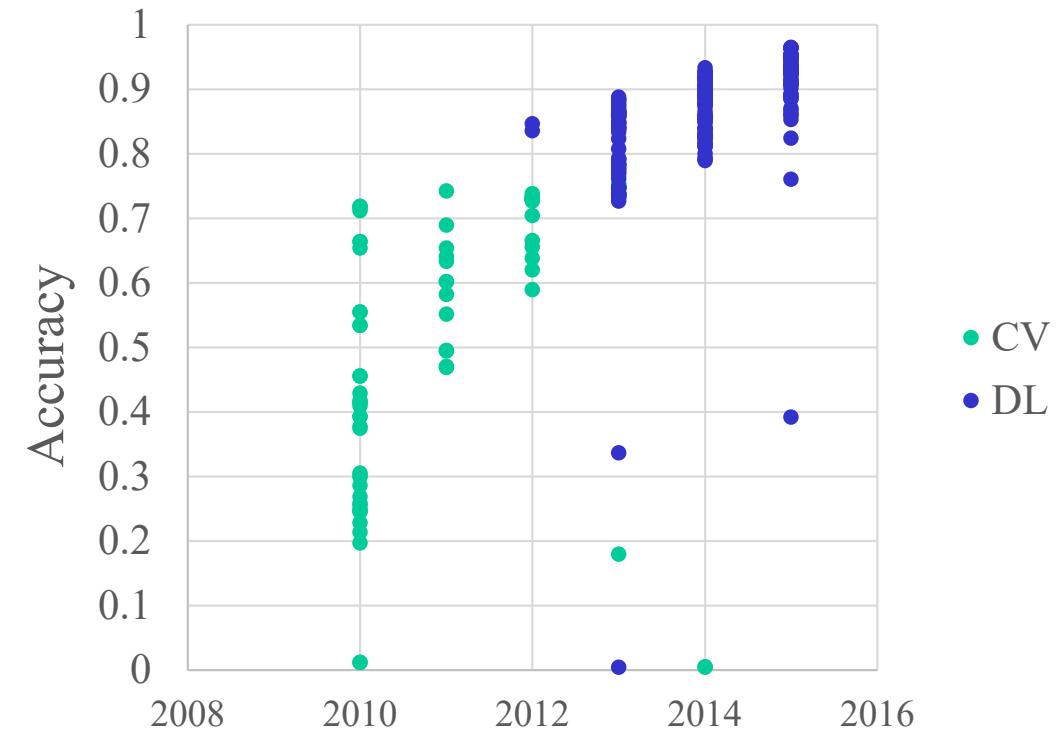
LeNet-5:CNN for hand-written digit recognition



Deep Learning Impact in Computer Vision

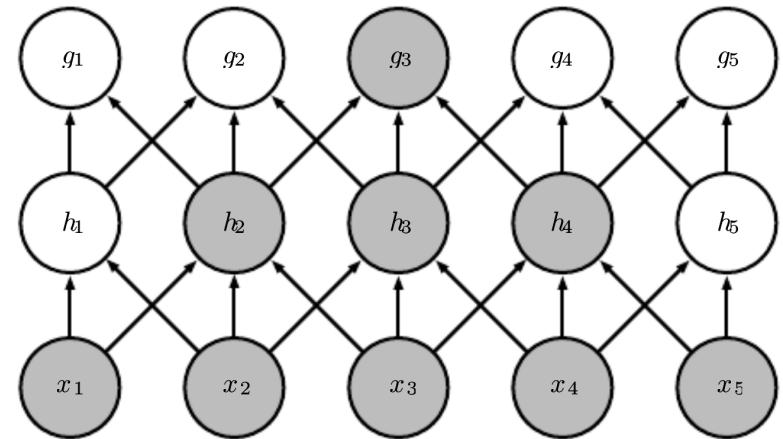


The Toronto team used GPUs and trained on 1.2M images in their 2012 winning entry at the Large Scale Visual Recognition Challenge



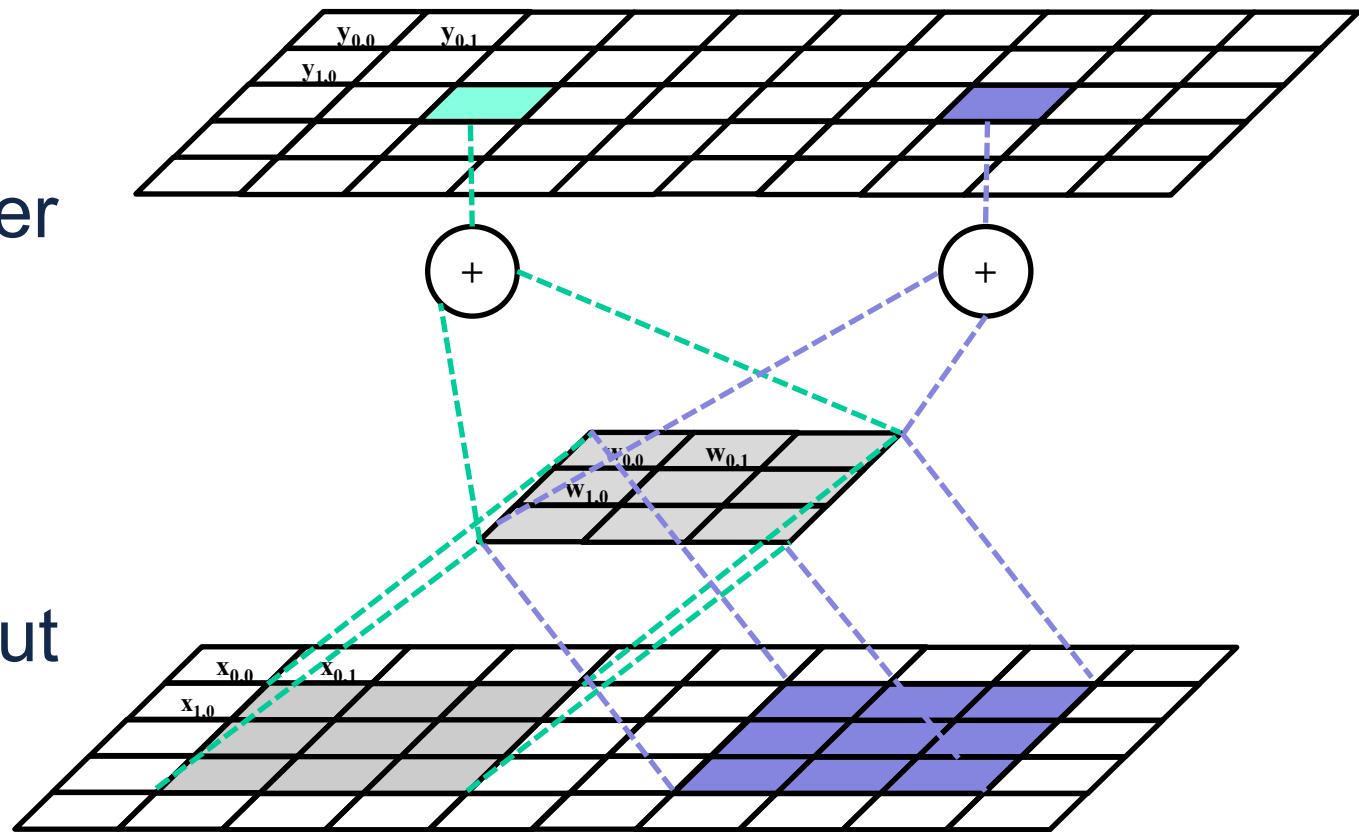
Why Convolution

- Sparse interactions
 - Meaningful features in small spatial regions
 - Need fewer parameters (less storage, better statistical characteristics, faster training)
 - Need multiple layers for wide receptive field



Why Convolution

- Parameter sharing
 - Kernel mask is applied repeatedly computing layer output
- Equivariant Representations
 - If input is translated, output is similarly translated
 - Output is a map of where features appear in input



Convolution

- 2-D Matrix
- $Y = W \otimes X$
- Kernel smaller than input:
smaller receptive field
- Fewer Weights

MLP

- Vector
- $Y = w x + b$
- Maximum receptive
field
- More weights



ANY MORE QUESTIONS?



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 10: Machine Learning and Deep Learning

Course Reminders

- Lab 3 is due this week
- Lab 4 is due next week
- Midterm 1 is on Tuesday, March 7th
 - On-line, everybody will be taking it at the same time
 - Tuesday, March 7th 7:00pm-8:30pm US Central time
 - Includes materials from Lecture 1 through Lecture 9
- Project Milestone 1: Baseline CPU implementation is due Friday March 10th
 - Project details to be posted next week

Objective

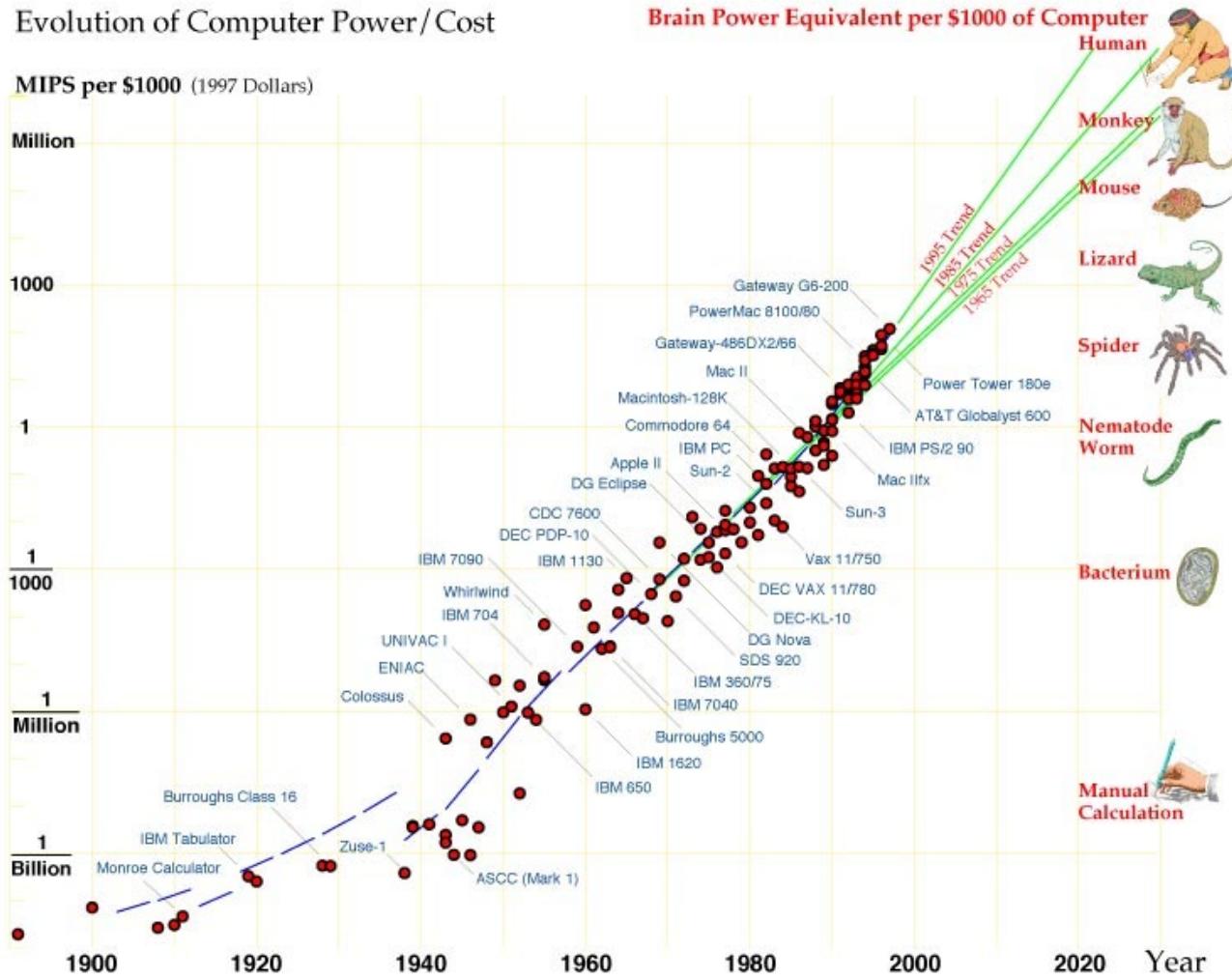
- To understand the application areas for machine learning.
- To learn the basic strategy for machine learning applications.
- To understand the extension to deep learning (mostly a research pitch).
- To learn about a Multi-Layer Perceptron.

Perspective is Important

- **Chips are cheaper than ever**
- Unlike humans, digital systems offer
 - **high-speed computation**,
 - **low capital investment**
(purchase vs. training a human), and
 - **negligible operations cost** (no salary!)
- **If computer outperforms** (or even matches) a **human, use a computer**
- **Industry has done so for 40-50 years now**

Evolution of Computer Power/Cost

Evolution of Computer Power/Cost



Computing has evolved under the premise that some day, computing machines will be able to mimic general human intelligence.

From a computing power perspective, Moore's Law has fueled the idea of the intelligent machine. Hardware has gotten 2x faster every 18 months.

The software, though, has been a vexing open question.

<https://jetpress.org/volume1/moravec.htm>

What is Machine Learning?

- **Machine learning**: important method of building applications whose logic is not fully understood
- Typically **by example**:
 - **use labeled data** (matched input-output pairs)
 - **to represent** desired **relationship**
- **Iteratively adjust program logic** to produce desired/approximate answers (called **training**)

Types of Learning Tasks

- classification
 - Map each input to a category
 - Ex: object recognition, chip defect detection
- regression
 - Numerical prediction from a sequence
 - Ex: predict tomorrow's temperature
- transcription
 - Unstructured data into textual form
 - Ex: optical character recognition

More Advanced Learning Tasks

- translation
 - Convert a sequence of symbols in one language to a sequence of symbols in another
- structured output
 - Convert an input to a vector with important relationships between elements
 - Ex: natural language sentence into grammatical structure
- others
 - Anomaly detection, synthesis, sampling, imputation, denoising, density estimation, genetic variant calling

Why Machine Learning Now?

- **Computing Power**
 - GPU computing hardware and programming interfaces such as CUDA has enabled very fast research cycle of deep neural net training
- **Data**
 - Lots of cheap sensors, cloud storage, IoT, photo sharing, etc..
- **Needs**
 - Autonomous Vehicles, Smart Devices, Security, Societal Comfort with Tech, Health Care

Test Cycle Time is Important

You've all written code...

- code, test, code, test, code, test
- integrate, test, test, test
- and test again!

But how long is the code, test cycle?

Depends what you're building.

What's your longest?

Your Cycle Times are Probably Small

- In college, **10k lines** took **$\frac{1}{2}$ hour** to compile on my PC.
- In grad. school, **100k lines** took
 - **$\frac{1}{2}$ hour** to compile on my workstation, or
 - **2 minutes** on our cluster (research platform)
- In ECE 435 (networking lab), students needed
 - **$\frac{1}{2}$ hour** to reinstall Linux after a bad bug
 - (Ever had a good bug?)
- Gene sequencing / applications can take **two weeks**

We're all a little spoiled...

Why Machine Learning Again?

- In 2007, **programmable GPUs accelerated the training cycle**
- Today, **new chip designs** for learning applications **have further accelerated**
- Led to a resurgence of interest
 - in Computer Vision, Speech Recognition, Document Translation, Self Driving Cars, Data Science...
 - all tasks that **human brains solve regularly, but** for which **we have struggled to express solutions** systematically.

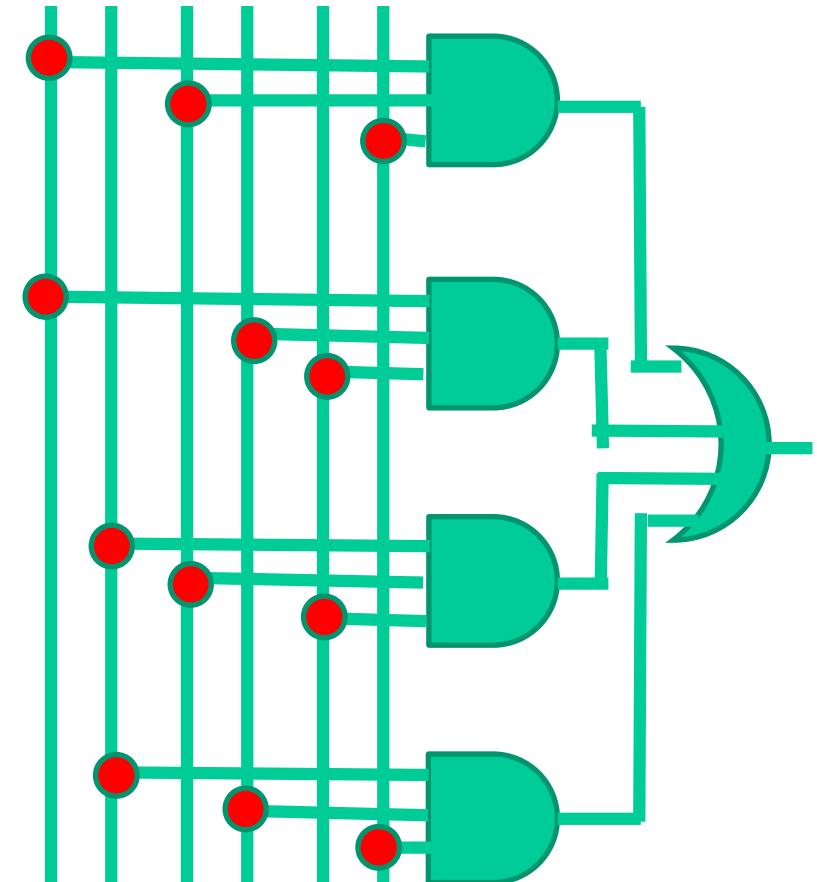
Many Problems are Still Hard

- Speed is not a panacea
- Many tasks still require human insight
 - for network structure and feature selection
 - for effective input and output formats, and
 - for production of high-quality labeled data.
- Other trends sometimes help: ubiquitous computing enables crowdsourcing, for example.

Many Problems Have Systematic Solutions

Example: building a Boolean function from a truth table

Input			output
a	b	c	
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



What if We Lack a Truth Table?

- Make enough observations to construct a rule
 - $000 \rightarrow 0$
 - $011 \rightarrow 0$
 - $100 \rightarrow 1$
 - $110 \rightarrow 0$
- If we cover all input patterns,
we can construct a truth table!

Many Problems are Too Large

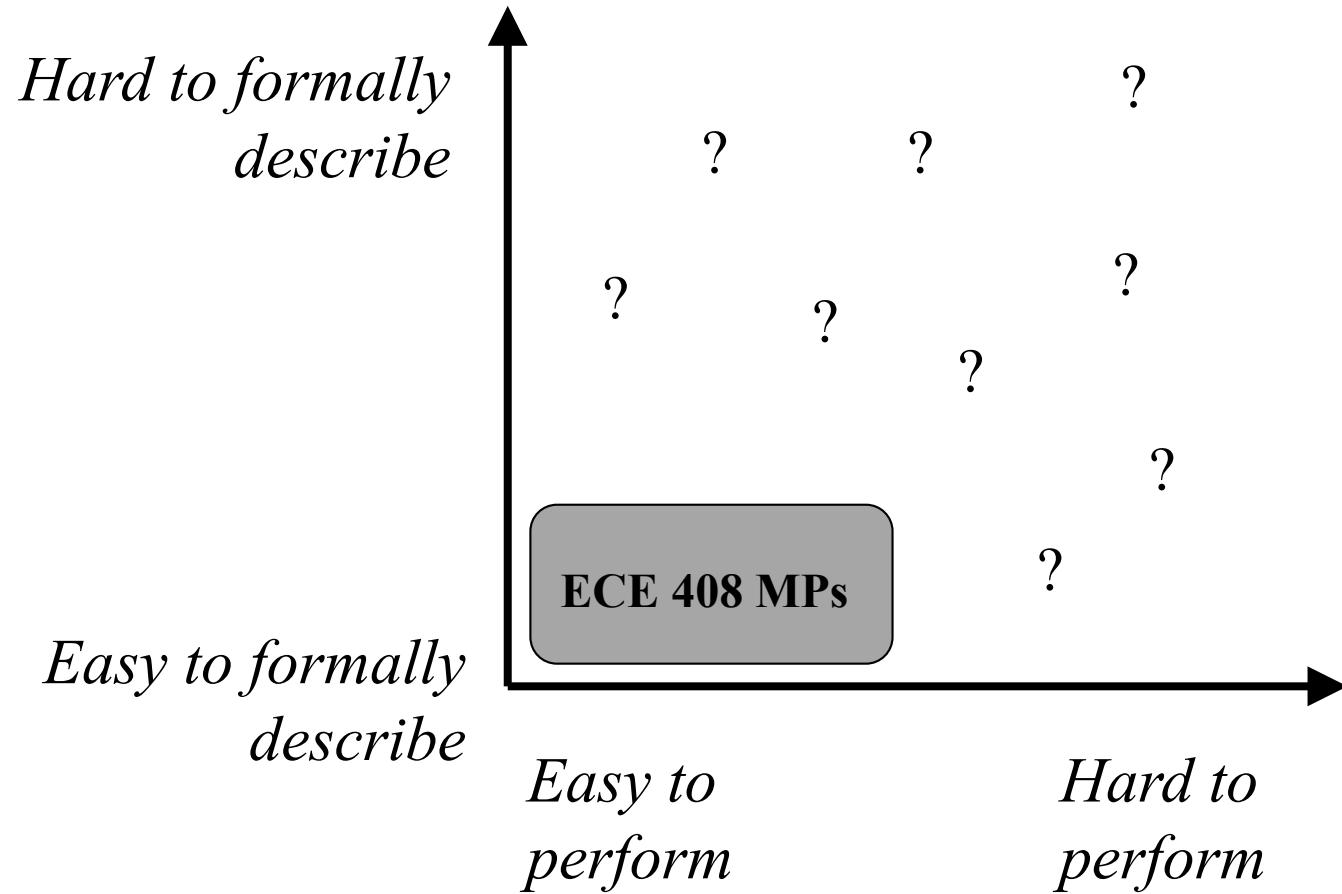
- The logic formulation of a 32x32-pixel (small) image recognition problem involves
 - 1024*8 bit input,
 - which will have a truth table of 2^{8196} entries
- If we managed to collect and label 1 billion ($\sim 2^{32}$) images as training data
 - We cover only $2^{32} / 2^{8196} = 1 / 2^{8164}$ of the truth table
 - Solution - learning processes that exploits features

Features in our logic example

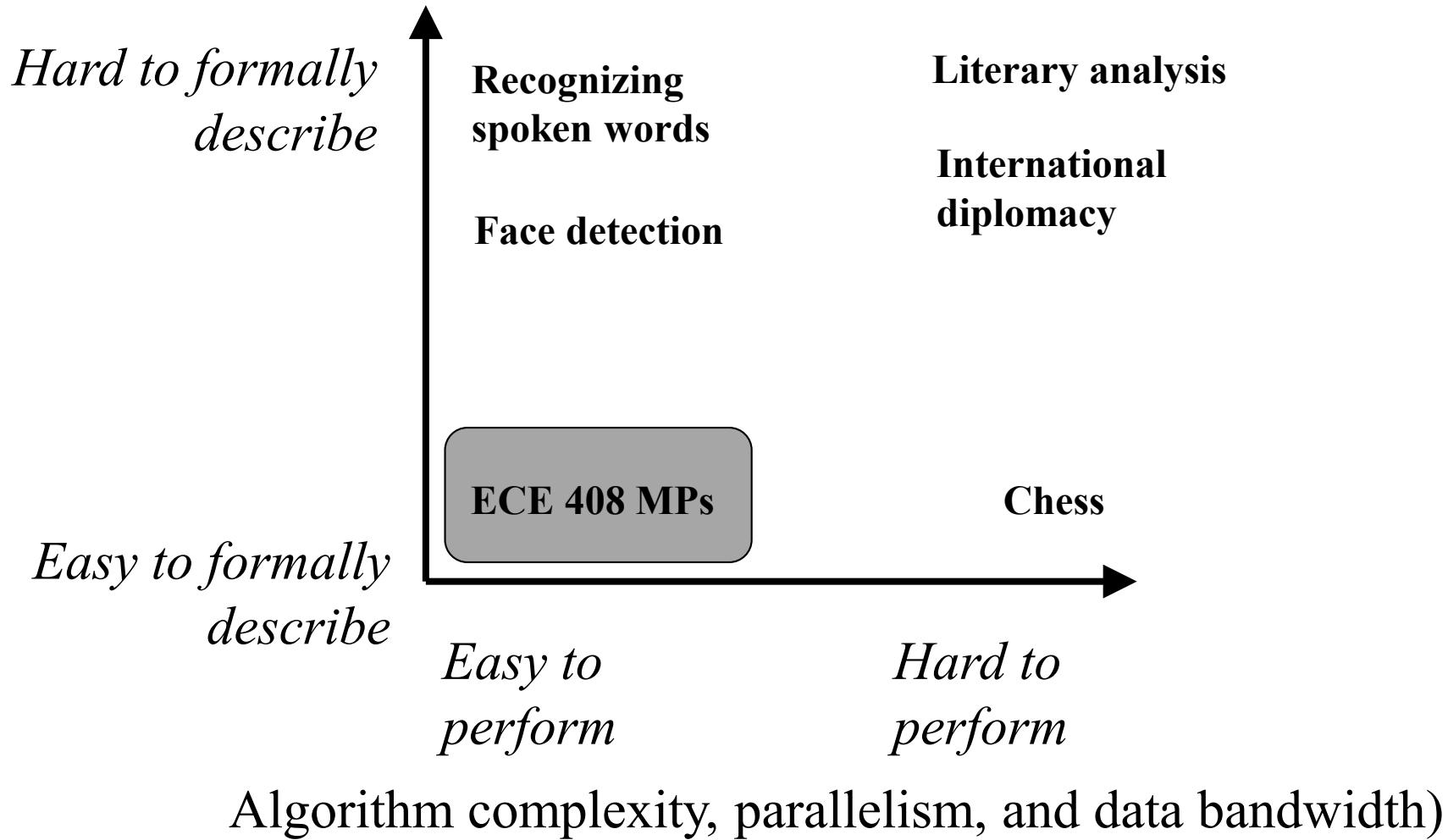
Input			output
a	b	c	
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Feature 1: bit patterns with odd number of 1's result in output 1
Feature 2: bit patterns with even number of 1's result in output 0

Types of Problems

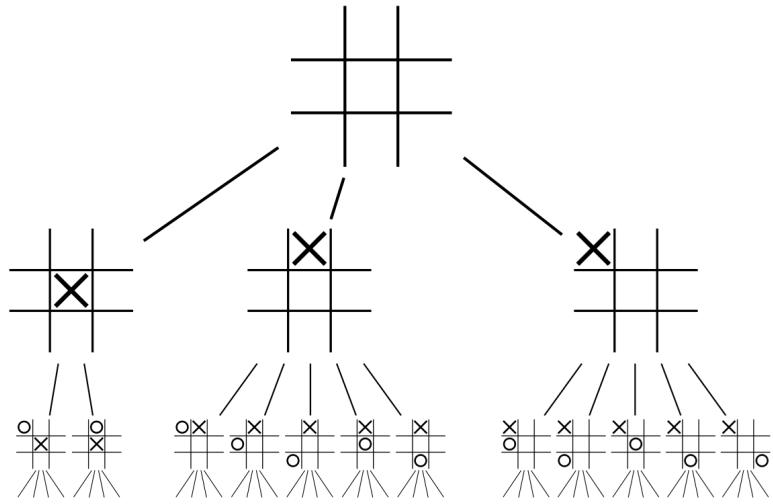


Types of Problems



Chess as an AI Success (1)

- Easy to formalize
 - 64 locations, 32 pieces
 - Well-defined, allowable moves
- Score each leaf in a tree of possible board positions
- Proceed down path that results in best position



2-ply game tree for tic-tac-toe

Chess as an AI Success (2)

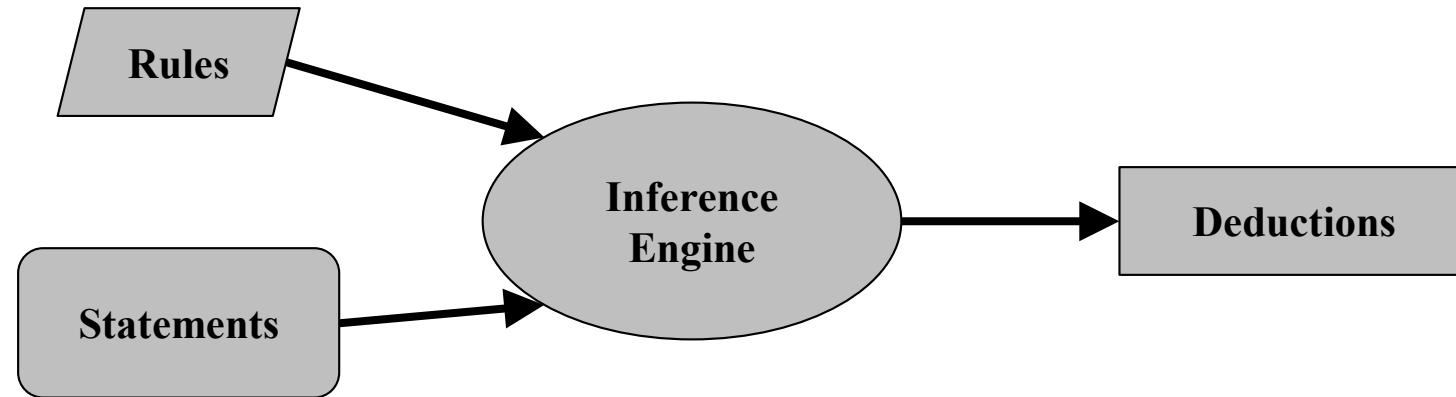


Deep Blue defeated Gary Kasparov
in 1997

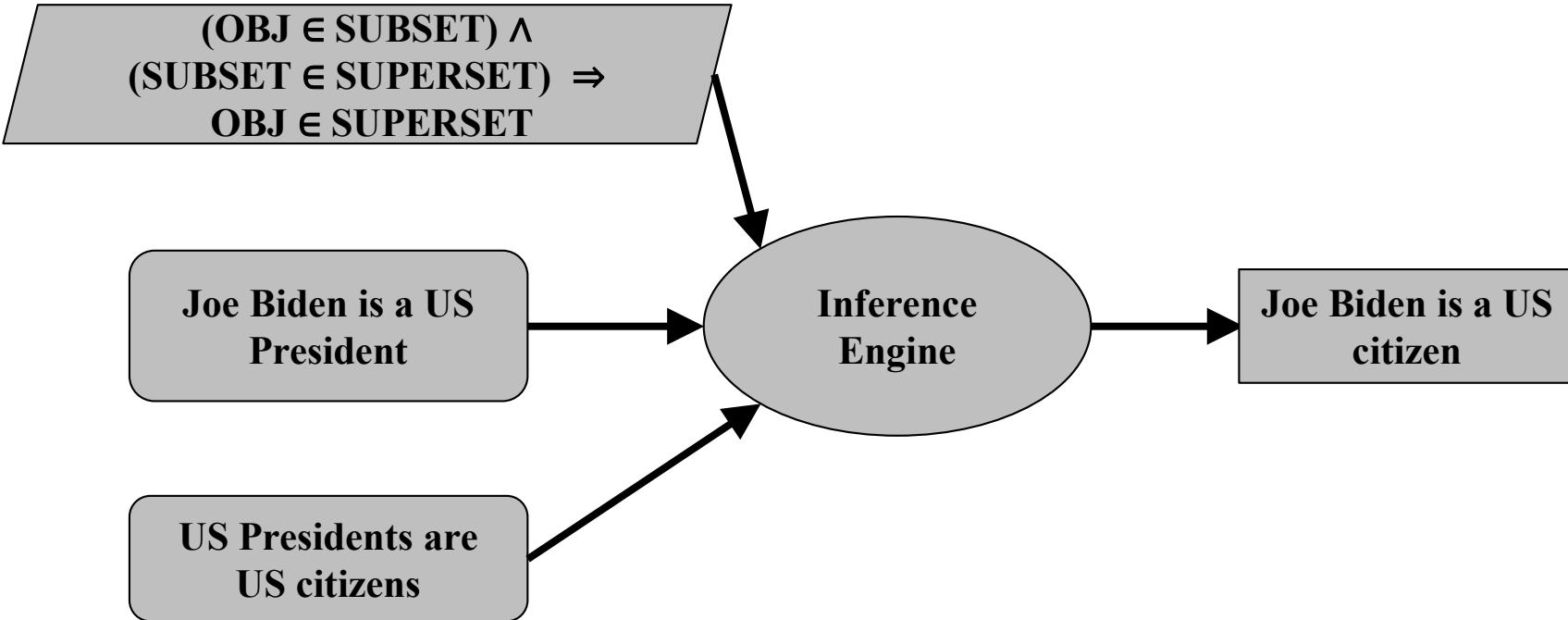
- Hard to perform
 - ~30 legal moves per position
 - 1,015 moves for 10-ply lookahead
 - 30 years of compute at 1M positions/sec
- Heuristics, pruning, parallel search, fast computers

Cyc: Extending Rule-based Systems to the Real World

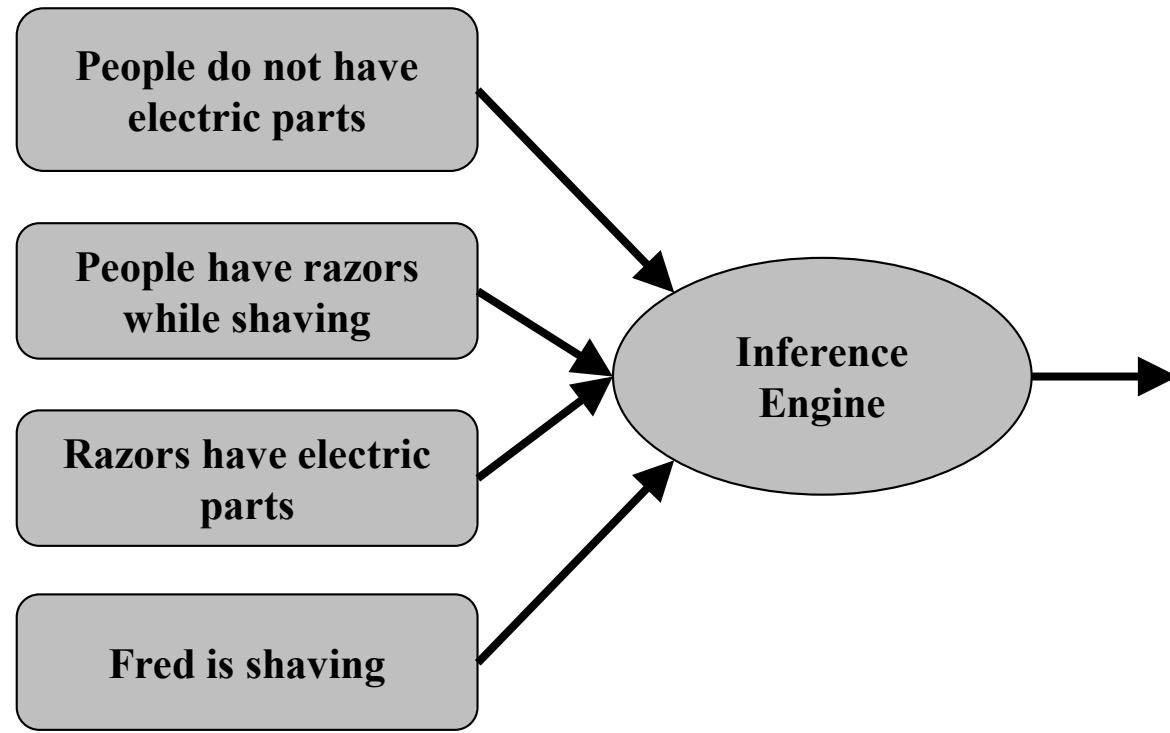
- Comprehensive ontology and knowledge base of common sense
- Cyc reasons about formal statements about the world



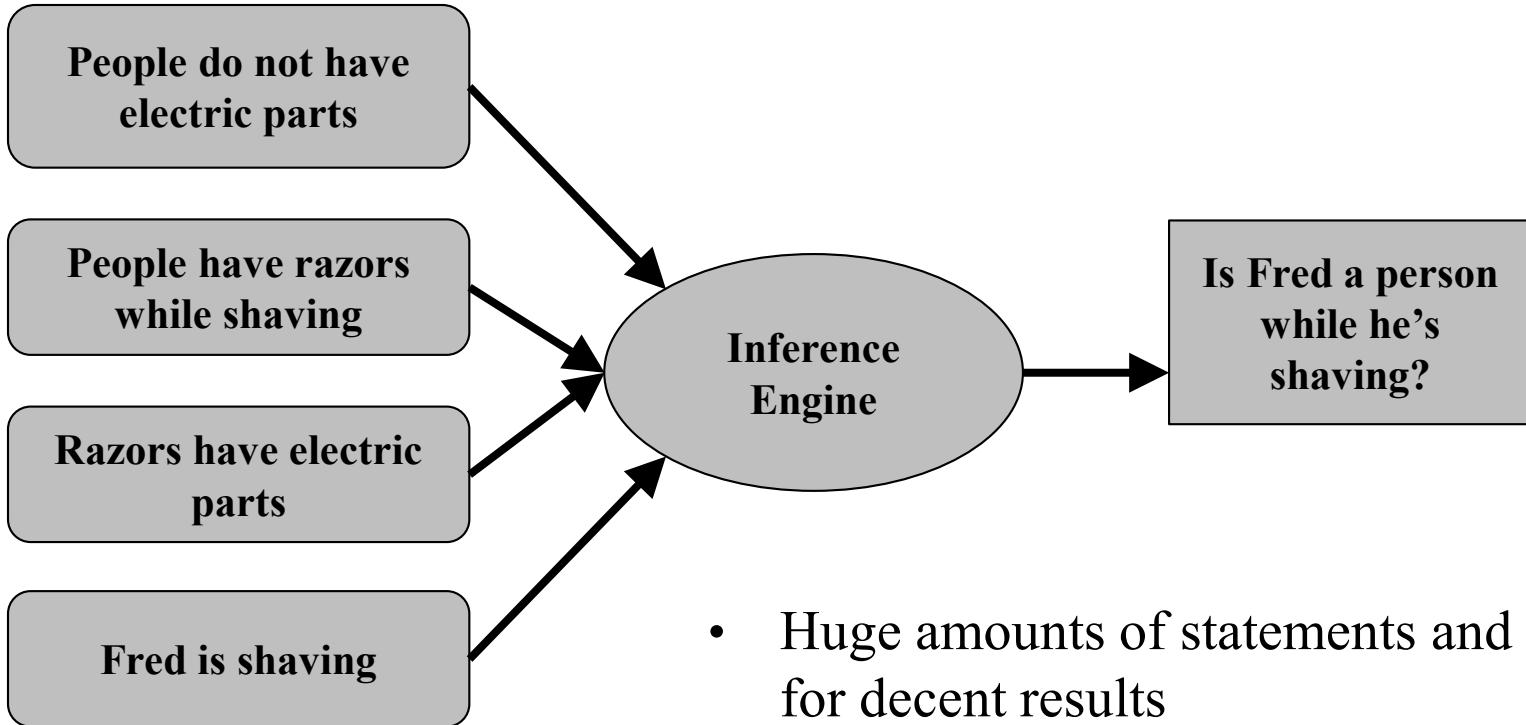
Cyc: A Simple Example



Cyc: FredWhileShaving

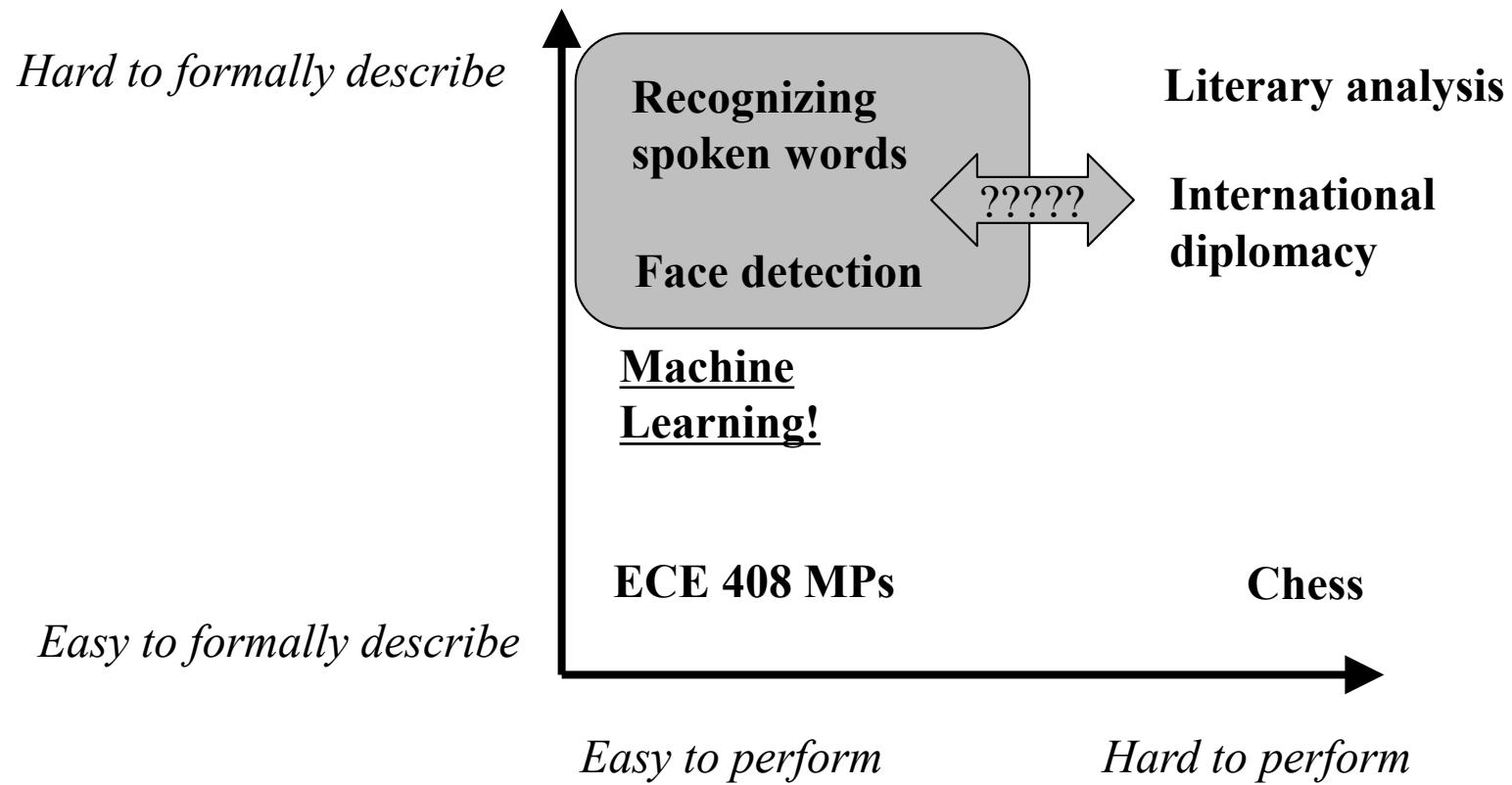


Cyc: FredWhileShaving



- Huge amounts of statements and rules for decent results
- Cannot learn new rules or statements on its own

Types of Problems



The “Machine Learning” Approach

Challenge

Hard to formalize the problem.

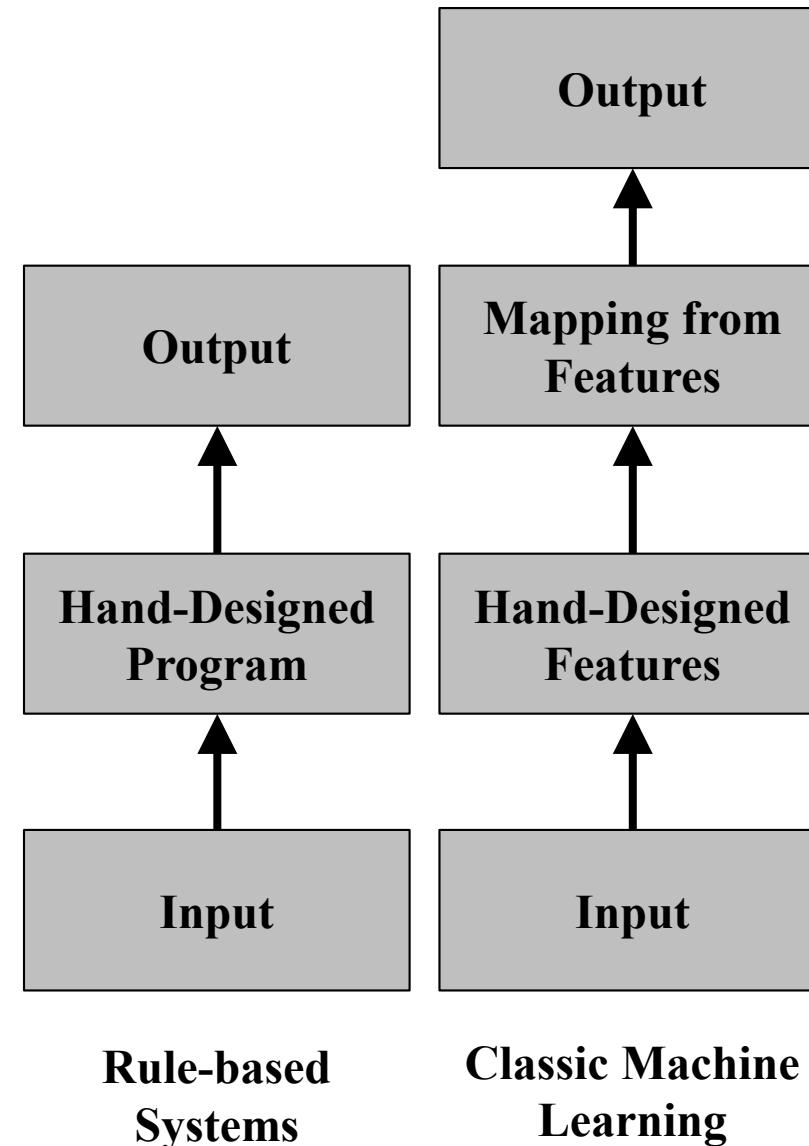
Solution

Don’t formalize the problem.

Let the machine learn from experience.

Classic Machine Learning

- Humans choose features
- Learn how features are associated with outputs

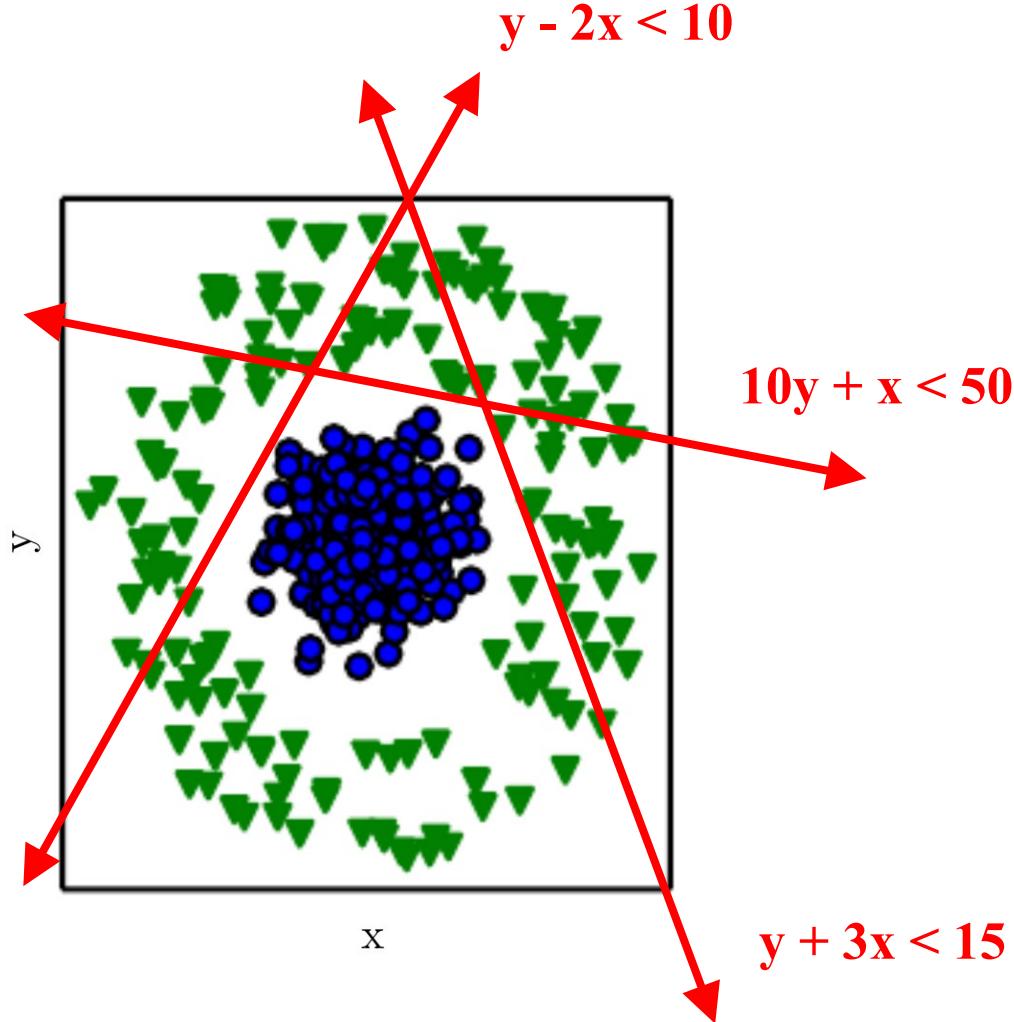


You may have heard of...

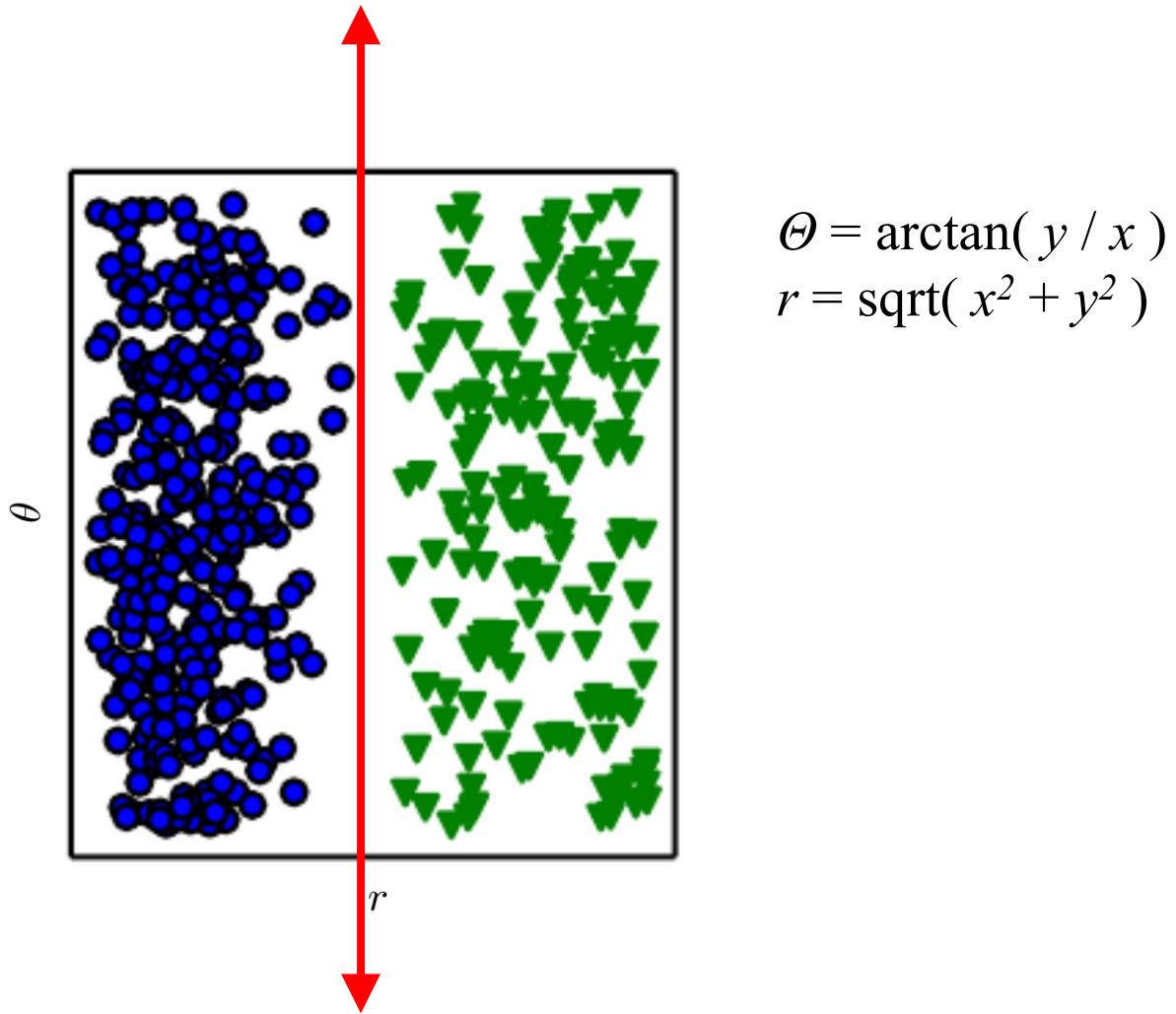
- Naïve Bayes:
features as independent contributors to output
- Logistic Regression:
 - learn how to weight each feature's contribution to output,
 - usually through gradient descent*

*more on this topic later in these slides

Data Representation is important!



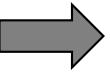
Data Representation is important!



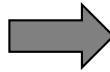
Different Features for Different Tasks



Image



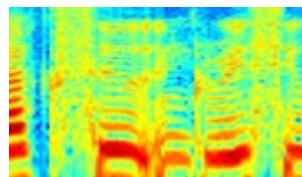
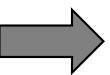
Vision Features



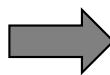
Detection



Audio



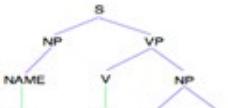
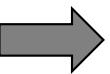
Audio Features



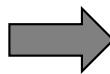
Identify
Speaker



Text



Text Features



Text classification, machine
translation, information retrieval

Which Data Features are Relevant

- Detecting a car in an image
- Cars have wheels → presence of a wheel?
- Can we describe pixel values that make up a wheel?
 - Circle-shaped?
 - Dark around perimeter?
- But what about?
 - Occlusion, perspective, shadows, white-walled tires, ...

Identify Factors of Variation that Explain Data

- Unobserved objects or forces that affect observed quantities
- Mental constructs that provide simplifying explanations or inferred causes
- Ex: speech
 - Age, sex, accent, words being spoken
- Ex: car
 - Position, color, angle of sun
- Many factors influence each piece of observed data

Representation Learning Approach

Challenge

Which data features are relevant?

Solution

Learn the features too!

(Looking ahead)

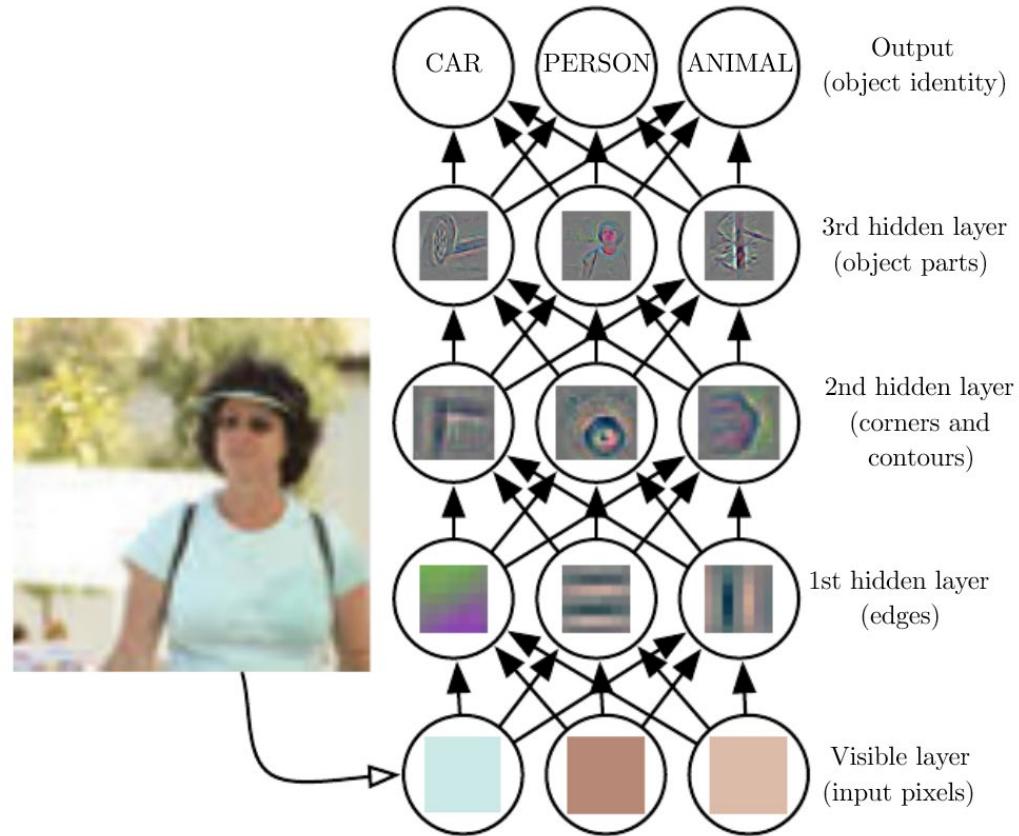
Deep Learning: a deep hierarchy of features

Machine Learning

- Ability to acquire knowledge by extracting patterns from data

Deep Learning

- A type of representation learning
- Representations expressed in terms of other representations



Deep Learning Approach

Challenge

Hard to formalize the problem?

Which data features are relevant?

Solution

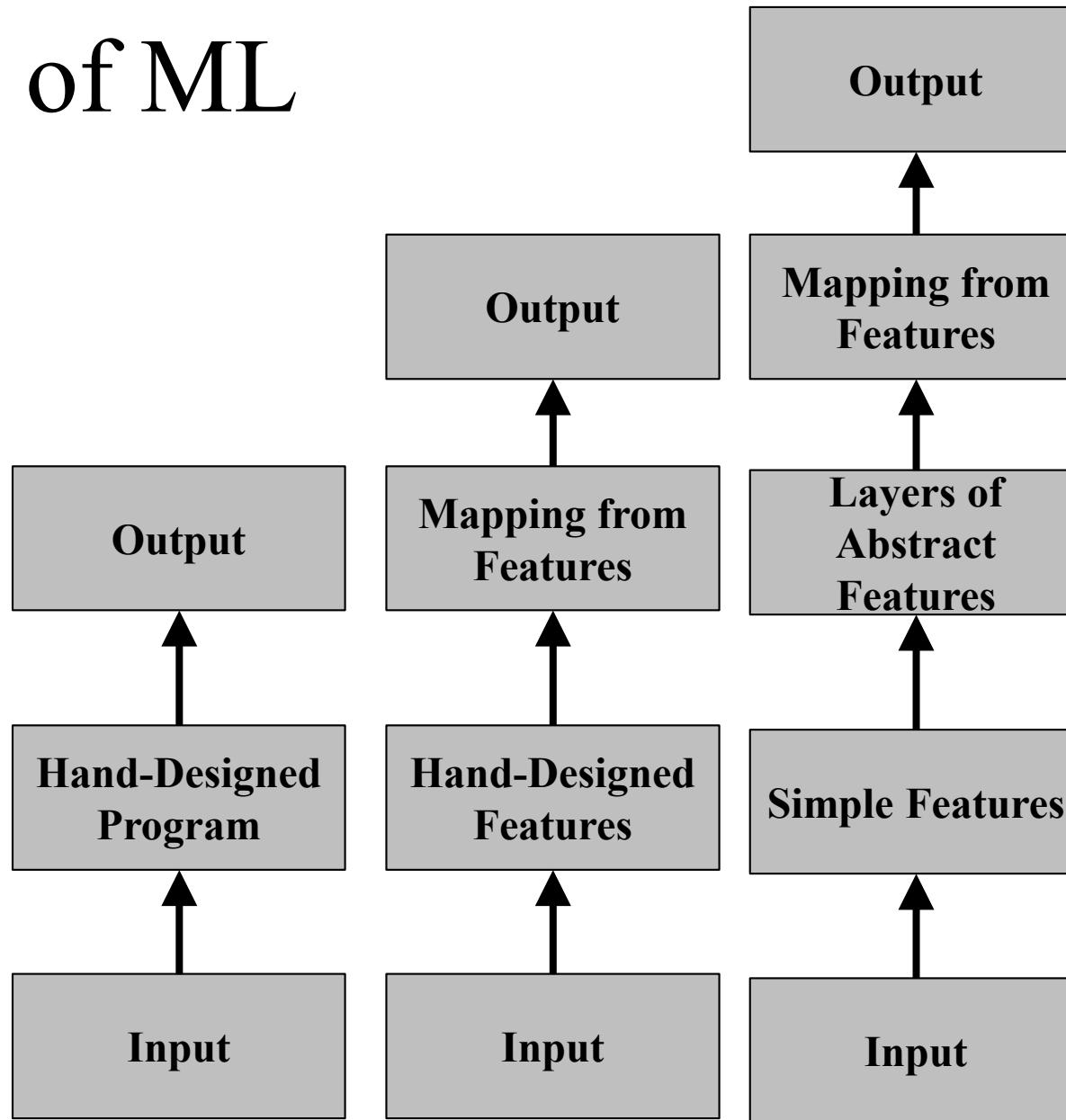
Don't formalize the problem

Let the machine learn from experience

Hierarchy of concepts to capture simple and complicated features

Learn the hierarchy too!

Evolution of ML



Let's Look at Classification

In a **classification problem**, we model

- a function mapping an input vector to a set of C categories: $F: \mathbb{R}^N \rightarrow \{1, \dots, C\}$,
- where the function **F is unknown.**

We **approximate F using a set of functions f**

- parametrized by a (large) set of weights, Θ
- that map from a vector of N real values* to an integer value representing a category:
- for category i , **$\text{prob}(i) = f(x, \Theta)$**

*floating-point values

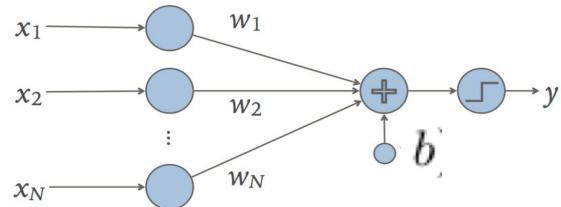
Linear Classifier (Perceptron)

- our formulation: $y = f(\mathbf{x}, \Theta)$

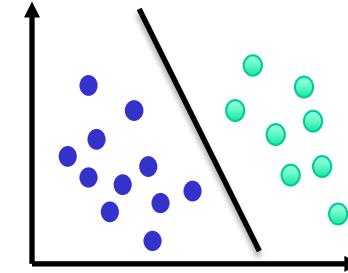
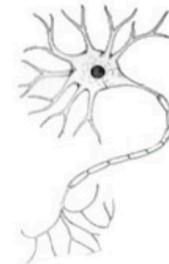
$$\Theta = \{W, b\}$$

$$y = \text{sign}(W \cdot \mathbf{x} + b)$$

The perceptron



The neuron

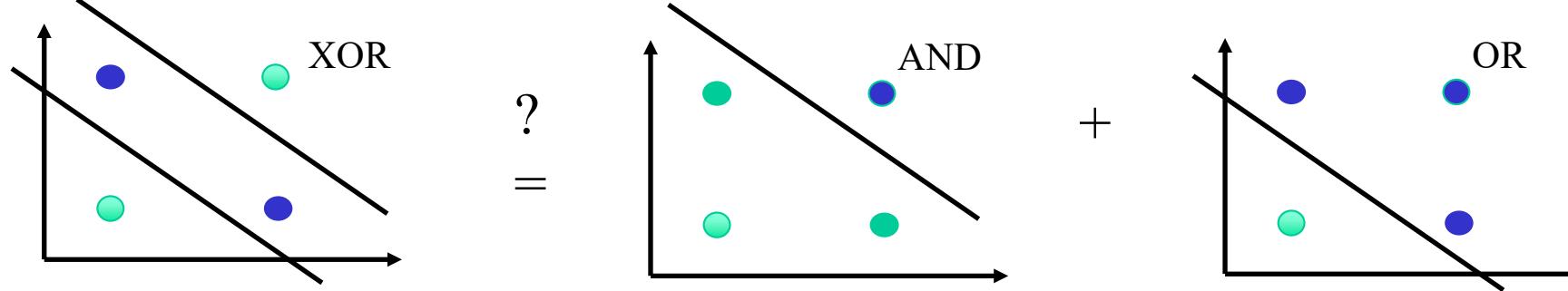


- Dot product + Scalar addition:

$$y = W \cdot \mathbf{x} + b$$

Diagram labels: output, weight, bias, input.

Can we learn XOR with a Perceptron?

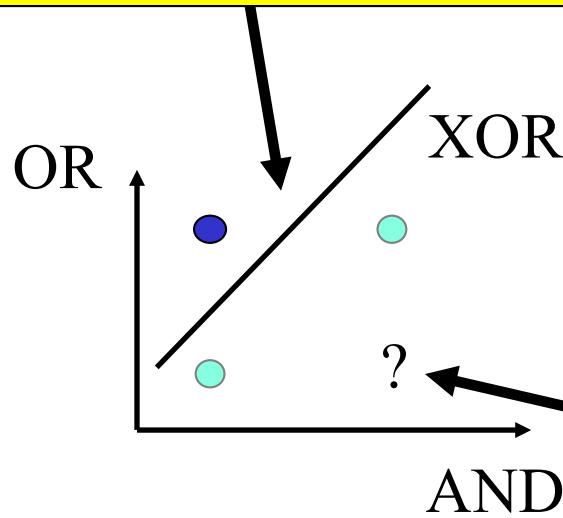


Multiple Layers Solve More Problems

What if input dimensions are AND and OR?

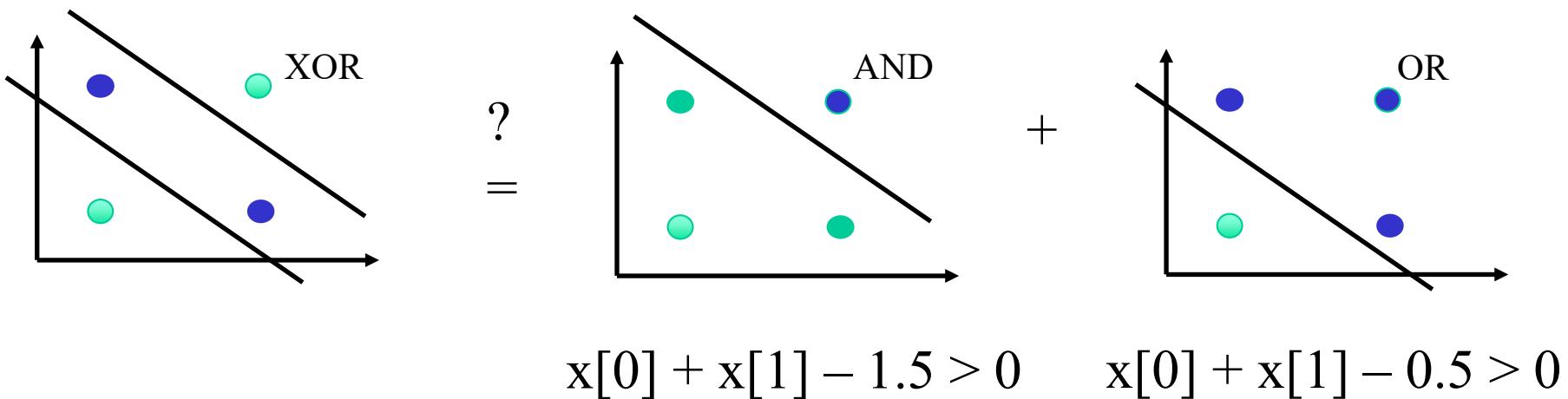
Now we can divide
with one line.

- FALSE
- TRUE



This combination
is impossible!

Perceptron



x[1]	x[0]	AND	OR	XOR
0	0	-1 ($-1.5 < 0$)	-1 ($-0.5 < 0$)	-1 ($-2.0 < 0$)
0	1	-1 ($-0.5 < 0$)	1 ($0.5 > 0$)	?
1	0	-1 ($-0.5 < 0$)	1 ($0.5 > 0$)	?
1	1	1 ($0.5 > 0$)	1 ($1.5 > 0$)	1 ($2.0 > 0$)

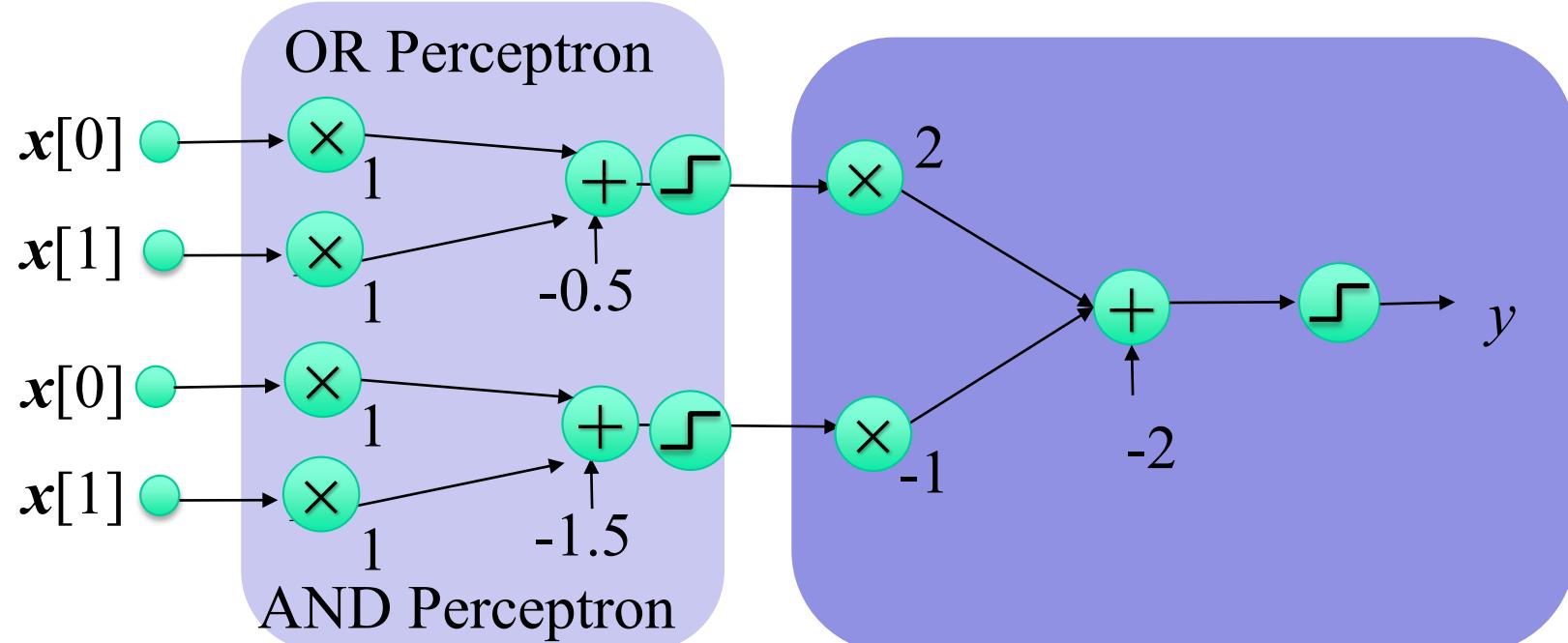
XOR is not a linear combination of AND and OR functions.

x[1]	x[0]	AND	OR	XOR
0	0	-1	-1	-1 ($-3 < 0$)
0	1	-1	+1	1 ($1 > 0$)
1	0	-1	+1	1 ($1 > 0$)
1	1	+1	+1	-1 ($-1 < 0$)

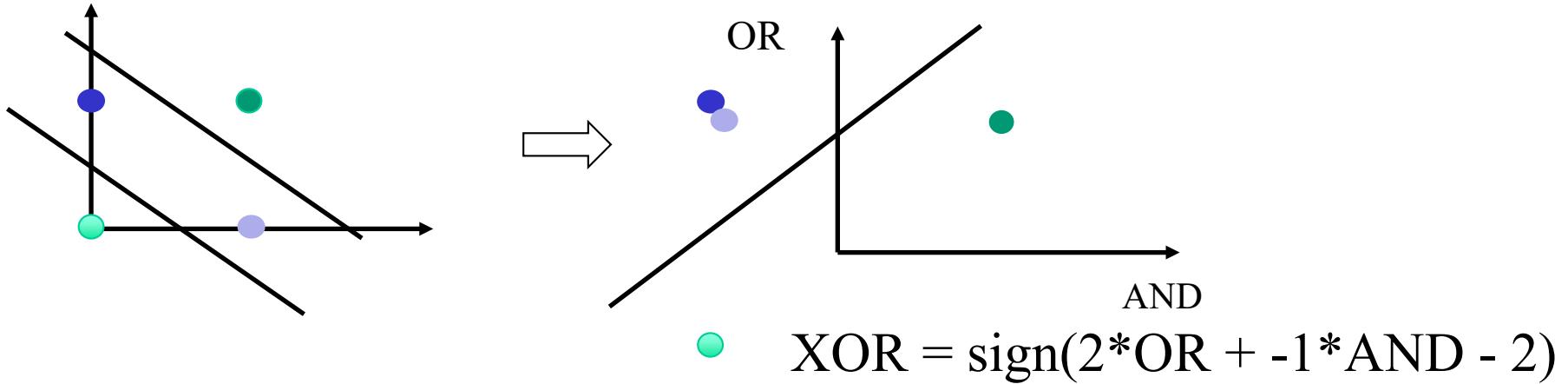
$$\text{OR} = \text{sign}(x[0] + x[1] - 0.5)$$

$$\text{AND} = \text{sign}(x[0] + x[1] - 1.5)$$

`sign()` function adds non-linearity to
“reposition” data points for the next layer.

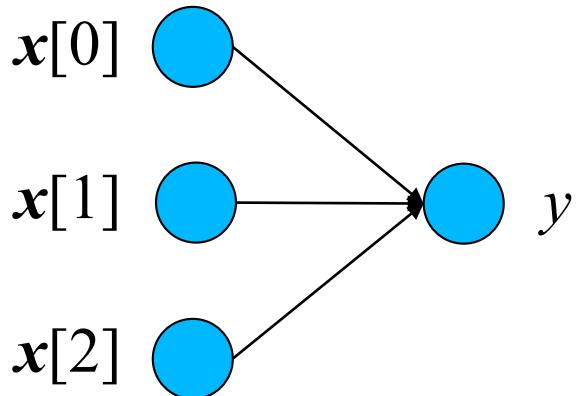


Multi-Layer Perceptron – data repositioning



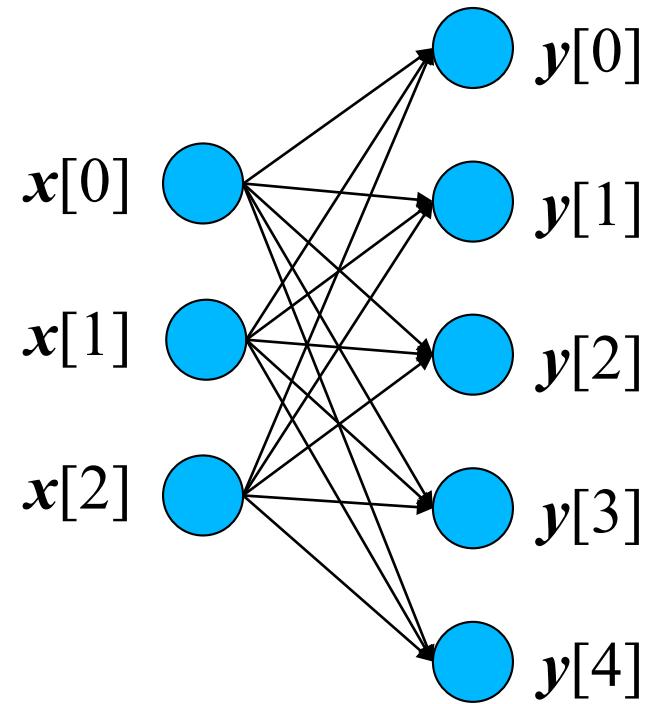
x[1]	x[0]	AND	OR	XOR
0	0	-1	-1	-1 (-3 < 0)
0	1	-1	+1	1 (1 > 0)
1	0	-1	+1	1 (1 > 0)
1	1	+1	+1	-1 (-1 < 0)

Generalize to Fully-Connected Layer



Linear Classifier:

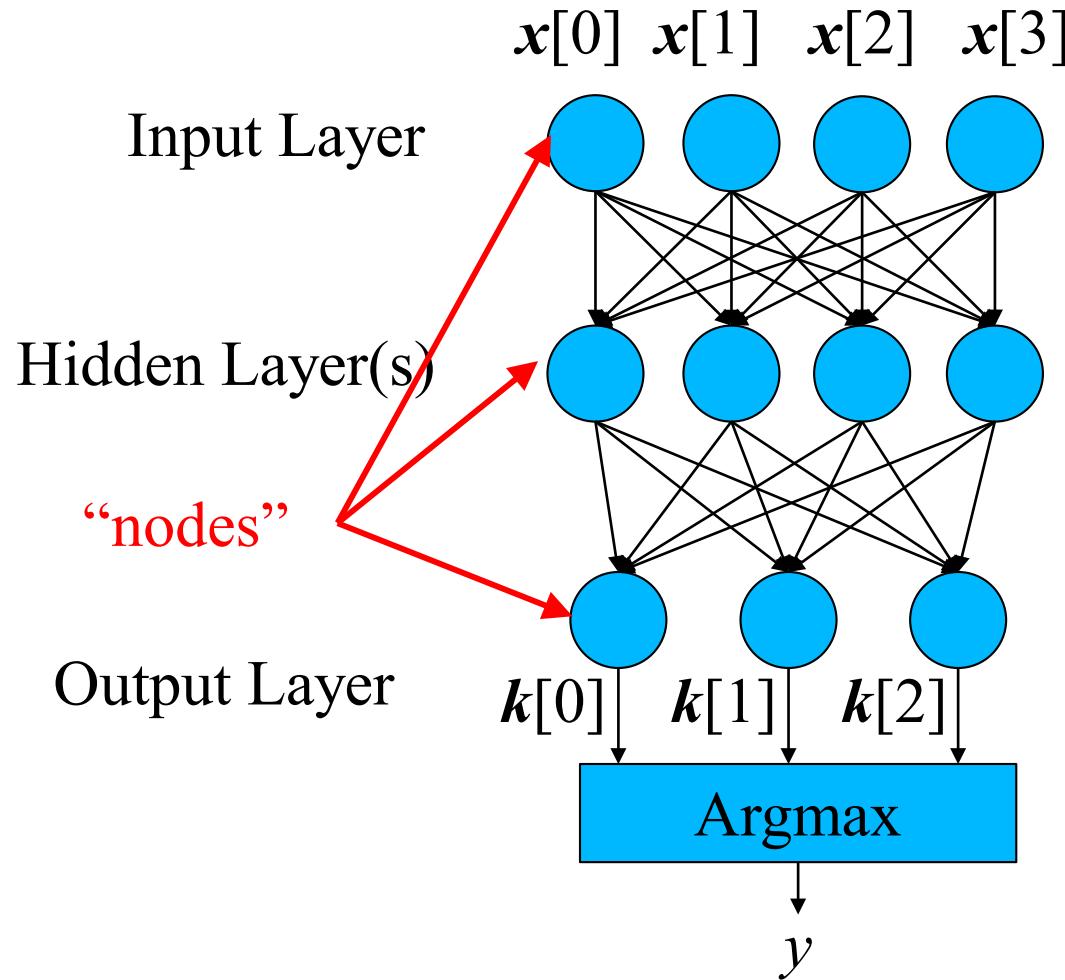
Input vector $\mathbf{x} \times$ weight vector \mathbf{w} to produce scalar output y



Fully-connected:

Input vector $\mathbf{x} \times$ weight matrix \mathbf{w} to produce vector output y

Multilayer Terminology



$W_k[i, j]$: weight between i^{th} input and j^{th} output of the k^{th} layer

W_1 is $[4 \times 4]$, b_1 is $[4 \times 1]$

W_2 is $[4 \times 3]$, b_2 is $[3 \times 1]$

Probability that input is class $k[i]$

How to determine the weights?

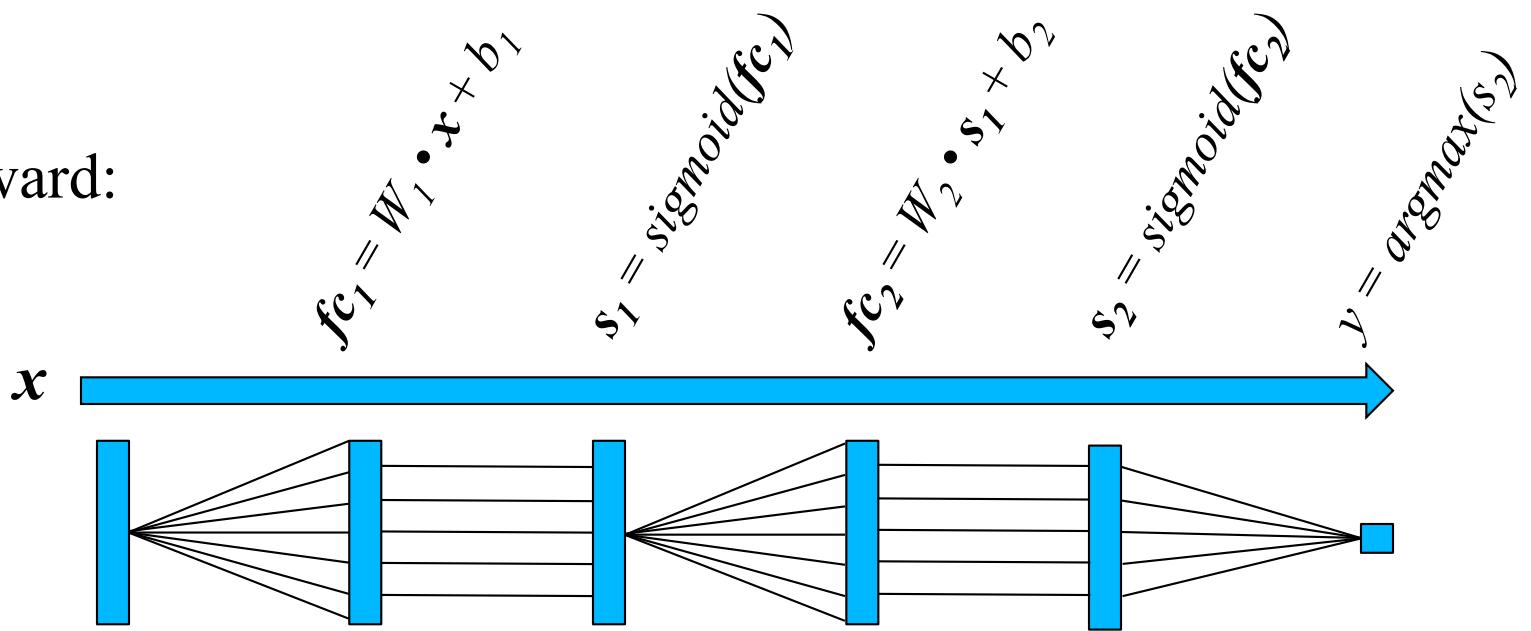
- Look at observational data to determine the weights?
- Pick some random values?
- Start with something that partially works?
- With enough *labeled* data, we can automatically *encode* the relationship between inputs and outputs.

Forward and Backward Propagation

- Forward (inference)
 - Given parameters Θ and input x , produce label y
- Backward (training)
 - Need a way to assess correctness (loss function)
 - Example: $(x - y)^2$
 - Find Θ , such that loss is minimized over all input data

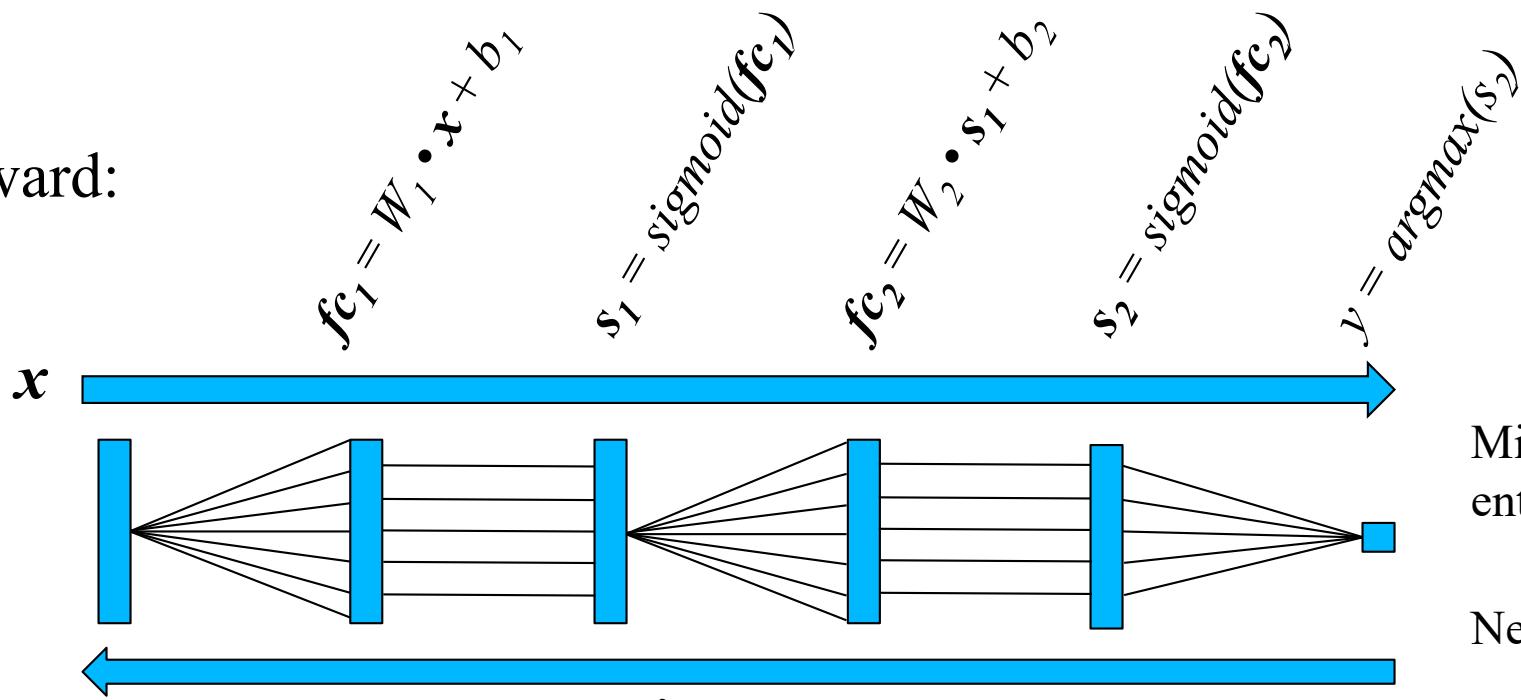
Forward Propagation (Inference)

Forward:



Backward Propagation (Training)

Forward:



Backward:

$$\frac{dE}{df_{c_1}} = \frac{dE}{ds_1} \frac{ds_1}{dfc_{c_1}}$$
$$\frac{dE}{ds_1} = \frac{dE}{df_{c_2}} \frac{df_{c_2}}{ds_1}$$
$$\frac{dE}{df_{c_2}} = \frac{dE}{ds_2} \frac{ds_2}{df_{c_2}}$$
$$\frac{dE}{ds_2} = \frac{dE}{dy} \frac{dy}{ds_2}$$
$$\frac{dE}{dy}$$

Minimize loss over entire training set:

Need $\frac{dE}{dW}$



ANY MORE QUESTIONS?



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 9: Tiled Convolution Analysis

Course Reminders

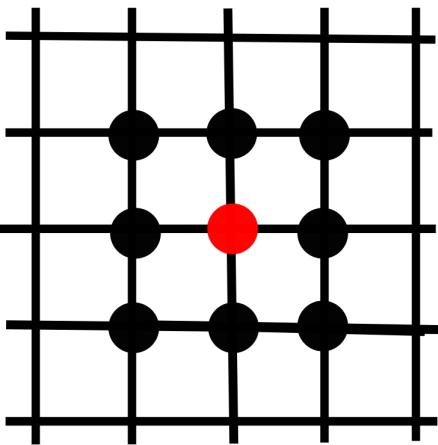
- Grades for Lab 1 are uploaded to Canvas soon
 - Check and let us know if they are missing
- Lab 4 out, it is due next week
- Midterm 1 is on Tuesday, March 7th
 - On-line, everybody will be taking it at the same time
 - Tuesday, March 7th 7:00pm-8:30pm US Central time
 - Includes materials from Lecture 1 through Lecture 9
- Project Milestone 1: Baseline CPU implementation is due Friday March 10th
 - Project details to be posted next week

Objective

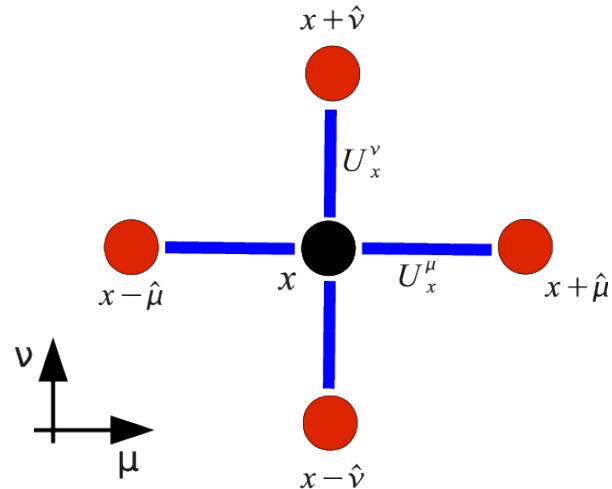
- To learn more about the analysis of tiled convolution/stencil algorithms

Stencil Algorithms

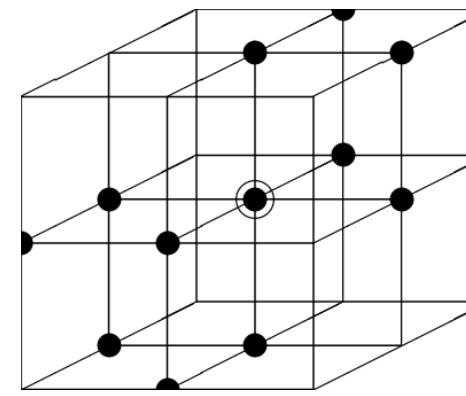
- Numerical data processing algorithms which update array elements according to some fixed pattern, called a *stencil*
 - Convolution is just one such example



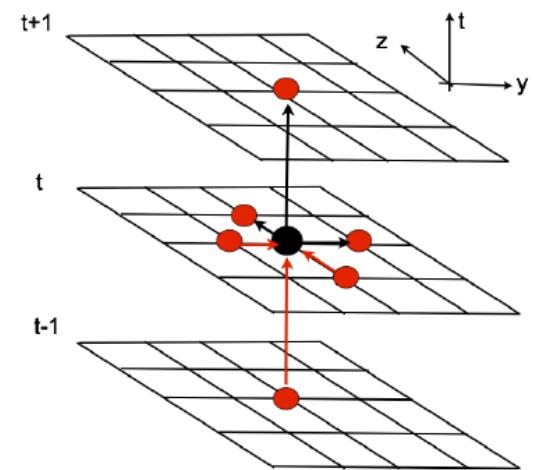
2D convolutional
kernel (9-point 2D
stencil)



Nearest neighbor lattice
Dirac operator (5-point 2D
stencil)

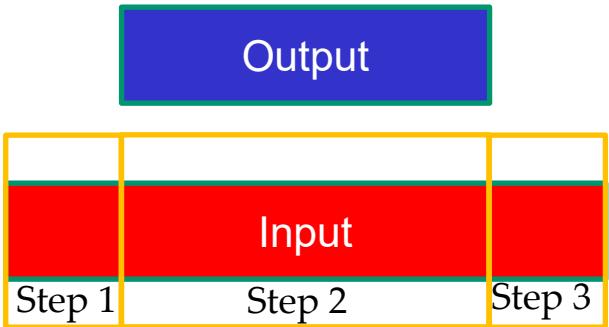


Finite difference stencil
for 3D explicit time-
marching (13-point 3D
stencil)



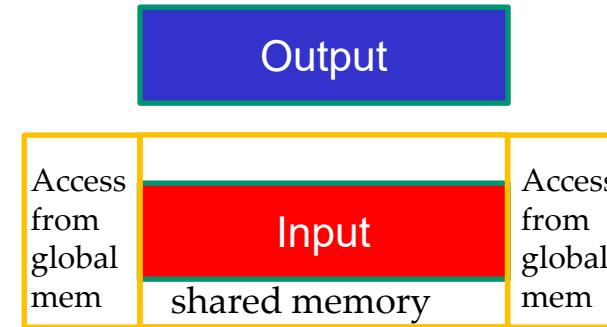
The Wilson-Dslash
operator
(4D stencil)

Review: Three Tiling Strategies



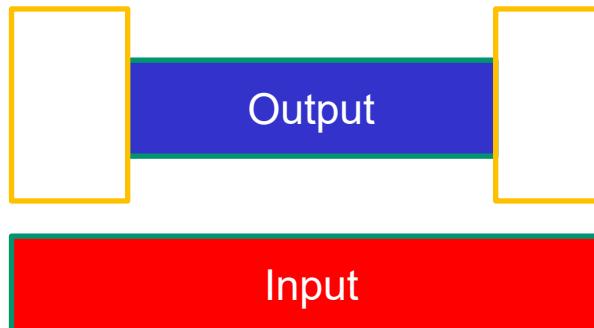
Strategy 1

1. Block size covers **output** tile
2. Use multiple steps to load input tile



Strategy 3

1. Block size covers **output** tile
2. Load only “core” of input tile
3. Access halo cells from global memory

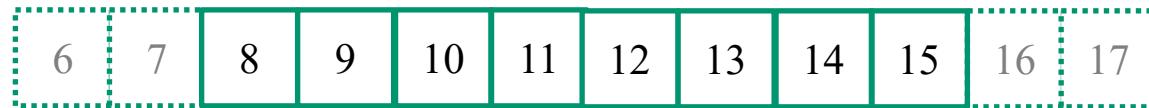


Strategy 2

1. Block size covers **input** tile
2. Load input tile in one step
3. Turn off some threads when calculating output

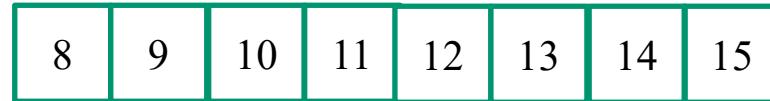
A Small 1D Convolution Example

tile



MASK_WIDTH is 5

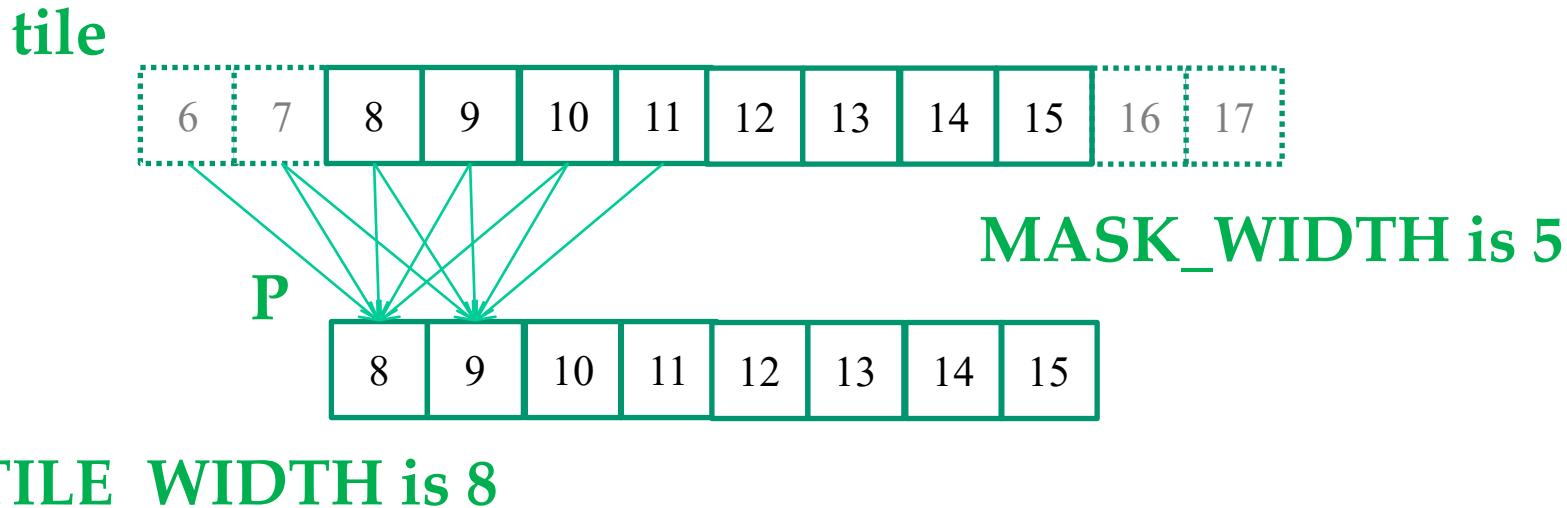
P



TILE_WIDTH is 8

- output and input tiles for block 1
- For MASK_WIDTH of 5, each block loads $8 + (5 - 1) = 12$ elements (**12 memory loads**)

Each Output Uses MASK_WIDTH Inputs



- $P[8]$ uses $N[6], N[7], N[8], N[9], N[10]$
- $P[9]$ uses $N[7], N[8], N[9], N[10], N[11]$
- ...
- $P[15]$ uses $N[13], N[14], N[15], N[16], N[17]$

Total of **8 * 5 values** from tile used for the output.

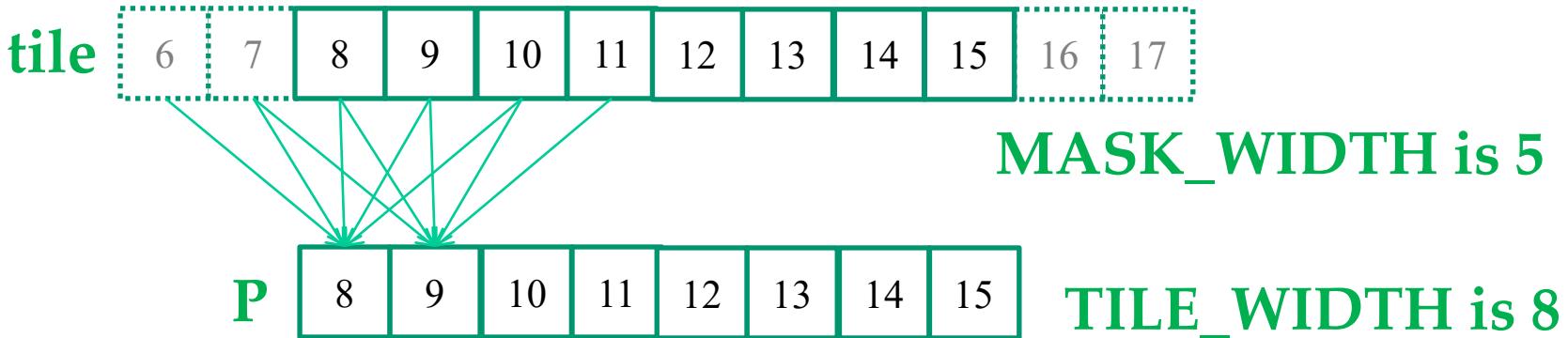
A simple way to calculate tiling benefit

- $8+(5-1) = 12$ unique elements of input array N loaded
- $8*5$ global memory accesses potentially replaced by shared memory accesses
- This gives a bandwidth reduction of **$40/12=3.3$**
- This is independent of the size of N

In General, for 1D convolution kernels

- Load $(\text{TILE_WIDTH} + \text{MASK_WIDTH} - 1)$ elements from global memory to shared memory
- Replace $(\text{TILE_WIDTH} * \text{MASK_WIDTH})$ global memory accesses with shared memory accesses
- This leads to bandwidth reduction of
 $(\text{TILE_SIZE} * \text{MASK_WIDTH}) / (\text{TILE_SIZE} + \text{MASK_WIDTH} - 1)$

Another Way to Look at Reuse



- tile[6] is used by P[8] (1×)
- tile[7] is used by P[8], P[9] (2×)
- tile[8] is used by P[8], P[9], P[10] (3×)
- tile[9] is used by P[8], P[9], P[10], P[11] (4×)
- tile[10] is used by P[8], P[9], P[10], P[11], P[12] (5×)
- ... (5×)
- tile[14] is used by P[12], P[13], P[14], P[15] (4×)
- tile[15] is used by P[13], P[14], P[15] (3×)
- tile[16] is used by P[14], P[15] (2×)
- tile[17] is used by P[15] (1×)

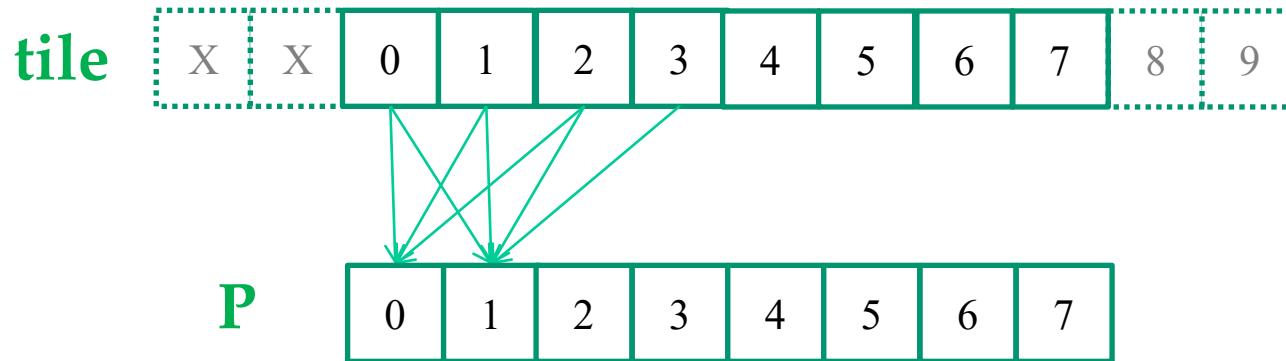
Another Way to Look at Reuse

- Each access **tile** replaces an access to input N
- The total number of global memory accesses (to the $(8+5-1)=12$ input elements) replaced by shared memory accesses is

$$\begin{aligned} & 1 + 2 + 3 + 4 + 5 * (8-5+1) + 4 + 3 + 2 + 1 \\ & = 10 + 20 + 10 \\ & = 40 \end{aligned}$$

- There are 12 elements of input N, so the average reduction is
40/12 = 3.3

What about Boundary Tiles?



- $P[0]$ uses $N[0], N[1], N[2]$
- $P[1]$ uses $N[0], N[1], N[2], N[3]$
- $P[2]$ uses $N[0], N[1], N[2], N[3], N[4]$

Less than $8 * 5$ elements of N are used for the output tile

Ghost elements change ratios

- For a boundary tile, we load $\text{TILE_WIDTH} + (\text{MASK_WIDTH}-1)/2$ elements
 - 10 in our example of TILE_WIDTH of 8 and MASK_WIDTH of 5
- Computing boundary elements do not access global memory for ghost cells
 - Total accesses is $6*5 + 4 + 3 = 37$ accesses (when computing the P elements)

The reduction is **37/10 = 3.7**

In General for 1D, internal tiles

- The total number of global memory accesses to the $(TILE_WIDTH+Mask_Width-1)$ elements of input N replaced by shared memory accesses is

$$\begin{aligned} & 1 + 2 + \dots + Mask_Width-1 + Mask_Width * (TILE_WIDTH - Mask_Width + 1) + Mask_Width-1 + \dots + 2 + 1 \\ &= ((Mask_Width-1) * Mask_Width)/2 + Mask_Width * (TILE_WIDTH - Mask_Width + 1) + ((Mask_Width-1) * Mask_Width)/2 \\ &= (Mask_Width-1) * Mask_Width + Mask_Width * (TILE_WIDTH - Mask_Width + 1) \\ &= Mask_Width * TILE_WIDTH \end{aligned}$$

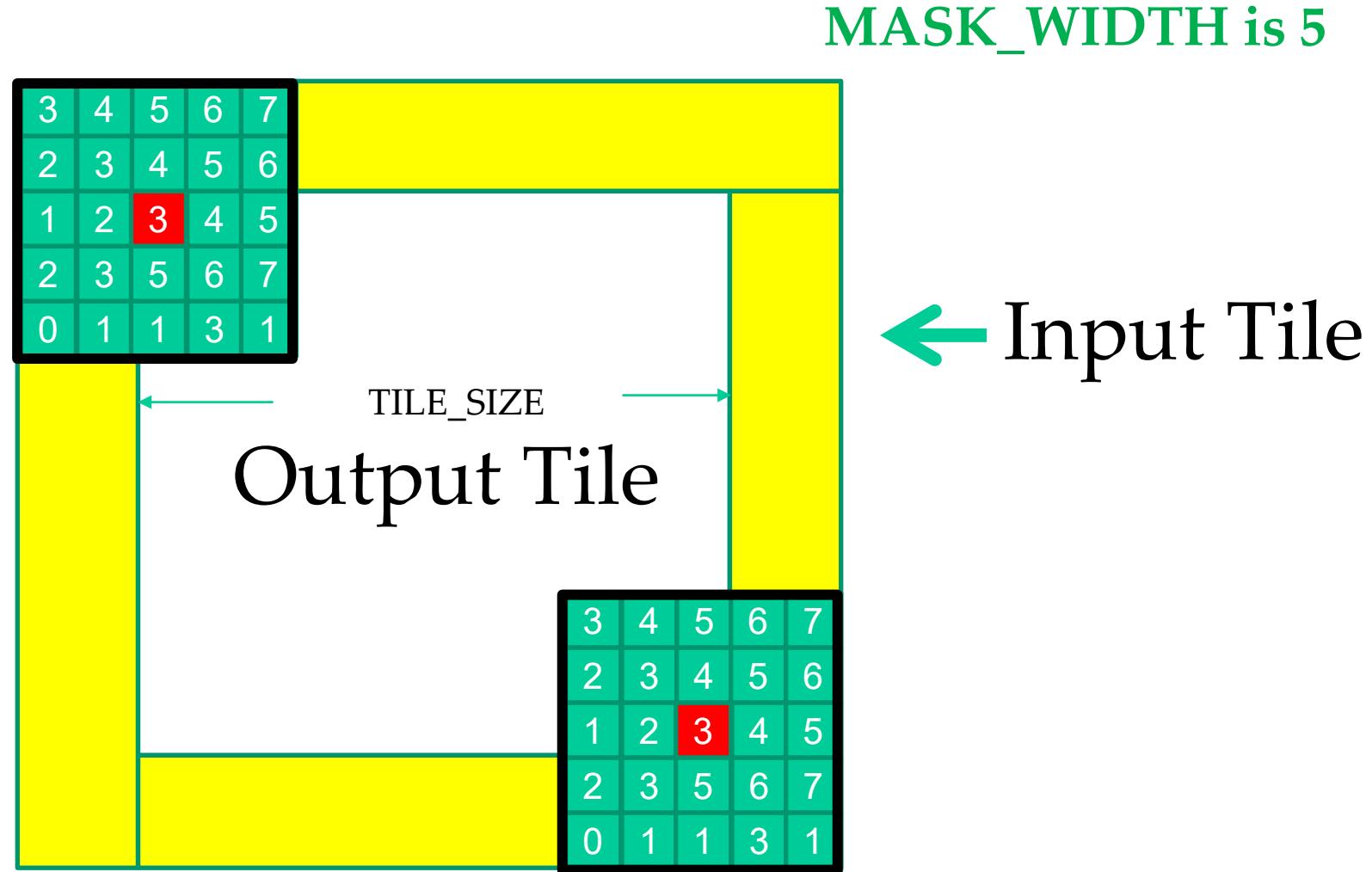
Bandwidth Reduction for 1D

- The reduction is

$$(\text{TILE_SIZE} * \text{MASK_WIDTH}) / (\text{TILE_SIZE} + \text{MASK_WIDTH} - 1)$$

TILE_WIDTH	16	32	64	128	256
Reduction MASK_WIDTH = 5	4.0	4.4	4.7	4.9	4.9
Reduction MASK_WIDTH = 9	6.0	7.2	8.0	8.5	8.7

Review: Parallelization of Tile Load



Analysis for an 8×8 Output Tile, MASK_WIDTH of 5

- Loading input tile requires **(8+5-1)² = 144 reads**
- Calculation of each output requires $5^2 = 25$ input elements
- **8×8×25 = 1,600 global memory accesses** for computing output tile are converted to shared memory accesses
- Bandwidth reduction of **1,600/144 = 11.1×**

In General

- $(\text{TILE_WIDTH} + \text{MASK_WIDTH} - 1)^2$ elements need to be loaded from \mathbf{N} into shared memory
- The calculation of each \mathbf{P} element needs to access MASK_WIDTH^2 elements of \mathbf{N}
- $(\text{TILE_WIDTH} * \text{MASK_WIDTH})^2$ global memory accesses converted into shared memory accesses
- Bandwidth reduction of
$$(\text{TILE_WIDTH} * \text{MASK_WIDTH})^2 / (\text{TILE_WIDTH} + \text{MASK_WIDTH} - 1)^2$$

Bandwidth Reduction for 2D Convolution Kernel

- The reduction is

$$(\text{ TILE_WIDTH} * \text{ MASK_WIDTH})^2 / (\text{ TILE_WIDTH} + \text{ MASK_WIDTH} - 1)^2$$

TILE_WIDTH	8	16	32	64
Reduction MASK_WIDTH = 5	11.1	16	19.7	22.1
Reduction MASK_WIDTH= 9	20.3	36	51.8	64

Ghost elements change ratios

- 2D calculation left for your enjoyment

2B/FLOP for Untiled Convolution

- How much global memory per FLOP is in untiled convolution?
- In untiled convolution,
 - each value from **N** (**4B** from global memory)
 - is multiplied by a value from **M** (**4B** from constant cache, **1 FLOP**),
 - then added to a running sum (**1 FLOP**)
- That gives **2B / FLOP**

Full Use of Compute Requires 13.3× Reuse

- Recall our reuse discussion from matrix multiply:
 - 1,000 **GFLOP/s** for GPU from ~2010, and
 - 150 **GB/s** memory bandwidth.
- Dividing memory bandwidth by **2B/FLOP**,
$$\frac{150 \text{ GB/s}}{2 \text{ B/FLOP}} = 75 \text{ GFLOP/s} = 7.50\% \text{ of peak.}$$
- **Need at least $100/7.50 = 13.3\times$ reuse** to make full use of compute resources

In 2020, Need 52.1× Reuse

- In 2020, the **GRID K520** offers
 - nearly **5,000 GFLOP/s**, but only
 - **192 GB/s** memory bandwidth
- Dividing memory bandwidth by **2B/FLOP**,
$$\frac{192 \text{ GB/s}}{2 \text{ B/FLOP}} = \textbf{96 GFLOP/s} = \textbf{1.92\% of peak}$$
- **Need at least $100/1.92 = 52.1\times$ reuse** to make full use of compute resources

Need Really Big Mask to Balance Resources

- Let's make another table: **% of peak compute**
 - for **1D** tiled convolution,
 - with **TILE_WIDTH 1024**

MASK_WIDTH	~2010 GPU with 1,000 GFLOP/s 150 GB/s	~2020 GPU with 5,000 GFLOP/s 192 GB/s
5	37%	9.6%
9	67%	17%
15	100%	28%
55	100%	100%

Need Really Big Mask to Balance Resources

- And one more: **% of peak compute**
 - for **2D** tiled convolution,
 - with **TILE_WIDTH 32×32**

MASK_WIDTH	~2010 GPU with 1,000 GFLOP/s 150 GB/s	~2020 GPU with 5,000 GFLOP/s 192 GB/s
3	60%	15%
5	100%	37%
7	100%	67%
9	100%	almost 100%

Food for Thought

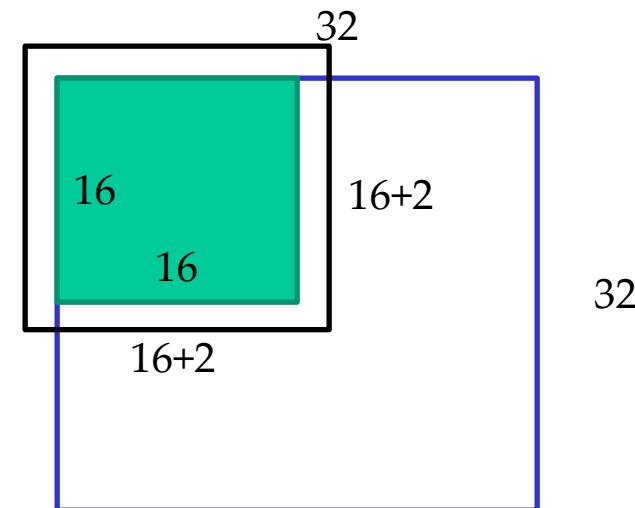
- Ratios are different for tiles on boundaries
- More importantly,
 - Each thread loads 4B to shared memory
 - 2,048 threads load only 8kB
 - Shared memory is usually 64kB or larger
 - What can one do with the rest?

Improved approach left as homework

(For example, can raise MW=7 from 67% to 81%)

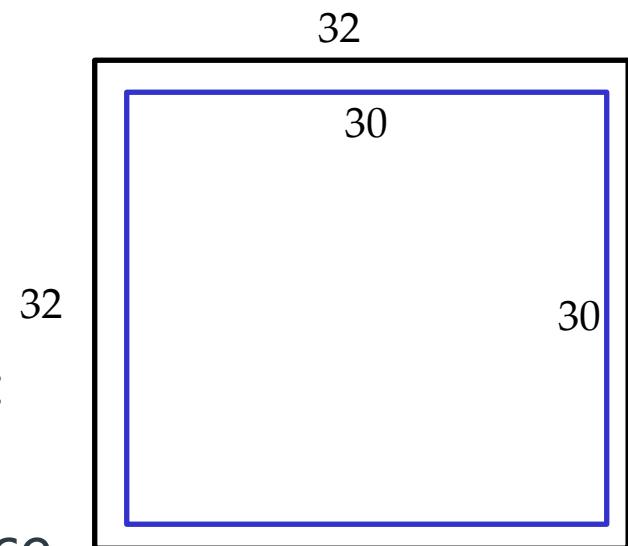
Problem Solving

- Q: Consider 2D tiled convolution with mask width of 5x5 and output tile width of 16x16 applied to an input image of size 32x32. Assume we use Strategy 1. The use of shared memory reduces the number of global memory accesses. What is the reduction in global memory accesses for thread block (0,0)?
- A:
 - Count # of global memory accesses:
row 0: $9+12+15*14$
row 1: $12+16+20*14$
other 14 rows: $15+20+25*14$
 - Count # of input elements to be used in the calculation: $18*18$



Problem Solving

- Q: For a tiled 2D convolution kernel with 30x30 output tiles and 3x3 mask (and thus 32x32 input tile), how many warps in each thread block have control divergence? (Assume strategy 2: block size covers input tile.)
- A:
 - thread block size is 32x32
 - All 32x32 threads participate in data load
 - What does each warp do?
 - Warp 0: load 0s and do not compute
 - Warps 1-30: 30 threads compute, and 2 threads do not
 - Warp 31: load 0s and do not compute
 - Thus, only 2 warps do not have control divergence





**ANY MORE QUESTIONS?
READ CHAPTER 7**



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 8: Tiled Convolution



Course Reminders

- Labs
 - Lab 2 is due this Friday
 - Lab 3 is our and is due next Friday
- Midterm 1 is coming up, on March 7

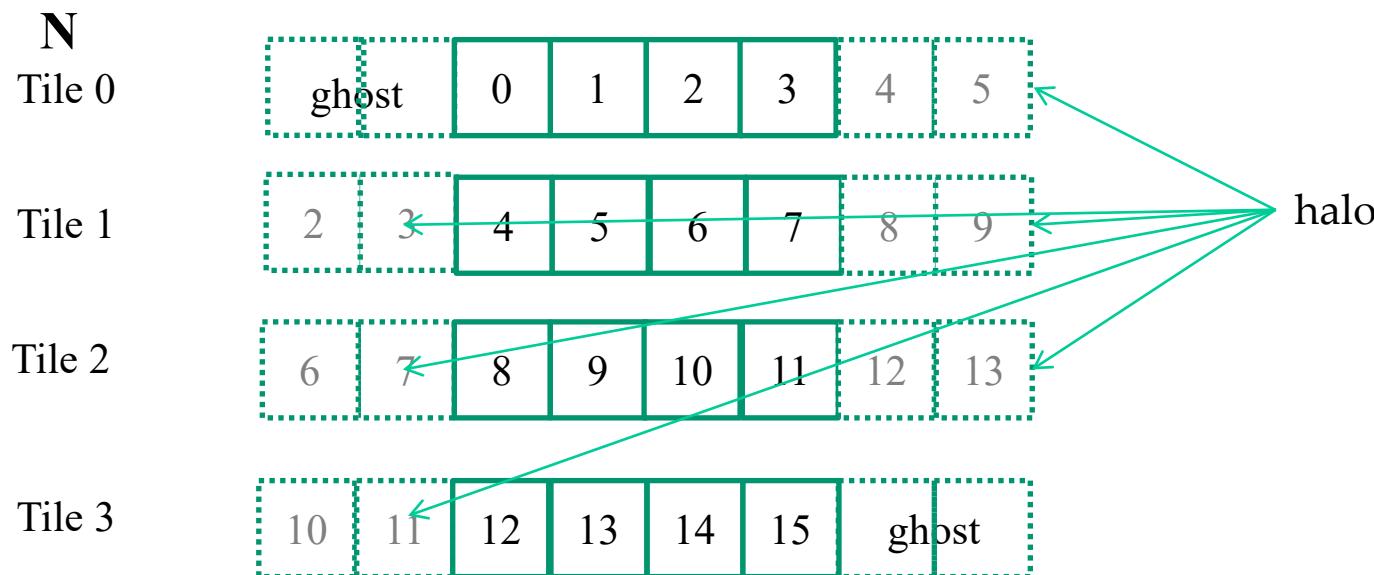
Objective

- To learn about tiled convolution algorithms
 - Some intricate aspects of tiling algorithms
 - Output tiles versus input tiles
 - Three different styles of input tile loading
 - To prepare for Lab 4

Tiled 1D Convolution Basic Idea

P	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----



What Shall We Parallelize?

In other words,

What should one thread do?

One answer:

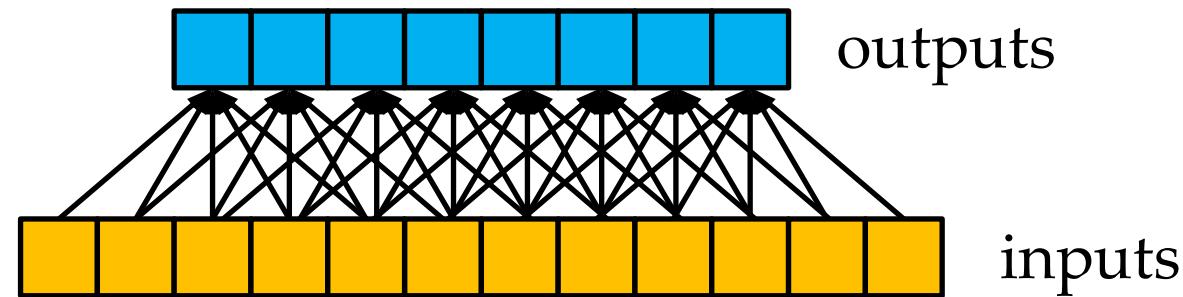
- (same as with vector sum and matrix multiply)
- **compute an output element!**

Should We Use Shared Memory?

In other words,

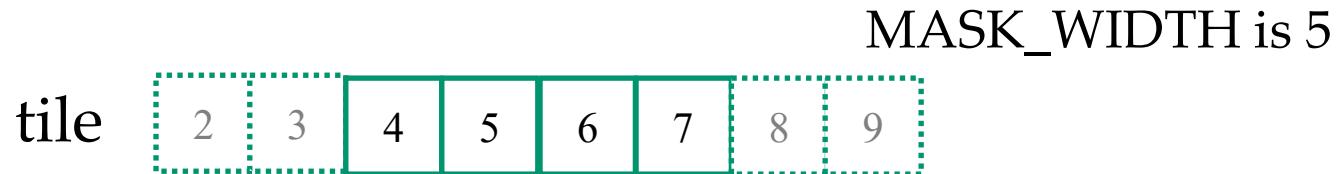
Can we reuse data read from global memory?

Let's look at the computation again...



Reuse reduces global memory bandwidth,
so **let's use shared memory**.

How Much Reuse is Possible?

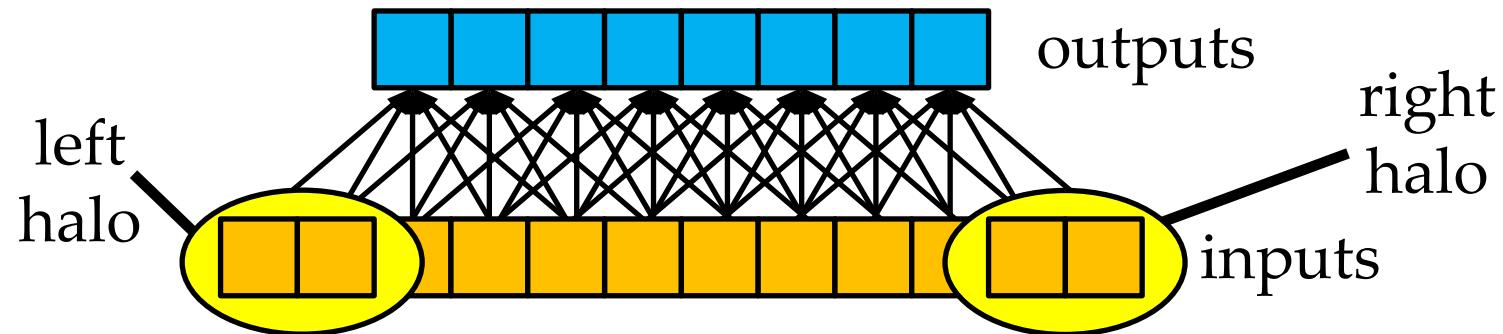


- Element 2 is used by thread 4 (1×)
- Element 3 is used by threads 4, 5 (2×)
- Element 4 is used by threads 4, 5, 6 (3×)
- Element 5 is used by threads 4, 5, 6, 7 (4×)
- Element 6 is used by threads 4, 5, 6, 7 (4×)
- Element 7 is used by threads 5, 6, 7 (3×)
- Element 8 is used by threads 6, 7 (2×)
- Element 9 is used by thread 7 (1×)

What About the Halos?

In other words,

Do we also copy halos into shared memory?



Let's **consider both** possible answers.

Can Access Halo from Global Memory

Approach:

- threads **read halo values**
- directly **from global memory**.

Advantage:

- optimize reuse of shared memory
- (halo reuse is smaller).

Disadvantages:

- **Branch divergence!** (shared vs. global reads)
- Halo **too narrow to fill** a memory **burst**

Can Load Halo to Shared Memory

Approach:

- **load halos to shared memory.**

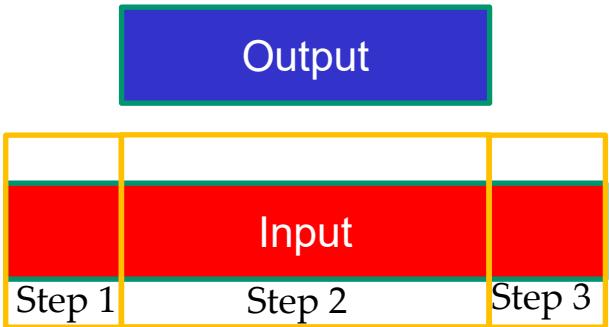
Advantages:

- **Coalesce global memory accesses.**
- **No branch divergence during computation.**

Disadvantages:

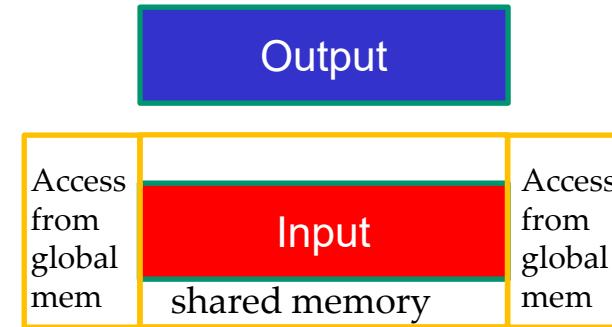
- Some threads must do >1 load, so
some branch divergence in reading data.
- Slightly more shared memory needed.

Three Tiling Strategies



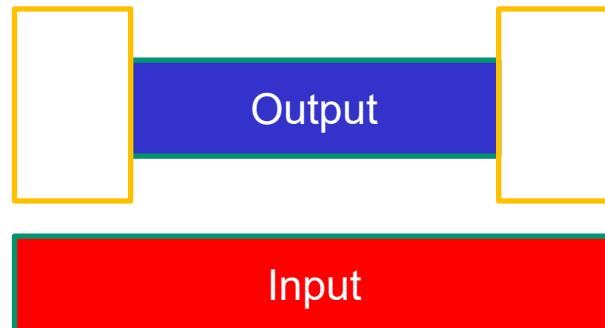
Strategy 1

1. Block size covers **output** tile
2. Use multiple steps to load input tile



Strategy 3

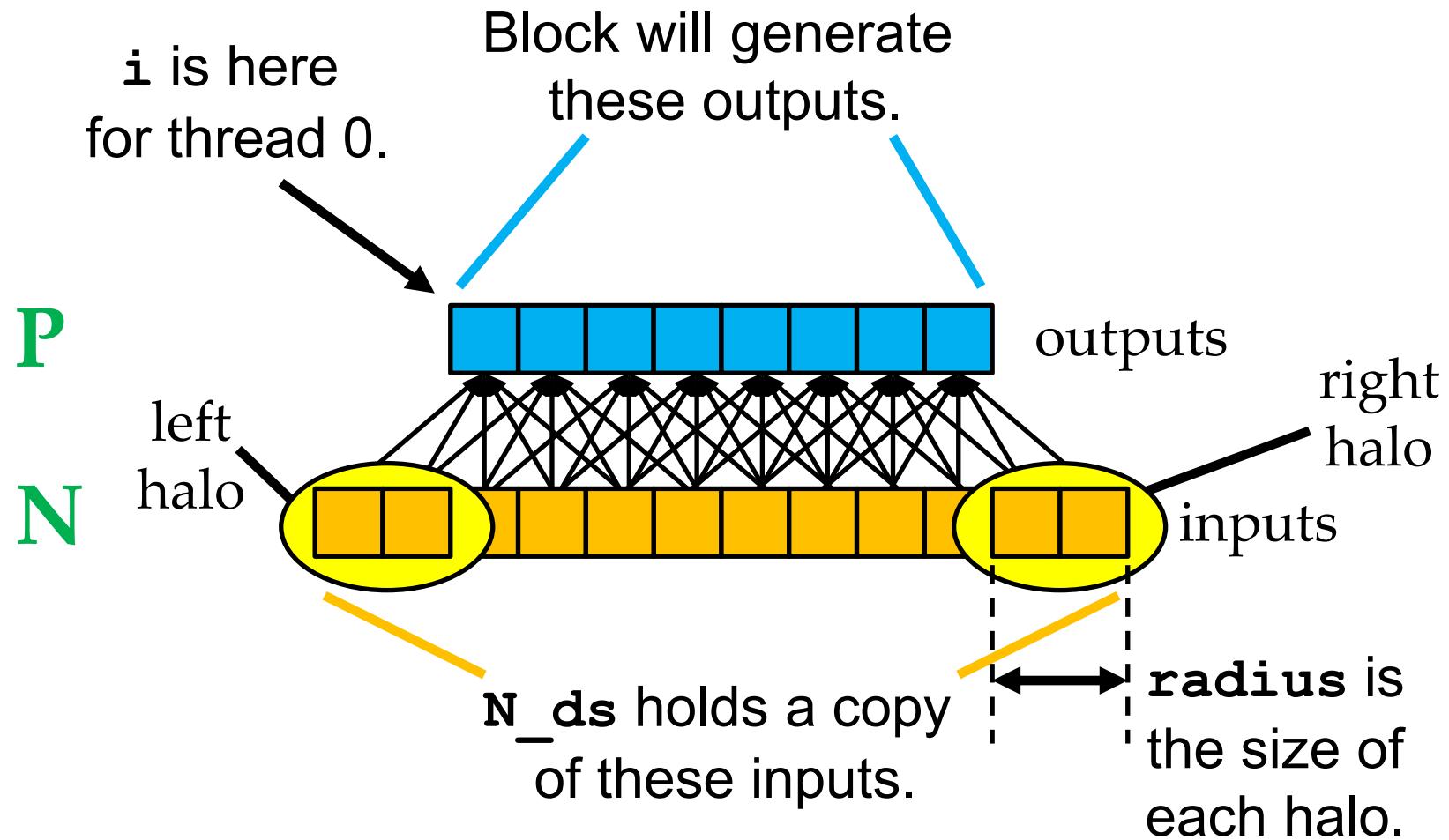
1. Block size covers **output** tile
2. Load only “core” of input tile
3. Access halo cells from global memory



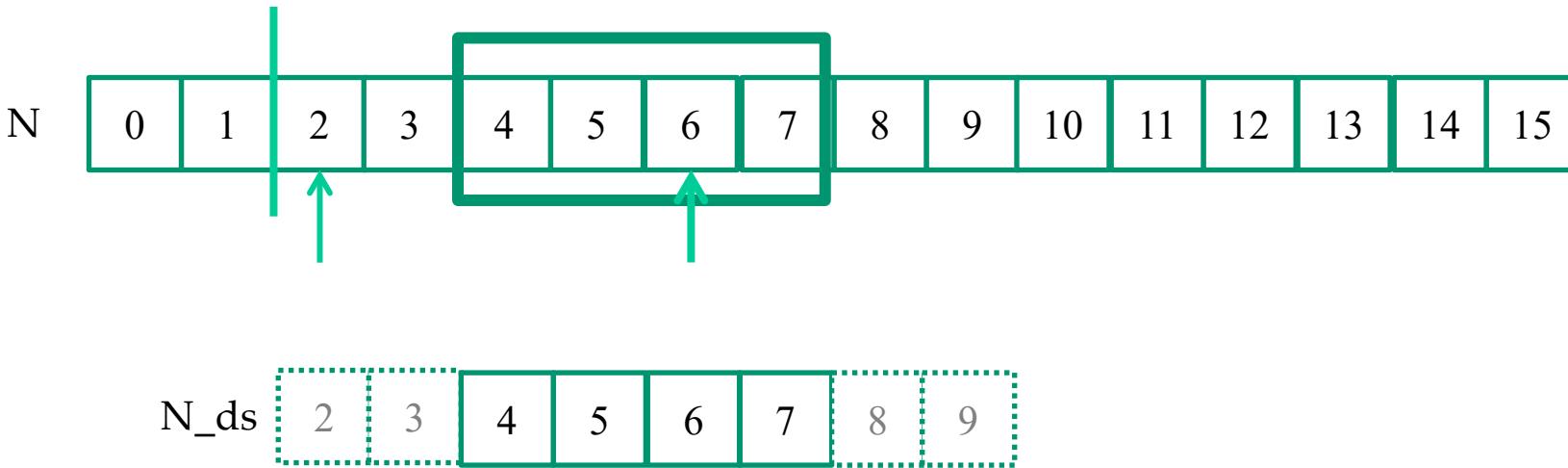
Strategy 2

1. Block size covers **input** tile
2. Load input tile in one step
3. Turn off some threads when calculating output

Strategy 1: Variable Meanings for a Block

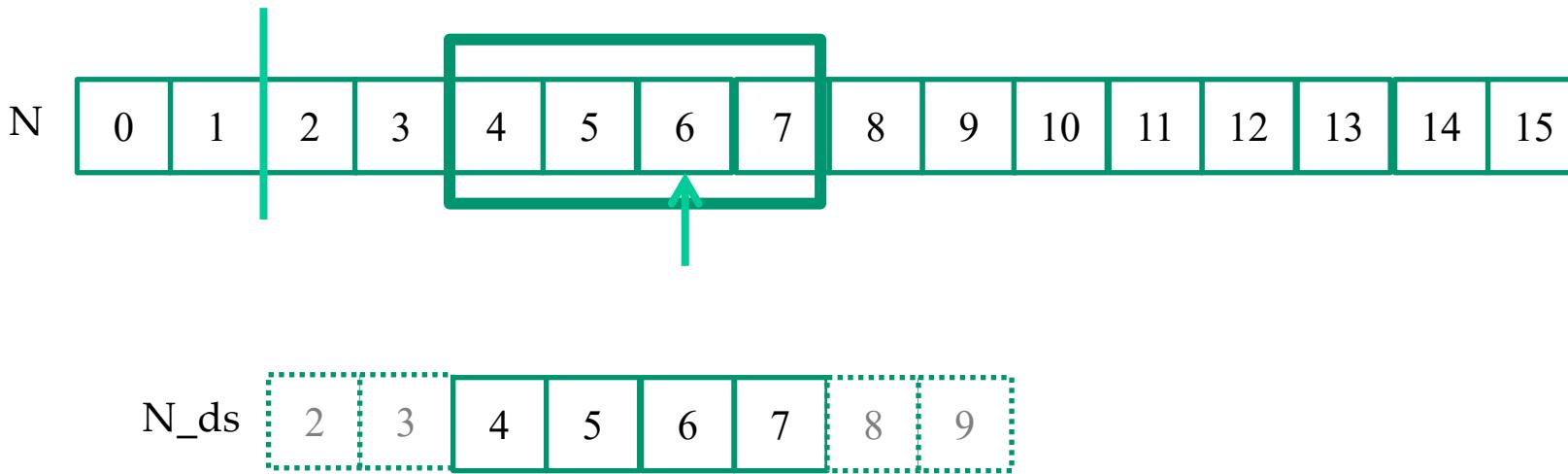


Loading the left halo



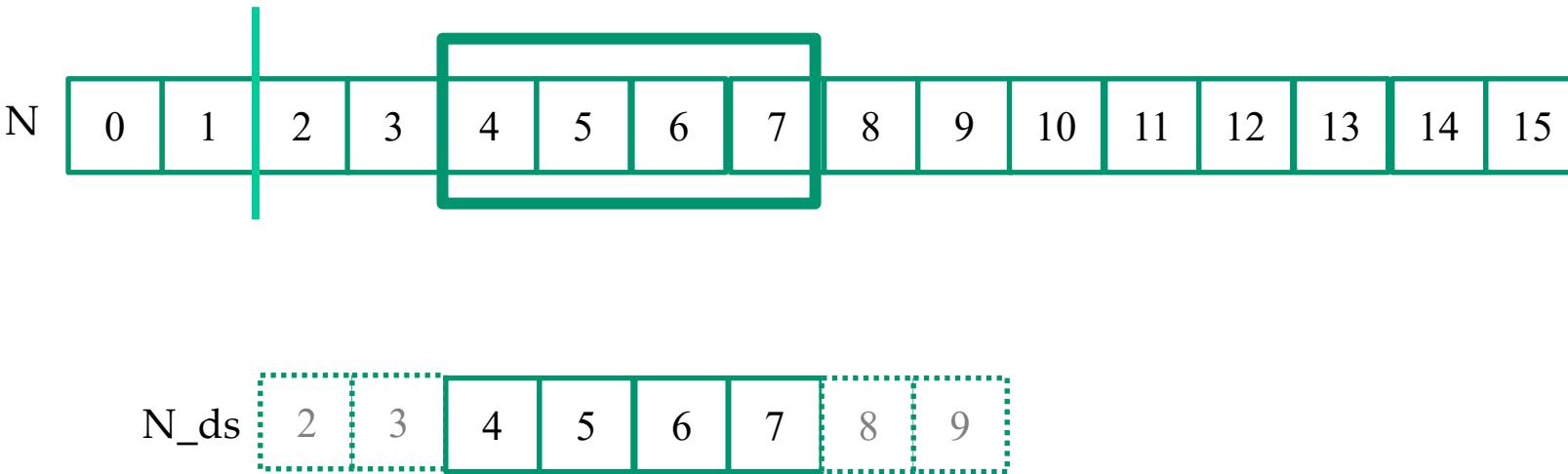
```
int radius = Mask_Width / 2;
int halo_index_left = (blockIdx.x - 1) * blockDim.x + threadIdx.x;
if (threadIdx.x >= (blockDim.x - radius)) {
    N_ds[threadIdx.x - (blockDim.x - radius)] =
        (halo_index_left < 0) ? 0 : N[halo_index_left];
}
```

Loading the internal elements



```
int index = blockIdx.x * blockDim.x + threadIdx.x;
if ((blockIdx.x * blockDim.x + threadIdx.x) < Width)
    N_ds[radius + threadIdx.x] = N[index];
else
    N_ds[radius + threadIdx.x] = 0.0f;
```

Loading the right halo



```
int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
if (threadIdx.x < radius) {
    N_ds[radius + blockDim.x + threadIdx.x] =
        (halo_index_right >= Width) ? 0 : N[halo_index_right];
}
```

```

__global__ void convolution_1D_tiled_kernel(float *N, float *P, int Mask_Width, int Width) {

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int radius = Mask_Width / 2;

    __shared__ float N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];

    int halo_index_left = (blockIdx.x - 1) * blockDim.x + threadIdx.x;
    if (threadIdx.x >= (blockDim.x - radius)) {
        N_ds[threadIdx.x - (blockDim.x - radius)] =
            (halo_index_left < 0) ? 0 : N[halo_index_left];
    }

    N_ds[radius + threadIdx.x] = N[i]; // bounds check is needed

    int halo_index_right = (blockIdx.x + 1) * blockDim.x + threadIdx.x;
    if (threadIdx.x < radius) {
        N_ds[radius + blockDim.x + threadIdx.x] =
            (halo_index_right >= Width) ? 0 : N[halo_index_right];
    }

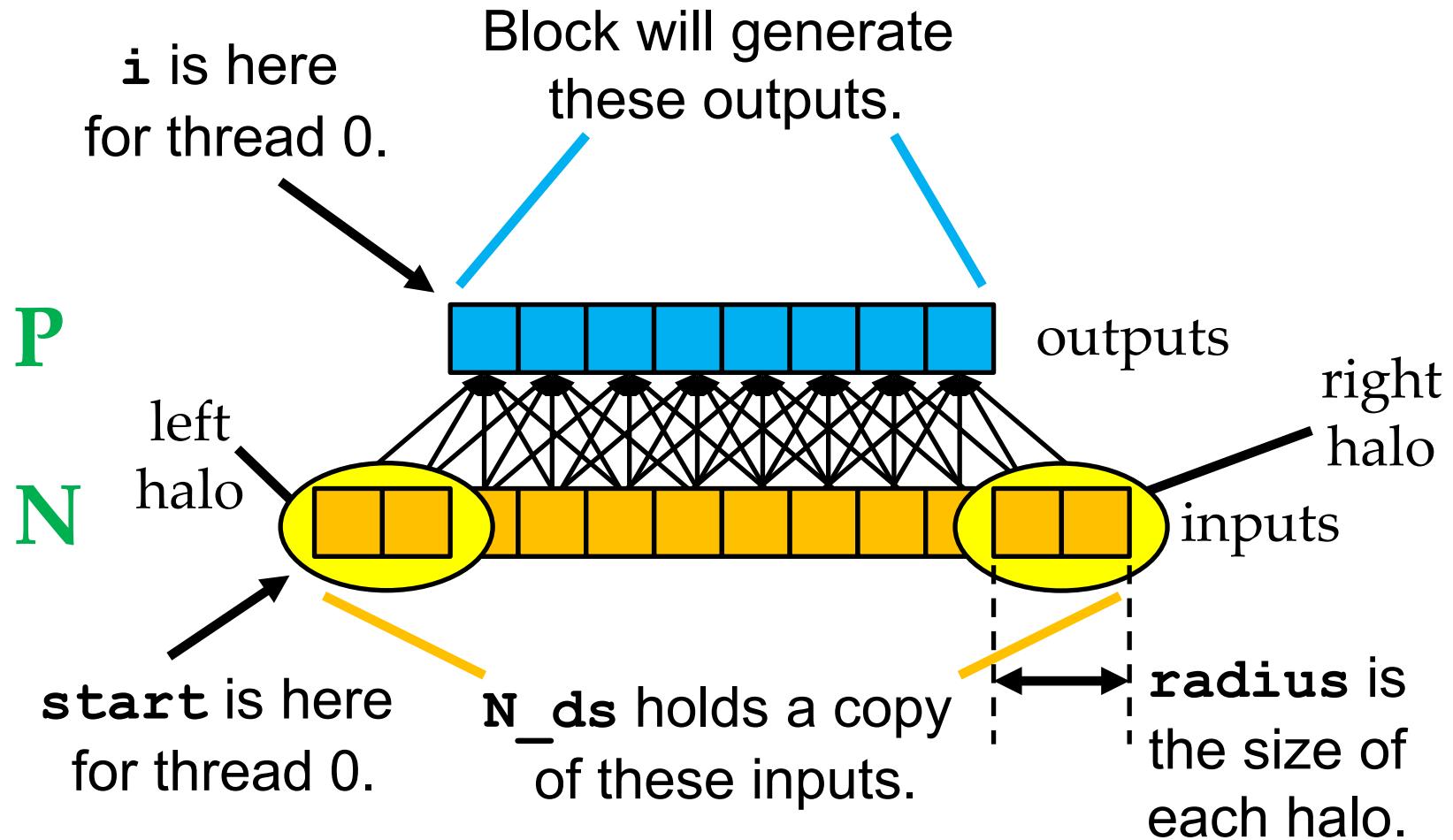
    __syncthreads();

    float Pvalue = 0;
    for(int j = 0; j < Mask_Width; j++) {
        Pvalue += N_ds[threadIdx.x + j]*M[j];
    }
    P[i] = Pvalue;
}

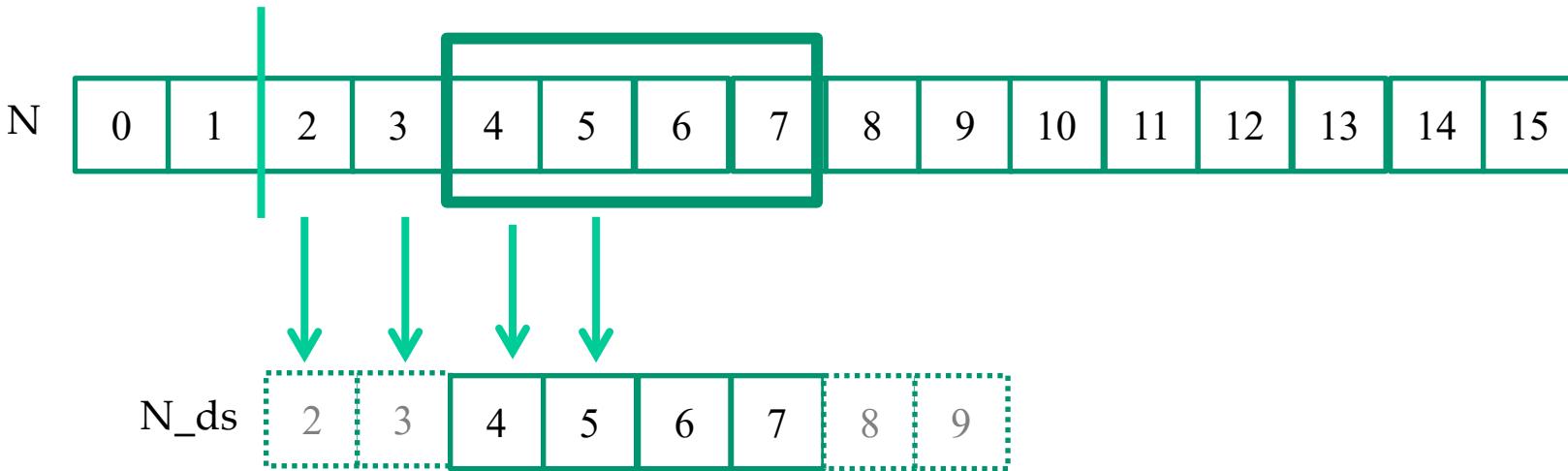
```

Strategy 1

Alternative implementation of Strategy 1: Variable Meanings for a Block

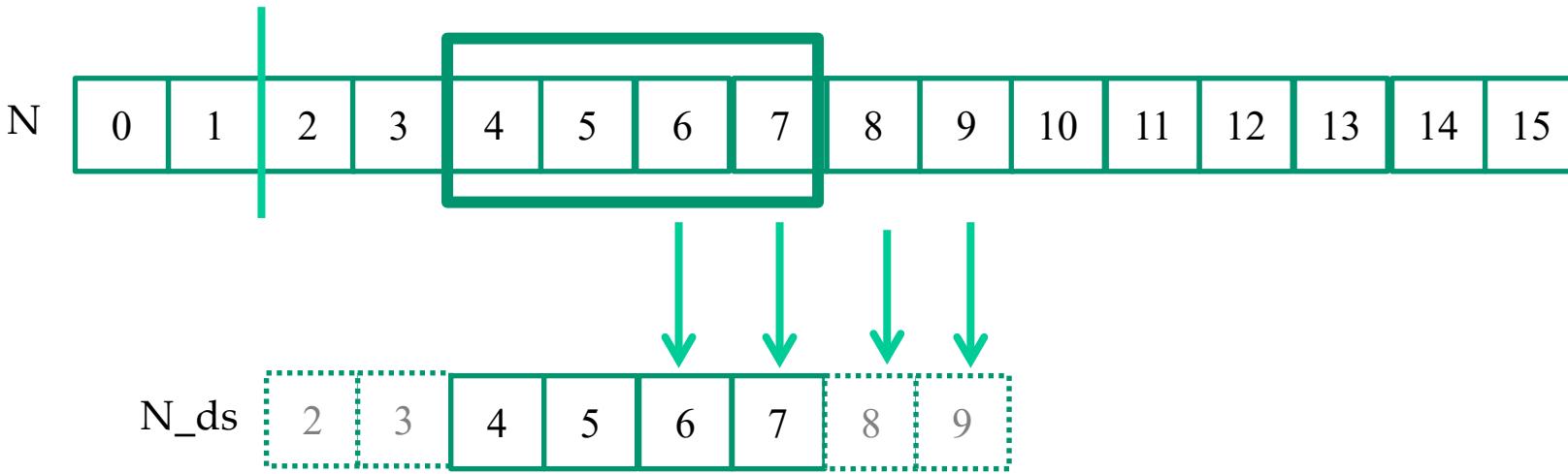


Load the Input Data – step 1



```
int start = i - radius;  
if (0 <= start && Width > start) {           // all threads  
    N_ds[threadIdx.x] = N[start];  
} else {  
    N_ds[threadIdx.x] = 0.0f;  
}
```

Load the Input Data – step 2



```
if (MASK_WIDTH - 1 > threadIdx.x) {           // some threads
    start += TILE_SIZE;
    if (Width > start) {
        N_ds[threadIdx.x + TILE_SIZE] = N[start];
    } else {
        N_ds[threadIdx.x + TILE_SIZE] = 0.0f;
    }
}
```

```

__global__ void convolution_1D_tiled_kernel float *N, float *P, int Width) {

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int radius = MASK_WIDTH / 2;
    int start = i - radius;

    __shared__ float N_ds[TILE_SIZE + MASK_WIDTH - 1];

    if (0 <= start && Width > start) {          // all threads
        N_ds[threadIdx.x] = N[start];
    } else
        N_ds[threadIdx.x] = 0.0f;

    if (MASK_WIDTH - 1 > threadIdx.x) {          // some threads
        start += TILE_SIZE;
        if (Width > start) {
            N_ds[threadIdx.x + TILE_SIZE] = N[start];
        } else
            N_ds[threadIdx.x + TILE_SIZE] = 0.0f;
    }

    __syncthreads();

    float Pvalue = 0.0f;
    for (int j = 0; MASK_WIDTH > j; j++) {
        Pvalue += N_ds[threadIdx.x + j] * Mc[j];
    }
    P[i] = Pvalue;
}

```

Alt. Strategy 1

```

__global__
void convolution_1D_tiled_cache_kernel(float *N, float *P, int Mask_Width, int Width) {

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

    __shared__ float N_ds[TILE_WIDTH];

    N_ds[threadIdx.x] = N[i];

    __syncthreads();

    int radius = Mask_Width / 2;
    int This_tile_start_point = blockIdx.x * blockDim.x;
    int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
    int N_start_point = i - radius;

    float Pvalue = 0;
    for (int j = 0; j < Mask_Width; j++) {
        int N_index = N_start_point + j;
        if (N_index >= 0 && N_index < Width) {
            if ((N_index >= This_tile_start_point) && (N_index < Next_tile_start_point))
                Pvalue += N_ds[threadIdx.x-radius+j] * M[j];
            else
                Pvalue += N[N_index] * M[j];
        }
    }
    P[i] = Pvalue;
}

```

Strategy 3

Review: What Shall We Parallelize?

In other words,

What should one thread do?

One answer:

- (same as with vector sum and matrix multiply)
- **compute an output element!**
 - **Strategy 1 & 3**

Is that our only choice? (What about Strategy 2?)

Strategy 2: Parallelize Loading of a Tile

Alternately,

- **each thread loads** one input element, and
- **some threads compute** an output.

(compared with previous approach)

Advantage:

- **No branch divergence for load** (high latency).
- **Avoid narrow global access** ($2 \times$ halo width).

Disadvantage:

- Branch **divergence for compute** (low latency).

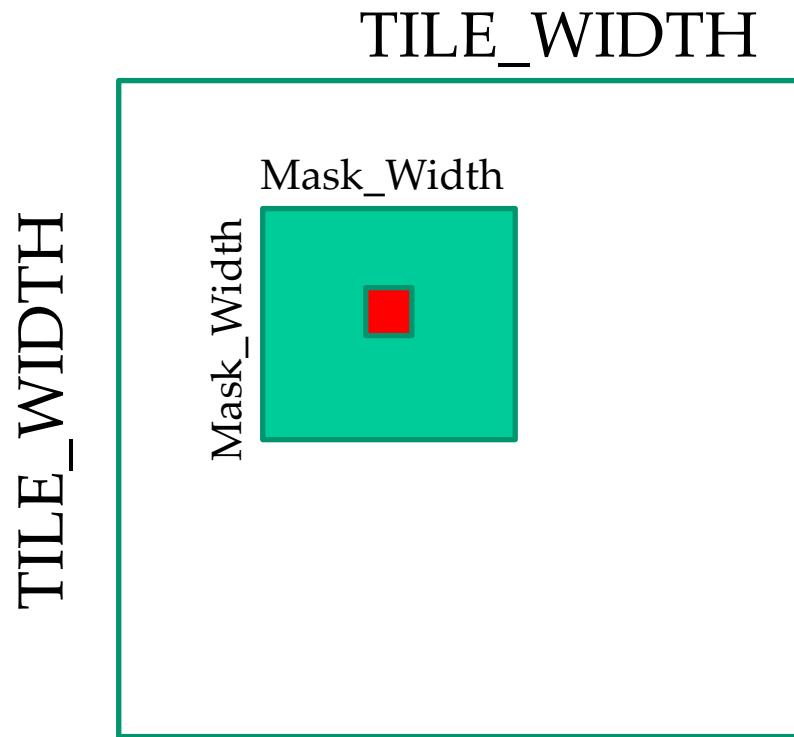
2D Example of Loading Parallelization

Let's do an example for 2D convolution

- Thread block matches input tile size
- Each thread loads one element of input tile
- Some threads do not participate in calculating output (Strategy 2)

Parallelizing Tile Loading

- Load a tile of N into shared memory
 - All threads participate in loading
 - A subset of threads then use each N element in shared memory

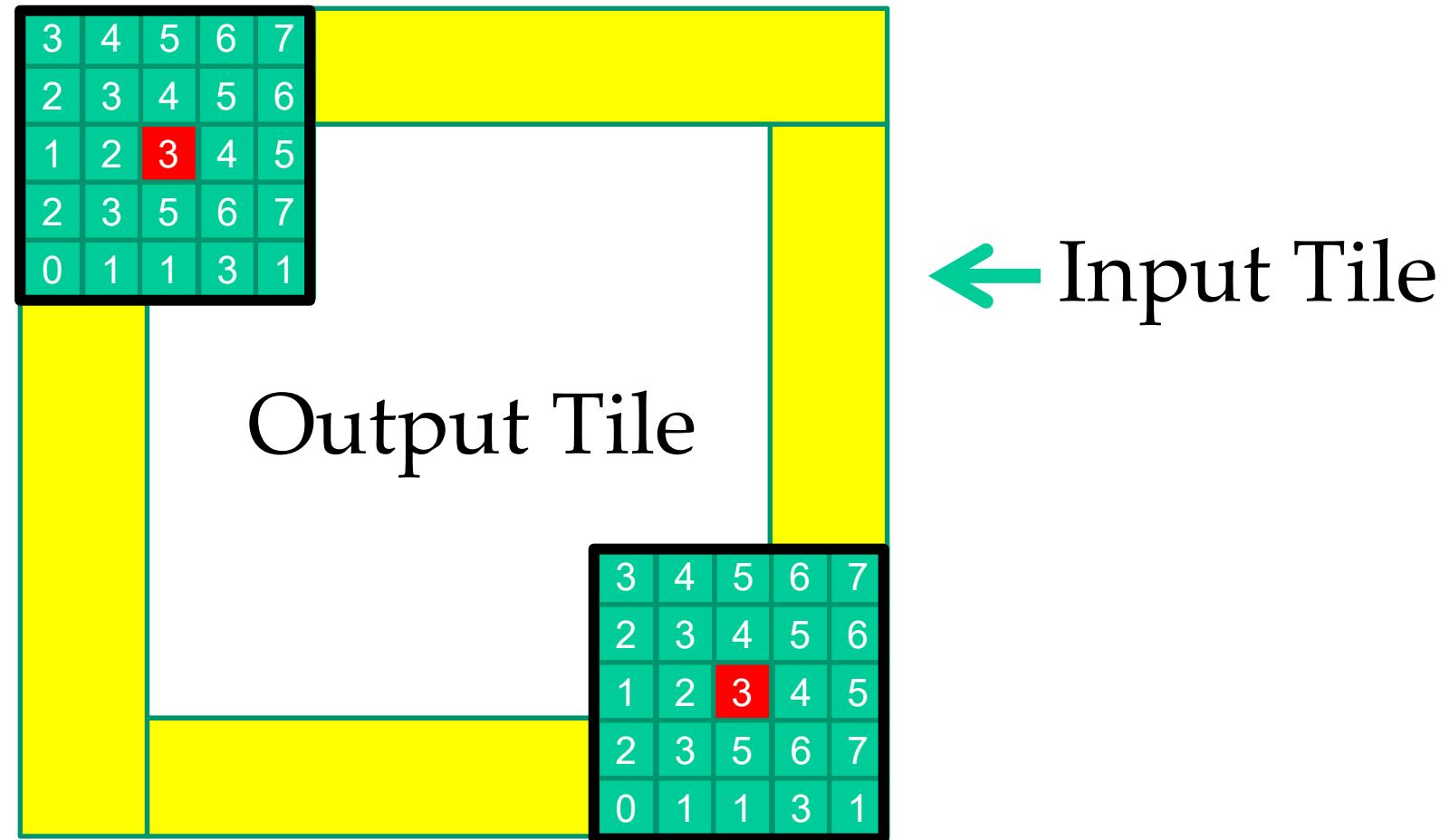


Output Tiles Still Cover the Output!

`row_o = blockIdx.y*TILE_WIDTH +
threadIdx.y;`



Input tiles need to be larger than output tiles



Setting Block Dimensions

```
dim3 dimBlock(TILE_WIDTH + 4, TILE_WIDTH + 4, 1);
```

In general, block width (square blocks) should be

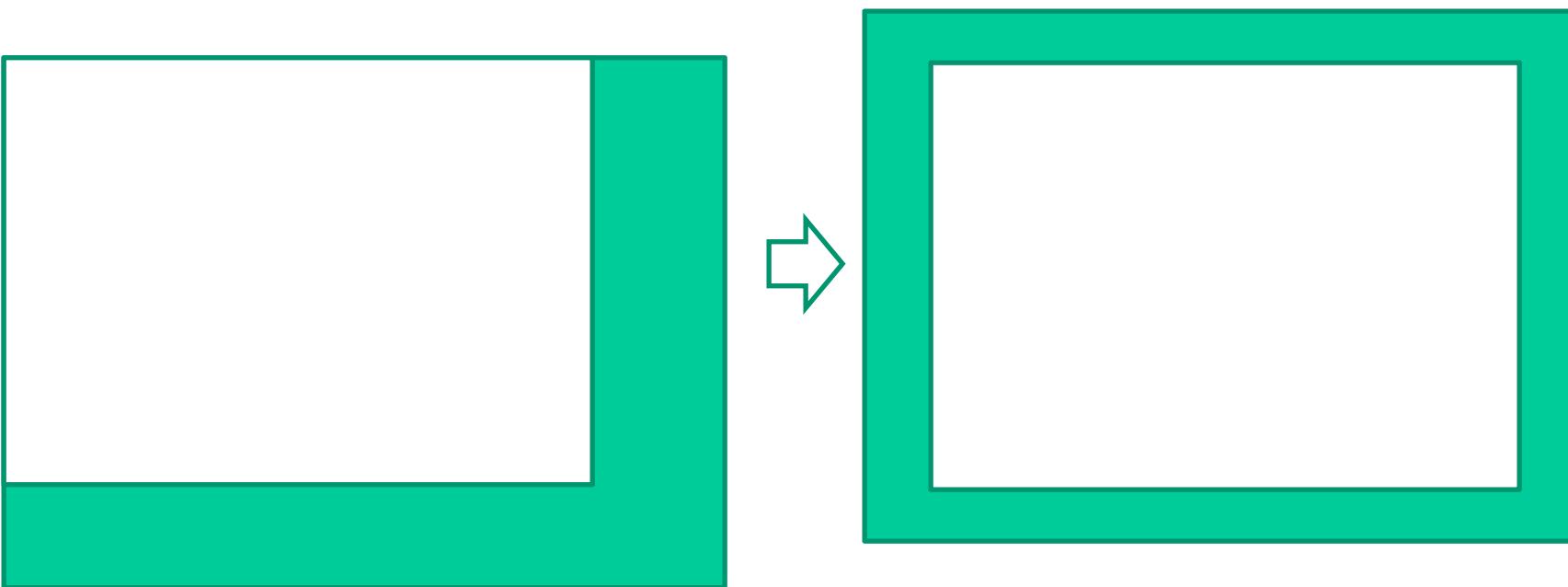
```
TILE_WIDTH + (MASK_WIDTH-1)
```

```
dim3 dimGrid(ceil(P.width/(1.0*TILE_WIDTH)) ,  
              ceil(P.height/(1.0*TILE_WIDTH)) , 1)
```

There need to be enough thread blocks to generate all P elements.

There need to be enough threads to load entire tile of input.

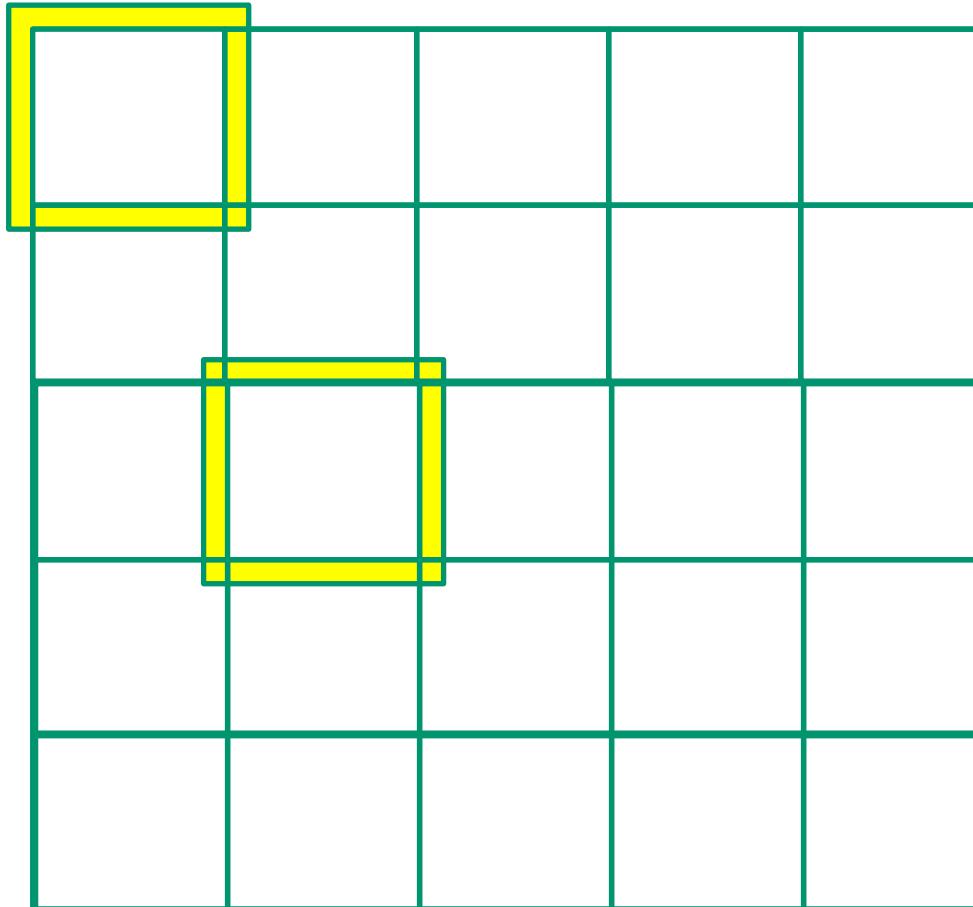
Shifting from output coordinates to input coordinates



Shifting from output coordinates to input coordinates

```
int tx = threadIdx.x;  
int ty = threadIdx.y;  
  
int row_o = blockIdx.y * TILE_WIDTH + ty;  
int col_o = blockIdx.x * TILE_WIDTH + tx;  
  
int row_i = row_o-2; // MASK_WIDTH / 2  
int col_i = col_o-2; // (radius in  
// prev. code)
```

Threads that loads halos outside N should return 0.0



Taking Care of Boundaries

```
float Pvalue = 0.0f;  
  
if((row_i >= 0) && (row_i < Width) &&  
    (col_i >= 0) && (col_i < Width)) {  
    tile[ty][tx] = N[row_i*Width + col_i];  
} else {  
    tile[ty][tx] = 0.0f;  
}  
__syncthreads () ; // wait for tile
```

Not All Threads Calculate Output

```
if(ty < TILE_WIDTH && tx <TILE_WIDTH) {  
    for(i = 0; i < 5; i++) {  
        for(j = 0; j < 5; j++) {  
            Pvalue += Mc[i][j] * tile[i+ty][j+tx];  
        }  
    }  
    // if continues on next page
```

Not All Threads Write Output

```
if(row_o < Width && col_o < Width)
    P[row_o * Width + col_o] = Pvalue;
}
} // end of if selecting output
// tile threads
```

Alternatively

- You can extend the 1D strategy 3 tiled convolution into a 2D strategy 3 tiled convolution.
 - Each input tile matches its corresponding output tile
 - All halo elements will be loaded from global memory
 - If condition and divergence during inner product computation



**ANY MORE QUESTIONS?
READ CHAPTER 7**



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 7: Convolution, Constant Memory and Constant Caching

Course Reminders

- Lab updates
 - We are grading Lab 1 now, expect it to be graded soon
 - Lab 2 is due this Friday
 - Lab 3 is out and is due next Friday
- Take a note of Midterm 1 date
 - March 7
 - Other details to be provided soon

Objective

- To learn convolution, an important parallel computation pattern
 - Widely used in signal, image and video processing
 - Foundational to stencil computation used in many science and engineering applications
 - Critical component of Neural Networks and Deep Learning
- Important GPU technique
 - Taking advantage of cache memories

Convolution Mathematics

$$f(x) * g(x) = \int_{-\infty}^{\infty} f(\tau) \cdot g(x - \tau) d\tau$$

$$f[x] * g[x] = \sum_{k=-\infty}^{\infty} f[k] \cdot g[x - k]$$

Convolution Applications

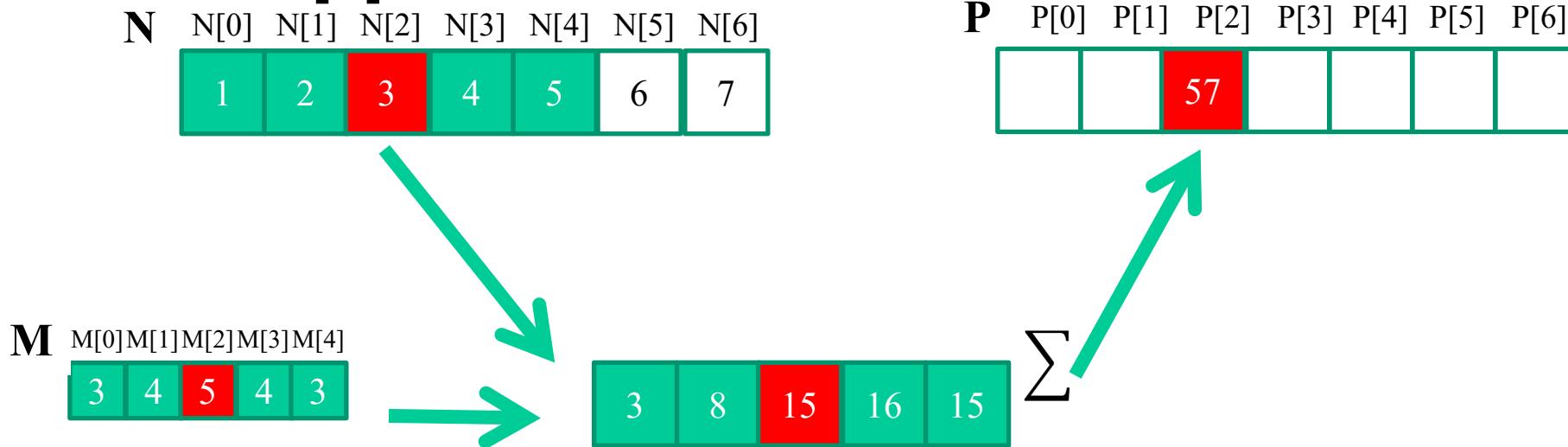
- A popular operation that is used in various forms in signal processing, digital recording, image processing, video processing, computer vision, and machine learning.
- Convolution is often performed as a **filter** that transforms the input signal (audio, video, etc) in some context-aware way.
 - Some filters smooth out the signal values so that one can see the big-picture trend
 - Or Gaussian filters to blur images, backgrounds

Convolution Computation

- An array operation where each output data element is a weighted sum of a collection of neighboring input elements
- The weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the *convolution kernel*
 - We will refer to these mask arrays as convolution masks or convolution filters to avoid confusion.
 - The same convolution mask is typically used for all elements of the array.

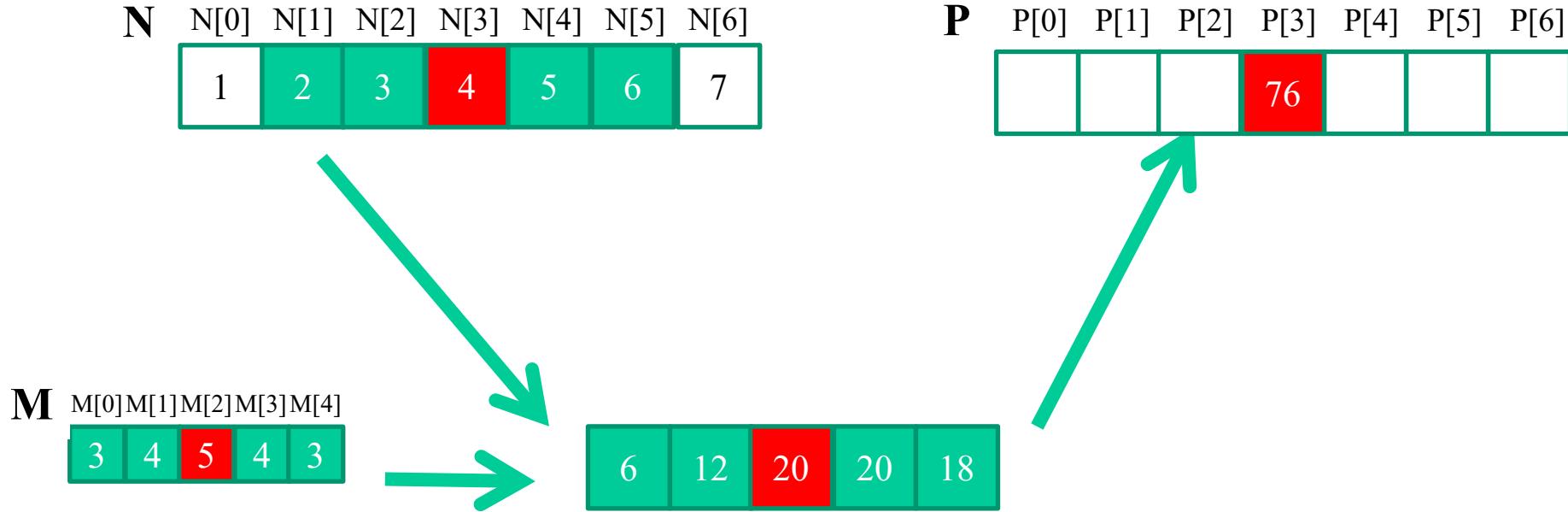
1D Convolution Example

- Commonly used for audio processing
 - MASK_WIDTH is usually an odd number of elements for symmetry (5 in this example)
 - MASK_RADIUS is the number of elements used in convolution on each side of the pixel being calculated (2 in this example).
- Calculation of P[2]:



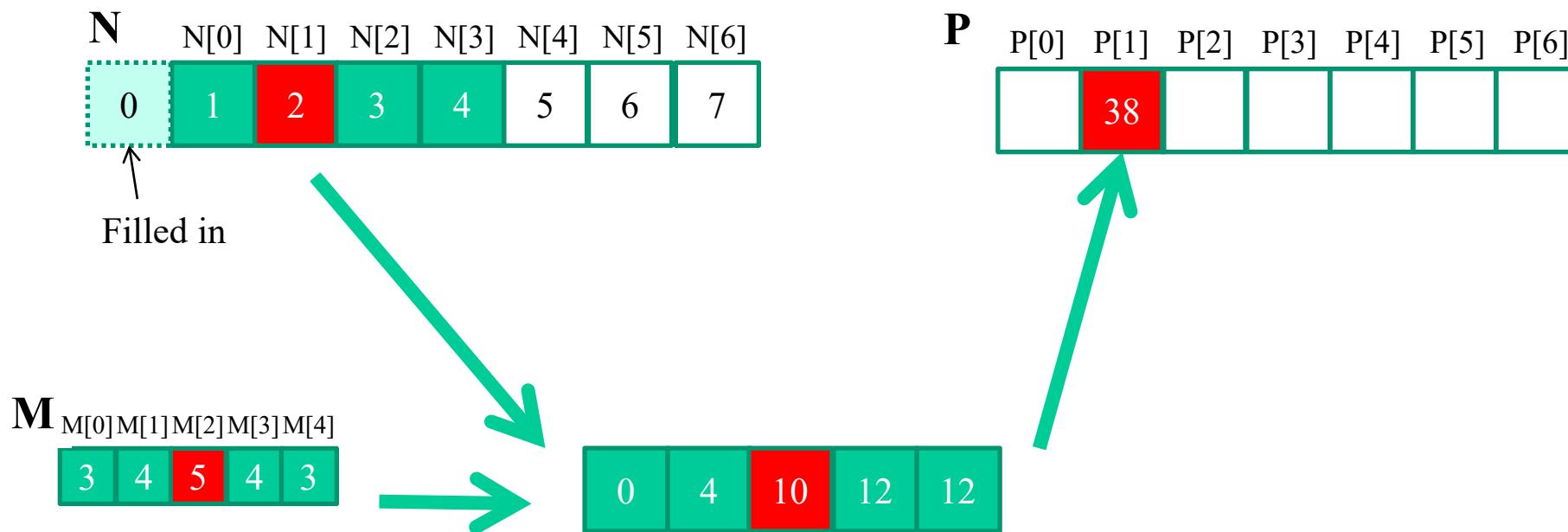
1D Convolution Example

- Calculation of $P[3]$



1D Convolution Boundaries

- Calculation of output elements near the boundaries of the input array need to deal with “ghost” elements
 - Different policies (0, replicates of boundary values, etc.)



A 1D Convolution Kernel with Boundary Handling

- This kernel forces all elements outside the valid range to 0

```
__global__ void
convolution_1D_basic_kernel(float *N, float *M, float *P, int Mask_Width, int Width)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    float Pvalue = 0;
    int N_start_point = i - (Mask_Width/2);

    for (int j = 0; j < Mask_Width; j++) {
        if (((N_start_point + j) >= 0) && ((N_start_point + j) < Width)) {
            Pvalue += N[N_start_point + j]*M[j];
        }
    }

    P[i] = Pvalue;
}
```

2D Convolution

N

1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	5	6
5	6	7	8	5	6	7
6	7	8	9	0	1	2
7	8	9	0	1	2	3

P

		321		

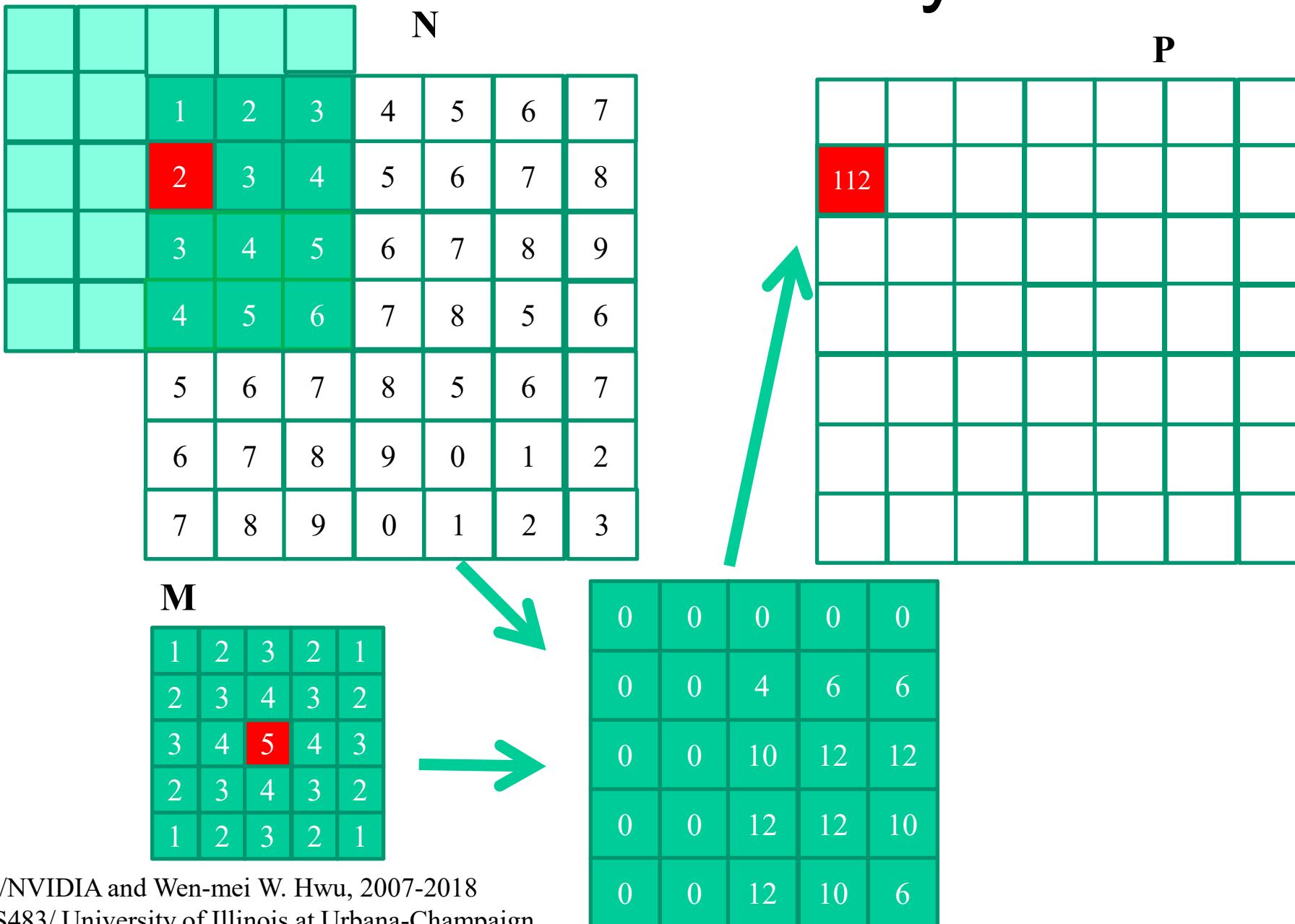
M

1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

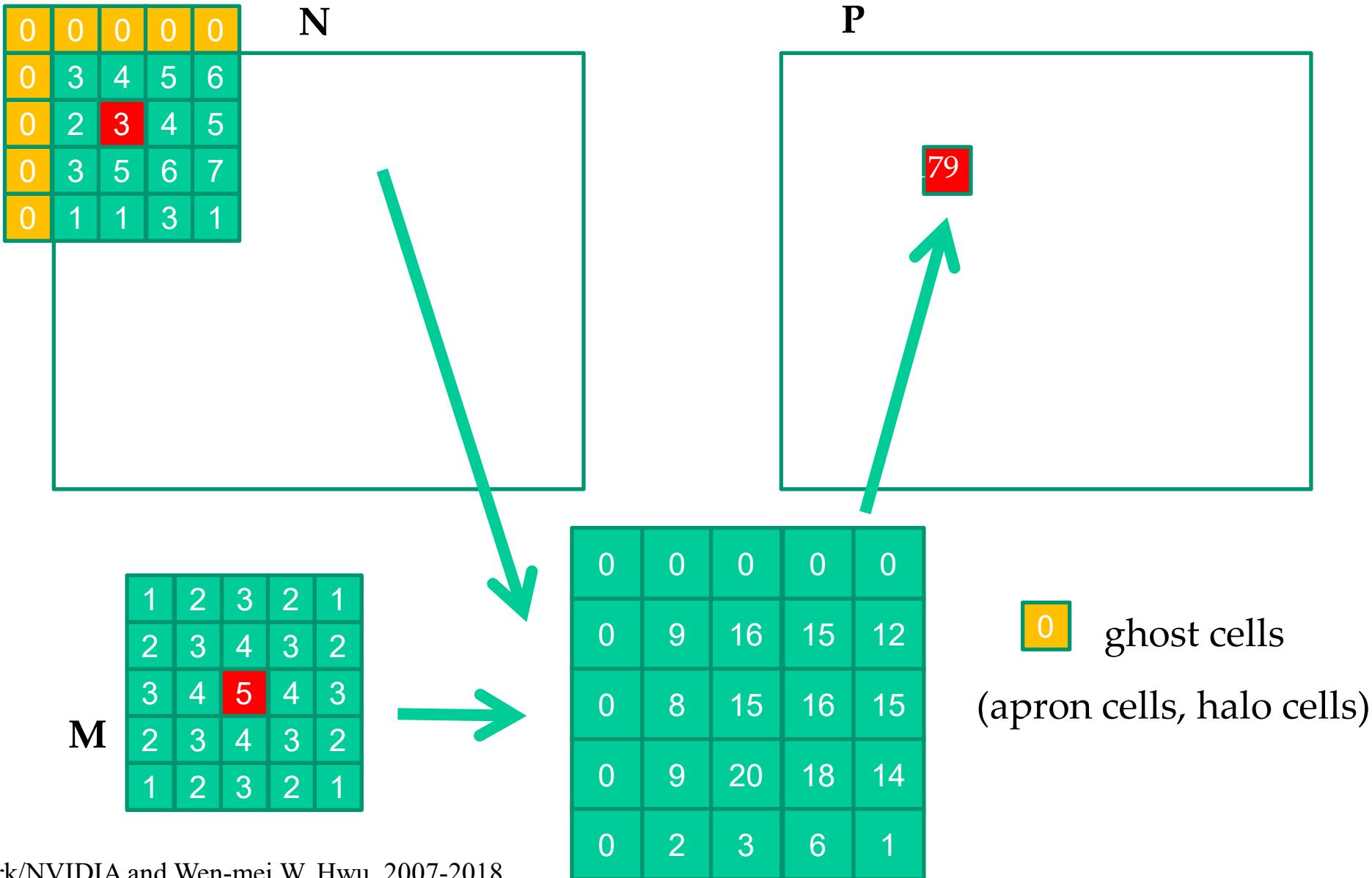


1	4	9	8	5
4	9	16	15	12
9	16	25	24	21
8	15	24	21	16
5	12	21	16	5

2D Convolution Boundary Condition



2D Convolution – Ghost Cells



What does this kernel accomplish?

$$\mathbf{M} = \frac{1}{273} \times$$

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

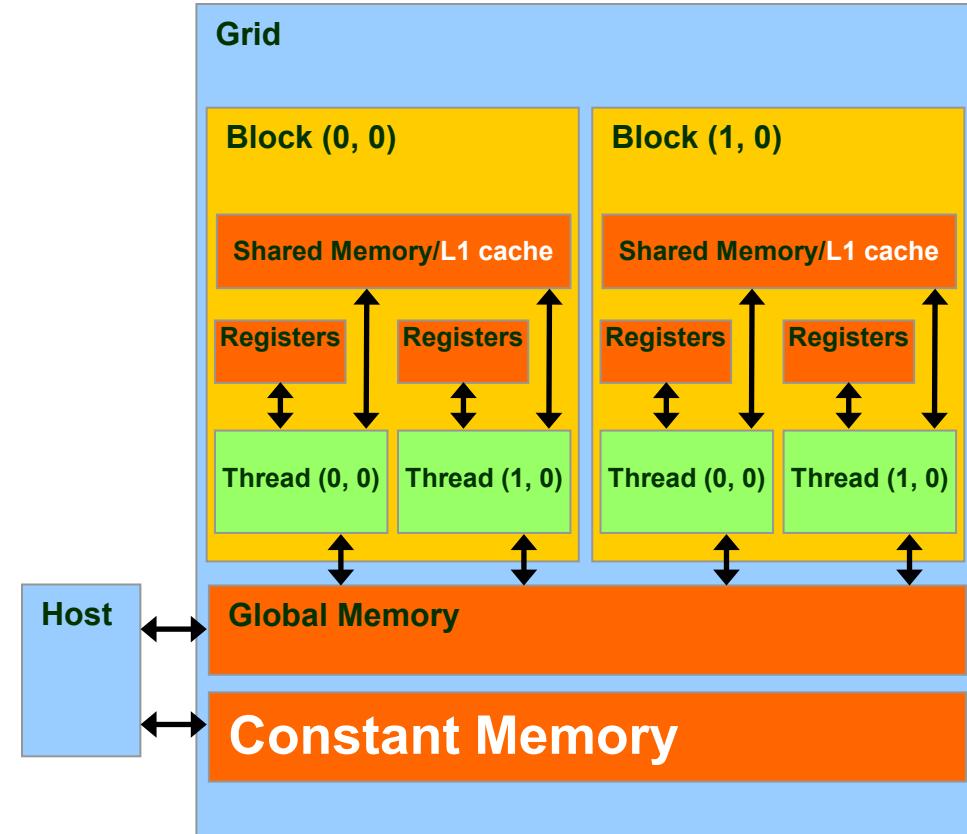
Hint: Assume input N is an image

Access Pattern for M

- Elements of M are called **mask** (kernel, filter) **coefficients** (weights)
 - Calculation of all output P elements needs M
 - M is not changed during grid execution
- Bonus - M elements are accessed in the same order when calculating all P elements
- M is a good candidate for **Constant Memory**

Programmer View of CUDA Memories (Review)

- Each thread can:
 - Read/write per-thread **registers (~1 cycle)**
 - Read/write per-block **shared memory (~5 cycles)**
 - Read/write per-grid **global memory (~500 cycles)**
 - Read/only per-grid **constant memory (~5 cycles with caching)**



Memory Hierarchies

- Review: If we had to go to global memory to access data all the time, the execution speed of GPUs would be limited by the global memory bandwidth
 - We saw the use of shared memory in tiled matrix multiplication to reduce this limitation
- Another important solution: Caches

Cache

- A cache is an “array” of cache lines
 - A cache line can usually hold data from several consecutive memory addresses
- When data is requested from the global memory, an entire cache line that includes the data being accessed is loaded into the cache, in an attempt to reduce global memory requests
 - The data in the cache is a “copy” of the original data in global memory
 - Additional hardware is used to remember the addresses of the data in the cache line

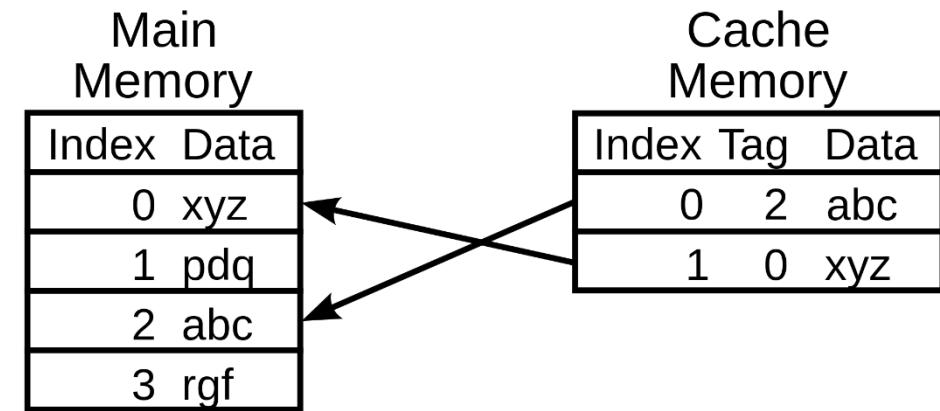
Caches Store Lines of Memory

Recall: memory bursts

- contain around **1024 bits (128B)** from
- consecutive (linear) addresses.
- Let's call a single burst a **line**.

What's a **cache**?

- An **array of cache lines** (and tags).
- Memory **read produces** a **line**,
- **cache stores** a **copy** of the line, and
- tag records line's memory address.



Caches and Locality

- Some definitions:
 - Spatial locality: when the data elements stored in consecutive memory locations are accessed consecutively
 - Temporal locality: when the same data element is accessed multiple times in short period of time
- Both spatial locality and temporal locality improve the performance of caches

Memory Accesses Show Locality

An executing program

- loads and store data from memory.
- Consider sequence of addresses accessed.

Sequence usually shows two types of locality:

- spatial: accessing X implies accessing X+1 (and X+2, and so forth) soon
- temporal: accessing X implies accessing X again soon

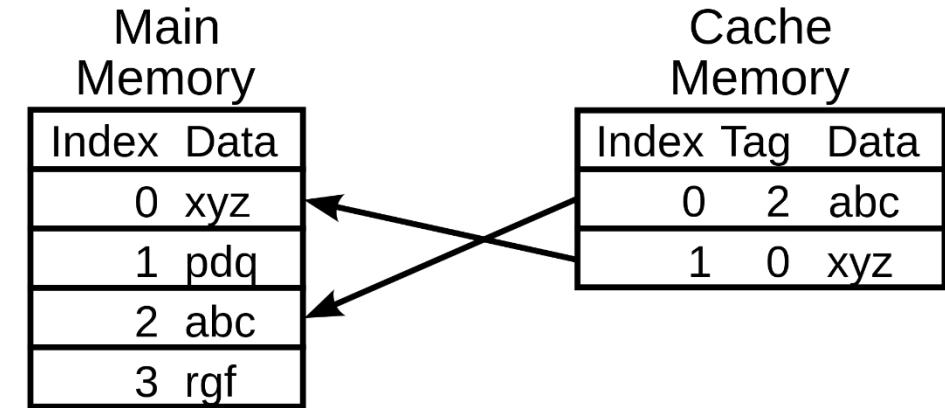
(Caches improve performance for both types.)

Caches Can't Hold Everything

Caches are smaller than memory.

When cache is full,

- must make room for new line,
- usually by **discarding least recently used line**.



Shared Memory vs. Cache

- Shared memory in CUDA is another type of temporary storage used to relieve main memory contention.
 - In terms of distance from the SMs, shared memory is similar to L1 cache.
- Unlike cache, shared memory does not necessarily hold a copy of data that is also in main memory
 - Shared mem requires explicit data transfer instructions into locations in the shared mem, whereas cache doesn't

Shared Memory vs. Cache - Cont'd

- Caches vs. shared memory
 - Both on chip*, with similar performance
 - (As of Volta generation, both using the same physical resources, allocated dynamically!)

What's the difference?

- **Programmer controls shared memory** contents (called a scratchpad)
- **Microarchitecture** automatically **determines contents of cache**.

*Static RAM, not DRAM, by the way—see ECE120/CS233.

Constant Cache in GPUs

- Modification to cached data needs to be (eventually) reflected back to the original data in global memory
 - Requires logic to track the modified status, etc.
- Constant cache is a special cache for constant data that will not be modified during kernel execution by a grid
 - Data declared in the constant memory not modified during kernel execution.
 - Constant cache can be accessed with higher throughput than L1 cache for some common patterns

GPU Has Constant and L1 Caches

To support writes (modification of lines),

- changes must be copied back to memory, and
- cache must track modification status.
- L1 cache in GPU (for global memory accesses) supports writes.

Cache for constant / texture memory

- Special case: lines are read-only
- Enables higher-throughput access than L1 for common GPU kernel access patterns.

How to Use Constant Memory

Host code is **similar** to previous versions, but...

Allocate device memory for **M** (the mask)

- **outside of all functions**
- **using constant**
(tells GPU that caching is safe).

For copying to device memory, **use**

- **cudaMemcpyToSymbol(dest, src, size, offset = 0, kind = cudaMemcpyHostToDevice)**
- with destination defined as above.

Host Code Example

```
(MASK_WIDTH is the size of the mask.)  
// global variable, outside any kernel/function  
__constant__ float Mc[MASK_WIDTH][MASK_WIDTH];  
  
// Initialize Mask  
float Mask[MASK_WIDTH][MASK_WIDTH]  
for(unsigned int i = 0; i < MASK_WIDTH * MASK_WIDTH; i++) {  
    Mask[i] = (rand() / (float)RAND_MAX);  
    if(rand() % 2) Mask[i] = - Mask[i]  
}  
cudaMemcpyToSymbol(Mc, Mask, MASK_WIDTH*MASK_WIDTH*sizeof(float));  
  
ConvolutionKernel<<<dimGrid, dimBlock>>>(Nd, Pd);
```



**ANY MORE QUESTIONS?
READ CHAPTER 7**



ECE408/CS483/CSE408 Spring 2023 Applied Parallel Programming

Lecture 6: Generalized Tiling & DRAM Bandwidth

Course Reminders

- Lab 1 is due this Friday
- Lab 2 is due next Friday

Objectives

- To learn to handle boundary conditions in tiled algorithms.
- To understand the organization of memory based on dynamic RAM (DRAM).
- To understand the use of burst mode and multiple banks (both sources of parallelism) to increase DRAM performance (data rate).
- To understand memory access coalescing, which connects GPU kernel performance to DRAM organization.

How to Handle Matrices of Other Sizes?

- Lecture 5's tiled kernel
 - assumed integral number of tiles (thread blocks)
 - in all matrix dimensions.

How can we avoid this assumption?

- One answer: add padding, but not easy to reformat data, and adds transfer time.

Other ideas?

Let's Review Our Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
1. __shared__ float subTileM[TILE_WIDTH] [TILE_WIDTH];
2. __shared__ float subTileN[TILE_WIDTH] [TILE_WIDTH];

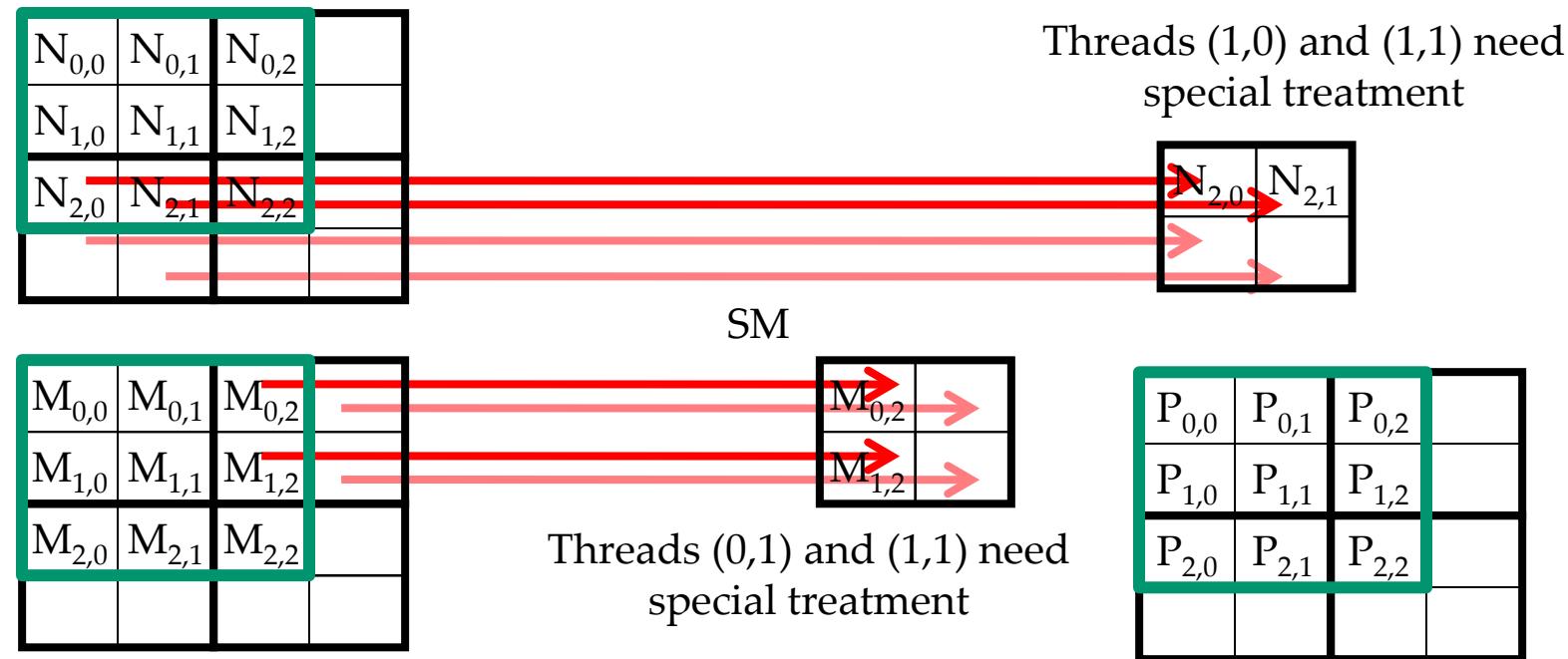
3. int bx = blockIdx.x;  int by = blockIdx.y;
4. int tx = threadIdx.x; int ty = threadIdx.y;

        // Identify the row and column of the P element to work on
5. int Row = by * TILE_WIDTH + ty; // note: blockDim.x == TILE_WIDTH
6. int Col = bx * TILE_WIDTH + tx; //          blockDim.y == TILE_WIDTH
7. float Pvalue = 0;

        // Loop over the M and N tiles required to compute the P element
        // The code assumes that the Width is a multiple of TILE_WIDTH!
8. for (int m = 0; m < Width/TILE_WIDTH; ++m) {
            // Collaborative loading of M and N tiles into shared memory
9.     subTileM[ty][tx] = M[Row*Width + m*TILE_WIDTH+tx];
10.    subTileN[ty][tx] = N[(m*TILE_WIDTH+ty)*Width+Col];
11.    __syncthreads();
12.    for (int k = 0; k < TILE_WIDTH; ++k)
13.        Pvalue += subTileM[ty][k] * subTileN[k][tx];
14.    __syncthreads();
15. }

16. P[Row*Width+Col] = Pvalue;
}
```

Second Tile Load for Block (0,0)

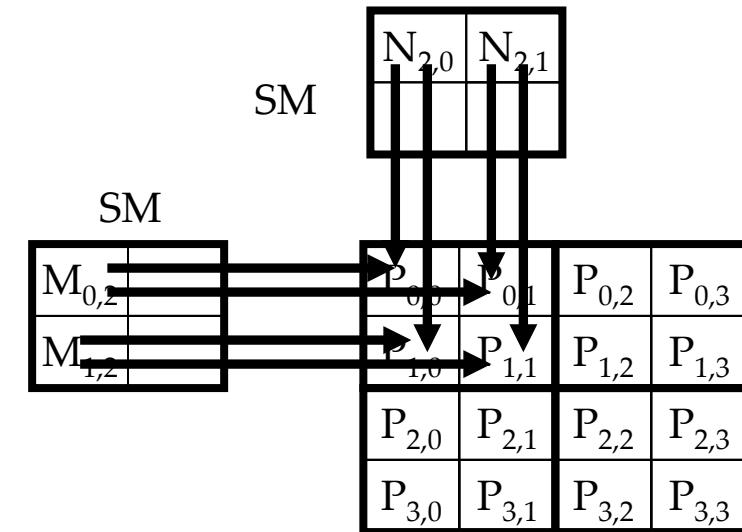


6

Second Tile Use for Block (0,0), k of 0

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

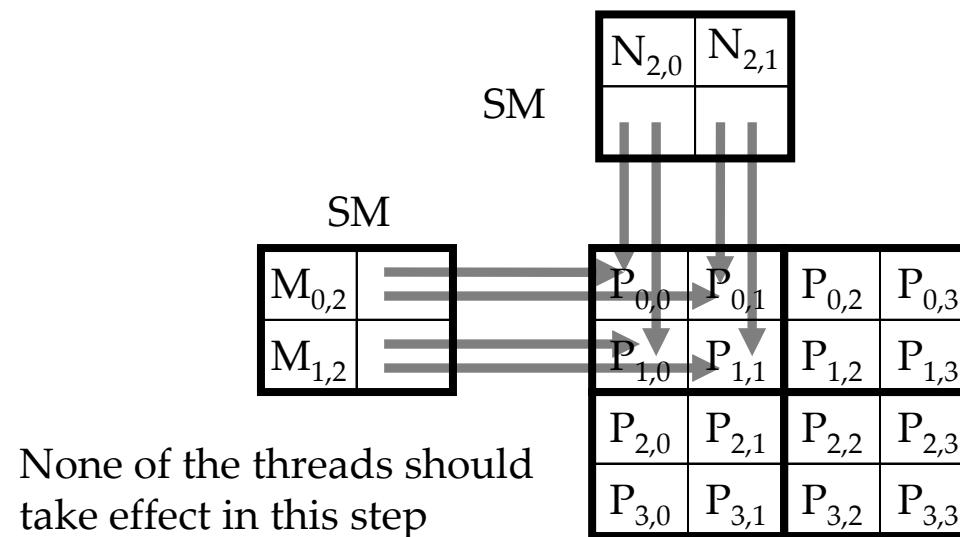
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	



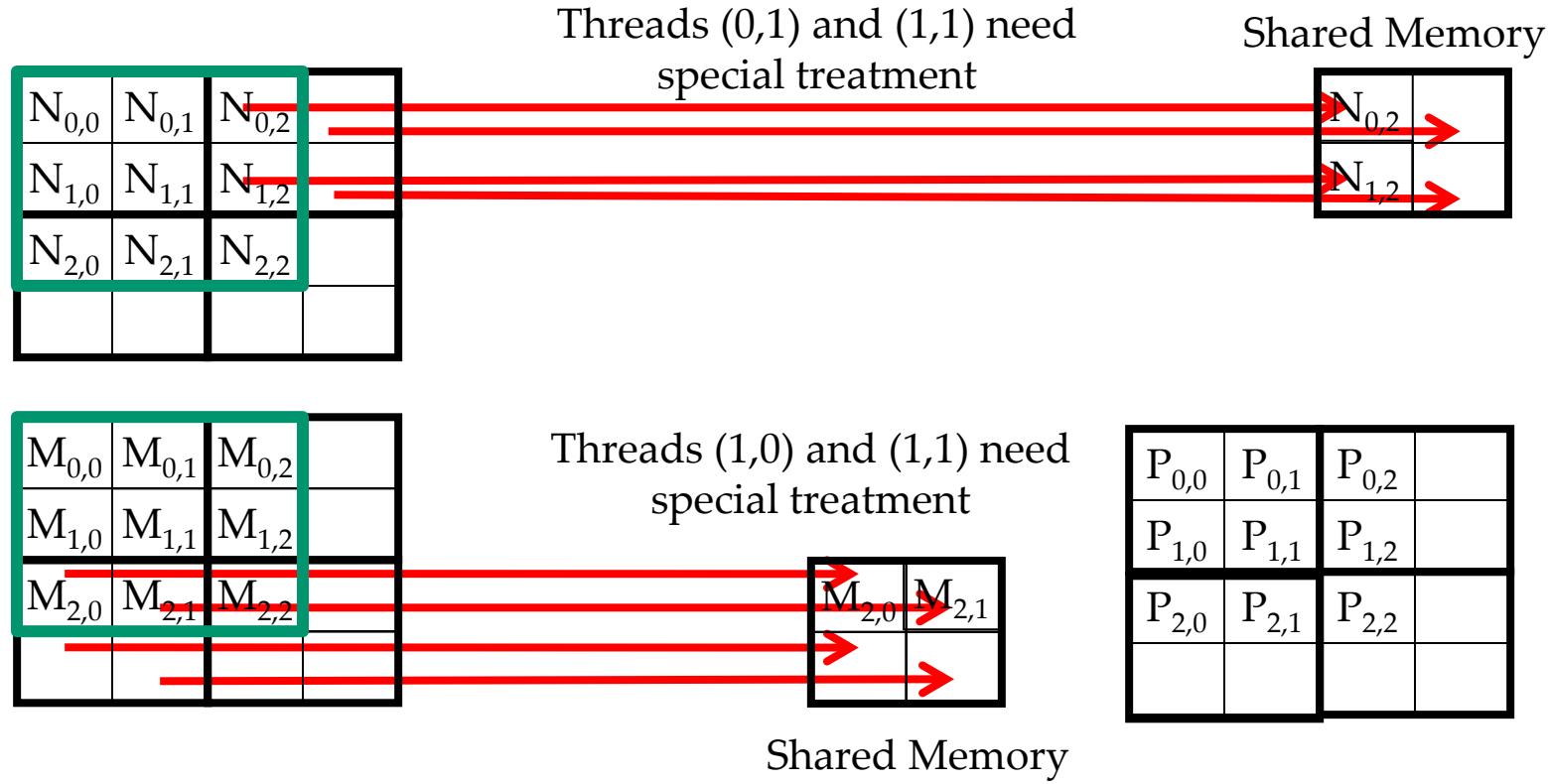
Second Tile Use for Block (0,0), k of 1

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	



First Tile Load for Block (1,1)

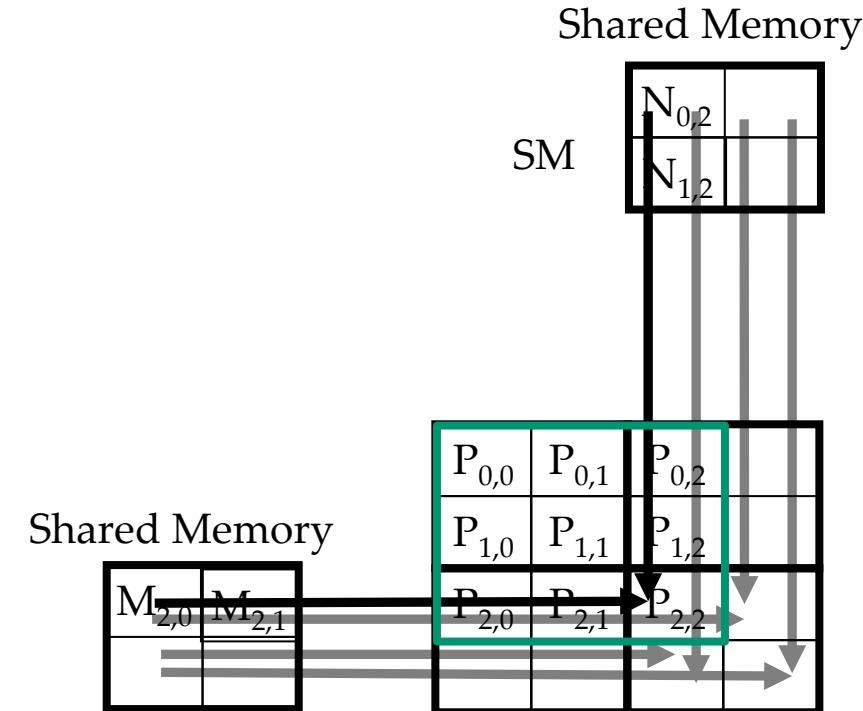


9

First Tile Use for Block (1,1), k of 0

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	

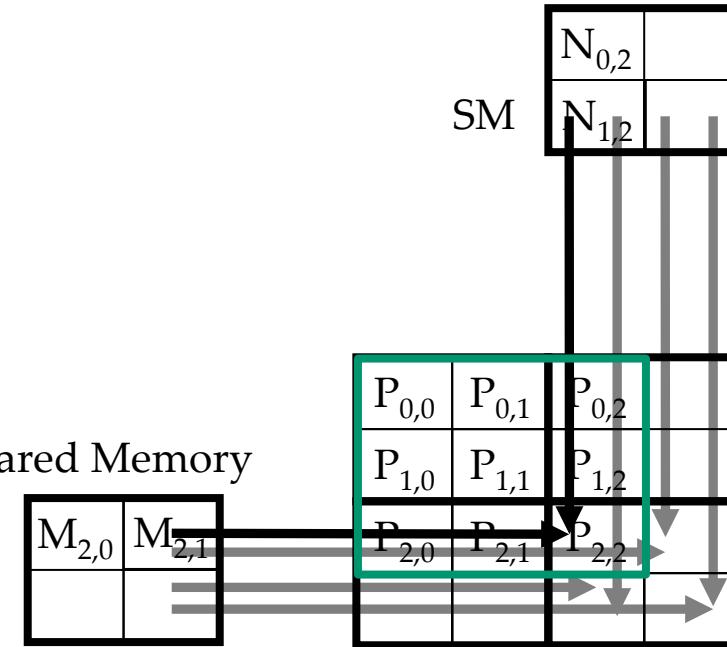


First Tile Use for Block (1,1), k of 1

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	

Shared Memory



Shared Memory

Major Cases in Toy Example

- Threads that calculate valid P elements but can step outside valid input
 - Second tile of Block(0,0), all threads when k is 1
- Threads that do not calculate valid P elements
 - Block(1,1), Thread(1,0), non-existent row
 - Block(1,1), Thread(0,1), non-existing column
 - Block(1,1), Thread(1,1), non-existing row/column

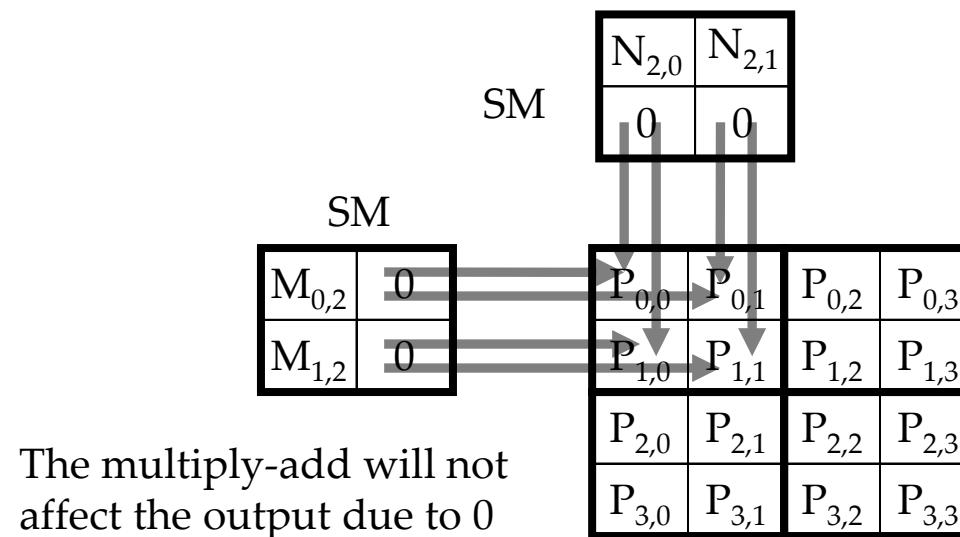
Solution: Write 0 for Missing Elements

- Test during tile load:
is **target within input matrix?**
 - If yes, proceed to **load**;
 - otherwise, just **write 0** to shared memory.
- The **benefit**?
 - **No specialization during tile use!**
 - Multiplying by 0 guarantees that unwanted terms do not contribute to the inner product.

Second Tile Use for Block (0,0), k of 1

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	



The multiply-add will not
affect the output due to 0

What About Threads Outside of P?

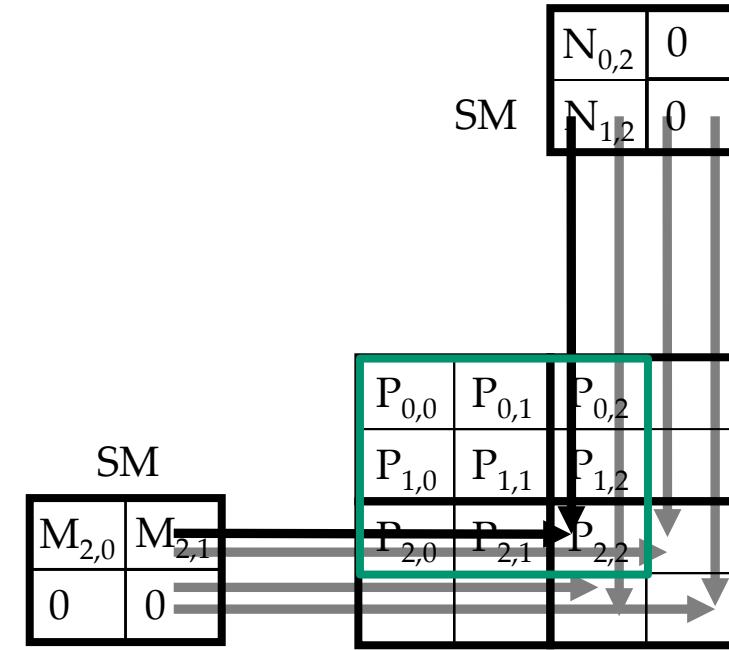
- If a **thread is not within P**,
 - All terms in sum are 0.
 - No harm in performing FLOPs.
 - No harm in writing to registers.
 - **Must not be allowed to write to global memory!**

So: **Threads outside of P calculate 0,
but store nothing.**

First Tile Use for Block (1, 1)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	



Modifying the Tile Count

```
8. for (int m = 0; m < Width/TILE_WIDTH; ++m) {
```

The bound for **m** implicitly assumes that **Width** is a multiple of **TILE_WIDTH**. We need to round up.

```
for (int m = 0; m < (Width - 1)/TILE_WIDTH + 1; ++m) {
```

For non-multiples of **TILE_WIDTH**:

- quotient is unchanged;
- add one to round up.

For multiples of **TILE_WIDTH**:

- quotient is now one smaller,
- but we add 1.

Modifying the Tile Loading Code

We had ...

```
// Collaborative loading of M and N tiles into shared memory  
9.    subTileM[ty][tx] = M[Row*Width + m*TILE_WIDTH+tx];  
10.   subTileN[ty][tx] = N[(m*TILE_WIDTH+ty)*Width+Col];
```

Note: the tests for M and N tiles are NOT the same.

```
if (Row < Width && m*TILE_WIDTH+tx < Width) {  
    // as before  
    subTileM[ty][tx] = M[Row*Width + m*TILE_WIDTH+tx];  
} else {  
    subTileM[ty][tx] = 0;  
}
```

And for Loading N...

We had ...

```
// Collaborative loading of M and N tiles into shared memory  
9.    subTileM[ty][tx] = M[Row*Width + m*TILE_WIDTH+tx];  
10.   subTileN[ty][tx] = N[ (m*TILE_WIDTH+ty)*Width+Col];
```

Note: the tests for M and N tiles are NOT the same.

```
if (m*TILE_WIDTH+ty < Width && Col < Width ) {  
    // as before  
    subTileN[ty][tx] = N[ (m*TILE_WIDTH+ty)*Width+Col];  
} else {  
    subTileN[ty][tx] = 0;  
}
```

Modifying the Tile Use Code

We had ...

```
12. for (int k = 0; k < TILE_WIDTH; ++k)
13.     Pvalue += subTileM[ty][k] * subTileN[k][tx];
```

Note: **no changes are needed**, but we might save a little energy (fewer floating-point ops)?

```
if (Row < Width && Col < Width) {
    // as before
    for (int k = 0; k < TILE_WIDTH; ++k)
        Pvalue += subTileM[ty][k] * subTileN[k][tx];
}
```

Modifying the Write to P

We had ...

```
16. P[Row*Width+Col] = Pvalue;
```

We must test for threads outside of P:

```
if (Row < Width && Col < Width) {  
    // as before  
    P[Row*Width+Col] = Pvalue;  
}
```

Some Important Points

- For each thread, conditions are different for
 - Loading M element
 - Loading N element
 - Calculation/storing output elements
- Branch divergence
 - affects only blocks on boundaries, and
 - should be small for large matrices.
- What about rectangular matrices?

Global Memory (DRAM) Bandwidth

Ideal

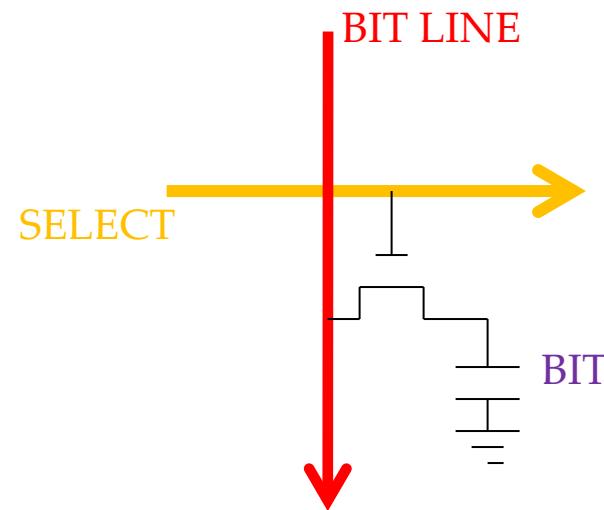


Reality



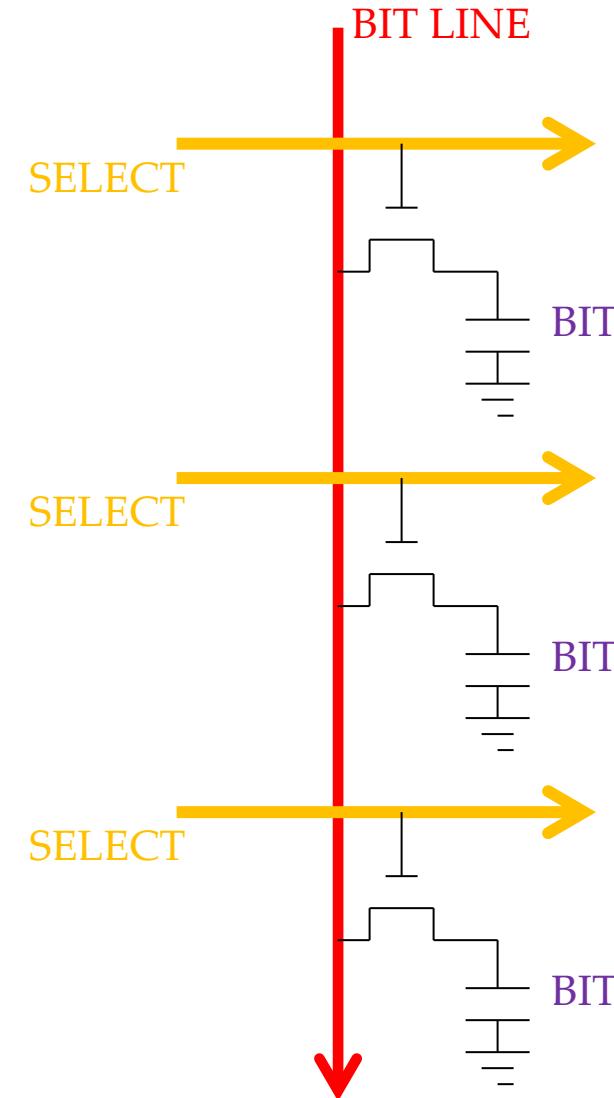
Most Large Memories Use DRAM

- **Random Access Memory (RAM):**
same time needed to read/write any address
- **Dynamic RAM (DRAM):**
 - **bit** stored on a capacitor
 - connected via transistor to **bit line** for read/write
 - **bits disappear** after a while
(around 50 msec, due to tiny leakage currents through transistor),
and must be rewritten (hence dynamic)



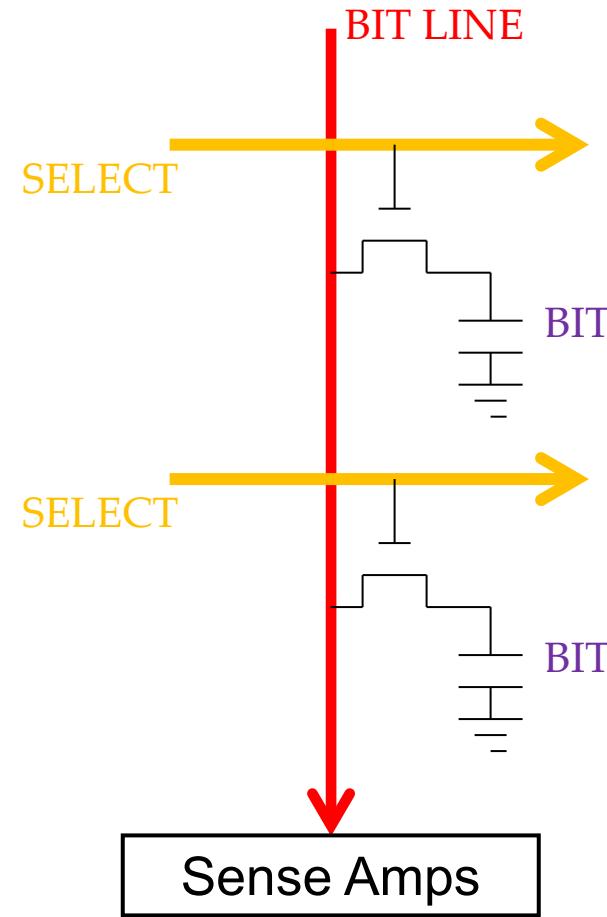
Many Cells (Bits) per Bit Line

- About **1,000 cells** connect to each **BIT LINE**.
- **Connection/disconnection depends** on **SELECT** line.
- Some **address bits** decoded to **connect exactly one cell** to the **BIT LINE**.

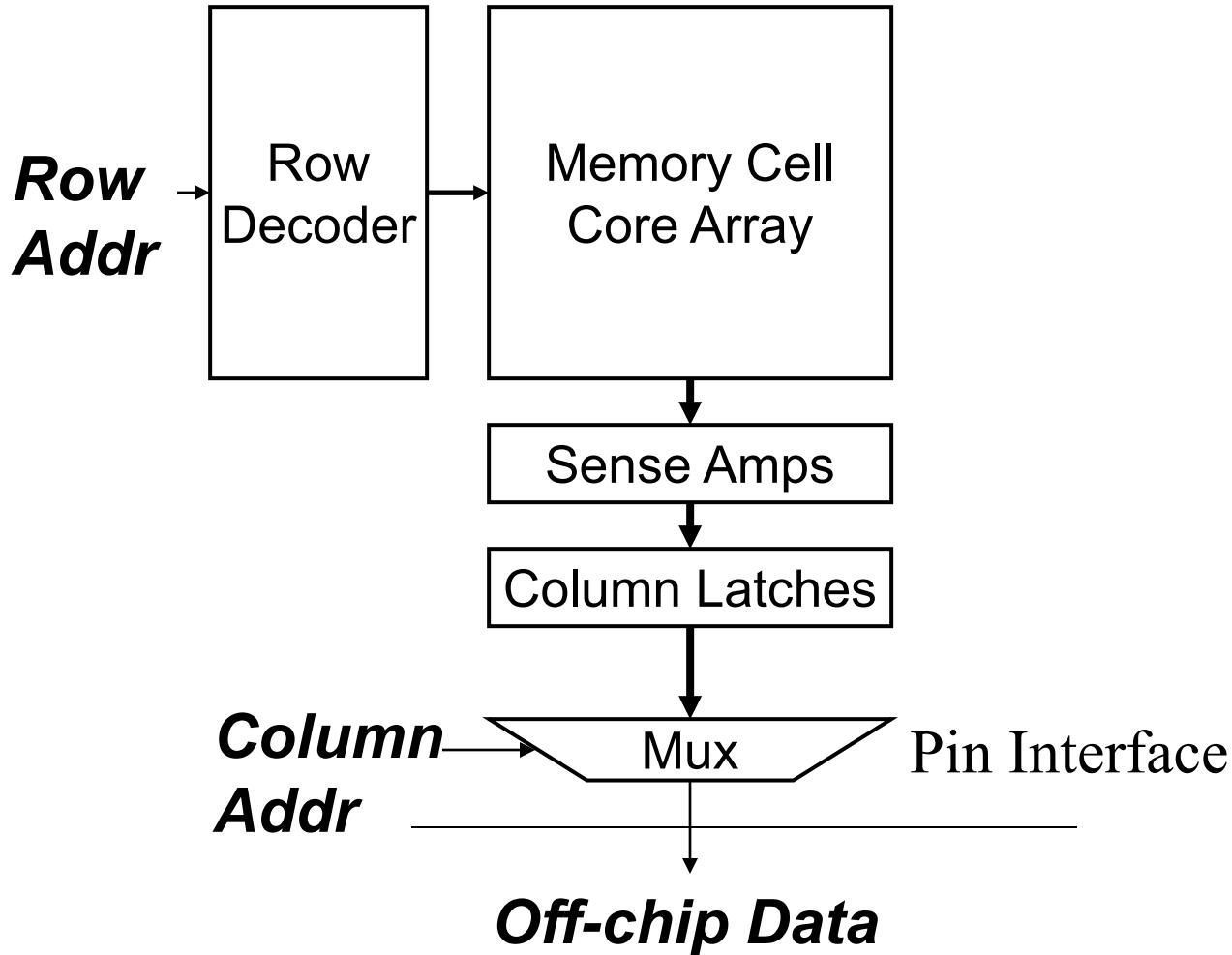


DRAM is Slow But Dense

- Capacitance...
 - tiny for the **BIT**, but
 - huge for the
BIT LINE
- Use an amplifier for higher speed!
- Still **slow**...
- But only need
1 transistor per bit.



DRAM Bank Organization

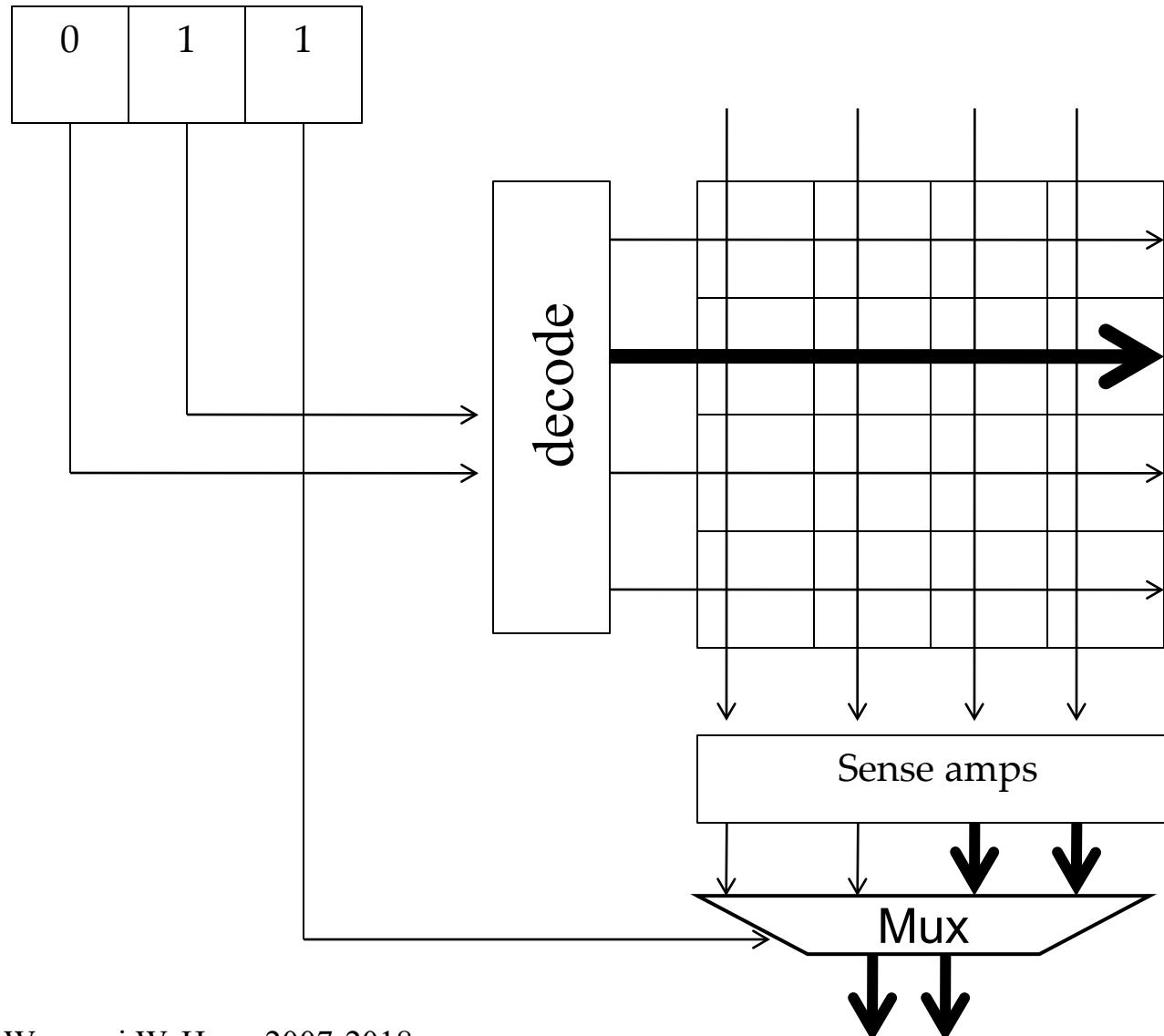


- SELECT lines connect to about 1,000 bit lines.
- Core array has about $O(1M)$ bits
- Use more address bits to choose bit line(s).

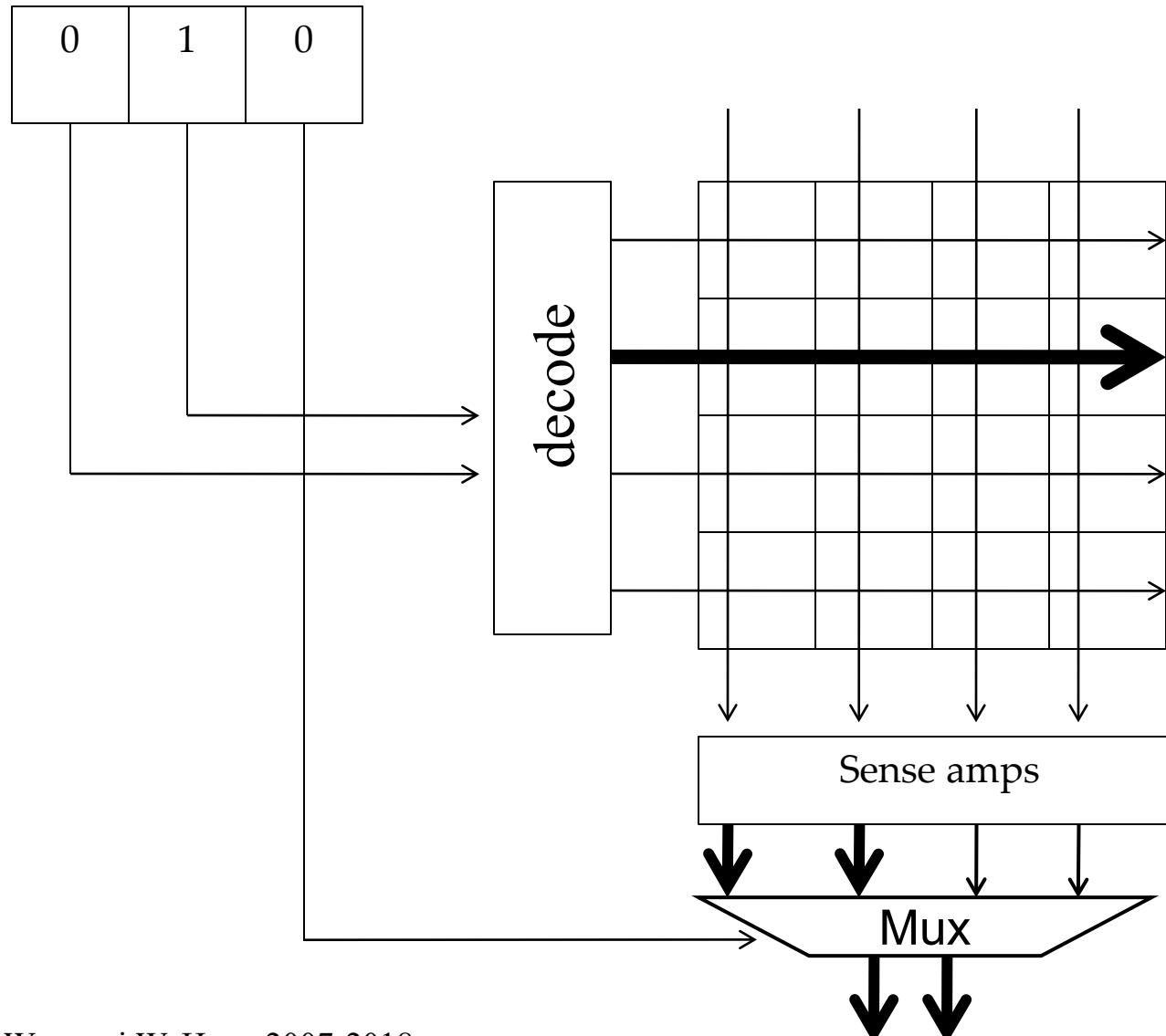
DRAM Interfaces are Clocked

- DRAM **cells** are **not clocked** (clocking requires transistors).
- DRAM **interfaces** are **clocked**.
 - DDR: Core speed = $\frac{1}{2}$ interface speed
 - DDR2/GDDR3: Core speed = $\frac{1}{4}$ interface speed
 - DDR3/GDDR4: Core speed = $\frac{1}{8}$ interface speed
 - ... likely to be worse in the future

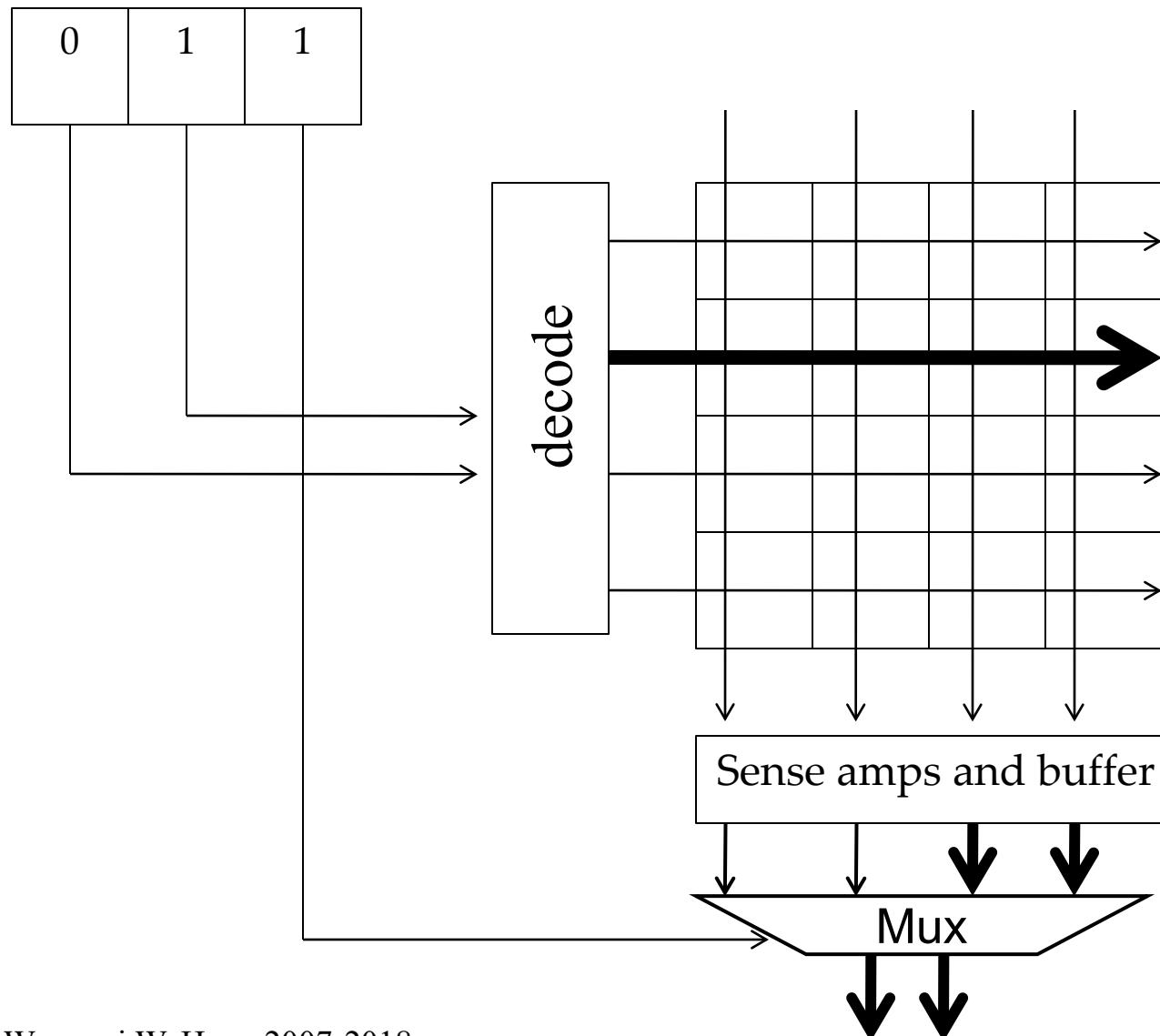
A very small (8x2 bit) DRAM Bank



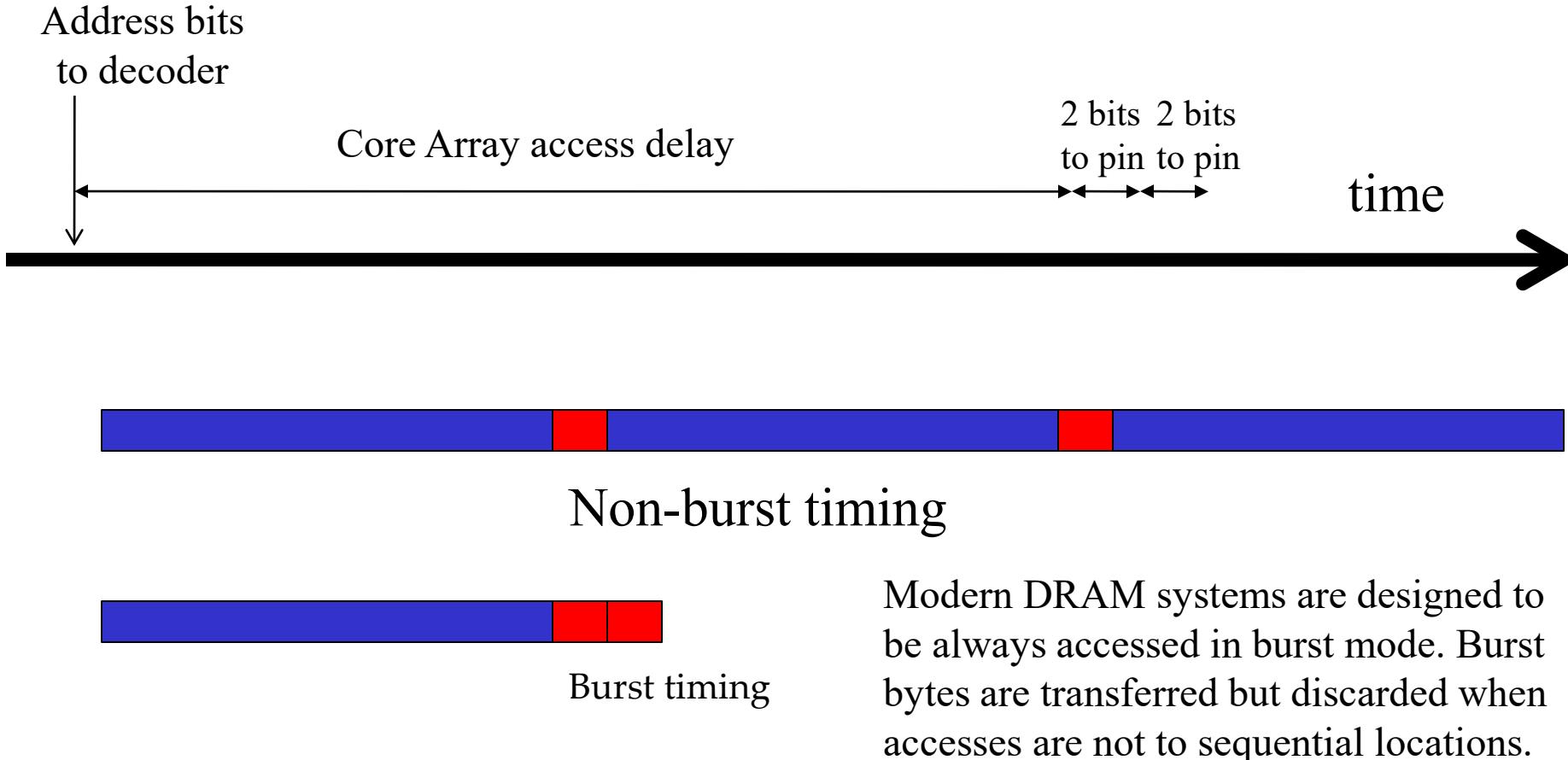
DRAM Bursting (burst size = 4 bits)



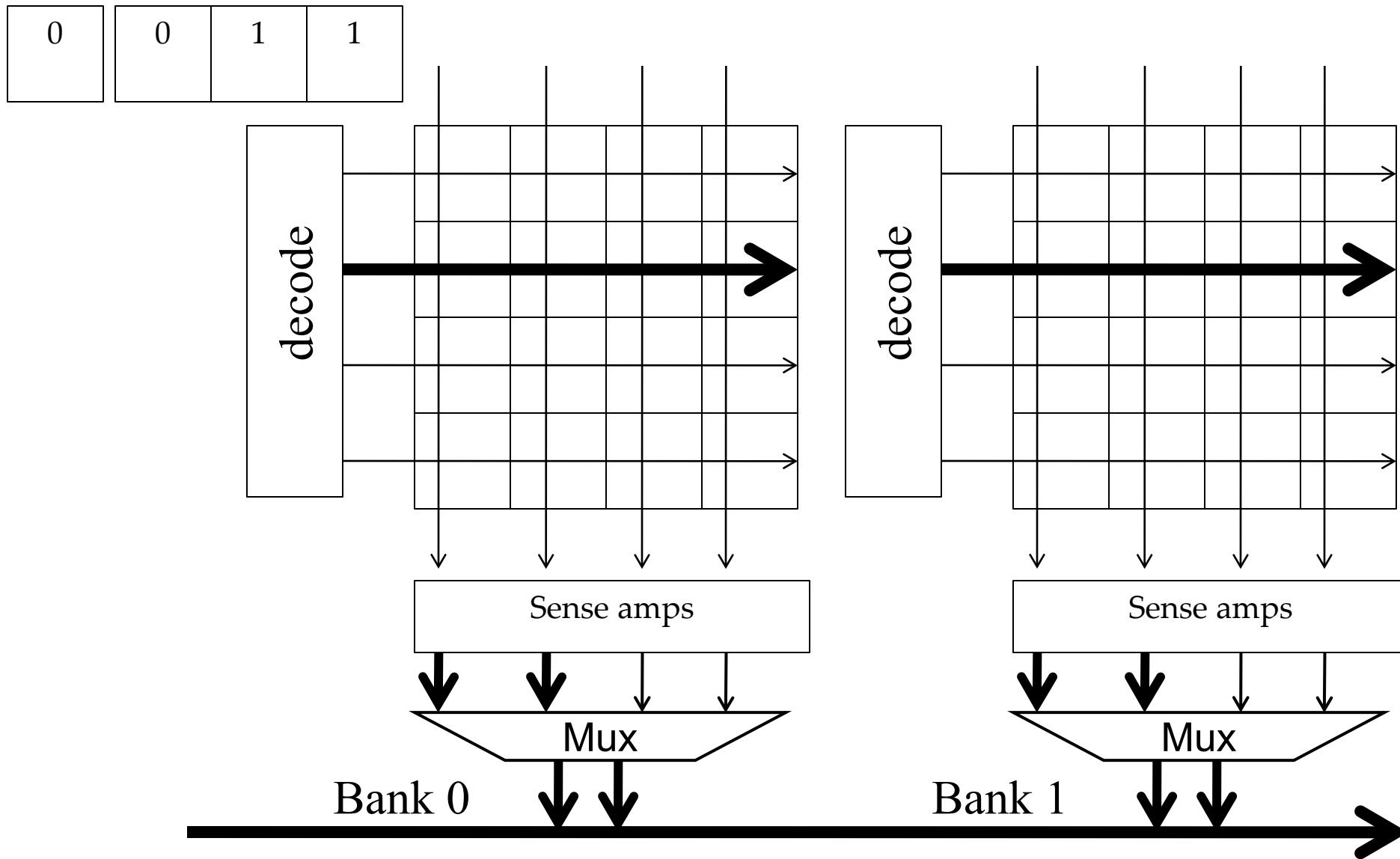
Second part of the burst



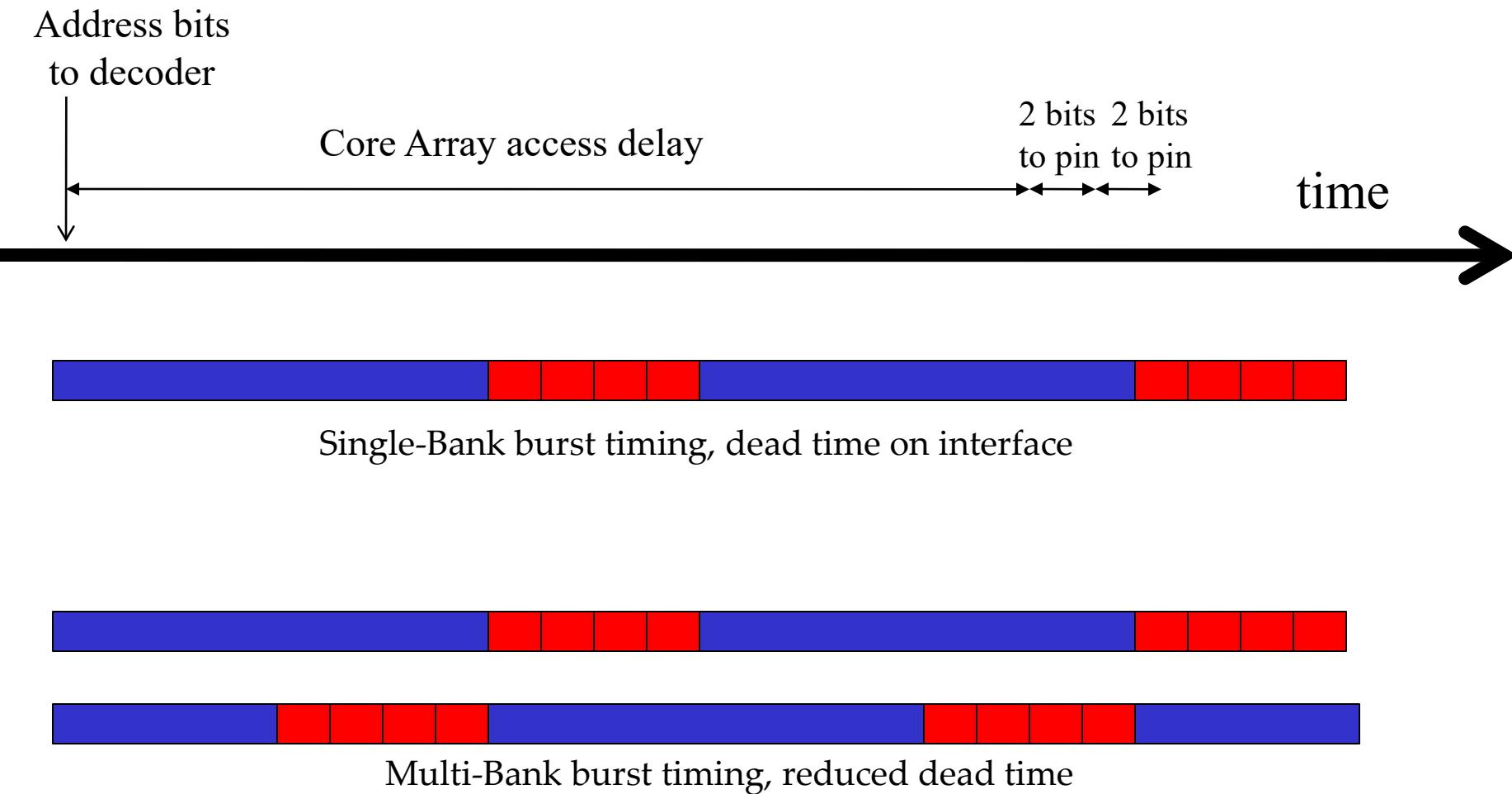
DRAM Bursting for the 8x2 Bank



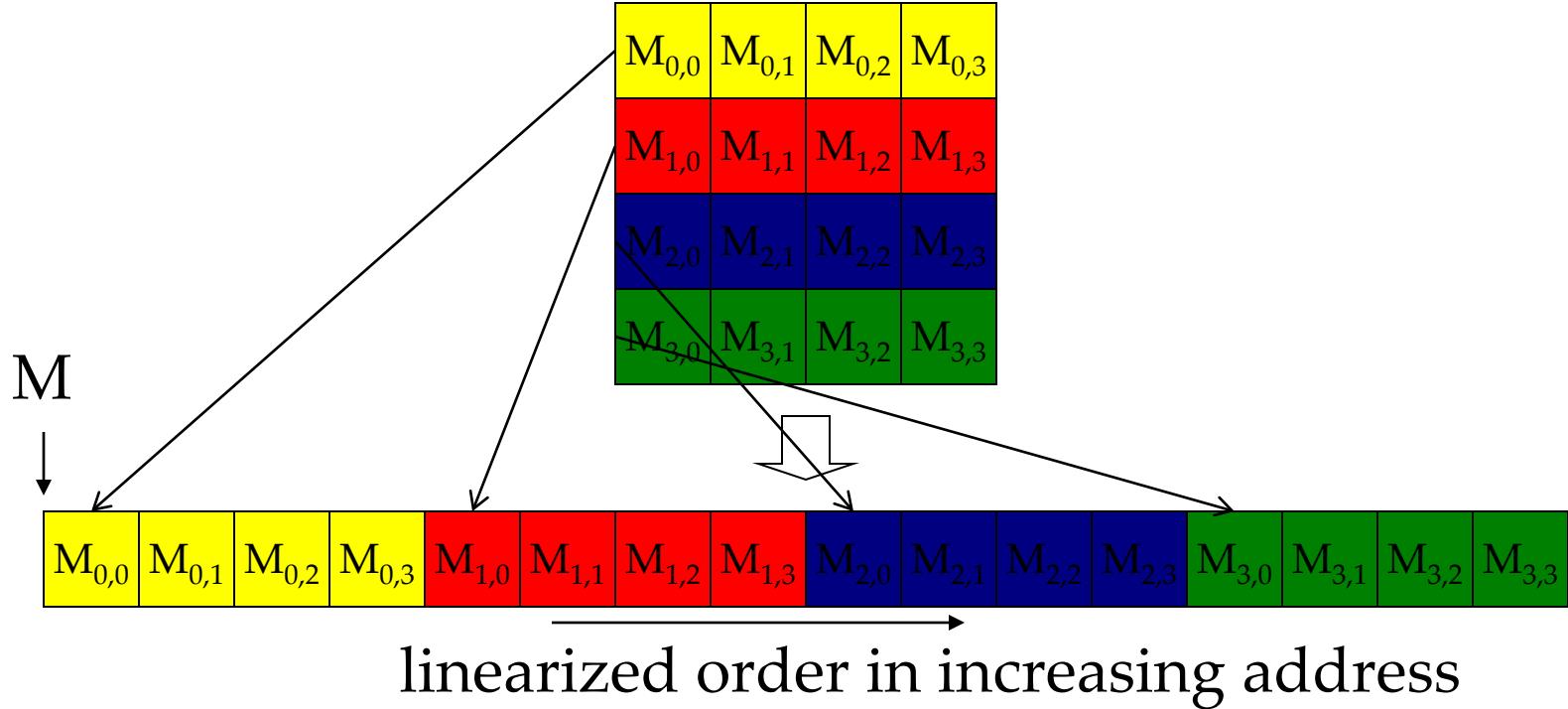
Multiple DRAM Banks



DRAM Bursting for the 8x2 Bank



Placing a 2D C array into linear memory space (review)



A Simple Matrix Multiplication Kernel (review)

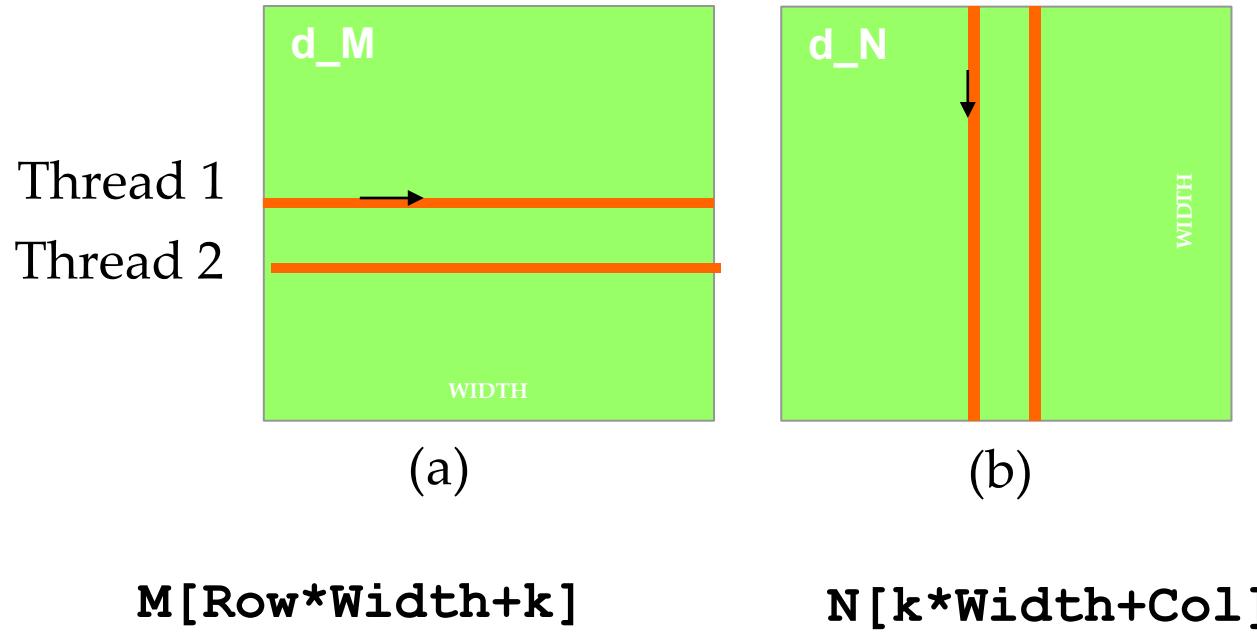
```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
    // Calculate the row index of the P element and M
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    // Calculate the column index of P and N
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;

        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k)
            Pvalue += M[Row*Width+k] * N[k*Width+Col];

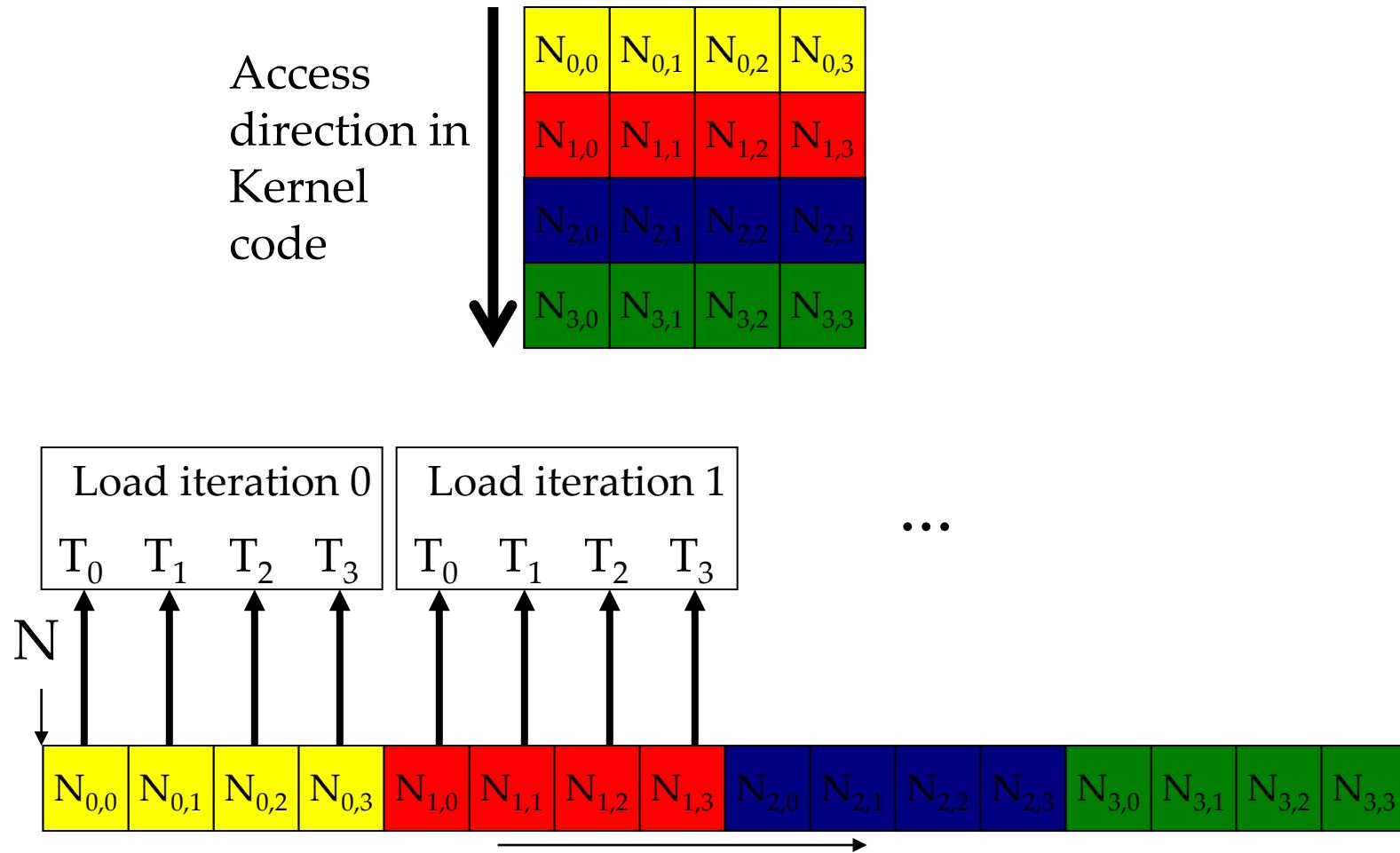
        P[Row*Width+Col] = Pvalue;
    }
}
```

Two Access Patterns

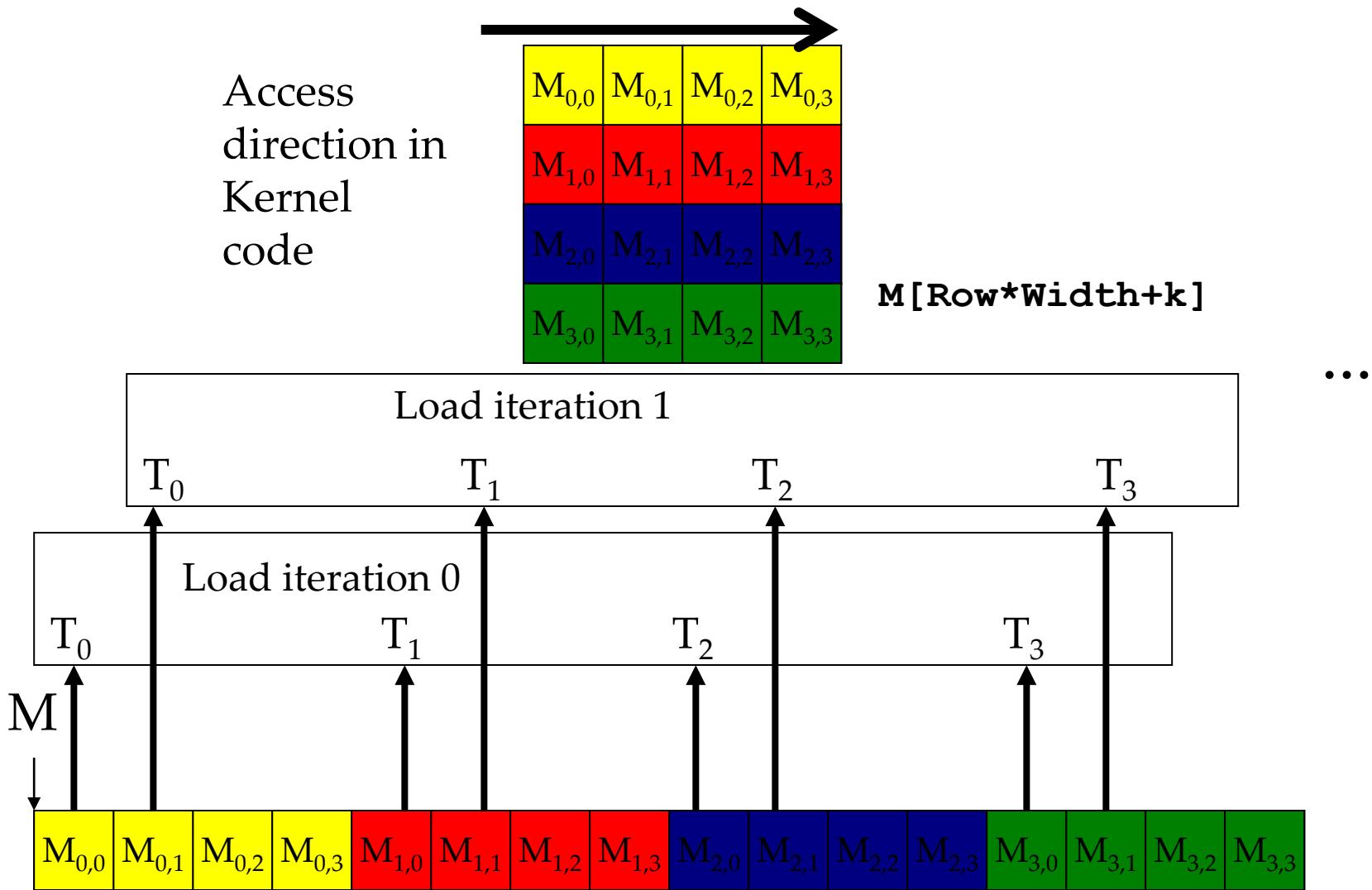


© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2018
ECE408/CS483/ University of Illinois at Urbana-Champaign

N accesses are coalesced

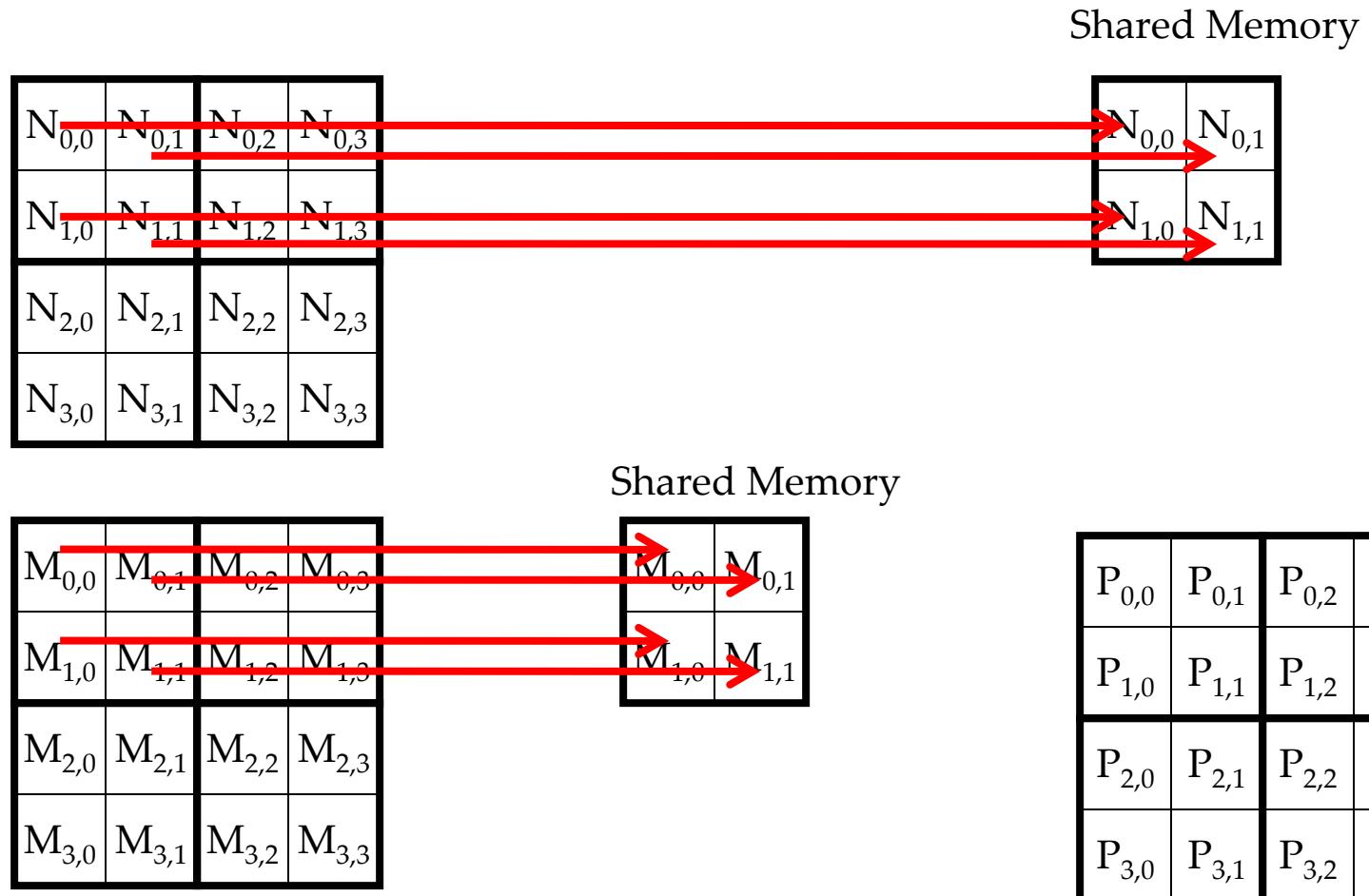


M accesses are not coalesced

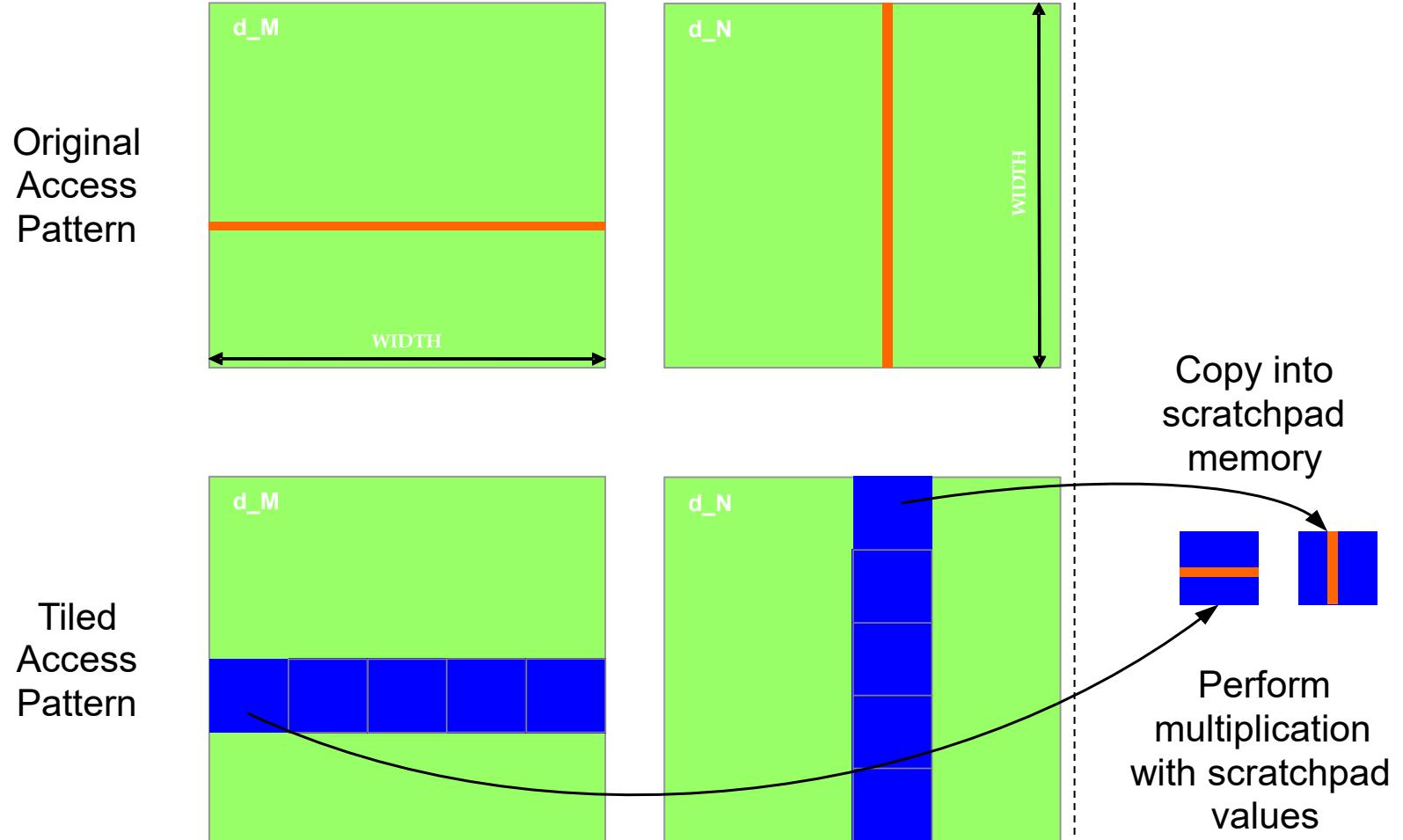


Work for Block (0,0)

Step 0



Use shared memory to enable coalescing in tiled matrix multiplication



Problem Solving

- Q: Which of the following two lines of a kernel code has better performance?

```
int tx = blockIdx.x * blockDim.x + threadIdx.x;
```

```
int ty = blockIdx.y * blockDim.y + threadIdx.y;
```

A) $B[ty * \text{Width} + tx] = 2 * A[ty * \text{Width} + tx];$

B) $B[tx * \text{Width} + ty] = 2 * A[tx * \text{Width} + ty];$

- A: Cannot tell. We do not know how the data is arranged in memory.

Problem Solving

- Q: Consider a DRAM system with a burst size of 512 bytes and a peak bandwidth of 240 GB/s. Assume a thread block size of 1024 and warp size of 32 and that A is a floating-point array in the global memory. What is the maximal memory data access throughput we can hope to achieve in the following access to A?

```
int i = blockIdx.x * blockDim.x + threadIdx.x;  
float temp = A[4*i] + A[4*i+1];
```

- A:
 - From a burst of 512 bytes, we will only use every other 8 bytes due to reading from $[4*i]$ and $[4*i+1]$ memory locations.
 - So, we can expect 120 GB/s.



**ANY MORE QUESTIONS?
READ CHAPTER 5**



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 5: Locality and Tiled Matrix Multiplication

Course Reminders

- Lab 1 is due this Friday
 - IMPORTANT: You need to add "--submit MP1" at the end of your RAI command to submit your final version of MP1 for grading.
 - When filling in the blank questions in Canvas quiz, please answer with the exact numbers. No expression is allowed.
 - You have single attempt on all questions. Your quiz results will only be available after the deadline.
- Lab 2 is out; it is due next Friday
- Lowest lab grade will be dropped from the final grade
 - Thus, no late submissions are allowed for labs!

Objective

- To learn to evaluate the performance implications of global memory accesses
- To prepare for MP3: tiled matrix multiplication
- To learn to assess the benefit of tiling

Kernel Invocation (Host-side Code)

```
// Setup the execution configuration
// BLOCK_WIDTH is a #define constant
dim3 dimGrid(ceil((1.0*Width)/BLOCK_WIDTH),
    ceil((1.0*Width)/BLOCK_WIDTH), 1);

dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

A Simple Matrix Multiplication Kernel

```
__global__
void MatrixMulKernel(float *d_M, float *d_N, float *d_P, int Width)
{
    // Calculate the row index of the d_P element and d_M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;

    // Calculate the column idenx of d_P and d_N
    int Col = blockIdx.x*blockDim.x+threadIdx.x;

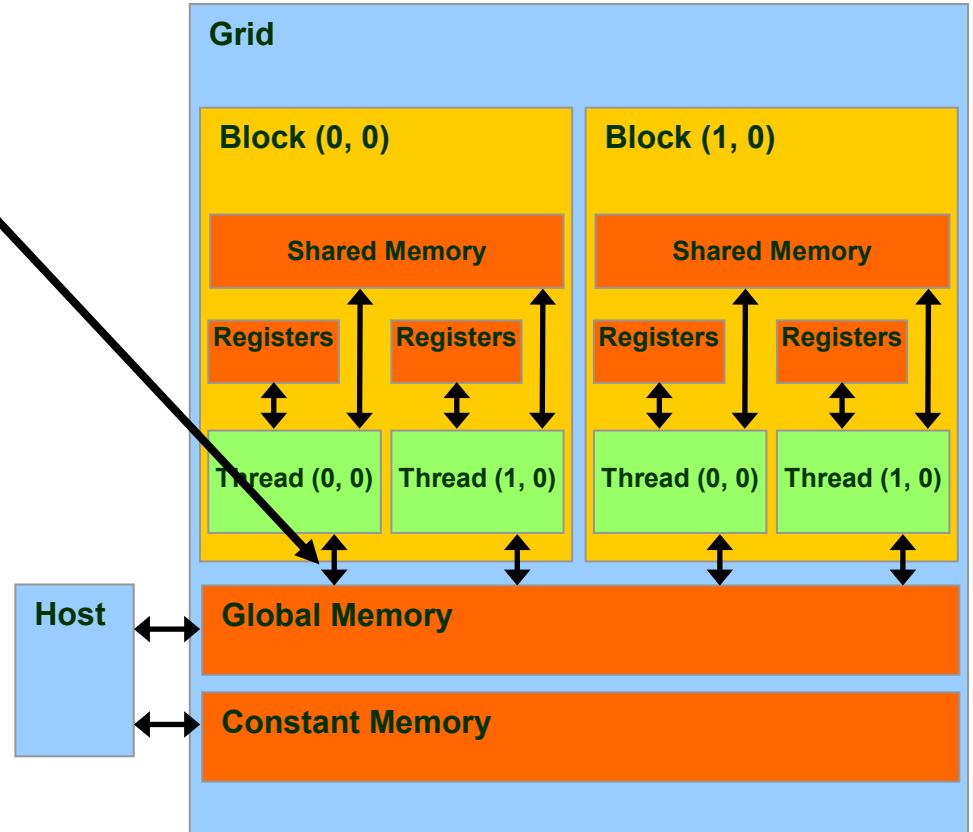
    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k)
            Pvalue += d_M[Row*Width+k] * d_N[k*Width+Col];
        d_P[Row*Width+Col] = Pvalue;
    }
}
```

Review: 4B of Data per FLOP

- Each threads access global memory
 - for elements of **M** and **N**:
 - 4B each**, or **8B per pair**.
 - (And once TOTAL to **P** per thread—ignore it.)
- With each pair of elements,
 - a thread does a single multiply-add,
 - 2 FLOP**—floating-point operations.
- So for every FLOP,
 - a thread needs** 4B from memory:
 - 4B / FLOP**.

How about performance on a device with 150 GB/s memory bandwidth?

- All threads access global memory for their input matrix elements
 - Two memory accesses (8 bytes) per floating point multiply-add (2 fp ops)
 - 4B/s of memory bandwidth/FLOPS
 - 150 GB/s limits the code at 37.5 GFLOPS
- The actual code runs at about 25 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak of more than 1,000 GFLOPS



A Common Programming Strategy

- Global memory is implemented with DRAM - slow
- To avoid Global Memory bottleneck, tile the input data to take advantage of Shared Memory:
 - Partition data into subsets (tiles) that fit into the (smaller but faster) shared memory
 - Handle each data subset with one thread block by:
 - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
 - Performing the computation on the subset from shared memory; each thread can efficiently access any data element
 - Copying results from shared memory to global memory
 - Tiles are also called blocks in the literature

A Common Programming Strategy

- In a GPU, **only threads in a block can** use **shared** memory.
- Thus, each **block** operates on **separate tiles**:
 - **Read tile(s)** into shared memory **using multiple threads** to exploit memory-level parallelism.
 - **Compute** based on shared memory tiles.
 - **Repeat**.
 - **Write results back to global memory**.

Declaring Shared Memory Arrays

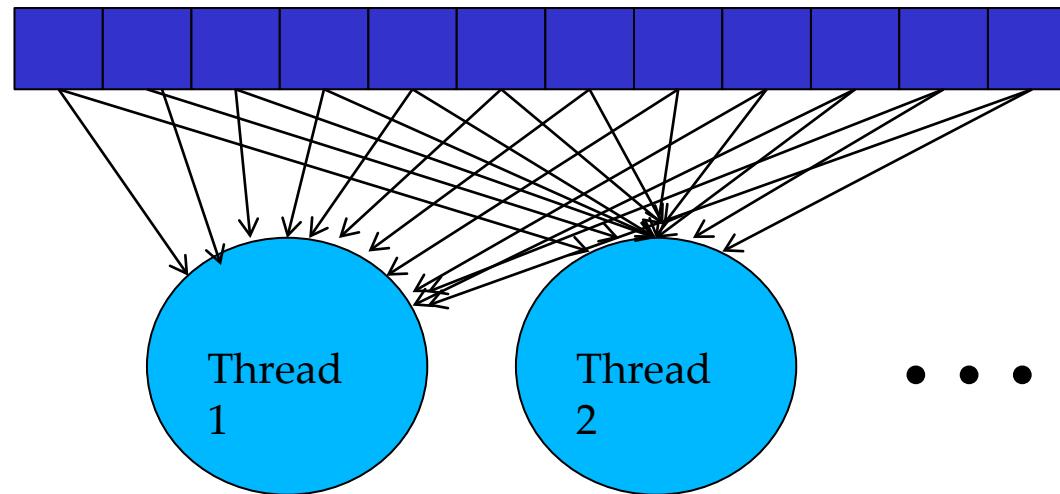
```
__global  
void MatrixMulKernel(float* M, float* N, float* P, int Width)  
{  
  
    __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];  
    __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];
```



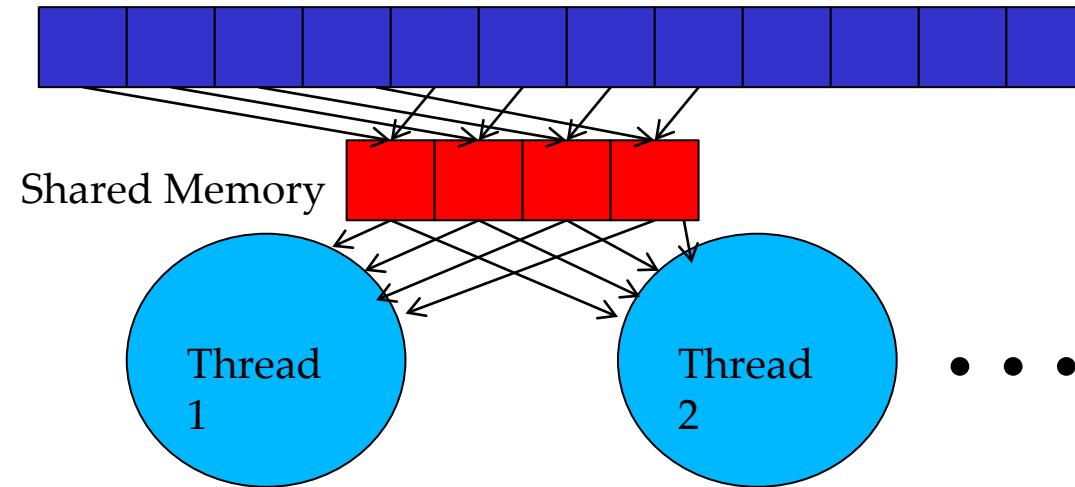
Common across all
threads in a block

Shared Memory Tiling Basic Idea

Data in Global Memory



Data in Global Memory

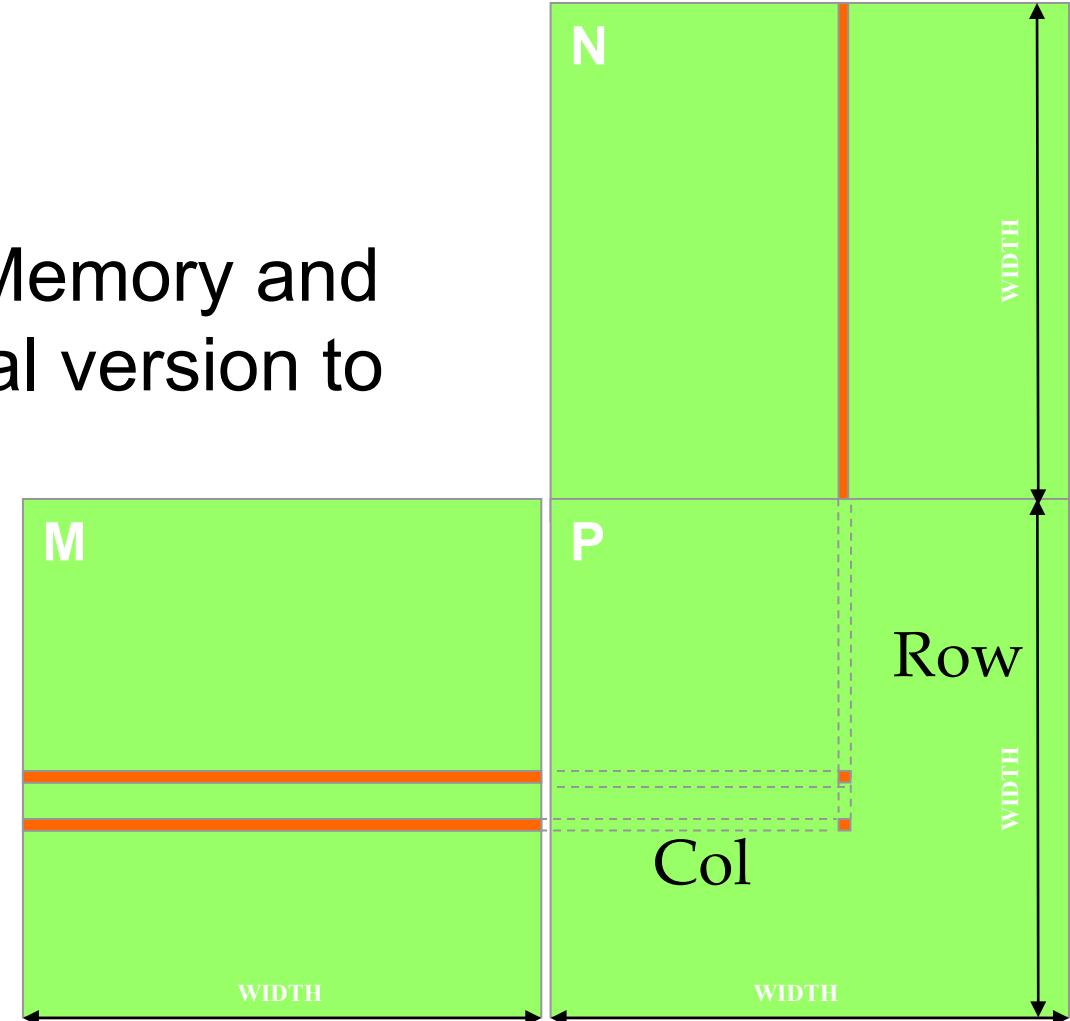


Outline of Technique

- Identify a tile of global data that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- Have the multiple threads to access their data from the on-chip memory
- Move on to the next block/tile

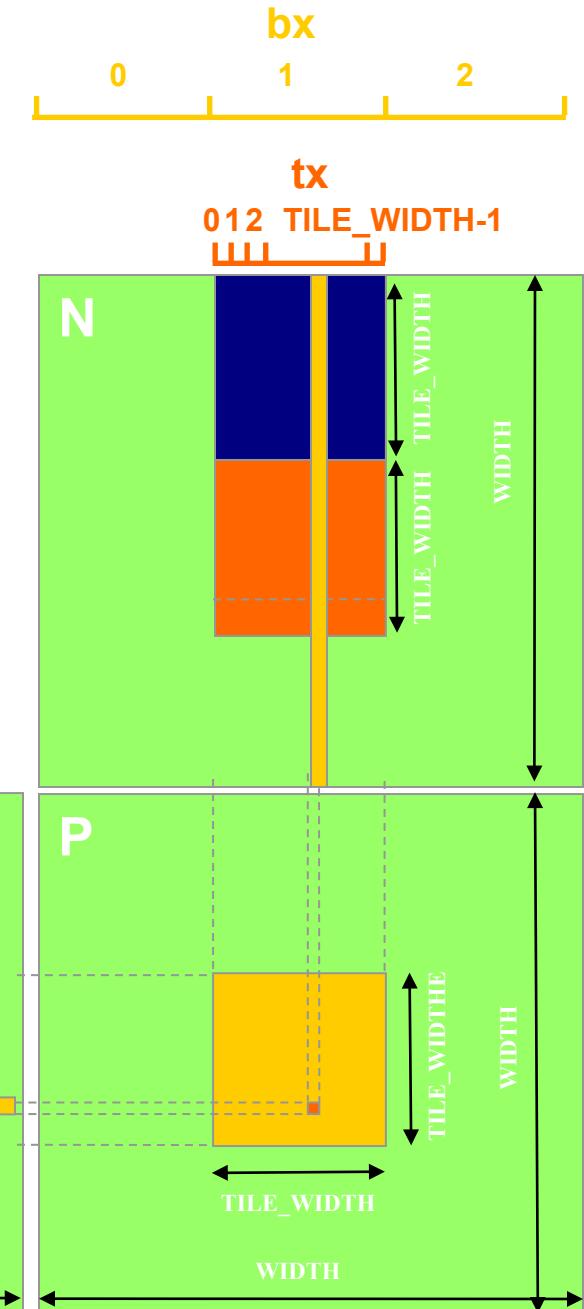
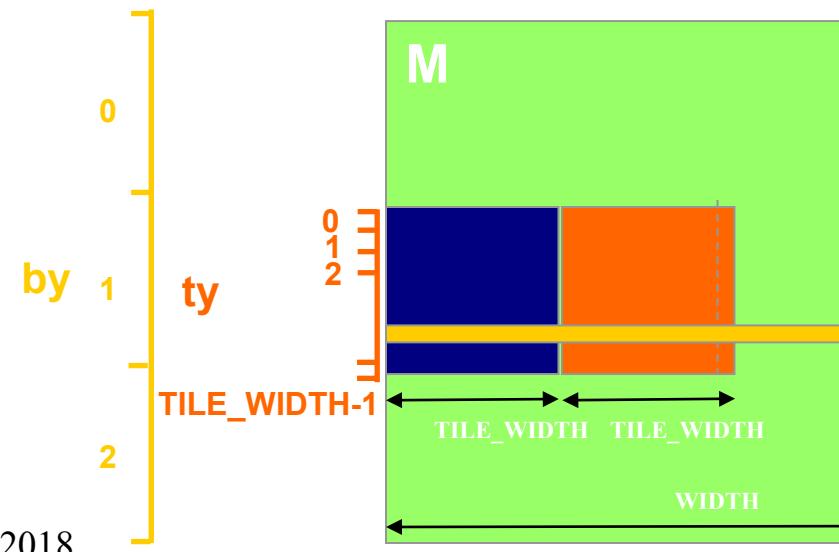
Use Shared Memory for data that will be reused

- Observe that each input element of M and N is used WIDTH times
- Load each element into Shared Memory and have several threads use the local version to reduce the memory bandwidth



Tiled Multiply

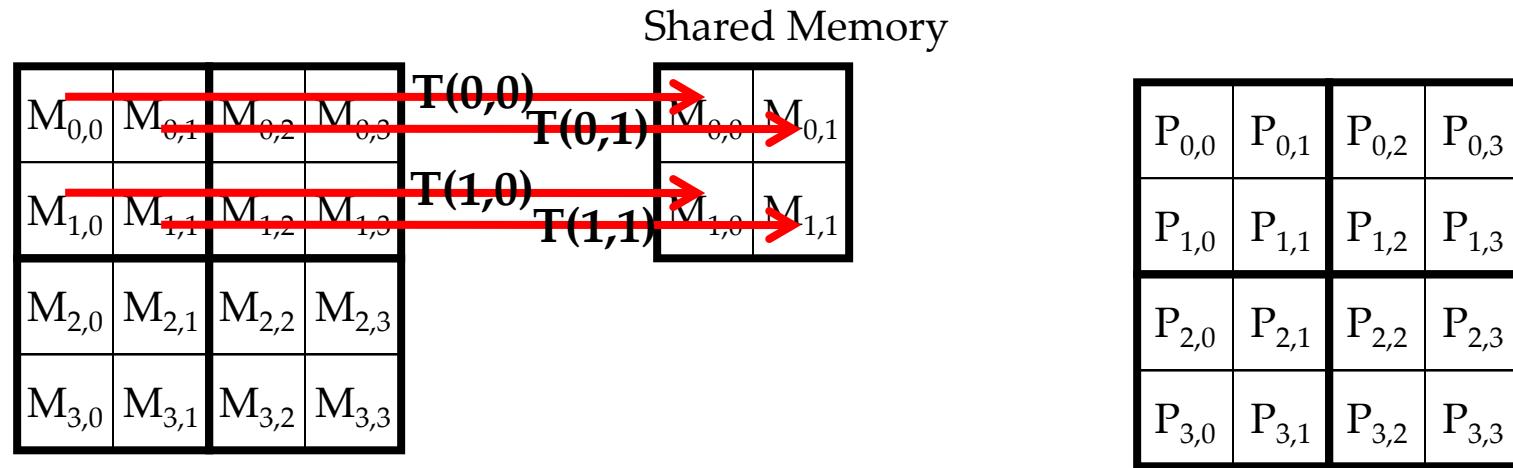
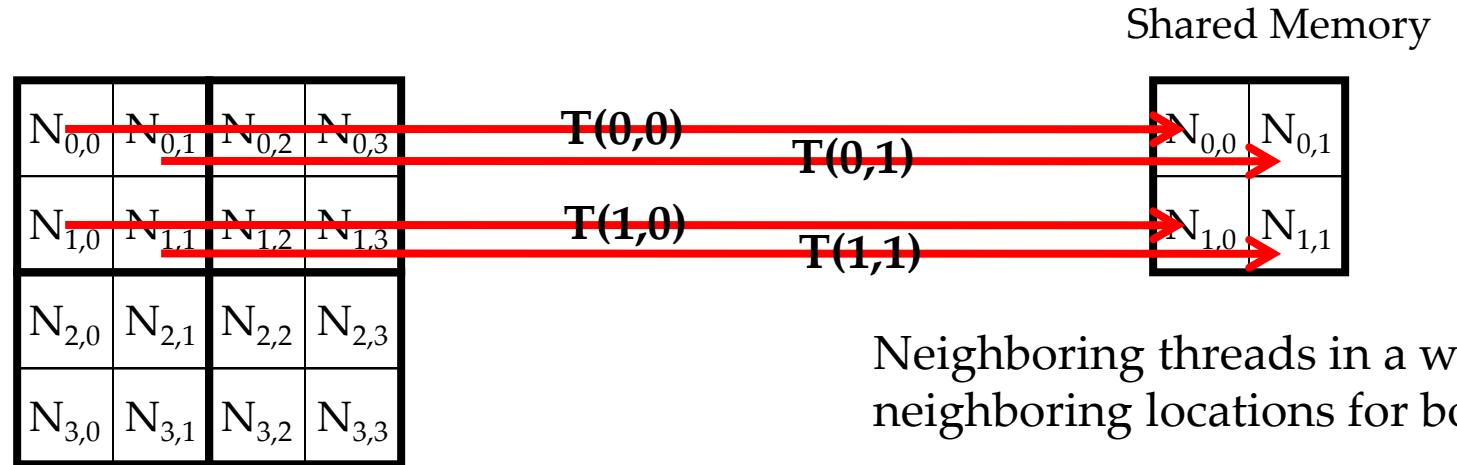
- Break up the execution of the kernel into phases so that the data accesses in each phase are focused on one tile of M and N
- For each tile:
 - Phase 1: Load tiles of M & N into share memory
 - Phase 2: Calculate partial dot product for tile of P



Loading a Tile

- All threads in a block participate
 - Each thread loads
 - one **M** element and
 - one **N** element
 - in basic tiling code.
- Assign the loaded element to each thread such that the accesses within each warp is coalesced (more later).

Loading Tiles for Block (0,0)



Work for Block (0,0)

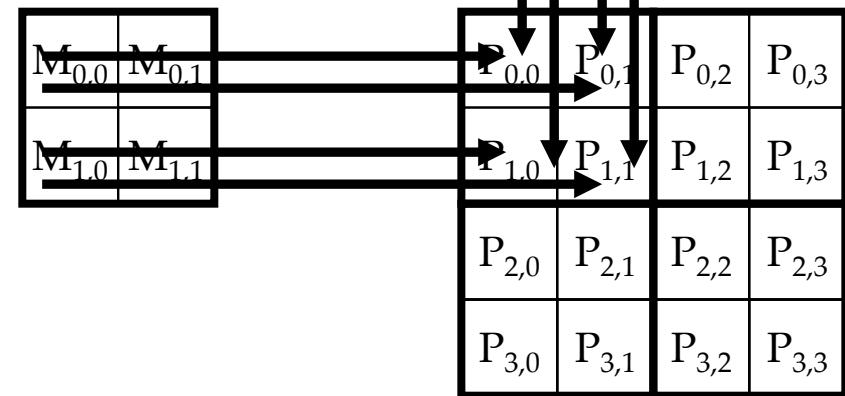
Step 1

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



Shared Memory

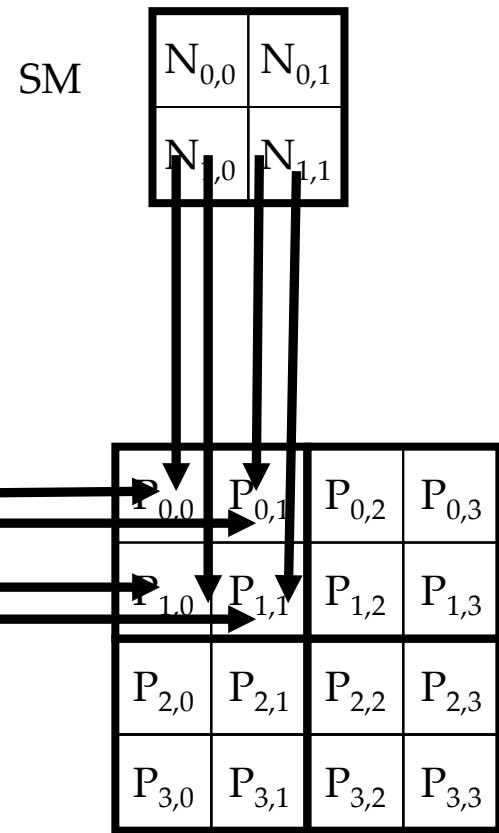


Work for Block (0,0)

Step 2

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



Work for Block (0,0)

Step 3

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

Shared Memory

$N_{2,0}$	$N_{2,1}$
$N_{3,0}$	$N_{3,1}$

Shared Memory

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

$M_{0,2}$	$M_{0,3}$
$M_{1,2}$	$M_{3,1}$

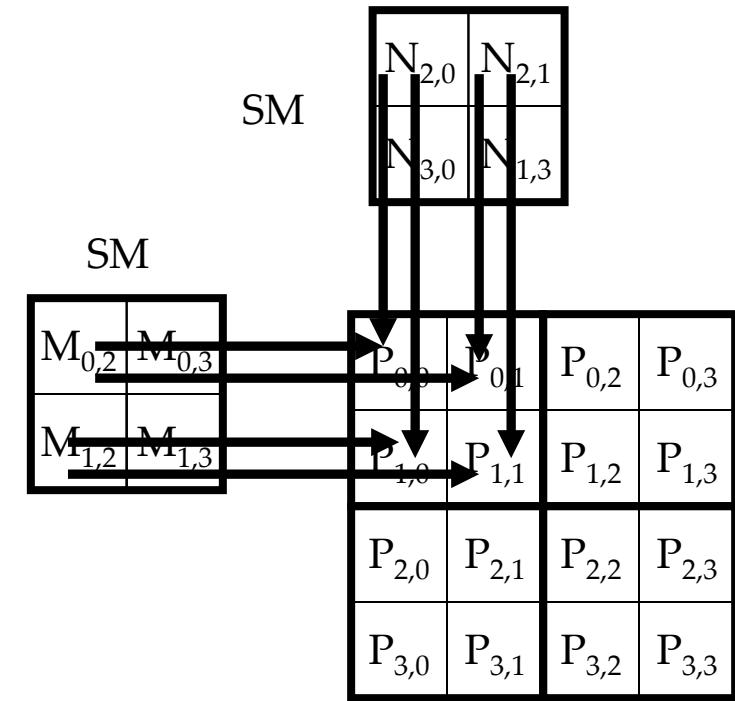
$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$

Work for Block (0,0)

Step 4

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

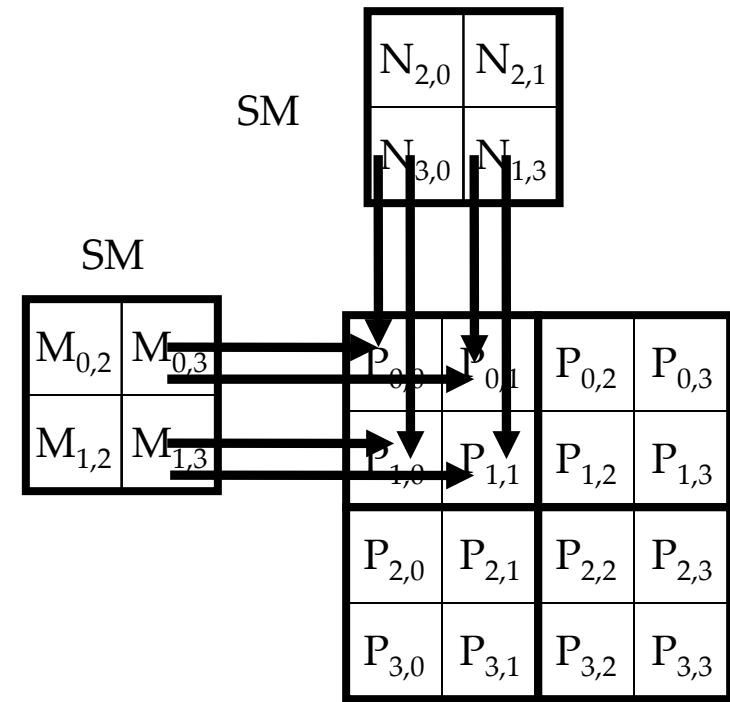


Work for Block (0,0)

Step 5

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



Phase 1: Loading a Tile

- All threads in a block participate
 - Each thread loads one M element and one N element in basic tiling code
- Assign the loaded element to each thread such that the accesses within each warp is coalesced (more later).

Loading an Input Tile 0

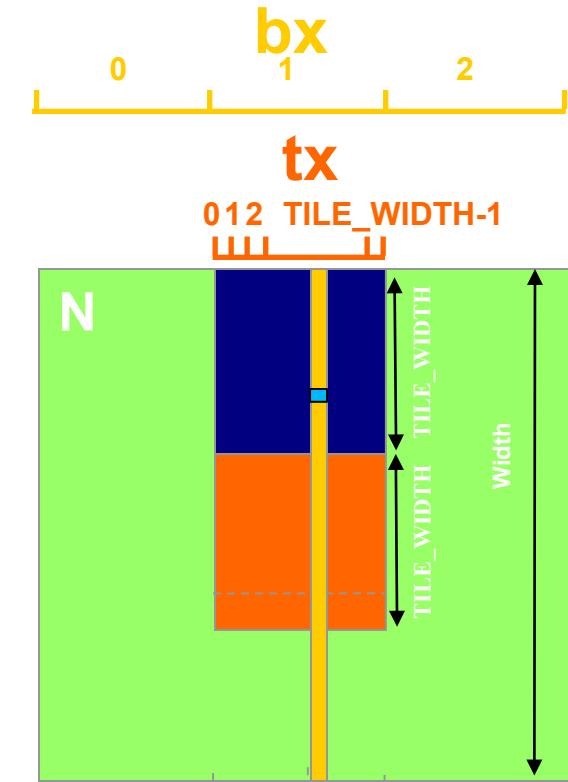
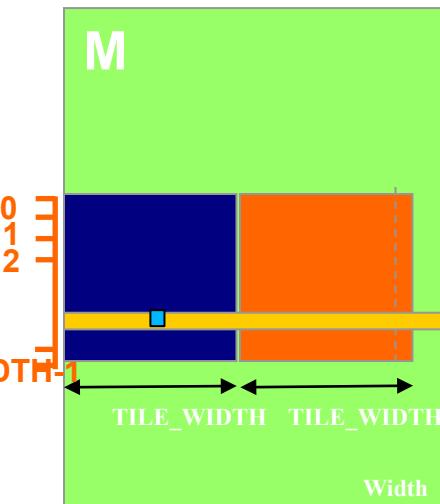
2D indexing for Tile 0

M [Row] [tx]

N [ty] [Col]

by
0
1
2
TILE_WIDTH-1

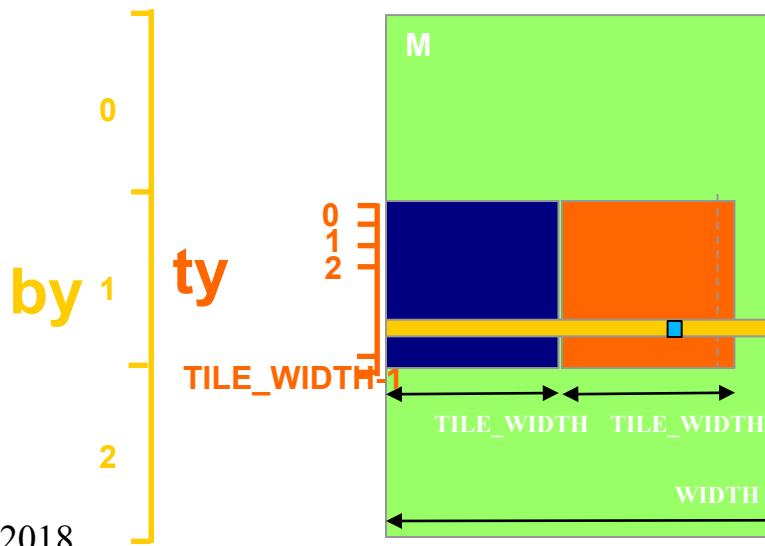
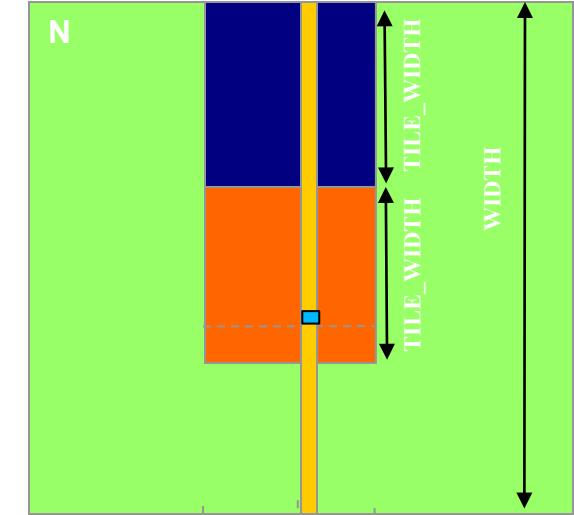
ty
0
1
2



Loading an Input Tile 1

Accessing tile 1 in 2D indexing:

$M[\text{Row}] [1 * \text{TILE_WIDTH} + \text{tx}]$
 $N[1 * \text{TILE_WIDTH} + \text{ty}] [\text{Col}]$



Loading an Input Tile q

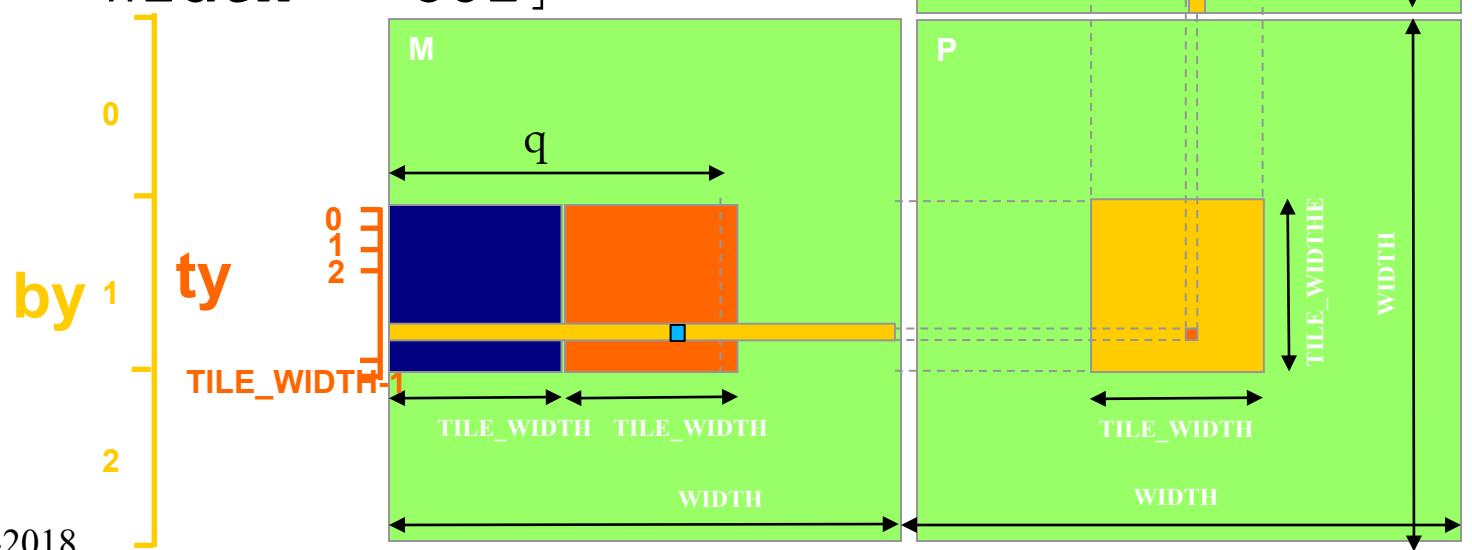
However, recall that M and N are dynamically allocated and can only use 1D indexing:

$M[\text{Row}] [m * \text{TILE_WIDTH} + tx]$

$M[\text{Row} * \text{Width} + q * \text{TILE_WIDTH} + tx]$

$N[q * \text{TILE_WIDTH} + ty] [\text{Col}]$

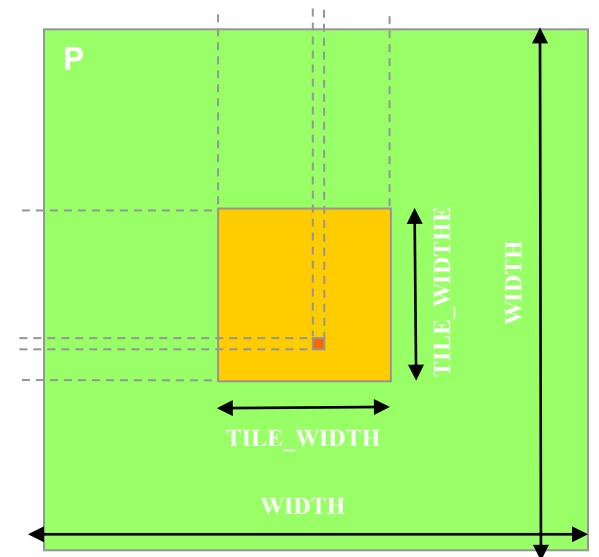
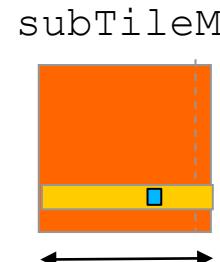
$N[(q * \text{TILE_WIDTH} + ty) * \text{Width} + \text{Col}]$



Phase 2: Compute partial product

To perform the k^{th} step of the product within the tile:

`subTileM[ty][k]`
`subTileN[k][tx]`



We're Not There Yet!

- But ...
- **How can a thread know ...**
 - **That another thread has finished** its part of the tile?
 - Or that another thread has finished using the previous tile?

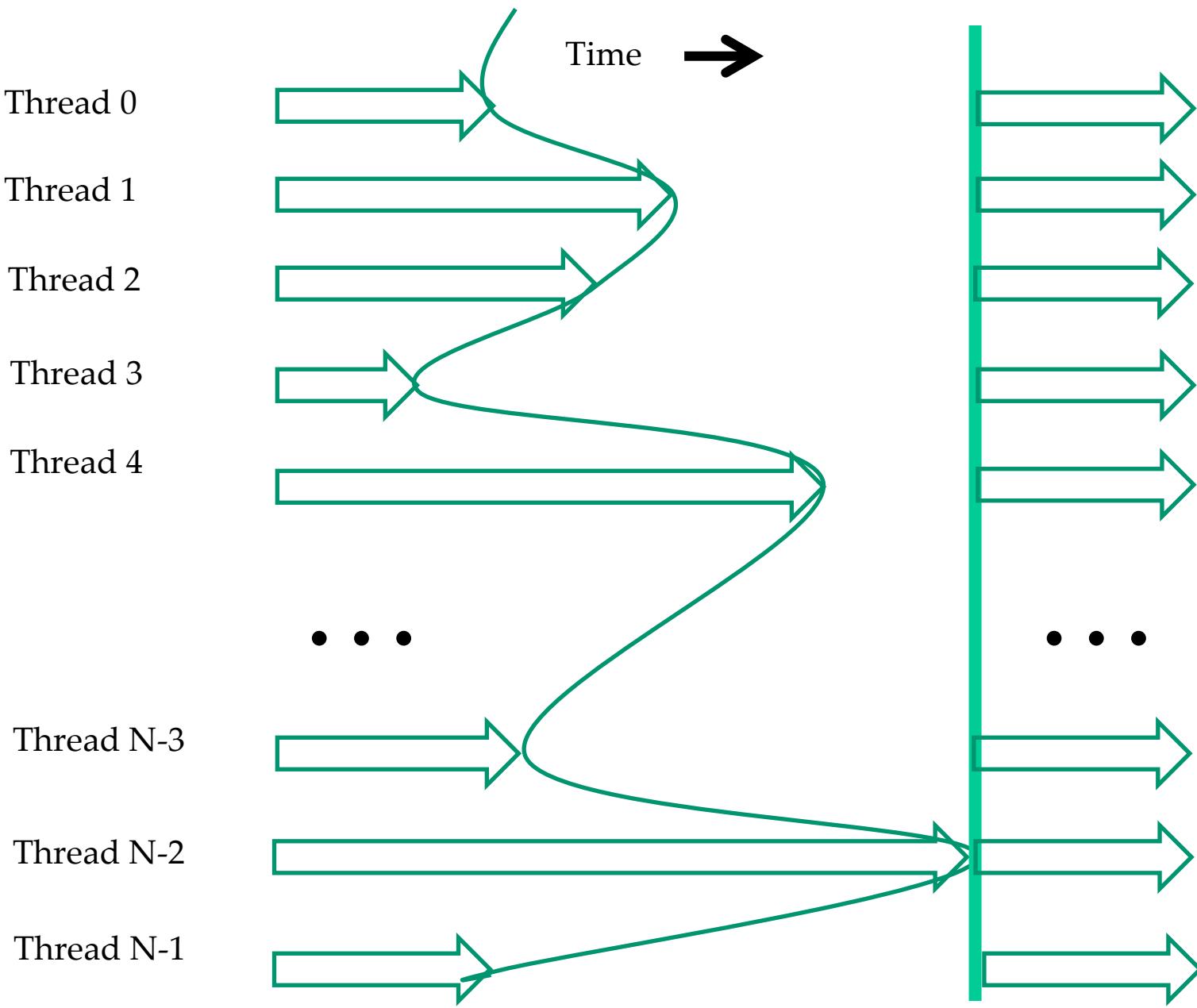
We need to synchronize!

Leveraging Parallel Strategies

- **Bulk synchronous execution:**
threads execute roughly in unison
 1. Do some work
 2. Wait for others to catch up
 3. Repeat
- **Much easier programming model**
 - Threads only parallel within a section
 - Debug lots of little programs
 - Instead of one large one.
- **Dominates high-performance applications**

Bulk Synchronous Steps Based on Barriers

- **How does it work?**
Use a barrier to wait for thread to ‘catch up.’
- A barrier is a synchronization point:
 - **each thread calls a function** to enter barrier;
 - **threads block** (sleep) in barrier function
until all threads have called;
 - **after last thread calls** function,
all threads continue past the barrier.



Barrier Synchronization

- An API function call in CUDA `__syncthreads()`
- All threads in the same block must reach the `__syncthreads()` before any can move on
- Can be used to coordinate tiled algorithms
 - To ensure that all elements of a tile are loaded
 - To ensure that certain computation on elements is complete

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
    1. __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
    2. __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];

    3. int bx = blockIdx.x; int by = blockIdx.y;
    4. int tx = threadIdx.x; int ty = threadIdx.y;

        // Identify the row and column of the P element to work on
    5. int Row = by * TILE_WIDTH + ty;
    6. int Col = bx * TILE_WIDTH + tx;
    7. float Pvalue = 0;

        // Loop over the M and N tiles required to compute the P element
        // The code assumes that the Width is a multiple of TILE_WIDTH!
    8. for (int q = 0; q < Width/TILE_WIDTH; ++q) {
            // Collaborative loading of M and N tiles into shared memory
        9. subTileM[ty][tx] = M[Row*Width + q*TILE_WIDTH+tx];
        10. subTileN[ty][tx] = N[(q*TILE_WIDTH+ty)*Width+Col];
        11. __syncthreads();
        12. for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += subTileM[ty][k] * subTileN[k][tx];
        13. __syncthreads();
    14. }
    15. }
    16. P[Row*Width+Col] = Pvalue;
}
```

Compare with Basic MM Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
    // Calculate the row index of the P element and M
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    // Calculate the column index of P and N
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;

        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k)
            Pvalue += M[Row*Width+k] * N[k*Width+Col];

        P[Row*Width+Col] = Pvalue;
    }
}
```

Use of Large Tiles Shifts Bottleneck

- Recall our example GPU: **1,000 GFLOP/s, 150 GB/s**
- **16x16 tiles** use each operand for 16 operations
 - **reduce global memory accesses by** a factor of **16**
 - **150GB/s** bandwidth supports
 $(150/4)*16 = \mathbf{600\ GFLOPS!}$
- **32x32 tiles** use each operand for 32 operations
 - **reduce global memory accesses by** a factor of **32**
 - **150 GB/s** bandwidth supports
 $(150/4)*32 = \mathbf{1,200\ GFLOPS!}$
 - **Memory bandwidth is no longer the bottleneck!**

Also Need Parallel Accesses to Memory

- Shared memory size
 - implementation dependent
 - **64kB** per SM in Maxwell (48kB max per block)
- Given **TILE_WIDTH of 16** (256 threads / block),
 - each thread block uses $2 \times 256 \times 4B = 2kB$ of shared memory,
 - which limits active blocks to 32;
 - max. of 2048 threads per SM,
 - which limits blocks to 8.
 - Thus up to $8 \times 512 = \mathbf{4,096 \text{ pending loads}}$
(2 per thread, 256 threads per block)

Another Good Choice: 32x32 Tiles

- Given **TILE_WIDTH of 32** (1,024 threads / block),
 - each thread block uses
 $2*1024*4B = 8kB$ of shared memory,
 - which limits active blocks to 8;
 - max. of 2,048 threads per SM,
 - which limits blocks to 2.
 - Thus up to $2*2,048 = \mathbf{4,096 pending loads}$
(2 per thread, 1,024 threads per block)

(same memory parallelism exposed)

Current GPU? Use Device Query

- Number of devices in the system

```
int dev_count;  
cudaGetDeviceCount( &dev_count);
```

- Capability of devices

```
cudaDeviceProp dev_prop;  
for (i = 0; i < dev_count; i++) {  
    cudaGetDeviceProperties( &dev_prop, i);  
  
    // decide if device has sufficient resources and capabilities  
}
```

- `cudaDeviceProp` is a built-in C structure type

- `dev_prop.dev_prop.maxThreadsPerBlock`
- `dev_prop.sharedMemoryPerBlock`
- ...



**ANY MORE QUESTIONS?
READ CHAPTER 4!**



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 4: Memory Model

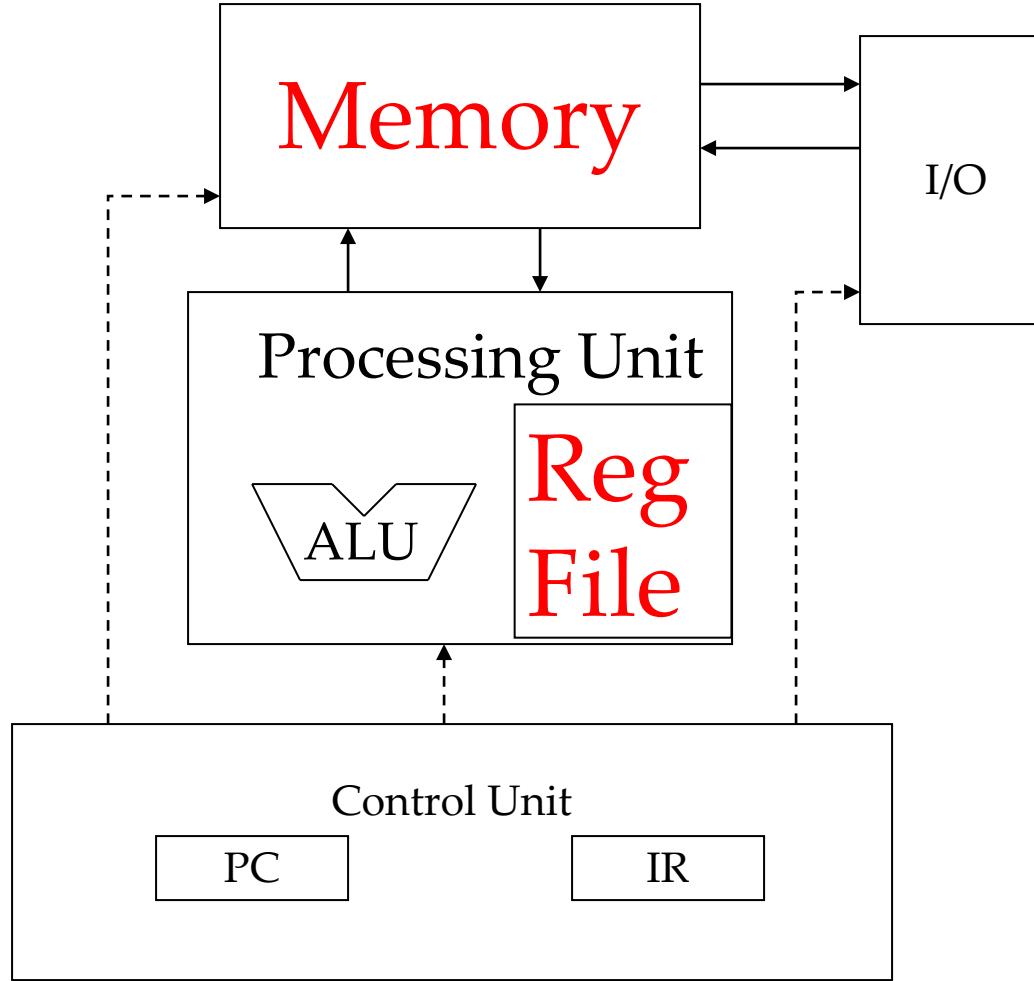
Course Reminders

- Lab 0 submission deadline is coming up
 - Make sure to run the code first **AND** only then answer the questions on Canvas
- Lab 1 is out, it is due next week on Friday
 - There is an additional step in submitting the code for this and all following labs, you must use the “`--submit MP1`” flag to submit the code for grading
- The course staff only replies to questions posted on Campuswire
- Check out office hours schedule, there are many options

Objective

- To learn the basic features of the memories accessible by CUDA threads
- To prepare for MP-2 - basic matrix multiplication
- To learn to evaluate the performance implications of global memory accesses

The Von-Neumann Model



Instructions are Stored in Memory

- Every instruction needs to be fetched from memory, decoded, then executed.
- Instruction processing breaks into steps:

Fetch | Decode | Execute | Memory

- Instructions come in three flavors:
Operate, Data Transfer, and Control Flow.

Example: Processing an Add Instruction

- Example of an (LC-3) operate instruction:
ADD R1, R2, R3
- meaning:
 - read R2 and R3
 - add them as unsigned/2's complement
 - write sum to R1
- Instruction processing for an operate instruction:
Fetch | Decode | Execute | Memory

Example: Processing a Load Instruction

- Example of an (LC-3) data transfer instruction:

LDR R4, R6, #3 ; a load

- meaning:

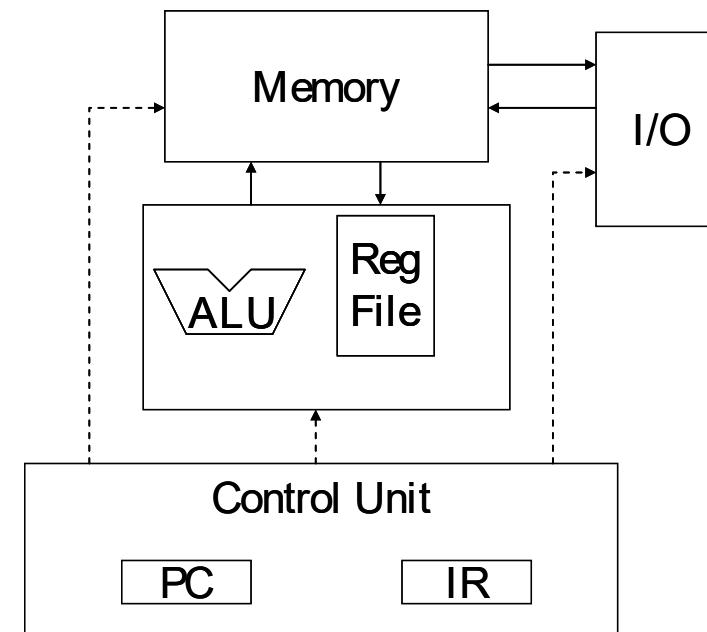
- read R6
 - add the number 3 to it
 - load the contents of memory at the resulting address
 - write the bits to R4

- Instruction processing for a load instruction:

Fetch | Decode | Execute | Memory

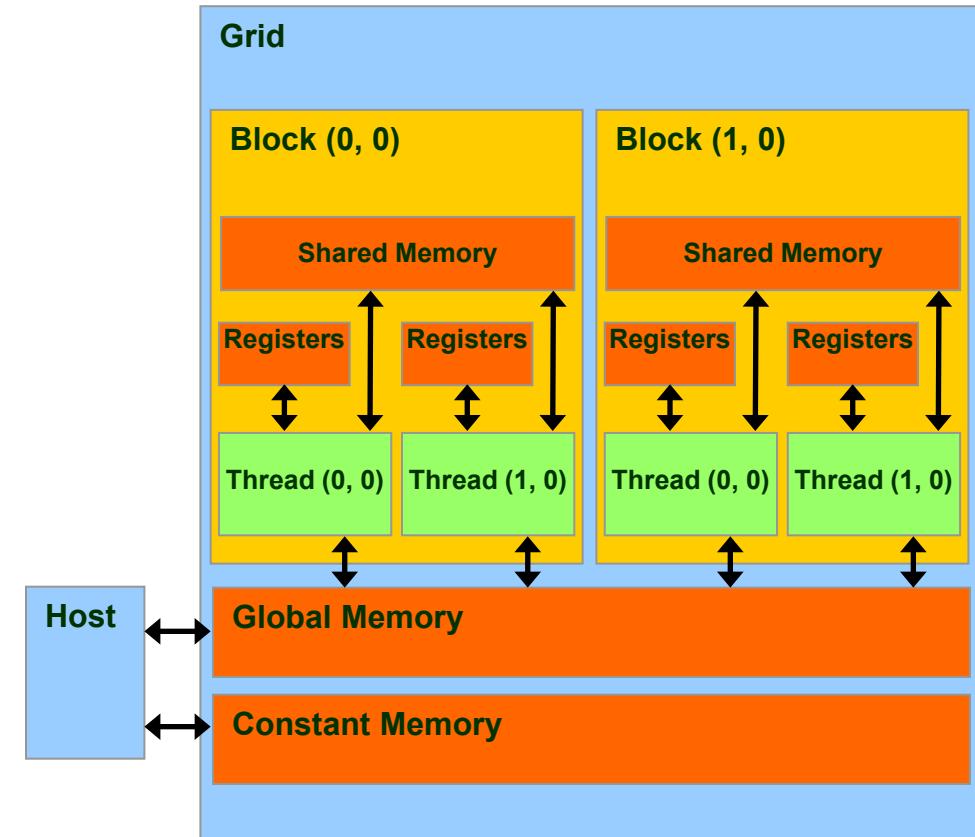
Registers vs Memory

- Registers
 - Fast: 1 cycle; no memory access required
 - Few: hundreds for CPU, $O(10k)$ for GPU SM
- Memory
 - Slow: hundreds of cycles
 - Huge: GB or more



Programmer View of CUDA Memories

- Each thread can:
 - read/write per-thread **registers** (**~1 cycle**)
 - read/write per-block **shared memory** (**~5 cycles**)
 - read/write per-grid **global memory** (**~500 cycles**)
 - read/only per-grid **constant memory** (**~5 cycles with caching**)



CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
<code>int LocalVar;</code>	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	app.	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	app.	application

- **device**
 - optional with **shared** or **constant**
 - not allowed by itself within functions
- **Automatic variables with no qualifiers**
 - in **registers** for primitive types and structures
 - in **global memory** for per-thread arrays

Next Application: Matrix Multiplication

- Given two $\text{Width} \times \text{Width}$ matrices, M and N,
 - we can multiply M by N
 - to compute a third $\text{Width} \times \text{Width}$ matrix, P:
 - $P = MN$

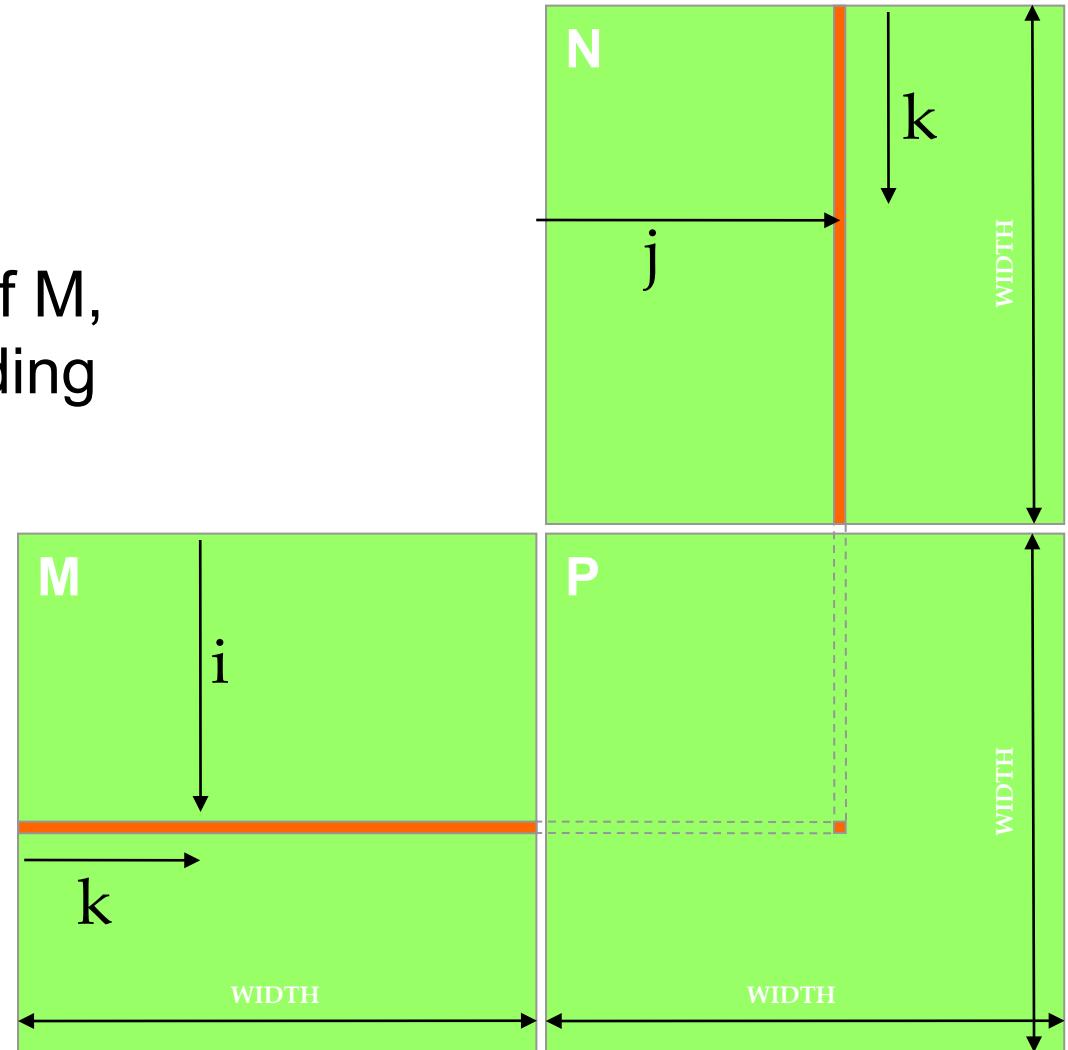
In terms of the elements of P, matrix multiplication implies computing...

$$P_{ij} = \sum_{k=1}^{\text{Width}} M_{ik} N_{kj}$$

Matrix Multiplication

$$P_{ij} = \sum_{k=1}^{Width} M_{ik} N_{kj}$$

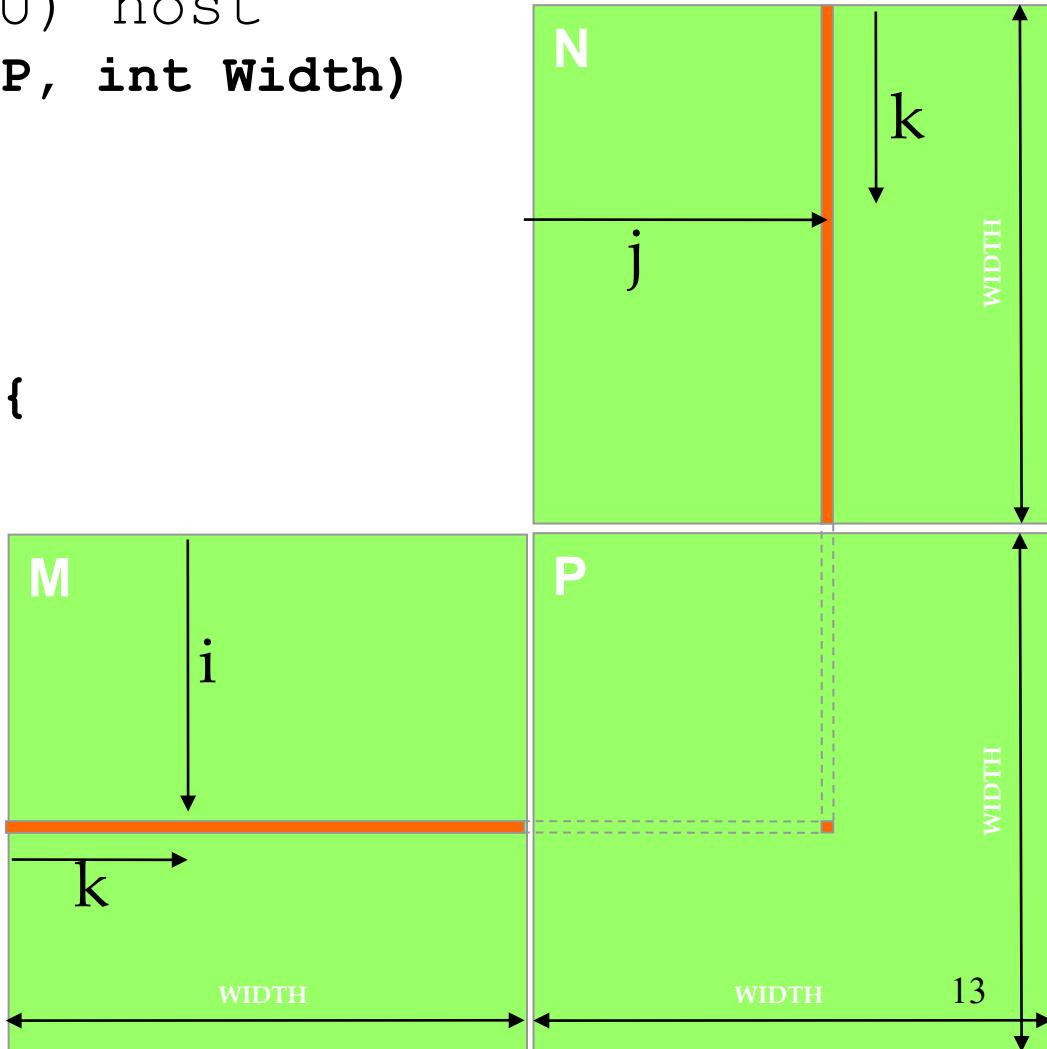
- Graphically, imagine
 - taking each element in a row of M,
 - multiplying it by the corresponding element in a column of N, and
 - summing up the products.
- Do that for every row and every column to produce P.



Matrix Multiplication Example

A Simple Host Version in C

```
// Matrix multiplication on the (CPU) host
void MatrixMul(float *M, float *N, float *P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            float sum = 0;
            for (int k = 0; k < Width; ++k) {
                float a = M[i * Width + k];
                float b = N[k * Width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



Parallelize Elements of P

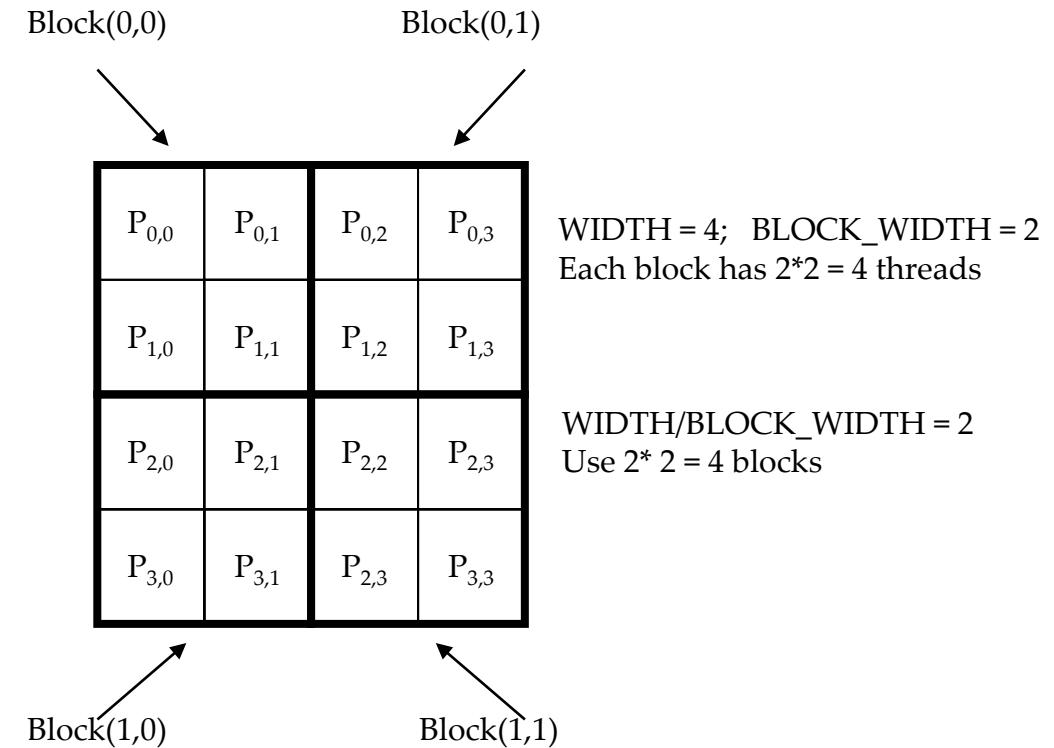
- What can we parallelize?
 - start with the **two outer loops**
 - parallelize **computation of elements of P**
- What about the inner loop?
 - Technically, floating-point is NOT associative.
 - The **parallel sum** is called a **reduction**—we'll come back to it in a few weeks.
 - For now, **use a single thread for each P_{ij} .**

Compute Using 2D Blocks in a 2D Grid

- **P** is 2D, so organize threads in 2D as well:
 - Split the output **P** into square **tiles**
 - **of size $\text{TILE_WIDTH} \times \text{TILE_WIDTH}$**
 - (a preprocessor constant).
 - **Each thread block produces one tile** of TILE_WIDTH^2 elements.
 - Create **$[\text{ceil}(\text{Width} / \text{TILE_WIDTH})]^2$** thread **blocks** to cover the output matrix.

Kernel Function - A Small Example

- Have each 2D thread block to compute a $(\text{BLOCK_WIDTH})^2$ sub-matrix of the result matrix
 - Each block has $(\text{BLOCK_WIDTH})^2$ threads
- Generate a 2D Grid of $(\text{WIDTH}/\text{BLOCK_WIDTH})^2$ blocks
- This concept is called **tiling**. Each block represents a **tile**.



Example: Width 8, TILE_WIDTH 2

P _{0,0}	P _{0,1}	P _{0,2}	P _{0,3}	P _{0,4}	P _{0,5}	P _{0,6}	P _{0,7}
P _{1,0}	P _{1,1}	P _{1,2}	P _{1,3}	P _{1,4}	P _{1,5}	P _{1,6}	P _{1,7}
P _{2,0}	P _{2,1}	P _{2,2}	P _{2,3}	P _{2,4}	P _{2,5}	P _{2,6}	P _{2,7}
P _{3,0}	P _{3,1}	P _{3,2}	P _{3,3}	P _{3,4}	P _{3,5}	P _{3,6}	P _{3,7}
P _{4,0}	P _{4,1}	P _{4,2}	P _{4,3}	P _{4,4}	P _{4,5}	P _{4,6}	P _{4,7}
P _{5,0}	P _{5,1}	P _{5,2}	P _{5,3}	P _{5,4}	P _{5,5}	P _{5,6}	P _{5,7}
P _{6,0}	P _{6,1}	P _{6,2}	P _{6,3}	P _{6,4}	P _{6,5}	P _{6,6}	P _{6,7}
P _{7,0}	P _{7,1}	P _{7,2}	P _{7,3}	P _{7,4}	P _{7,5}	P _{7,6}	P _{7,7}

Each block has
 $2 \times 2 = 4$ threads.

WIDTH/TILE_WIDTH = 4
Use $4 \times 4 = 16$ blocks.

Example: Same Matrix, Larger Tiles (Width 8, TILE_WIDTH 4)

P _{0,0}	P _{0,1}	P _{0,2}	P _{0,3}	P _{0,4}	P _{0,5}	P _{0,6}	P _{0,7}
P _{1,0}	P _{1,1}	P _{1,2}	P _{1,3}	P _{1,4}	P _{1,5}	P _{1,6}	P _{1,7}
P _{2,0}	P _{2,1}	P _{2,2}	P _{2,3}	P _{2,4}	P _{2,5}	P _{2,6}	P _{2,7}
P _{3,0}	P _{3,1}	P _{3,2}	P _{3,3}	P _{3,4}	P _{3,5}	P _{3,6}	P _{3,7}
P _{4,0}	P _{4,1}	P _{4,2}	P _{4,3}	P _{4,4}	P _{4,5}	P _{4,6}	P _{4,7}
P _{5,0}	P _{5,1}	P _{5,2}	P _{5,3}	P _{5,4}	P _{5,5}	P _{5,6}	P _{5,7}
P _{6,0}	P _{6,1}	P _{6,2}	P _{6,3}	P _{6,4}	P _{6,5}	P _{6,6}	P _{6,7}
P _{7,0}	P _{7,1}	P _{7,2}	P _{7,3}	P _{7,4}	P _{7,5}	P _{7,6}	P _{7,7}

Each block has
 $4 \times 4 = 16$ threads.

$\text{WIDTH}/\text{TILE_WIDTH} = 2$
Use $2 \times 2 = 4$ blocks.

Kernel Invocation (Host-side Code)

```
// TILE_WIDTH is a #define constant  
dim3 dimGrid(ceil((1.0*Width)/TILE_WIDTH),  
              ceil((1.0*Width)/TILE_WIDTH), 1);  
  
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);  
  
// Launch the device computation threads!  
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

Kernel Function

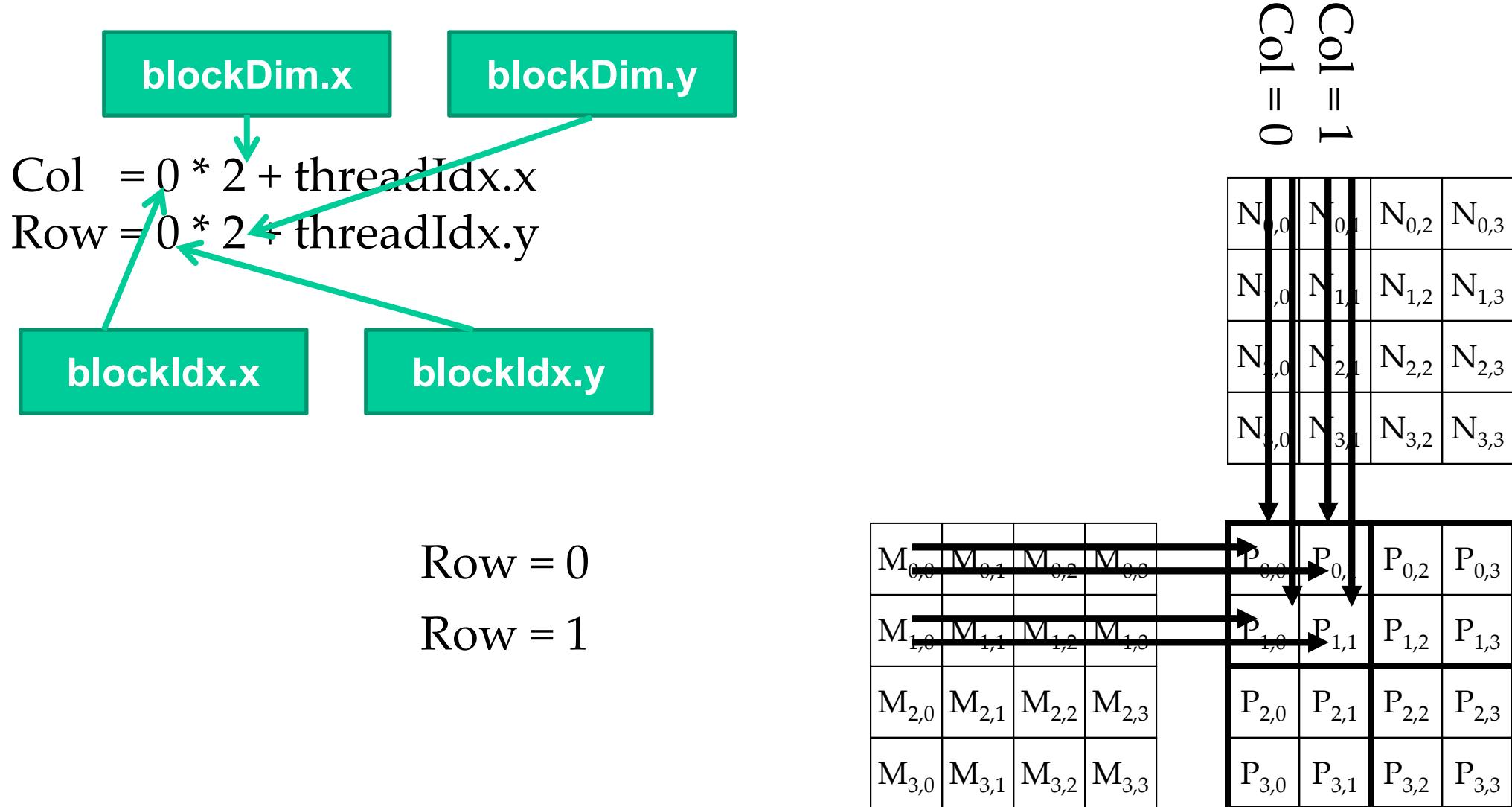
```
// Matrix multiplication kernel - per thread code

__global__
void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
    ...
    ...
    d_P[  ] = Pvalue;
```

Row & Col for each thread

```
// Calculate the row index of the d_P element and d_M  
int Row = blockIdx.y * blockDim.y + threadIdx.y;  
  
// Calculate the column idenx of d_P and d_N  
int Col = blockIdx.x * blockDim.x + threadIdx.x;
```

Work for Block (0,0) with TILE_WIDTH 2

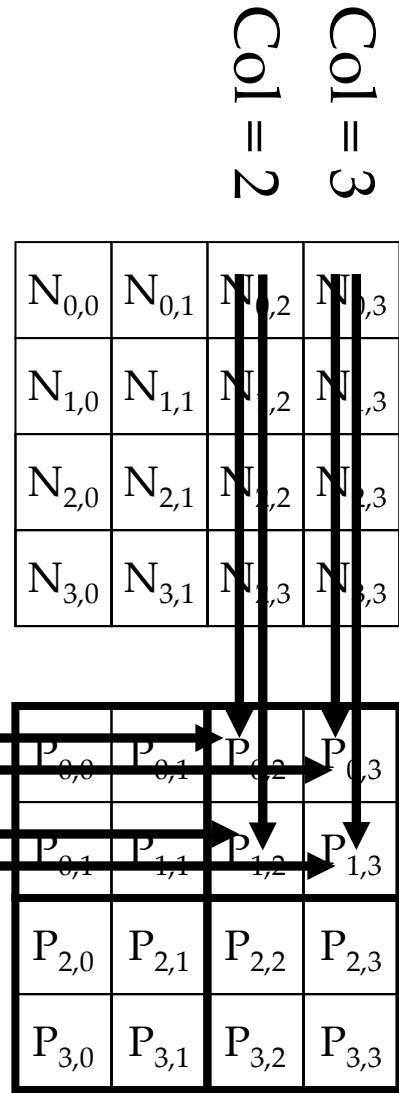


Work for Block (0,1)

Col = $1 * 2 + \text{threadIdx.x}$
Row = $0 * 2 + \text{threadIdx.y}$

blockIdx.x blockIdx.y

Row = 0
Row = 1



A Simple Matrix Multiplication Kernel

```
__global__
void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
    // Calculate the row index of the d_P element and d_M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    // Calculate the column idenx of d_P and d_N
    int Col = blockIdx.x*blockDim.x+threadIdx.x;

    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k)
            Pvalue += d_M[Row*Width+k] * d_N[k*Width+Col];
        d_P[Row*Width+Col] = Pvalue;
    }
}
```

Memory Bandwidth is Overloaded!

- That's a **simple implementation**:
 - GPU kernel is the **CPU code** with the **outer loops replaced with** per-thread **index calculations!**
- Unfortunately, performance is quite bad.
- Why?
- With the given approach,
 - global **memory bandwidth can't** supply enough data to **keep the SMs busy!**

Where Do We Access Global Memory?

```
__global__
void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
    // Calculate the row index of the d_P element and d_M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    // Calculate the column idenx of d_P and d_N
    int Col = blockIdx.x*blockDim.x+threadIdx.x;

    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k)
            Pvalue += d_M[Row*Width+k] * d_N[k*Width+Col];
        d_P[Row*Width+Col] = Pvalue;
    }
}
```

accesses
to global
memory

Each Thread Requires 4B of Data per FLOP

- Each threads access global memory
 - for elements of **M** and **N**:
 - 4B each**, or **8B per pair**.
 - (And once TOTAL to **P** per thread—ignore it.)
- With each pair of elements,
 - a thread does a single multiply-add,
 - 2 FLOP**—floating-point operations.
- So for every FLOP,
 - a thread needs** 4B from memory:
 - 4B / FLOP**.

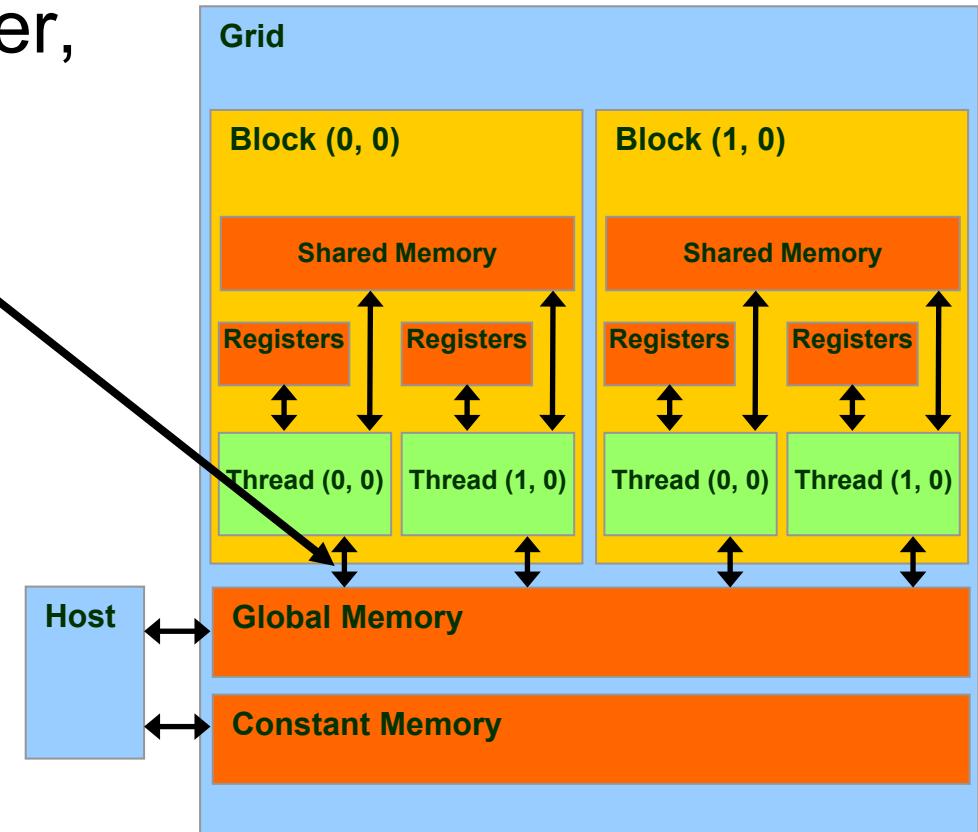
150 GB/s Bandwidth Implies 37.5 GFLOPs

- One generation of GPUs:
 - **1,000 GFLOP/s** of compute power, and
 - **150 GB/s** of memory bandwidth.

- Dividing bandwidth by memory requirements:

$$\frac{150 \text{ GB/s}}{4 \text{ B/FLOP}} = 37.5 \text{ GFLOP/s}$$

which **limits computation!**



What to Do? Reuse Memory Accesses!

But **37.5 GFLOPs** is a limit.

In an **actual execution**,

- memory is not busy all the time, and
- the code **runs at** about **25 GFLOPs**.

To get closer to 1,000 GFLOPs

- we **need to** drastically **cut down**
- **accesses to global memory**.

Problem Solving

- Q: Consider a 2D input matrix of 256 rows and 4 columns. We use 16×16 block of threads to perform operations on the input matrix so that each thread processes exactly one input element. How many blocks need to be launched?
- A:
 - $256 \text{ rows} * 4 \text{ columns} = 1024 \text{ elements to compute.}$
 - $16 * 16 = 256 \text{ threads per block.}$
 - $1024 / 256 = 4 \text{ blocks of threads.}$

Problem Solving

- Q: Suppose your kernel code requires certain threads to read data items written by other threads in the same thread block. Which type of memory will be most suitable (fastest accesses) for this purpose?
 - Register
 - Shared memory
 - Global memory
- A: Shared memory

Problem Solving

- Q: What are the possible values of *dst after this kernel execution?

```
__global__ void race_me(char *dst) {  
    dst[0] = threadIdx.x;  
}  
  
// host code  
cudaMalloc(&dst, 1);  
cudaMemset(dst, 3); // assign value 3 to dst  
race_me<<<1,2>>>(dst);
```

- A: Either 0 or 1



**ANY MORE QUESTIONS?
READ CHAPTER 4!**



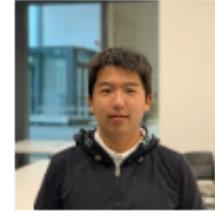
ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 3: Kernel-Based Data Parallel Execution Model

Course Reminders

- Lab 0 is due this Friday at 8pm US Central time
 - Easy lab, but make sure to start early!
- Lab 1 is out; it is due next Friday, 2/3/223
 - Requires Lab 0 to be completed first
- Post your questions on Campuswire, the course staff only replies to questions posted on Campuswire.

Name	Office Hours	Office Hours Location
 Xiaoyu Ma xiaoyum2@illinois.edu	Thursday 5 - 6 pm Friday 5 - 6 pm	zoom
 Xiyue Zhu xiyuez2@illinois.edu	Thursday 6 - 7 pm Friday 6 - 7 pm	zoom
 Huili Tao huilit2@illinois.edu	Wednesday 1 - 2 PM Friday 2 - 3 PM	zoom

Objective

- To learn more about the multi-dimensional logical organization of CUDA threads
- To learn to use control structures, such as loops in a kernel
- To learn the concepts of thread scheduling, latency tolerance, and hardware occupancy

Review – Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vecAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    dim3 DimGrid(ceil(n/256), 1, 1);
    dim3 DimBlock(256, 1, 1);
    A
    B
    vecAddKernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);
}
```

A Number of blocks per dimension

B Number of threads per dimension in a block

C Unique block # in x dimension

D Number of threads per block in x dimension

E Unique thread # in x dimension in the block

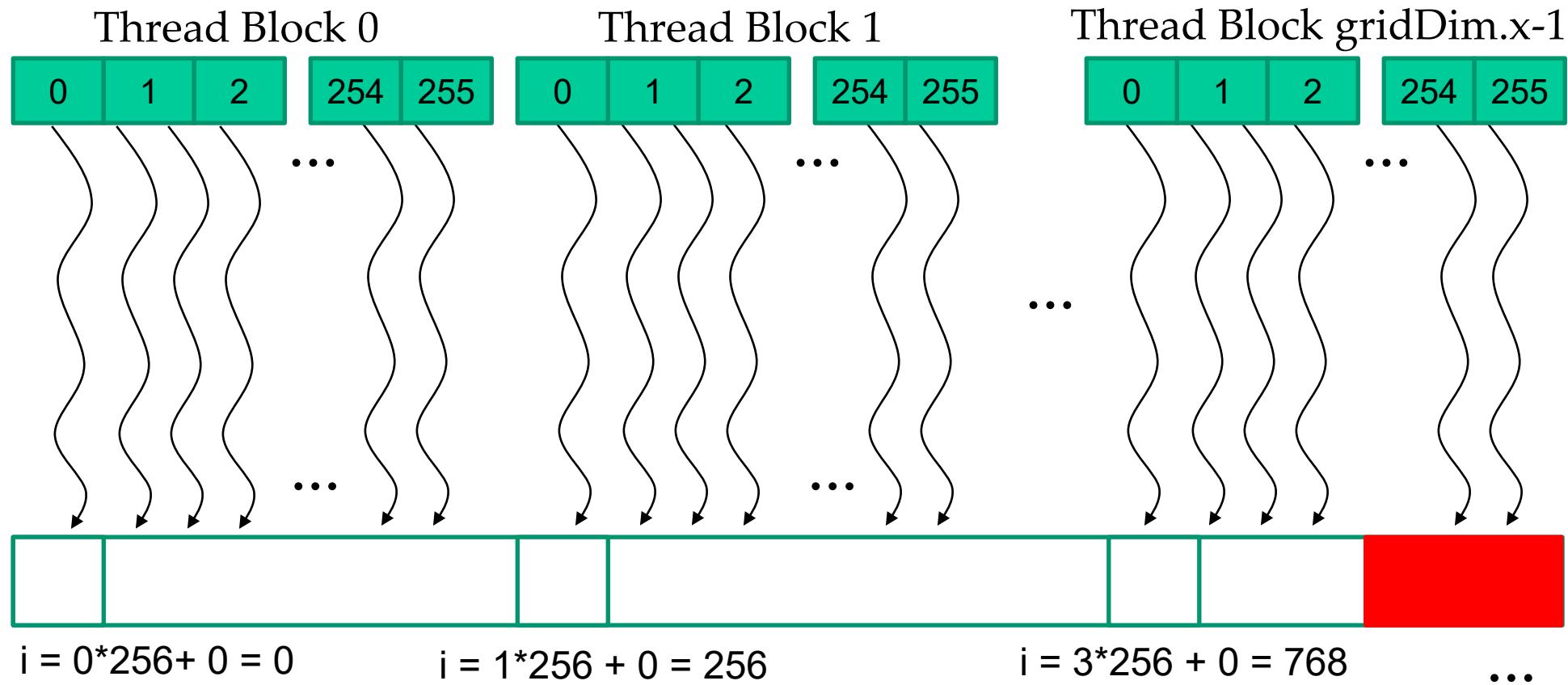
Q: How many threads in total will be executed in this example?

Review – Thread Assignment for vecAdd

where $N = 1,000$, block size = 256

```
vecAdd<<<ceil(N/256.0), 256>>>(...)
```

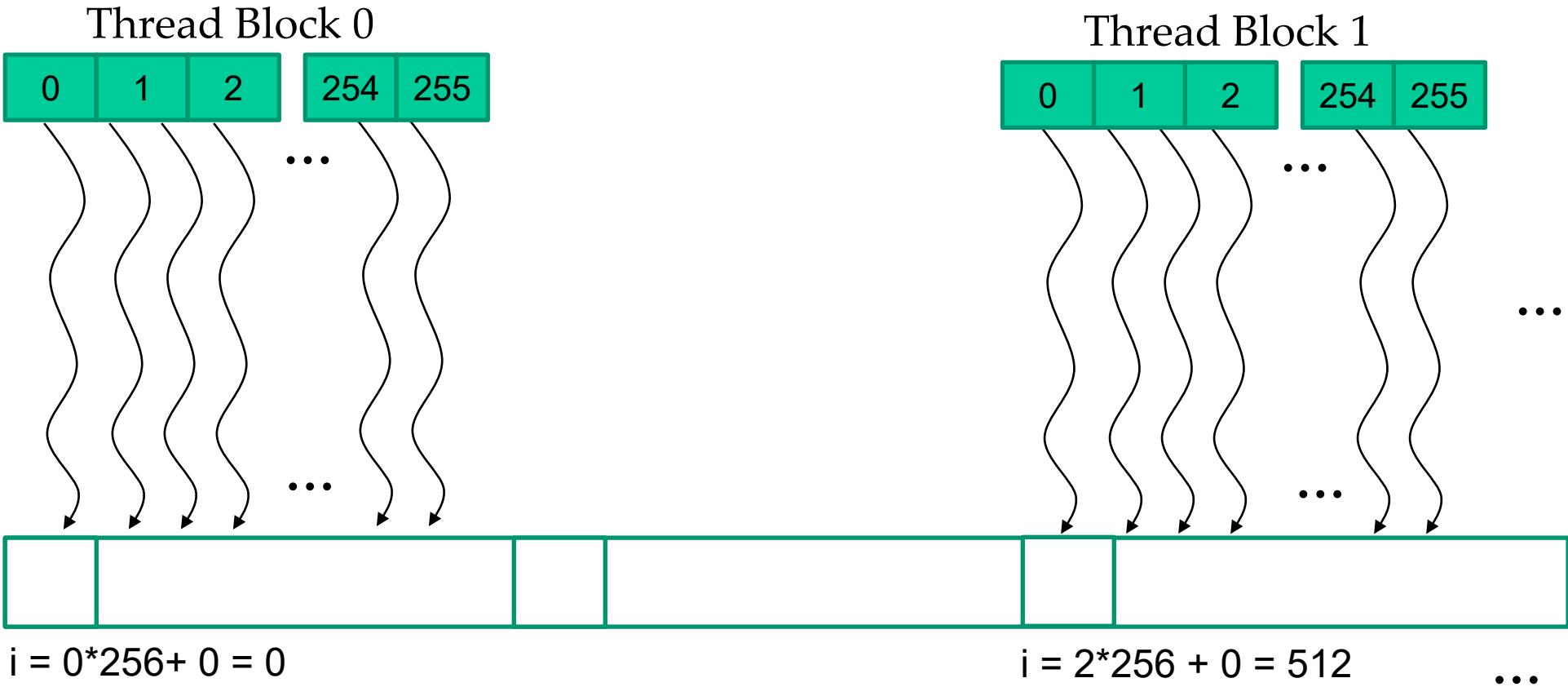
```
i = blockIdx.x * blockDim.x + threadIdx.x;  
if (i<n) C[i] = A[i] + B[i];
```



Coarser Grains: Thread Assignment for vecAdd with Two Elements per Thread

```
vecAdd<<<ceil(N/(2*256.0)), 256>>>(...)
```

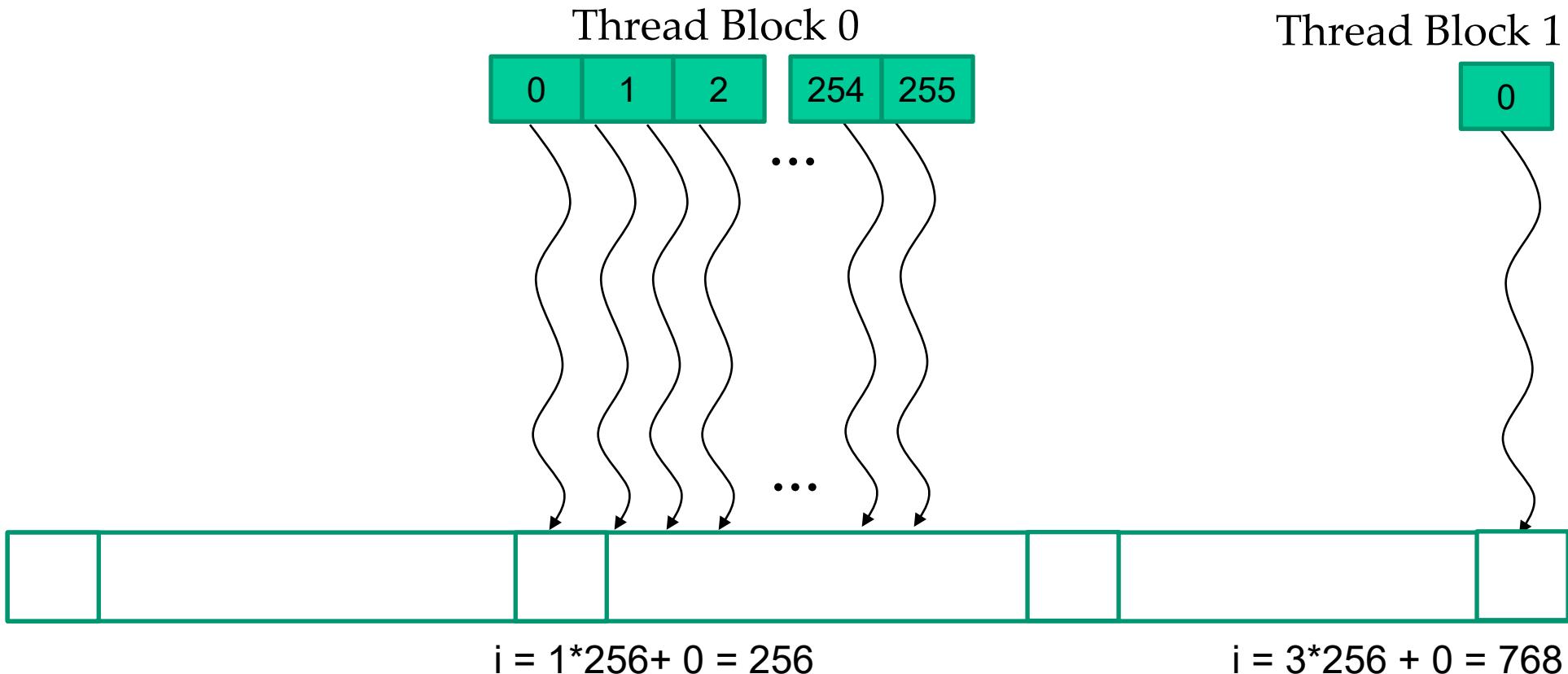
```
i = blockIdx.x * (2*blockDim.x) + threadIdx.x;  
if (i<n) C[i] = A[i] + B[i];
```



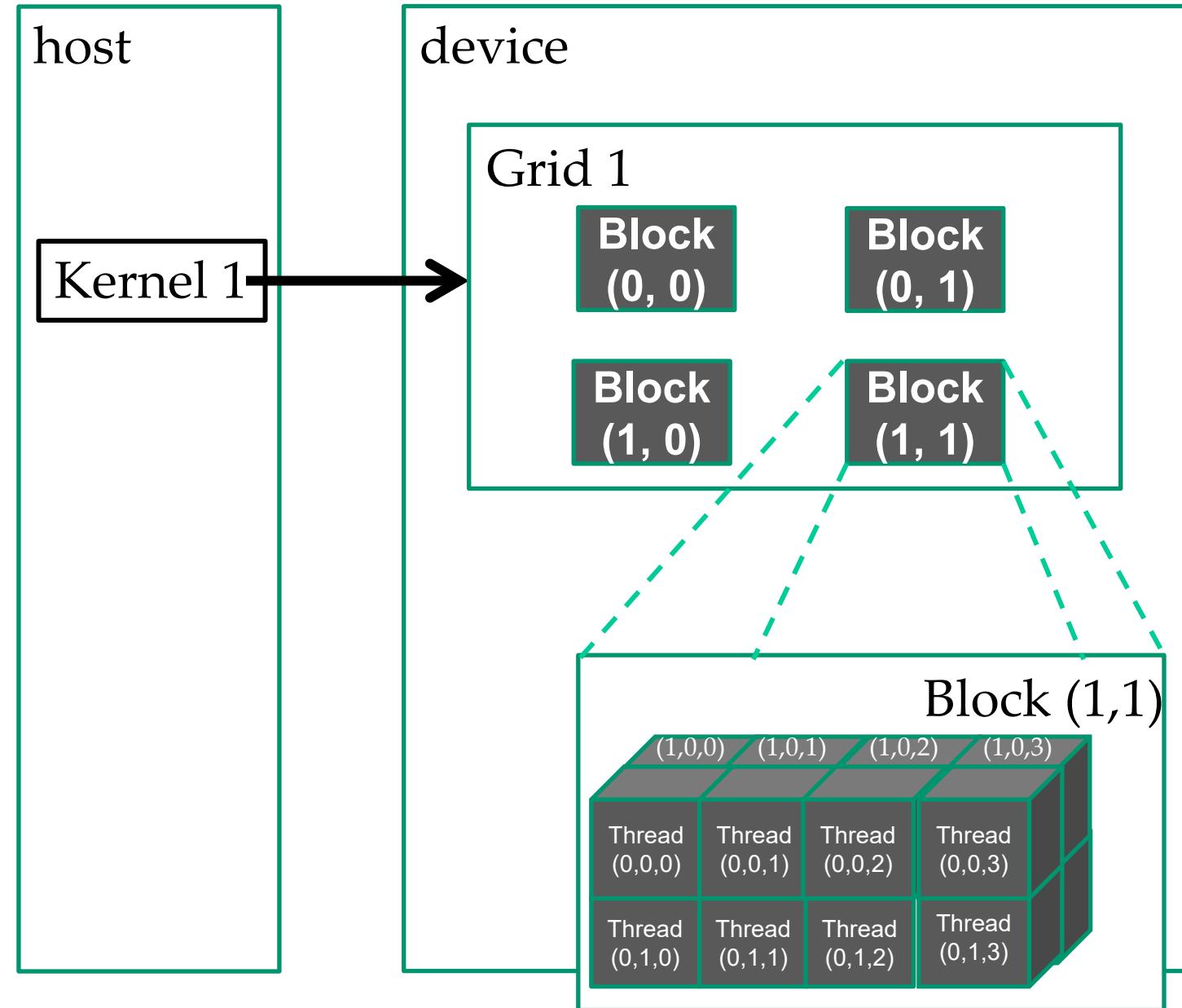
Coarser Grains: Thread Assignment for vecAdd with Two Elements per Thread

```
vecAdd<<<ceil(N/(2*256.0)), 256>>>(...)
```

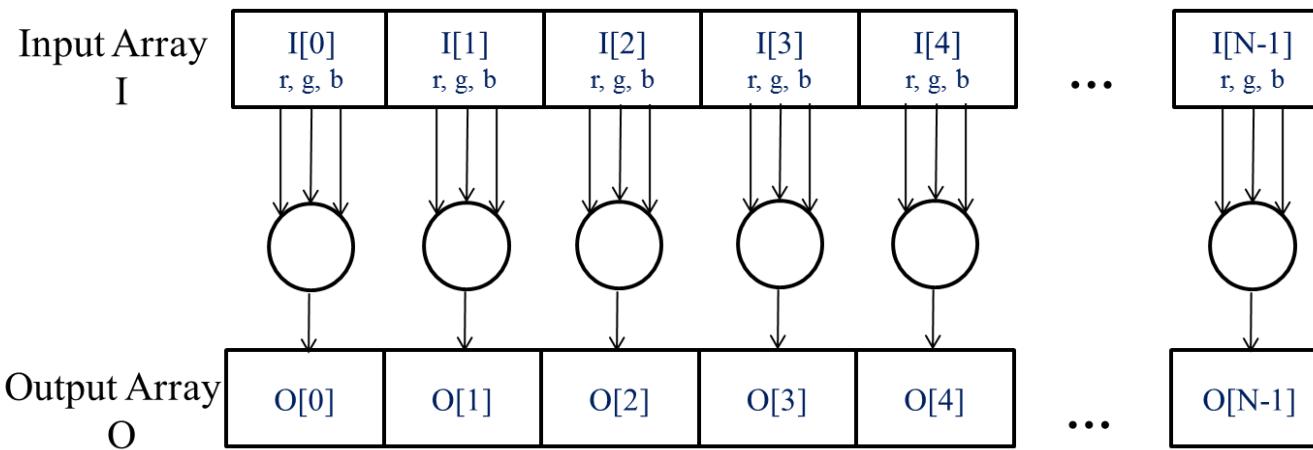
```
i = blockIdx.x * (2*blockDim.x) + threadIdx.x;  
if (i<n) C[i] = A[i] + B[i];  
i = i+blockDim.x;  
if (i<n) C[i] = A[i] + B[i];
```



CUDA Thread Grids are Multi-Dimensional

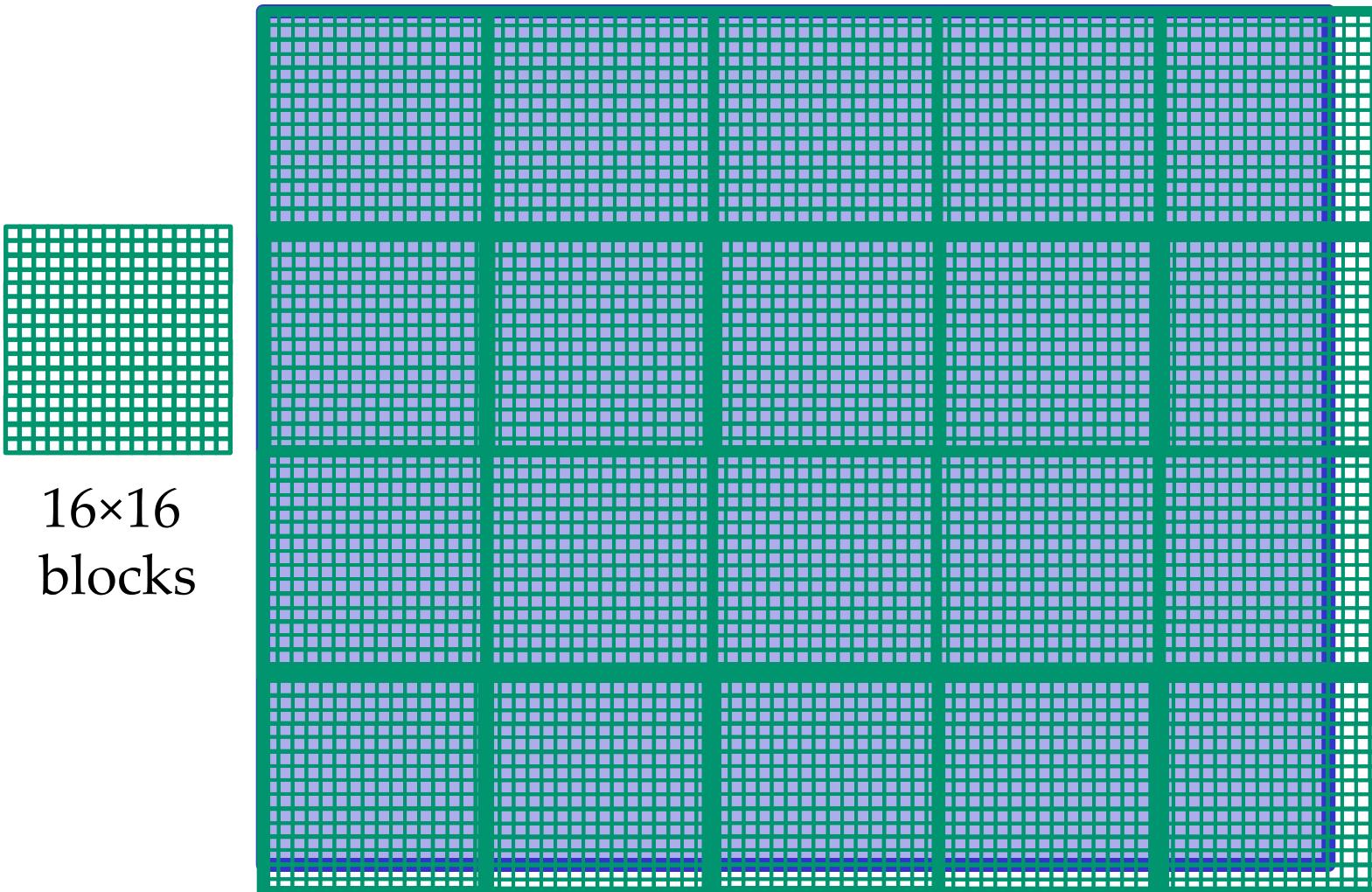


Example 1: Conversion of a color image to a grey-scale image

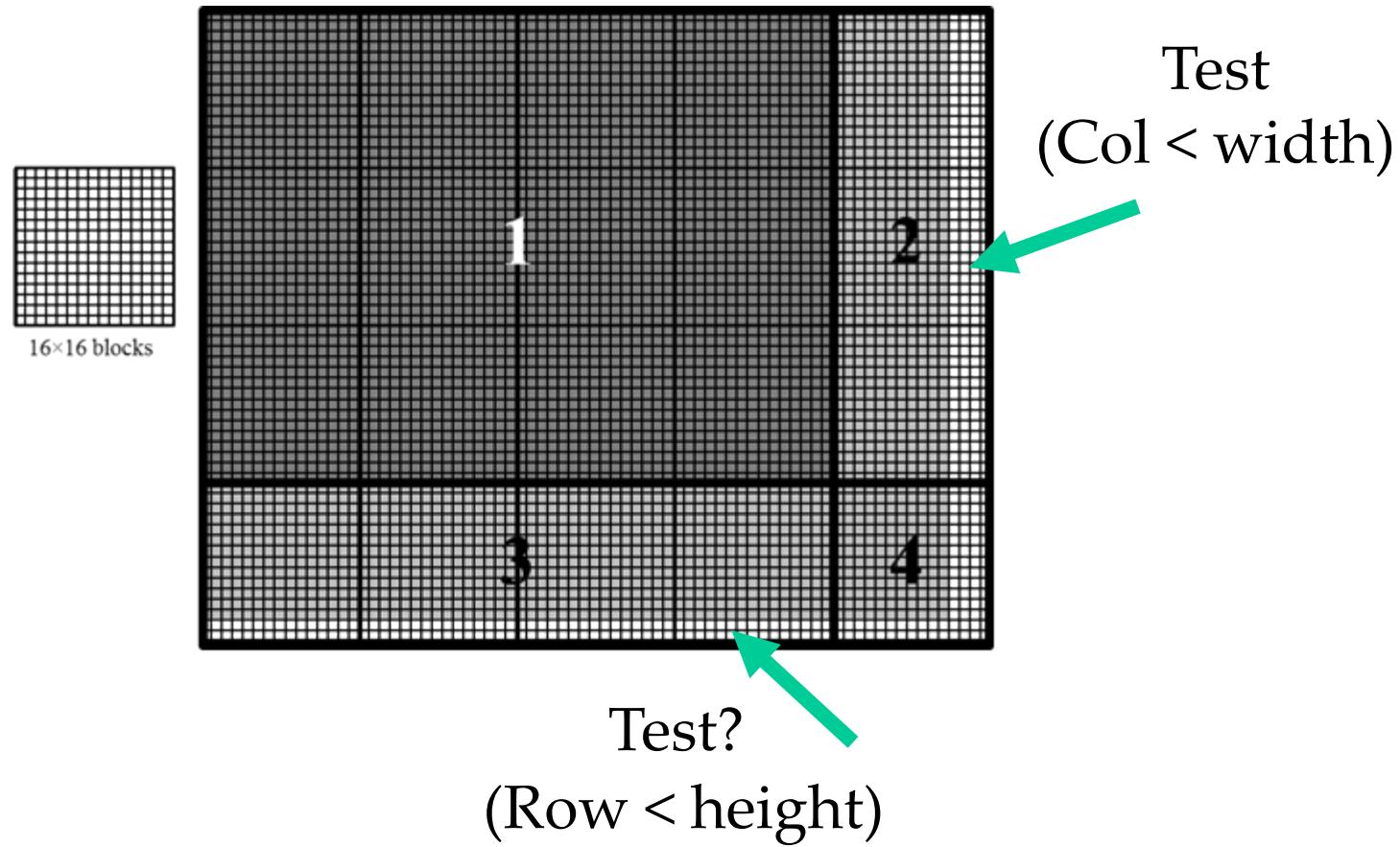


*Pixels can be
calculated
independently*

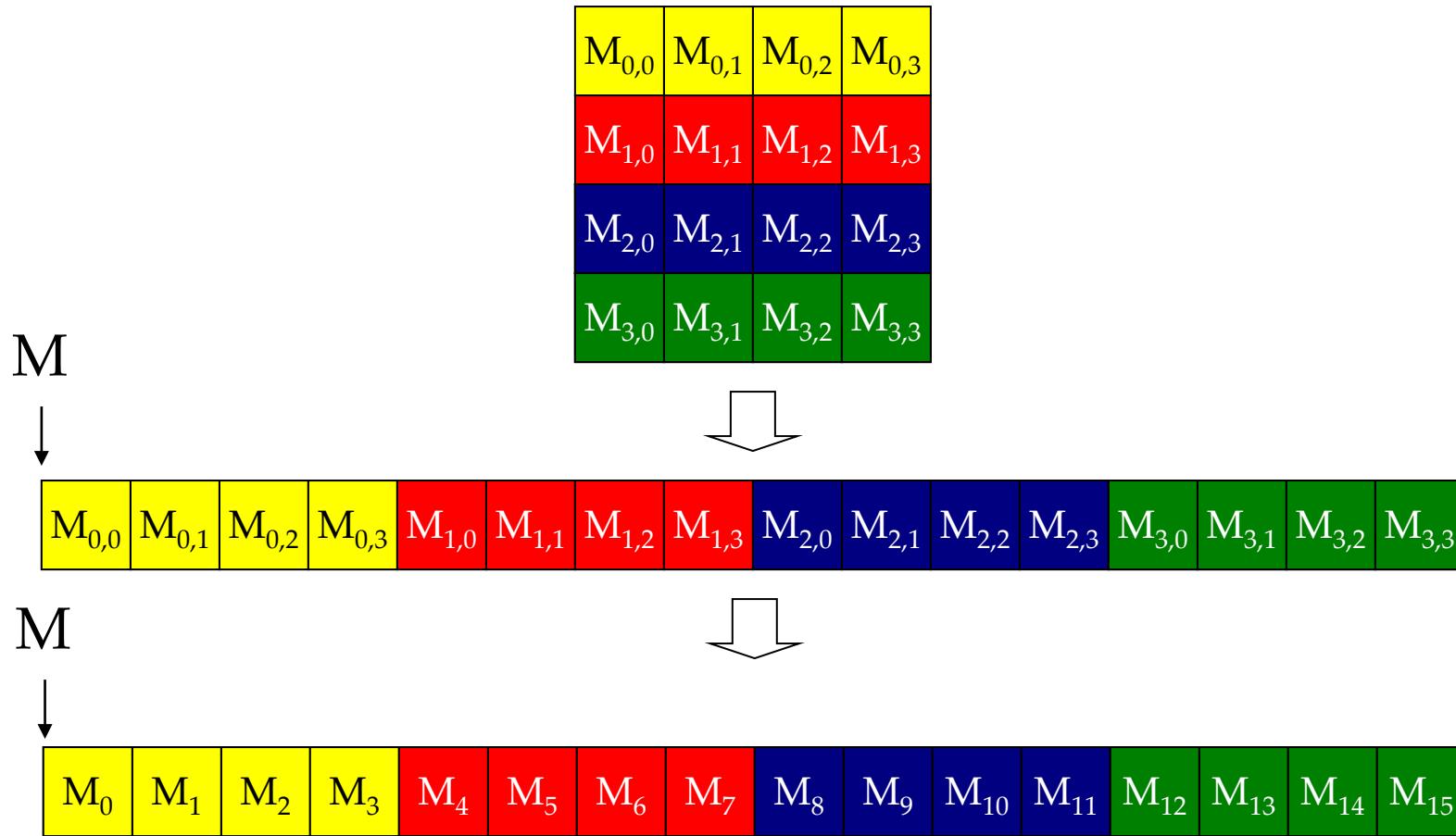
Processing a Picture with a 2D Grid



Covering a 76×62 picture with 16×16 blocks



Row-Major Layout of 2D Arrays in C/C++



$$M_{2,1} \rightarrow \text{Row} * \text{Width} + \text{Col} = 2 * 4 + 1 = 9$$

colorToGreyscaleConversion Kernel with 2D thread mapping to data

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConversion(unsigned char * grayImage, unsigned char * rgbImage,
                                 int width, int height)
{
    int Col = threadIdx.x + blockIdx.x * blockDim.x;
    int Row = threadIdx.y + blockIdx.y * blockDim.y;

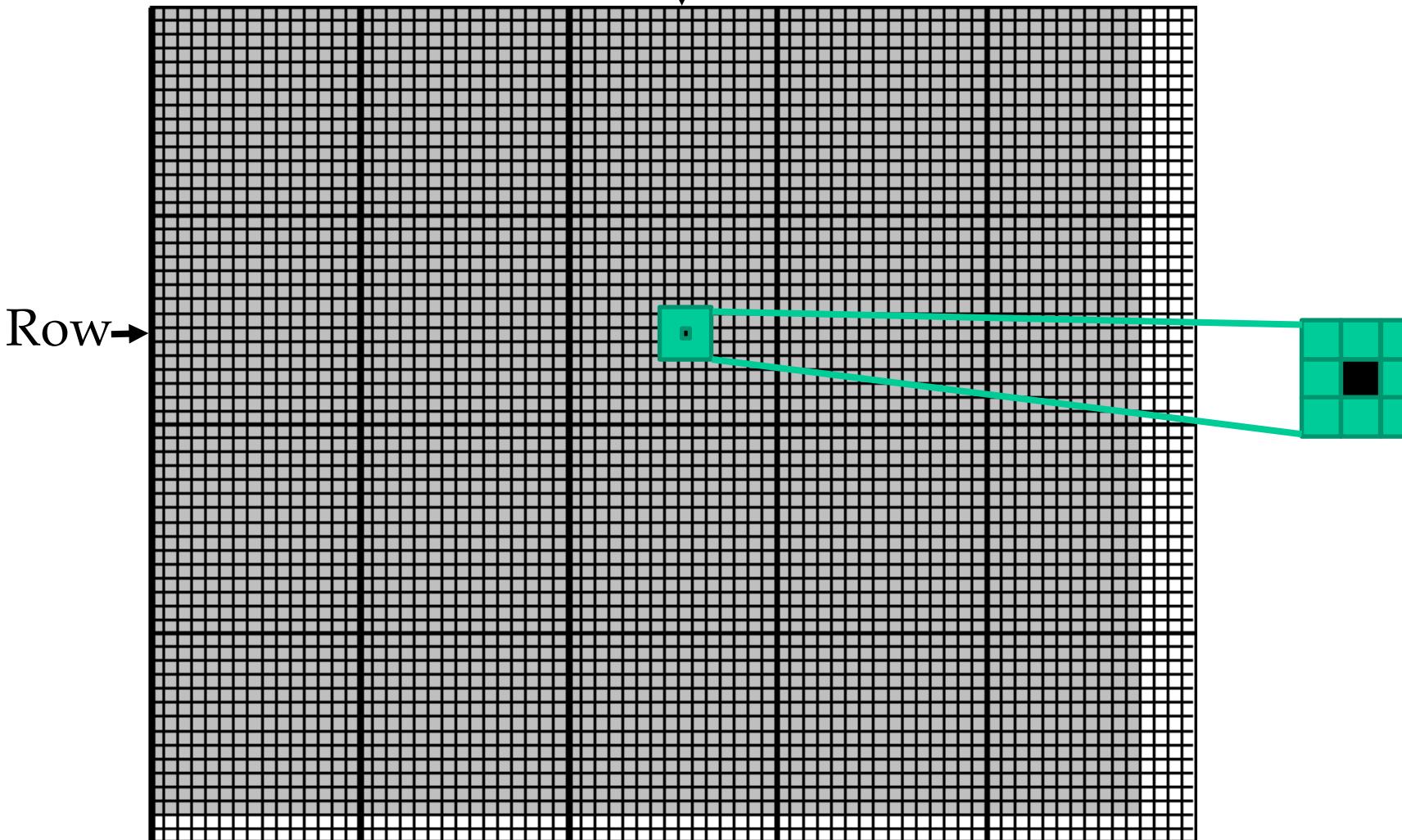
    if (Col < width && Row < height) {
        // get 1D coordinate for the grayscale image
        int greyOffset = Row*width + Col;
        // one can think of the RGB image having
        // THREE times as many columns of the gray scale image
        int rgbOffset = 3 * greyOffset;
        unsigned char r = rgbImage[rgbOffset]; // red value for pixel
        unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
        unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pixel
        // perform the rescaling and store it
        // We multiply by floating point constants
        grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
    }
}
```

Example 2: Image Blurring (Monochrome)



(BLUR_SIZE is 5)

Each output pixel is the average of pixels
around it (BLUR_SIZE = 1)



An Image Blur Kernel

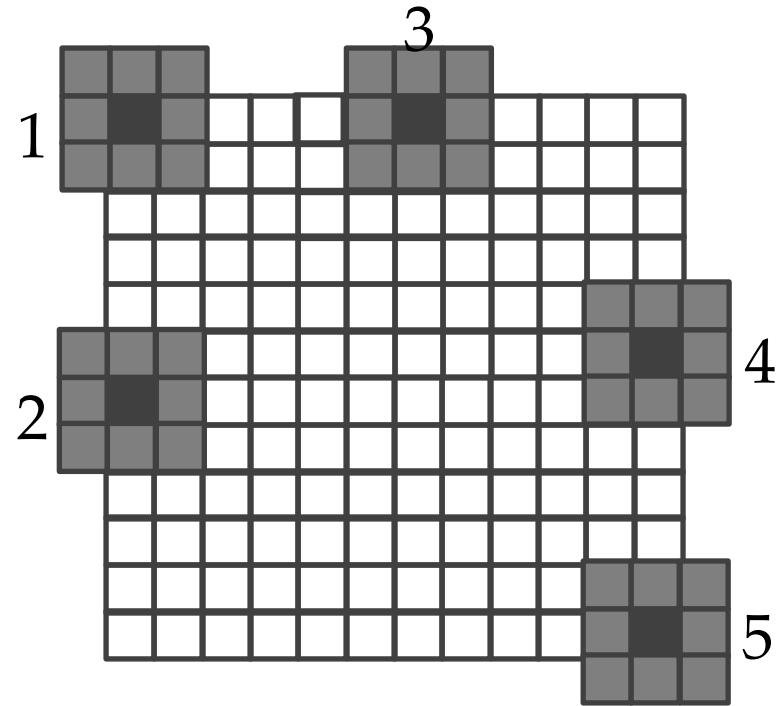
```
__global__
void blurKernel(unsigned char * in, unsigned char * out, int w, int h) {
    int Col  = blockIdx.x * blockDim.x + threadIdx.x;
    int Row  = blockIdx.y * blockDim.y + threadIdx.y;

    if (Col < w && Row < h) {
1.        int pixVal = 0;
2.        int pixels = 0;

        // Get the average of the surrounding BLUR_SIZE x BLUR_SIZE box
3.        for(int blurRow = -BLUR_SIZE; blurRow <= BLUR_SIZE; ++blurRow) {
4.            for(int blurCol = -BLUR_SIZE; blurCol <= BLUR_SIZE; ++blurCol) {

5.                int curRow = Row + blurRow;
6.                int curCol = Col + blurCol;
                // Verify we have a valid image pixel
7.                if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
8.                    pixVal += in[curRow * w + curCol];
9.                    pixels++; // Keep track of number of pixels in the avg
                }
            }
        }
        // Write our new pixel value out
10.       out[Row * w + Col] = (unsigned char)(pixVal / pixels);
    }
}
```

Handling boundary conditions for pixels near the edges of the image



An Image Blur Kernel

```
__global__
void blurKernel(unsigned char * in, unsigned char * out, int w, int h) {
    int Col  = blockIdx.x * blockDim.x + threadIdx.x;
    int Row  = blockIdx.y * blockDim.y + threadIdx.y;

    if (Col < w && Row < h) {
1.        int pixVal = 0;
2.        int pixels = 0;

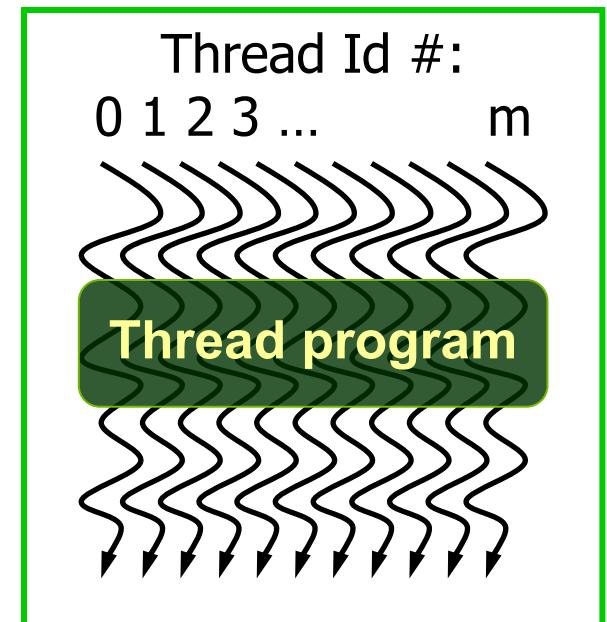
        // Get the average of the surrounding BLUR_SIZE x BLUR_SIZE box
3.        for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
4.            for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

5.                int curRow = Row + blurRow;
6.                int curCol = Col + blurCol;
                // Verify we have a valid image pixel
7.                if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
8.                    pixVal += in[curRow * w + curCol];
9.                    pixels++; // Keep track of number of pixels in the avg
                }
            }
        }
        // Write our new pixel value out
10.       out[Row * w + Col] = (unsigned char)(pixVal / pixels);
    }
}
```

CUDA Execution Model: Thread Blocks

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
 - Block size 1 to 1024 concurrent threads
 - Block shape 1D, 2D, or 3D
- Threads within block have **thread index** numbers
- Kernel code uses **thread index** and **block index** to select work and address shared data
- Threads in the same block **share data** and **synchronize** while doing their share of the work
- Threads in different blocks cannot cooperate
- Blocks **execute in arbitrary order!**

CUDA Thread Block



Courtesy: John Nickolls,
NVIDIA

Compute Capabilities are GPU-Dependent

Table 1. A Comparison of Maxwell GM107 to Kepler GK107

GPU	GK107 (Kepler)	GM107 (Maxwell)
CUDA Cores	384	640
Base Clock	1058 MHz	1020 MHz
GPU Boost Clock	N/A	1085 MHz
GFLOP/s	812.5	1305.6
Compute Capability	3.0	5.0
Shared Memory / SM	16KB / 48 KB	64 KB
Register File Size / SM	256 KB	256 KB
Active Blocks / SM	16	32
Memory Clock	5000 MHz	5400 MHz
Memory Bandwidth	80 GB/s	86.4 GB/s
L2 Cache Size	256 KB	2048 KB
TDP	64W	60W
Transistors	1.3 Billion	1.87 Billion
Die Size	118 mm ²	148 mm ²
Manufacturing Process	28 nm	28 nm

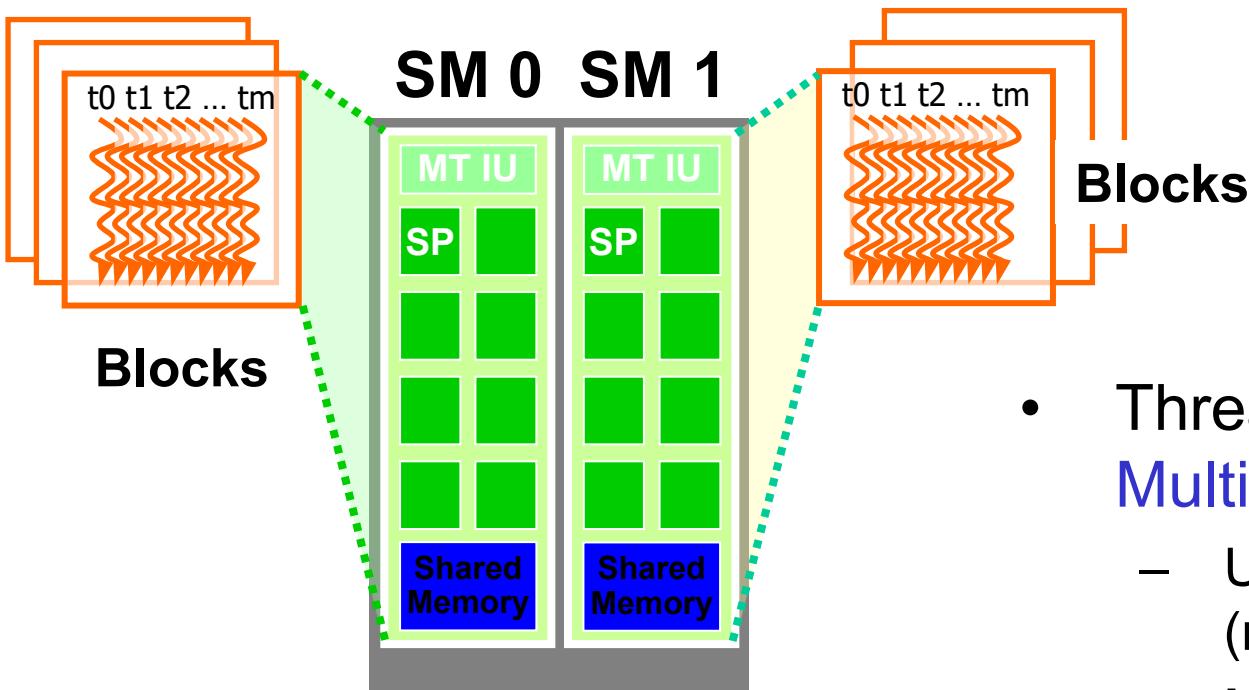
Compute Capabilities are GPU-Dependent

Table 1. A Comparison of Maxwell GM107 to Kepler GK107

CPU	GK107 (Kepler)	GM107 (Maxwell)
	GK107 (Kepler)	GM107 (Maxwell)
Shared Memory / SM	16 / 48 kB	64 kB
Register File Size / SM	256 kB	256 kB
Active Blocks / SM	16	32

TDP	64W	60W
Transistors	1.3 Billion	1.87 Billion
Die Size	118 mm ²	148 mm ²
Manufacturing Process	28 nm	28 nm

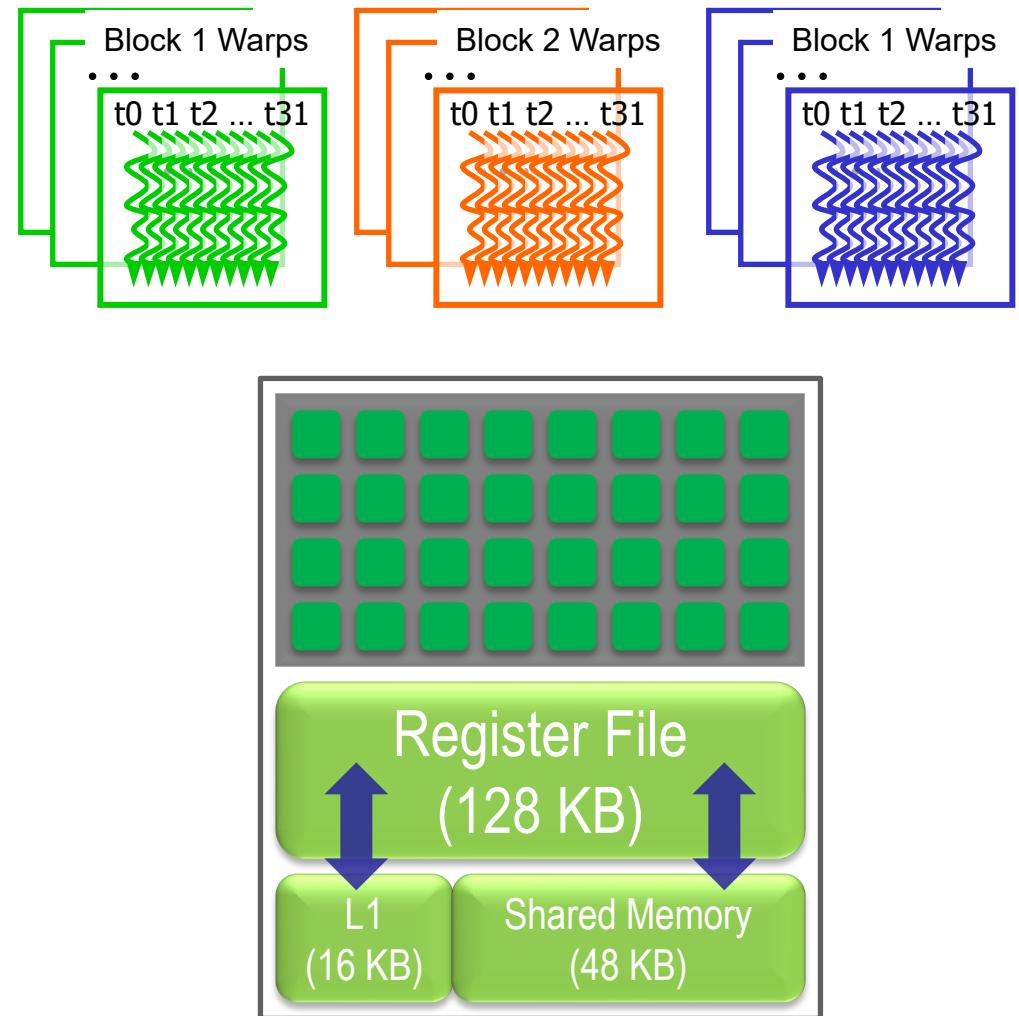
Executing Thread Blocks



- Threads are assigned to **Streaming Multiprocessors** in block granularity
 - Up to **32** blocks to each SM (resource limit for Maxwell)
 - Maxwell SM can take up to **2048** threads
- Threads run concurrently
 - SM maintains thread/block id #s
 - SM manages/schedules thread execution

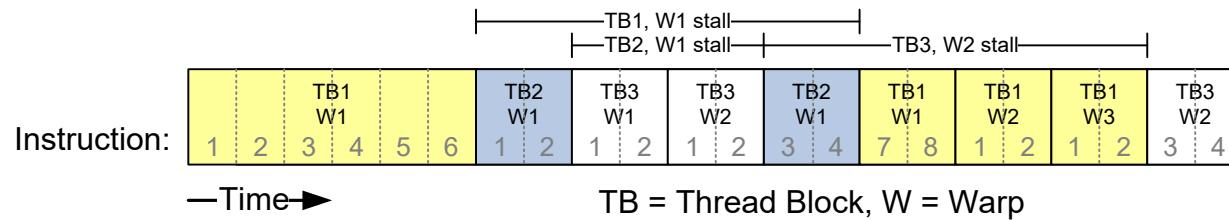
Thread Scheduling (1/2)

- Each block is executed as 32-thread warps
 - An implementation decision, not part of the CUDA programming model
 - Warps are divided based on their linearized thread index
 - Threads 0-31: warp 0
 - Threads 32-63: warp 1, etc.
 - Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each block has 256 threads, how many warps are there in an SM?
 - Each block is divided into $256/32 = 8$ warps
 - $8 \text{ warps/blk} * 3 \text{ blks} = 24 \text{ warps}$



Thread Scheduling (2/2)

- SM implements zero-overhead warp scheduling
 - Warps whose next instruction has its operands ready for consumption are eligible for execution
 - Eligible warps are selected for execution on a prioritized scheduling policy
 - **All threads in a warp execute the same instruction when selected**



Example execution timing of an SM

Pitfall: Control/Branch Divergence

- **branch divergence**
 - threads in a warp take different paths in the program
 - main performance concern with control flow
- GPUs use **predicated execution**
 - Each thread computes a yes/no answer for each path
 - **Multiple paths** taken by threads in a warp are **executed serially!**

Example of Branch Divergence

- Common case: use of thread ID as a branch condition

```
if (threadIdx.x > 2) {  
    // THEN path (lots of lines)  
} else {  
    // ELSE path (lots more lines)  
}
```

- Two control paths (THEN/ELSE) for threads in warp

***** ALL THREADS EXECUTE BOTH PATHS *****
(results kept only when predicate is true for thread)

Avoiding Branch Divergence

- Try to make branch granularity a multiple of warp size (remember, it may not always be 32!)

```
if (threadIdx.x / WARP_SIZE > 2) {  
    // THEN path (lots of lines)  
} else {  
    // ELSE path (lots of lines)  
}
```

- Still has two control paths
- But all threads in any warp follow only one path.

Block Granularity Considerations

- For `colorToGreyscaleConversion`, should one use 8x8, 16x16 or 32x32 blocks? Assume that in the GPU used, each SM can take up to 1,536 threads and up to 8 blocks.
 - For 8x8, we have 64 threads per block. Each SM can take up to 1,536 threads, which is $1,536/64=24$ blocks. But each SM can only take up to 8 blocks, so only 512 threads (16 warps) go into each SM!
 - For 16x16, we have 256 threads per block. Each SM can take up to 1,536 threads (48 warps), which is 6 blocks (within the 8 block limit). Thus, we use the full thread capacity of an SM.
 - For 32x32, we have 1,024 threads per Block. Only one block can fit into an SM, using only 2/3 of the thread capacity of an SM.

Problem Solving

- Q: A particular CUDA device's streaming multiprocessor (SM) can take up to 1536 threads and up to 4 thread blocks. Which of the following block configurations could guarantee the SM be fully utilized?
 - 256 threads per block, 384 threads per block, 576 threads per block
- A:
 - $1536 / 256 = 6$ thread blocks – too much for SM
 - $1536 / 384 = 4$ thread blocks per SM – just the right number
 - $1536 / 576 = 3$ thread blocks per SM – not enough to fully utilize the SM

Problem Solving

- Q: A 1D array of N floating point elements is to be processed in a one-element-per-thread fashion by a GPU. The target GPU has 8 SMs, each with 16 SPs. What is the best execution configuration for this kernel?
- A:
 - We do not know the max number of threads the SM can support.

Problem Solving

- Consider the following CUDA kernel:

```
__global__ void do_work(int q, int *A) {  
    int result = 0;  
    if (q < 5) result = threadIdx.x;  
    A[threadIdx.x] = result;  
}
```

- Q: Is there is a control divergence in this code?
- A: No since the value of q is the same for ALL threads in the thread block. For a thread to diverge, its execution path must depend on something unique to the thread, e.g., on threadIdx.



**ANY MORE QUESTIONS?
READ CHAPTER 3**



ECE408 /CS483/CSE408 Spring 2023 Applied Parallel Programming

Lecture 2: Introduction to CUDA C and Data Parallel Programming

Course Reminders

- Lab 0 is due next Friday at 8pm US Central time
 - It is an easy lab, but it may take some time to get the tools in place.
 - Its main purpose is to get familiar with the tools and the process.
 - If you miss this deadline, that's OK for Lab 0, but you must submit it ASAP. Remember, Lab 0 will be graded, but not counted towards your overall grade.
- Lab 1 will be out very soon, it is due on Friday, Feb. 3rd
 - See <https://wiki.illinois.edu/wiki/display/ECE408/Labs+and+Project> for details

Objective

- To learn the basic concept of data parallel computing
- To learn the basic features of the CUDA C programming interface

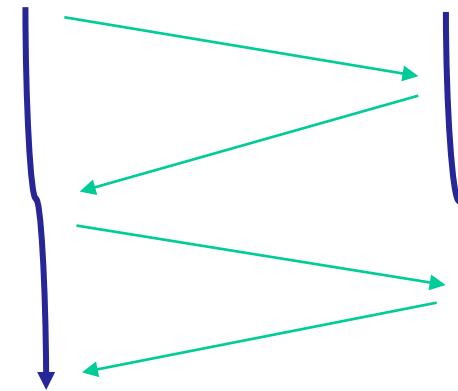
Thread as a basic unit of computing

- What is a thread?

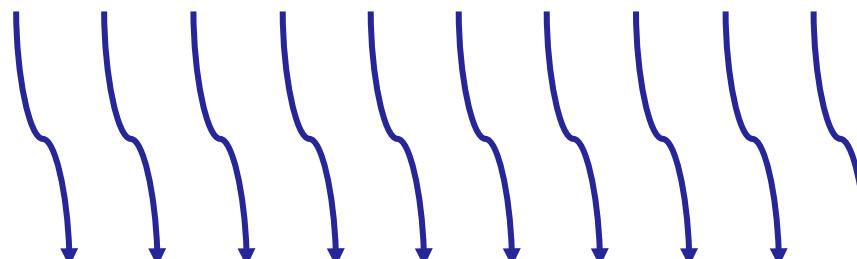
- Program
- PC
- Context
 - Memory
 - Registers
 - ...



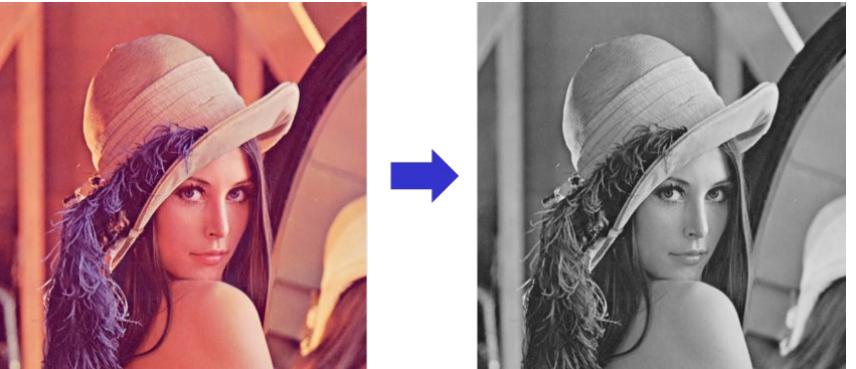
- Multiple threads



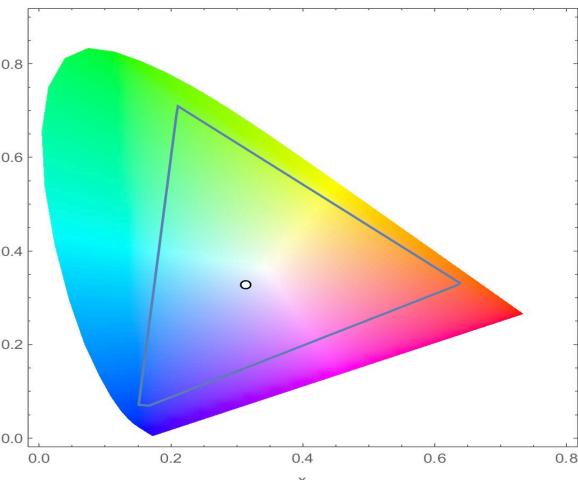
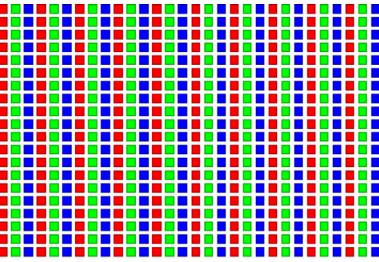
- Many threads



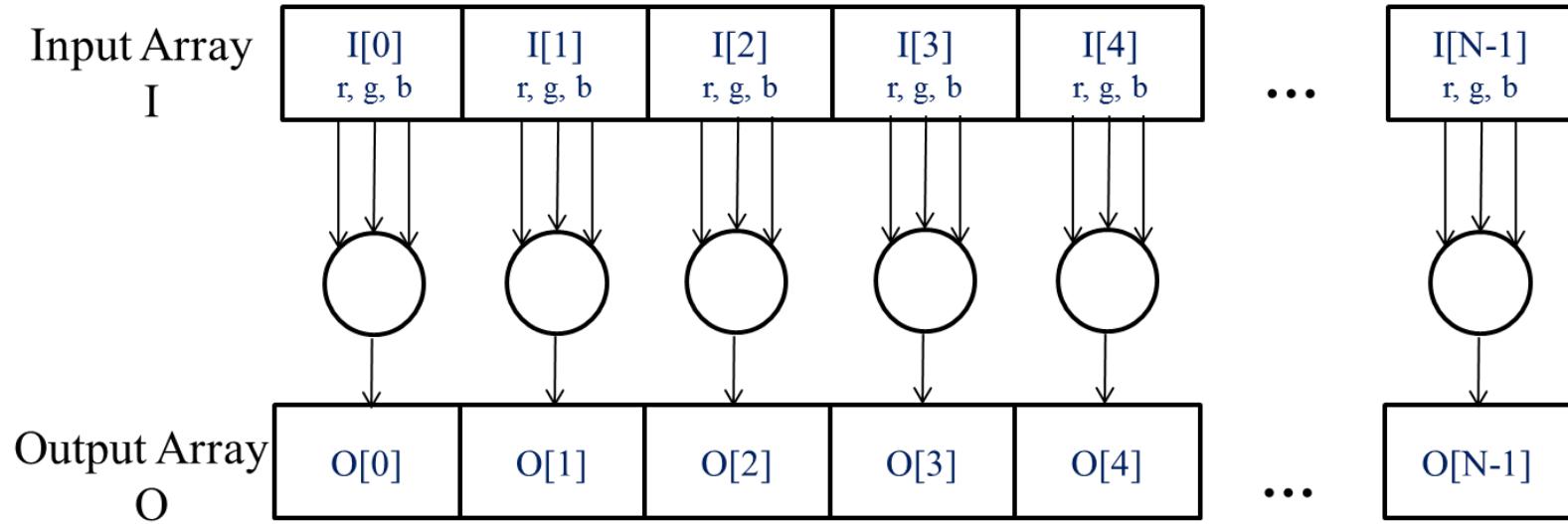
A Data Parallel Computation Example: Conversion of a color image to grey-scale image



```
for each pixel {  
    pixel = gsConvert(pixel)  
}  
// Every pixel is independent  
// of every other pixel
```



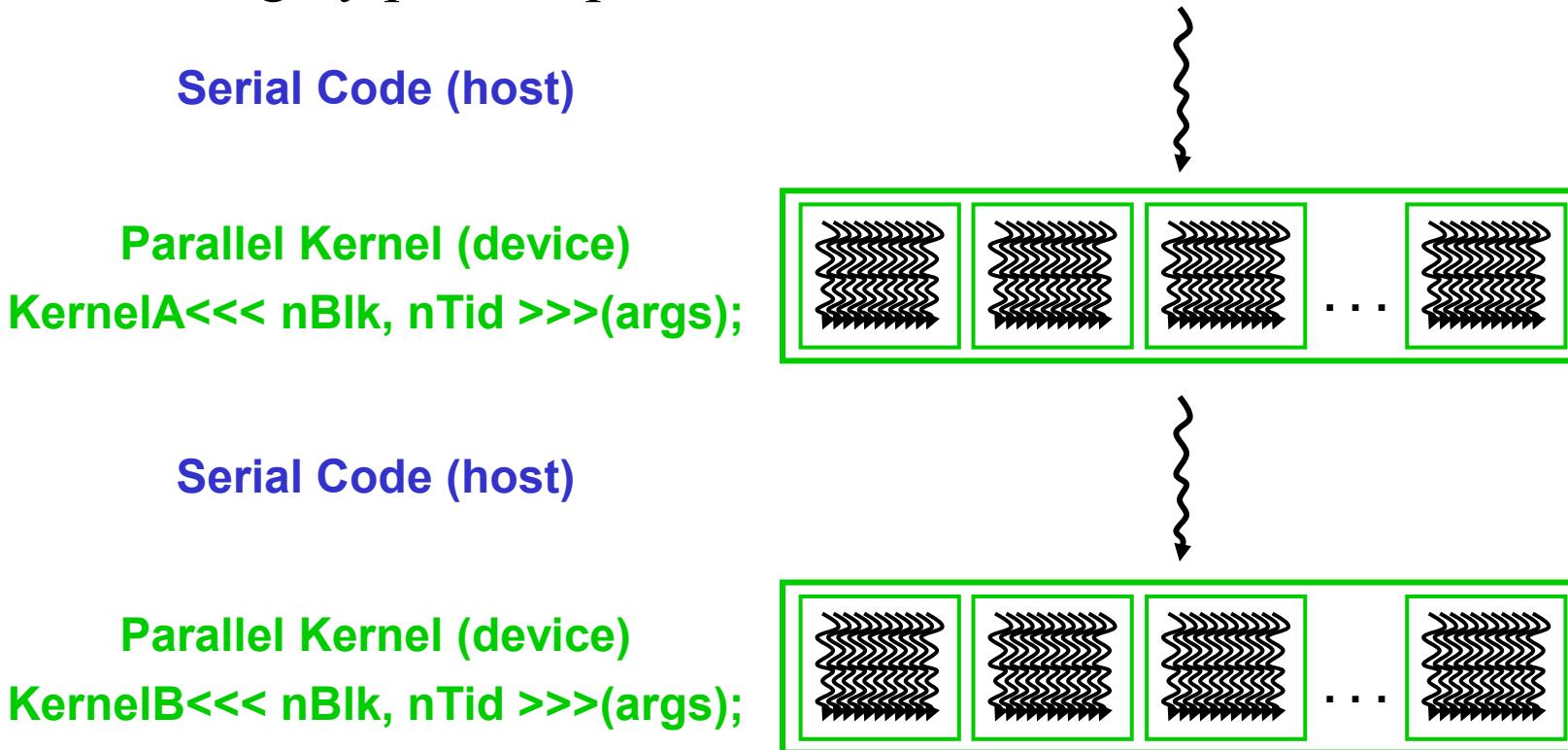
The pixels can be calculated independently of each other



```
for each pixel {  
    pixel = gsConvert(pixel)  
}  
// Every pixel is independent  
// of every other pixel
```

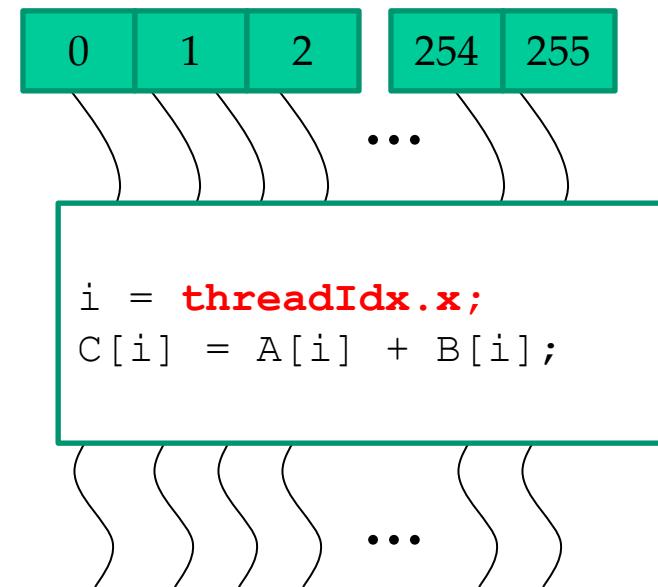
CUDA/OpenCL – Execution Model

- Integrated host+device app C program
 - Serial or modestly parallel parts in **host** C code
 - Highly parallel parts in **device** SPMD kernel C code



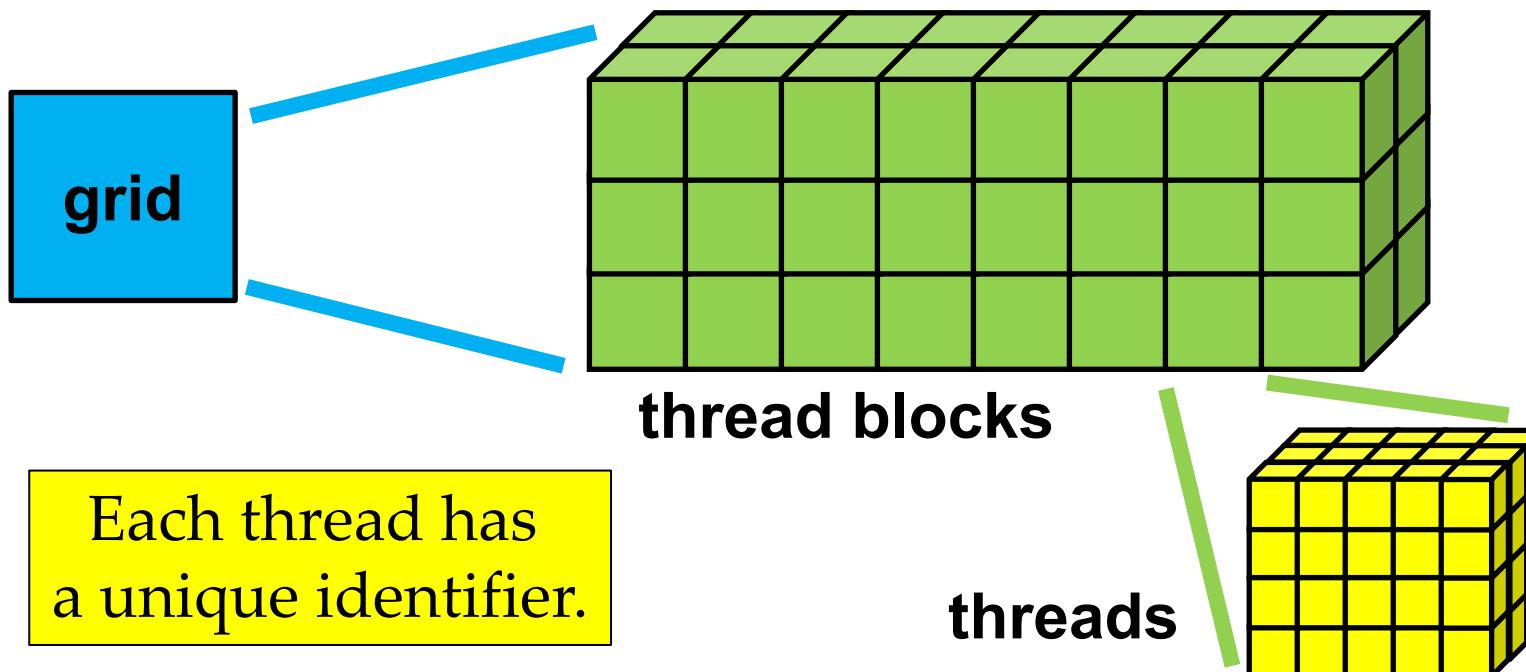
Arrays of Parallel Threads

- A CUDA kernel is executed as a **grid** (array) of threads
 - All threads in a grid run the same kernel code
 - Single Program Multiple Data (SPMD model)
 - Each thread has **a unique index** that it uses to compute memory addresses and make control decisions



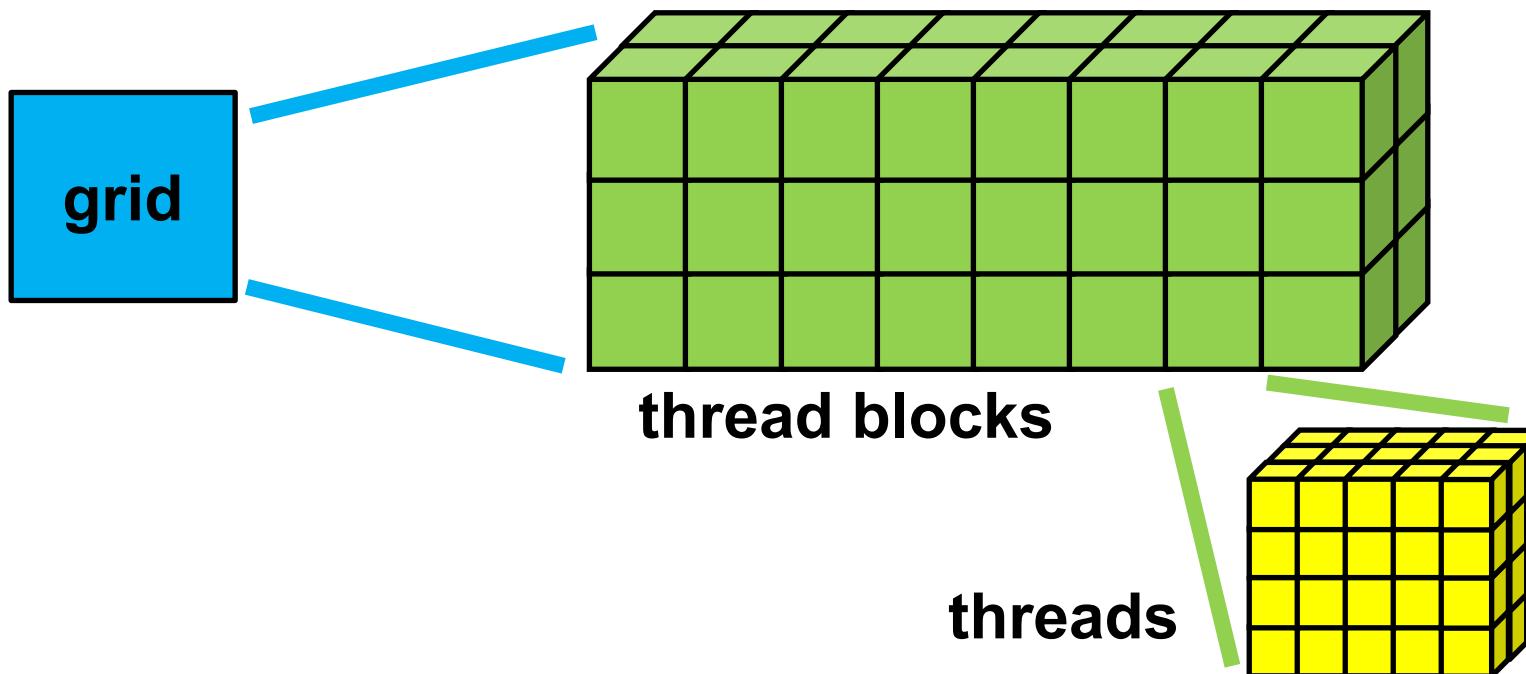
Logical Execution Model for CUDA

- Each CUDA kernel
 - is executed by a **grid**,
 - a 3D array of **thread blocks**, which are
 - 3D arrays of **threads**.



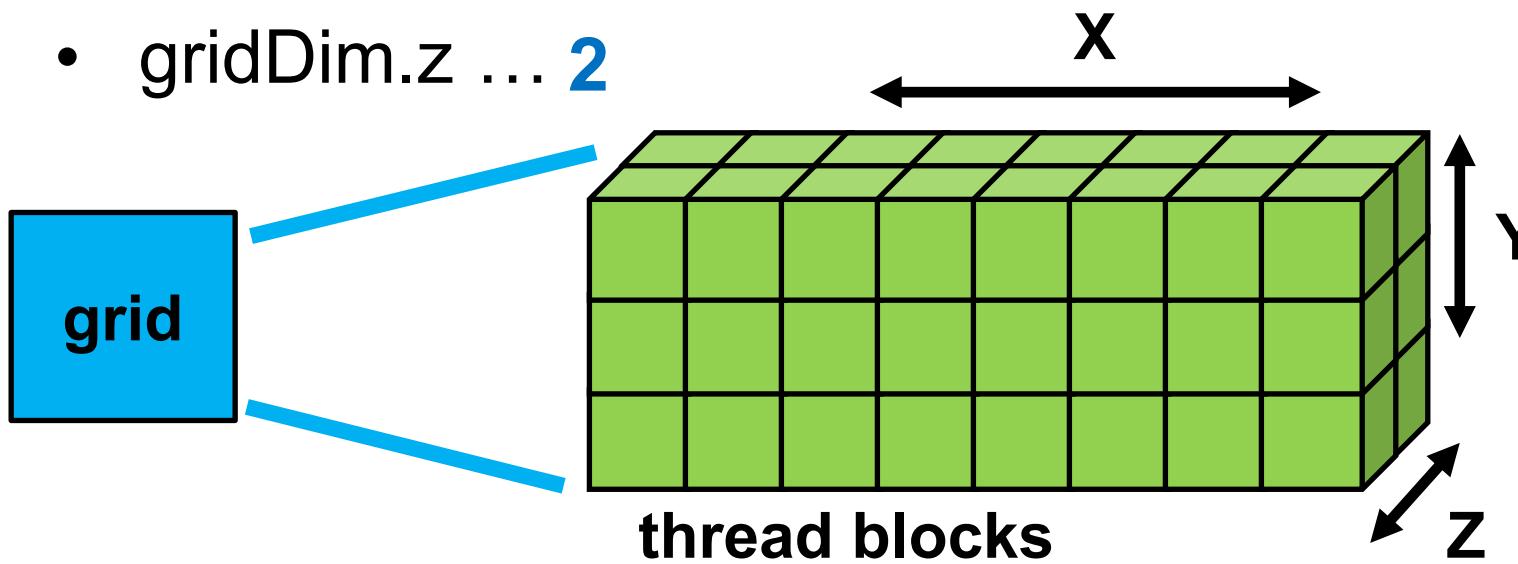
Single Program, Multiple Data

- Each thread
 - executes the **same program**
 - on **distinct data inputs**,
 - a single-program, multiple-data (**SPMD**) model



gridDim Gives Number of Blocks

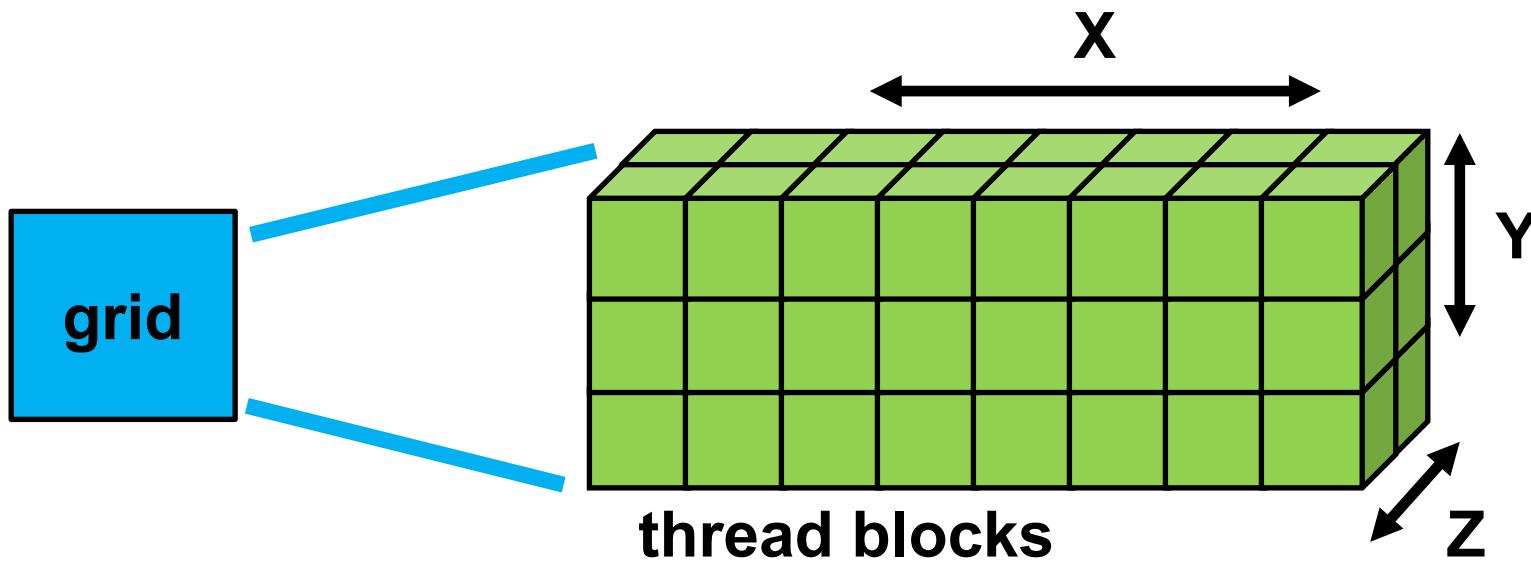
- Number of blocks in each dimension is
 - `gridDim.x` ... 8
 - `gridDim.y` ... 3
 - `gridDim.z` ... 2



For 2D (and 1D grids), simply use grid dimension 1 for Z (and Y).

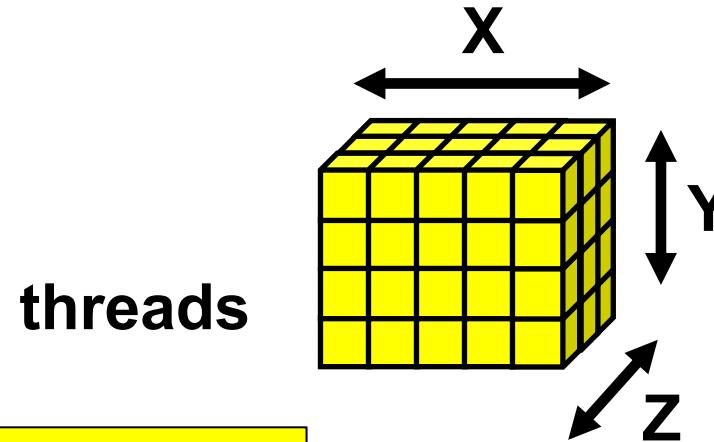
blockIdx is Unique for Each Block

- Each block has a unique index tuple
 - `blockIdx.x` (from 0 to $(\text{gridDim.x} - 1)$)
 - `blockIdx.y` (from 0 to $(\text{gridDim.y} - 1)$)
 - `blockIdx.z` (from 0 to $(\text{gridDim.z} - 1)$)



blockDim: # of Threads per Block

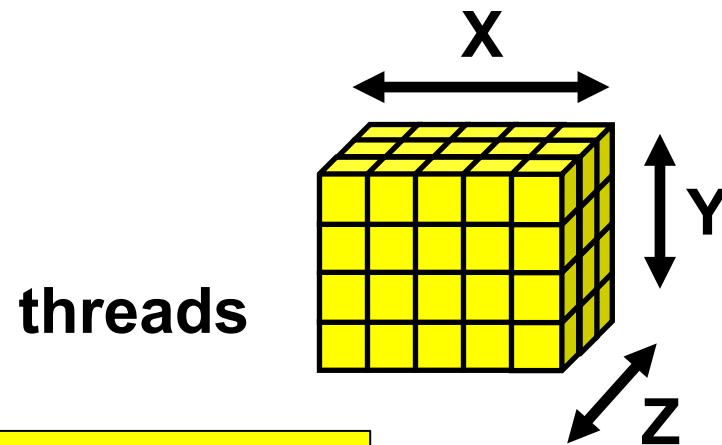
- Number of blocks in each dimension is
 - `blockDim.x` ... **5**
 - `blockDim.y` ... **4**
 - `blockDim.z` ... **3**



For 2D (and 1D blocks), simply use block dimension 1 for Z (and Y).

threadIdx Unique for Each Thread

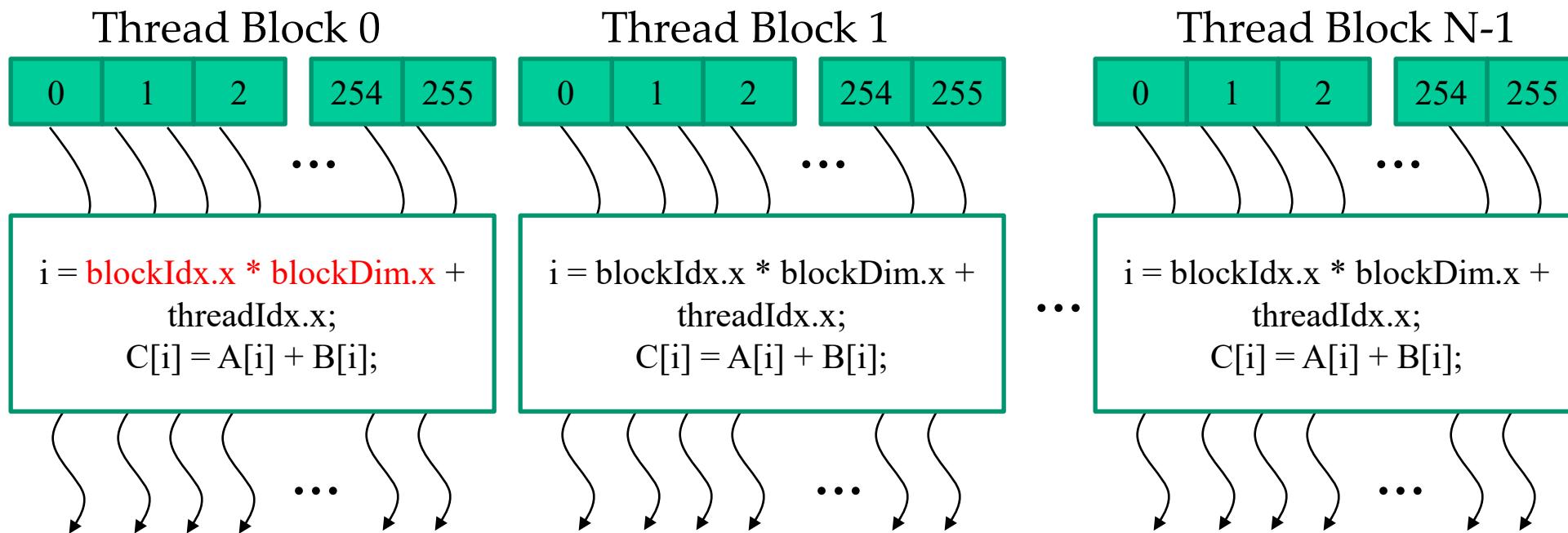
- Each thread has a unique index tuple
 - `threadIdx.x` (from 0 to $(\text{blockDim.x} - 1)$)
 - `threadIdx.y` (from 0 to $(\text{blockDim.y} - 1)$)
 - `threadIdx.z` (from 0 to $(\text{blockDim.z} - 1)$)



threadIdx tuple is unique to each thread
WITHIN A BLOCK.

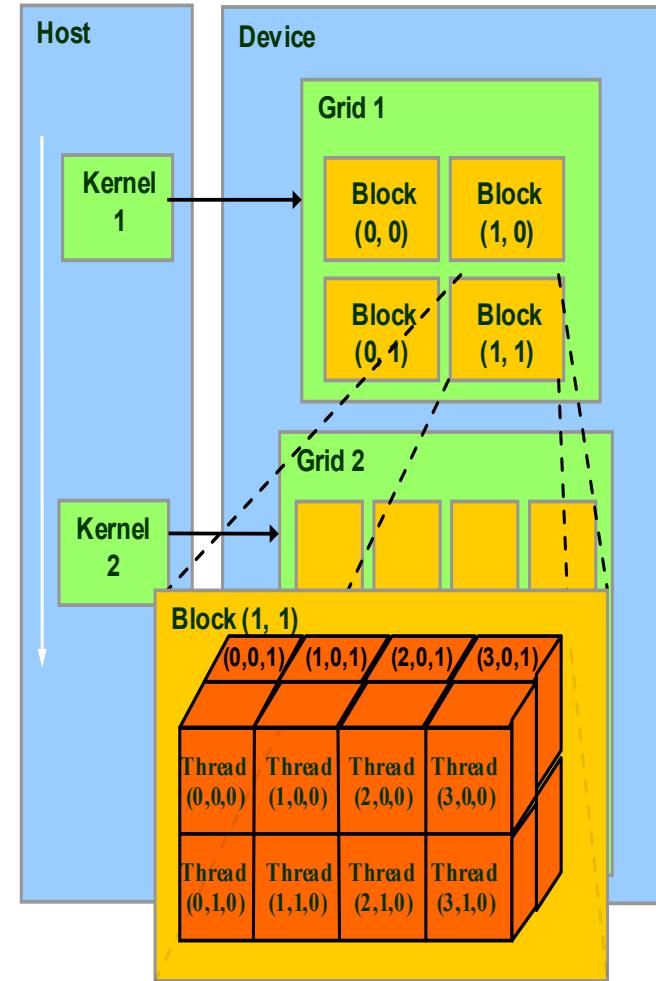
Thread Blocks: Scalable Cooperation

- Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization** (to be covered later)
- Threads in different blocks cooperate less.

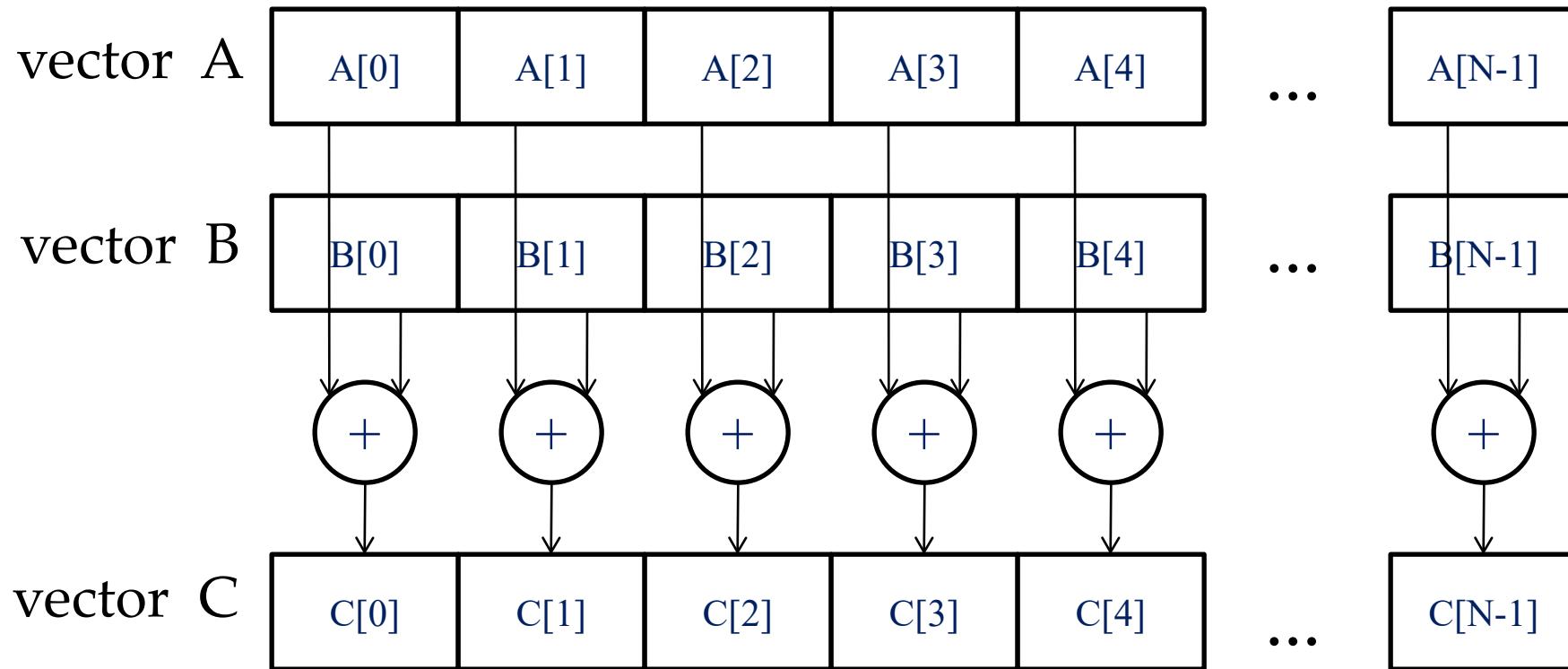


blockIdx and threadIdx

- Thread block and thread organization
 - simplifies memory addressing
 - when processing multidimensional data
- Image processing
- Vectors, matrices, tensors
- Solving PDEs on volumes
- ...



Vector Addition – Conceptual View



Vector Addition – Traditional C Code

```
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C, int n)
{
    for (i = 0, i < n, i++)
        C[i] = A[i] + B[i];
}

int main()
{
    // Memory allocation for A_h, B_h, and C_h
    // I/O to read A_h and B_h, N elements
    ...
    vecAdd(A_h, B_h, C_h, N);
}
```

Heterogeneous Computing: vecAdd Host Code

```
#include <cuda.h>
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n* sizeof(float);
    float *A_d, *B_d, *C_d;
    ...
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

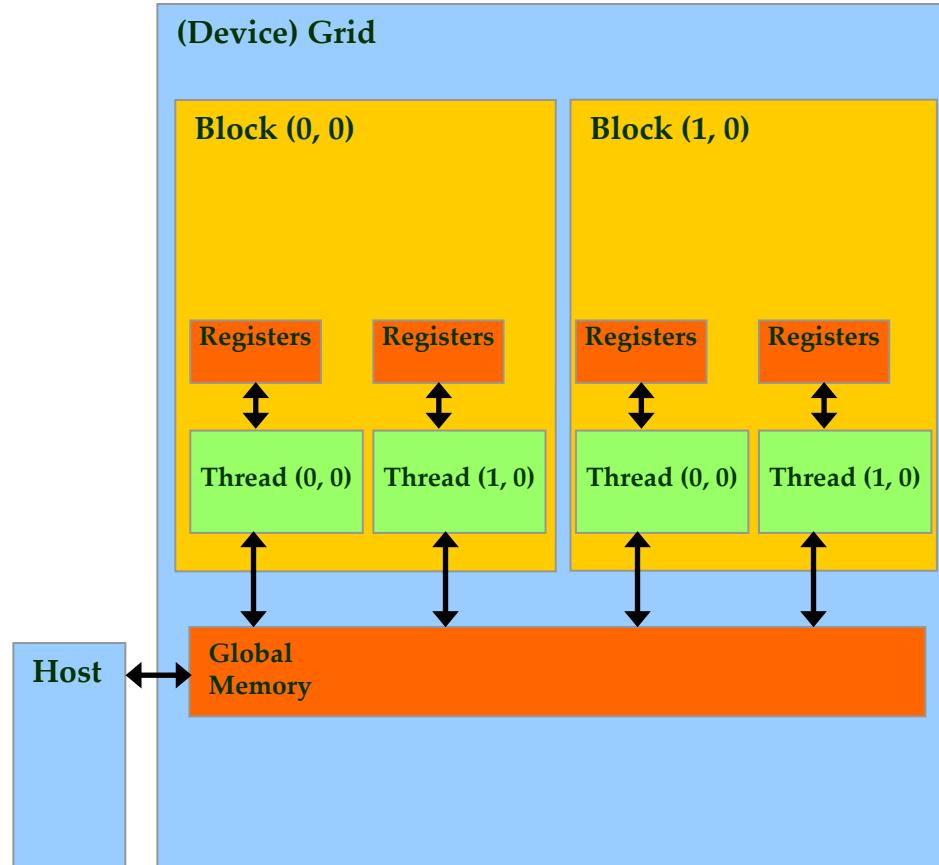
2. // Kernel launch code - to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```

Partial Overview of CUDA Memories

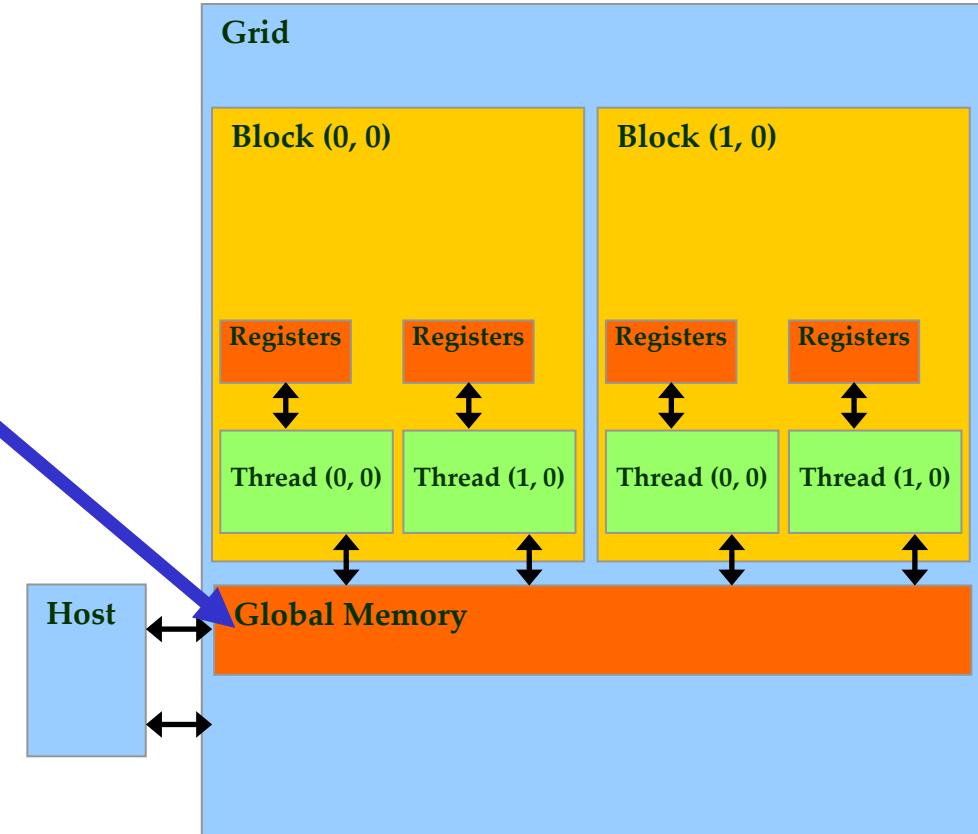
- Device code can:
 - R/W per-thread **registers**
 - R/W per-grid **global memory**
- Host code can
 - Transfer data to/from per grid **global memory**

We will cover more later.



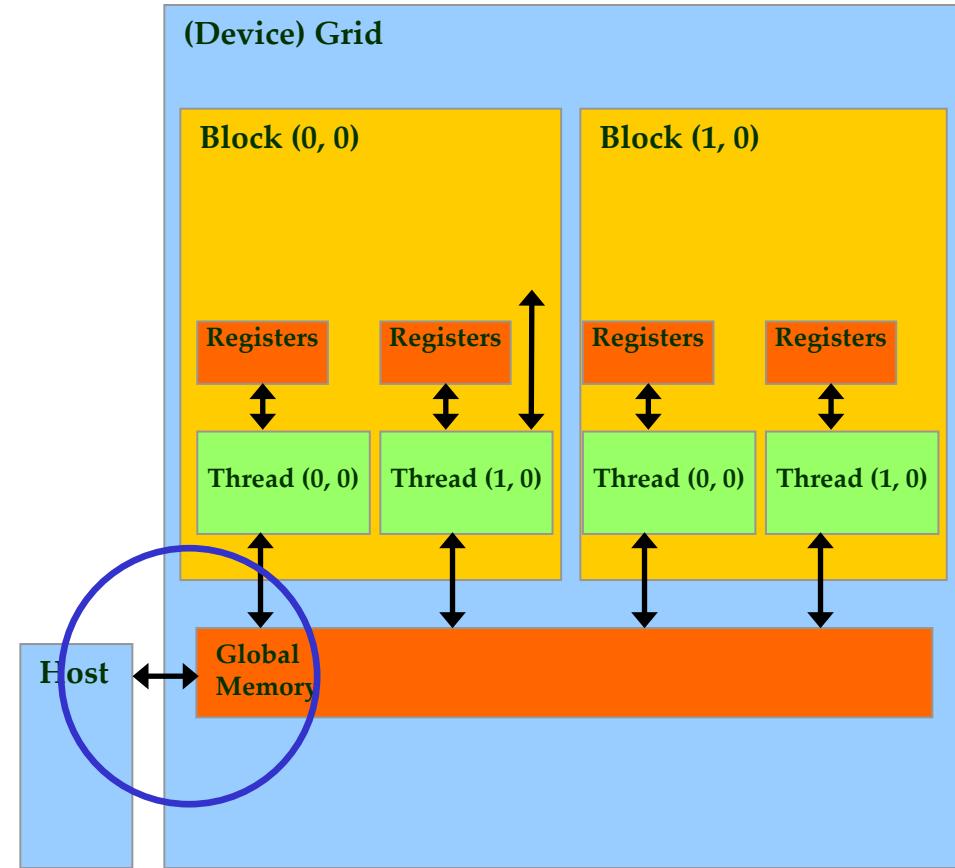
CUDA Device Memory Management API functions

- `cudaMalloc()`
 - Allocates object in the device global memory
 - Two parameters
 - **Address of a pointer** to the allocated object
 - **Size of** the allocated object in terms of bytes
- `cudaFree()`
 - Frees object from device global memory
 - **Pointer** to freed object



Host-Device Data Transfer API functions

- `cudaMemcpy()`
 - memory data transfer
 - Requires four parameters
 - Pointer to destination
 - Pointer to source
 - Number of bytes copied
 - Type/Direction of transfer



```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float *A_d, *B_d, *C_d;

1. // Transfer A and B to device memory
    // (error-checking omitted)
    cudaMalloc((void **) &A_d, size);
cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    // Allocate device memory for
    cudaMalloc((void **) &C_d, size);

2. // Kernel invocation code - to be shown later
    ...
3. // Transfer C from device to host
cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
    // Free device memory for A, B, C
    cudaFree(A_d); cudaFree(B_d); cudaFree(C_d);
}
```

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x ;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vectAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
}
```

Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vecAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0),256>>>(A_d, B_d, C_d, n);
}
```

Host Code

More on Kernel Launch

Equivalent Host Code

```
int vecAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    dim3 DimGrid(n/256, 1, 1);
    if (0 != (n % 256)) { DimGrid.x++; }
    dim3 DimBlock(256, 1, 1);

    vecAddKernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);
}
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vecAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    dim3 DimGrid(ceil(n/256), 1, 1);
    dim3 DimBlock(256, 1, 1);
    vecAddKernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);
}
```

A Number of blocks per dimension

B Number of threads per dimension in a block

C Unique block # in x dimension

D Number of threads per block in x dimension

E Unique thread # in x dimension in the block

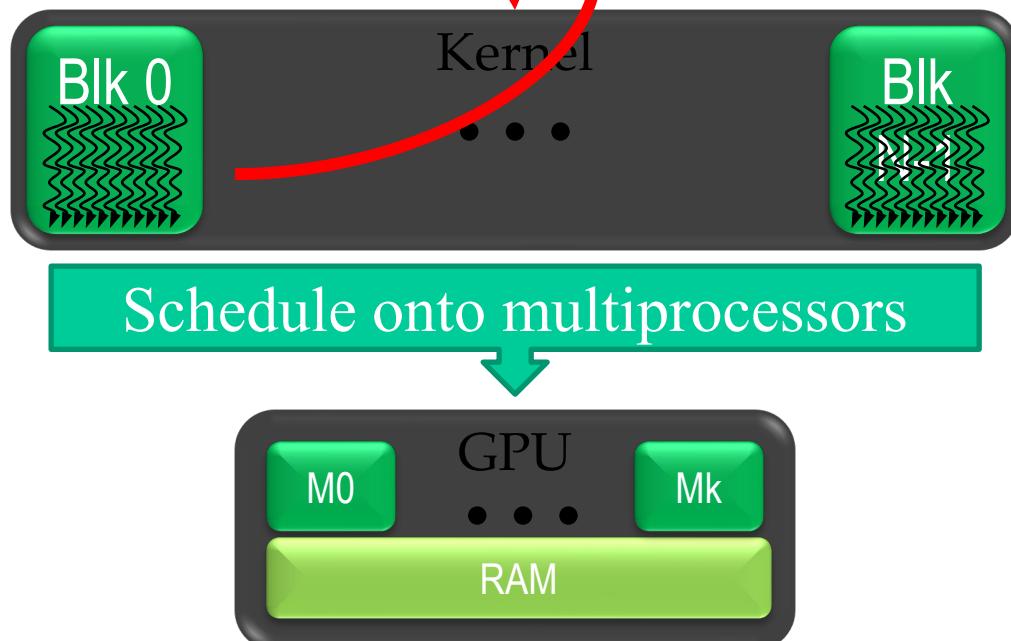
Kernel execution in a nutshell

```
__host__
void vecAdd()
{
    dim3 DimGrid(ceil(n/256.0),1,1);
    dim3 DimBlock(256,1,1);

    vecAddKernel<<<DimGrid,DimBlock>>>
    (A_d,B_d,C_d,n);
}
```

```
__global__
void vecAddKernel(float *A_d,
float *B_d, float *C_d, int n)
{
    int i = blockIdx.x * blockDim.x
            + threadIdx.x;

    if( i < n ) C_d[i] = A_d[i]+B_d[i];
}
```



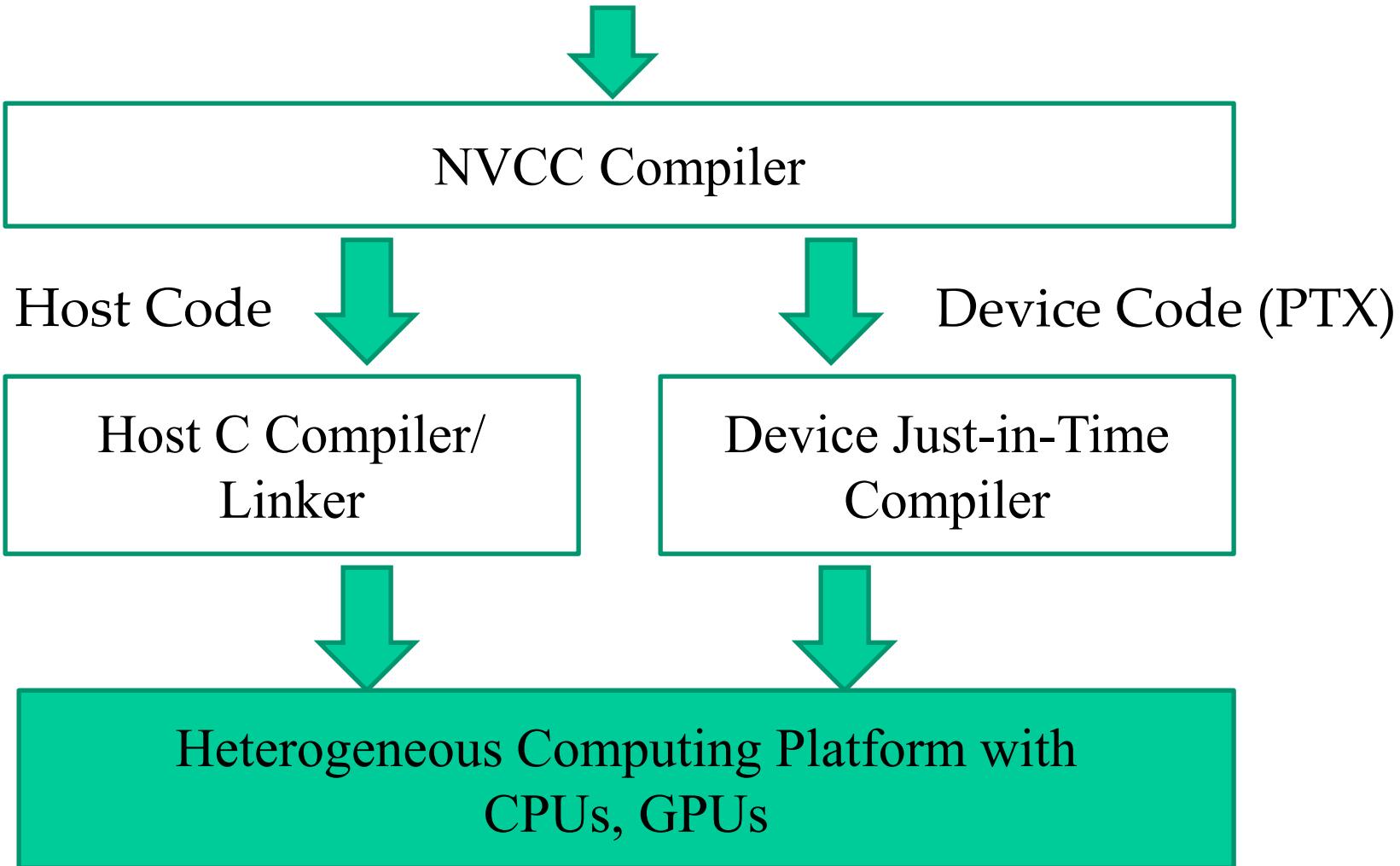
More on CUDA Function Declarations

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

- `__global__` defines a kernel function
 - Each “`__`” consists of two underscore characters
 - A kernel function must return `void`
- `__device__` and `__host__` can be used together

Compiling A CUDA Program

Integrated C programs with CUDA extensions



Problem Solving

- Consider the following code:

```
kernel<<VECTOR_N, ELEMENT_N>>>(d_C, d_A, d_B, ELEMENT_N);
```

- Q: How many CUDA threads are in each block as the result of the following kernel call?
- A: **ELEMENT_N**
- Q: How many CUDA threads will be created as the result of the following kernel call?
- A: **VECTOR_N * ELEMENT_N**

Problem Solving

- Q: For a vector addition, assume that the vector length is 16000, each thread calculates 8 output elements, and the thread block size is 256 threads. The programmer configures the kernel launch to have a minimal number of thread blocks to cover all output elements. How many **threads** will be in the **grid**?
- A:
 - How many threads do we need? $16000/8 = 2000$
 - How many blocks of threads do we need to run 2000 threads?
 $\text{ceil}(2000/256) = 8$
 - Thus, how many threads will be running? $8 * 256 = 2048$

Problem Solving

- Q: A CUDA kernel is launched with 512 thread blocks each of which has 256 threads. If a variable is declared as a local variable in the kernel, how many versions of the variable will be created through the lifetime of the execution of the kernel?
- A:
 - How many threads will be created? $512 * 256 = 131072$
 - So, there will be as many copies of the local variable, one in each thread.



**ANY MORE QUESTIONS?
READ CHAPTER 2**



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 1: Introduction



Before We Get Started

- Welcome to the course!
- The course is taught in mixed-mode (in-person and on-line)
 - Lectures are in-class and are streamed in real-time via zoom
 - They also will be recorded and posted on-line
 - Labs (MPs) & Projects are on your own out of class activities
 - Midterm exams are on-line
- Lecture slides will be posted on-line prior to the lecture on the course's wiki page
 - <https://wiki.illinois.edu/wiki/display/ECE408>



People

Instructor: **Prof. Volodymyr Kindratenko** (kindrtnk@illinois.edu)

TAs: **Xiaoyu Ma** (xiaoyum2@illinois.edu) (Labs TA)
 Xiyue Zhu (xiyuez2@illinois.edu) (Project TA)
 Huili Tao (huilit2@illinois.edu)

RAI administrator: **Andrew Schuh** (aschuh@illinois.edu)

About Prof. V. Kindratenko

- Ph.D. from University of Antwerp, Belgium, 1997
- At NCSA since 1997
 - Past: Director of Innovative Systems Lab
 - Current: Director of the Center for AI Innovation
- Research: Computing Systems, HPC, Computational Accelerators (FPGAs, GPUs), ML systems & applications



AC – first GPU
HPC cluster built in
2008 (used to teach
this course too)

- 32 S1070 GPUs



HAL – first AI-oriented cluster
built in 2018

- 64 V100 GPUs



Course Goals

- Learn to program massively parallel processors and achieve
 - High performance
 - Functionality and maintainability
 - Scalability across future generations
- Technical subjects
 - Parallel programming basics
 - Principles and patterns of parallel algorithms
 - Programming API, tools and techniques
 - Processor architecture features and constraints
 - Killer apps



Web Resources

- wiki space
 - <https://wiki.illinois.edu/wiki/display/ECE408>
 - Links to lecture slides/recordings
 - Links to labs/projects
- web board discussions in Campuswire
 - Channel for electronic announcements
 - Forum for Q&A – staff will read the board, and your classmates often have answers
- Canvas – grades & exams & lab quizzes & project reports



Grading

- Exams: 40%
 - Midterm 1: 20% -- ~ first 10+ lectures
 - Midterm 2: 20% -- ~ the remaining lectures
- Labs (Machine Problems): 35%
 - Passing Datasets 90%
 - Correct answers to questions
 - Lowest graded lab will be dropped
- Project: 25%
 - Demo/Functionality/Coding Style: ~50%
 - Performance with full functionality: ~50%
 - Detailed Rubric will be posted



Academic Honesty

- You are allowed and encouraged to discuss assignments with other students in the class. Getting verbal advice/help from people who've already taken the course is also fine.
- Any reference to assignments from previous terms or web postings is unacceptable.
- Any copying of non-trivial code is unacceptable
 - Non-trivial = more than a line or so
 - Copying includes reading someone else's code and then going off to write your own.
 - Those who have allowed copying will also be penalized.

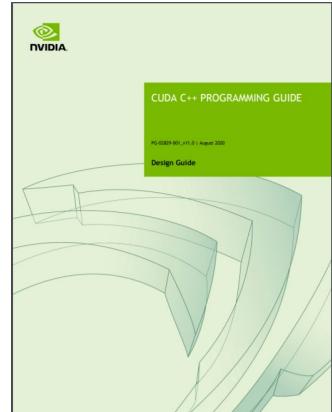
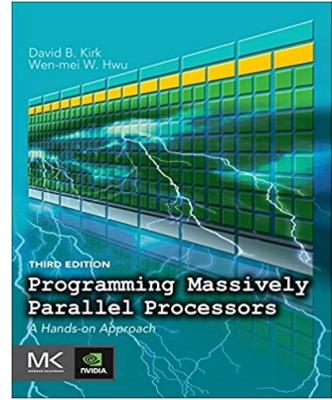


Academic Honesty (cont'd)

- Giving/receiving help on an exam is unacceptable.
- Deliberately sidestepping the lab requirements is unacceptable.
- Penalties for academic dishonesty:
 - Zero on the assignment/exam for the first occasion
 - Automatic failure of the course for repeat offenses

Text/Notes

1. D. Kirk and W. Hwu, “Programming Massively Parallel Processors – A Hands-on Approach,” Morgan Kaufman Publisher, 3rd edition, 2016, ISBN 978-0123814722
2. NVIDIA, *NVidia CUDA C Programming Guide*, version 7.5 or later (reference book)
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>



Tentative Schedule

- “Week 1”:
 - Tuesday: Lecture 1: Introduction
 - Thursday: Lecture 2: CUDA Intro
 - **Release: Lab 0, Installation, Test account**
 - “Week 2”:
 - Tuesday: Lecture 3: Data Parallelism Model
 - Thursday: Lecture 4: CUDA Memory Model
 - **Due: Lab 0, Installation, Test account**
 - **Release: Lab 1, Vector Addition**
 - Week 3:
 - Tuesday: Lecture 5: CUDA Memory Model
 - Thursday: Lecture 6: Performance Considerations
 - **Due: Lab 1, Vector Addition**
 - **Release: Lab 2, Simple Matrix Multiply**
-
- all *Labs (MPs)*
and *Project Milestones (PMs)* are
due on Fridays at
8:00pm US Central Time**



RAI

- Framework for submitting labs (MPs) and projects and grading them
 - You will receive an email with your RAI account instructions
 - Lab 0 will include all the details about deploying and using RAI client
- All Labs (MPs) and Project Milestones (PMs) are due on
 - Fridays at 8:00pm US Central Time
- Lab workflow:
 - #1: Get base code from GitHub → write code & compile & run (using RAI) → submit final code for grading (via RAI)
 - #2: Answer questions on Canvas AFTER #1 is completed
- **Lab 0 will have all the instructions to get started with RAI**

A major paradigm shift

- **In the 20th Century, we were able to understand, design, and manufacture what we can measure**
 - Physical instruments and computing systems allowed us to see farther, capture more, communicate better, understand natural processes, control artificial processes...

A major paradigm shift

- **In the 21st Century, we are able to understand, design, and create what we can compute**
 - Computational models are allowing us to see even farther, going back and forth in time, learn better, test hypothesis that cannot be verified any other way, create safe artificial processes...

Examples of Paradigm Shift

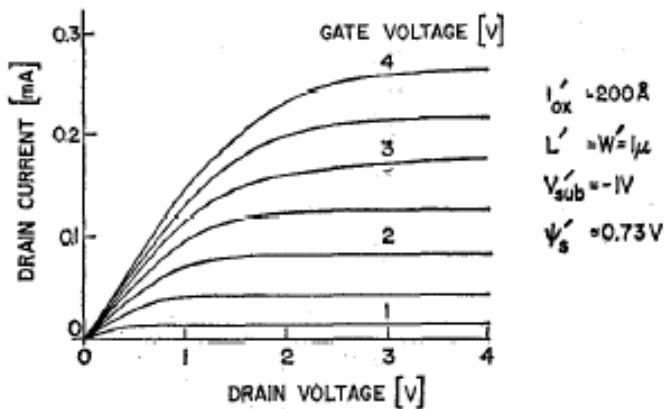
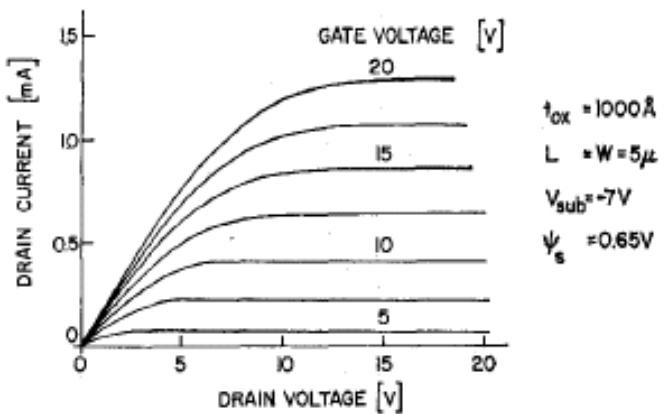
20th Century

- Small mask patterns
- Electronic microscope and Crystallography with computational image processing
- Anatomic imaging with computational image processing
- Teleconference
- GPS

21st Century

- Optical proximity correction
- Computational microscope with initial conditions from Crystallography
- Metabolic imaging sees disease before visible anatomic change
- Tele-immersion
- Self-driving cars

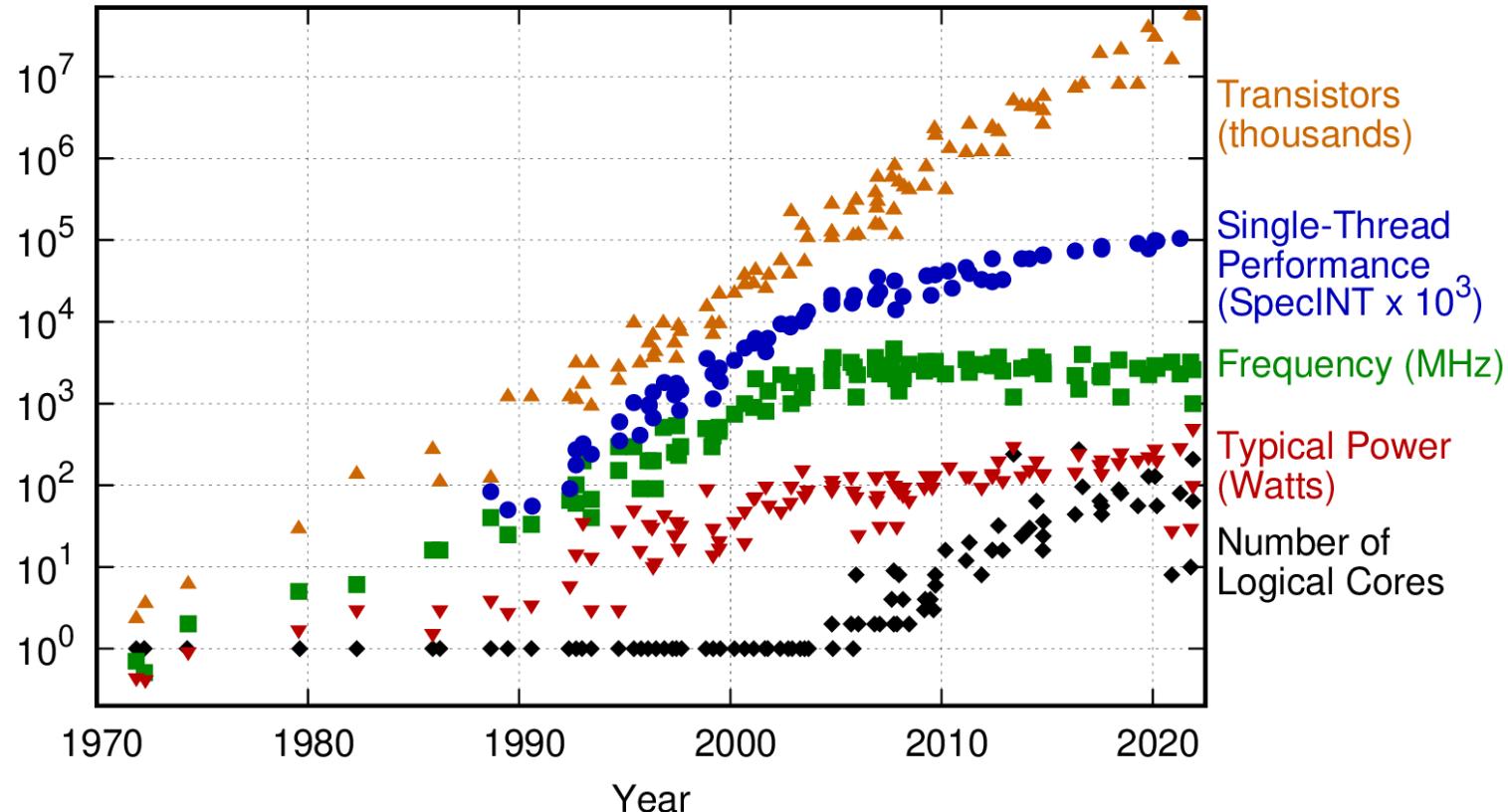
Dennard Scaling of MOS Devices



JSSC Oct 1974, page 256

- In this ideal scaling, as $L \rightarrow \alpha^* L$
 - $V_{DD} \rightarrow \alpha^* V_{DD}$, $C \rightarrow \alpha^* C$, $I \rightarrow \alpha^* I$
 - Delay = CV_{DD}/I scales by α , so $f \rightarrow 1/\alpha$
 - Power for each transistor is CV^2*f and scales by α^2
 - keeping total power constant for same chip area

Microprocessor Trends



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

<https://github.com/karlrupp/microprocessor-trend-data>

Post-Dennard Pivoting

- Multiple cores with more moderate clock frequencies
- Heavy use of vector execution
- Employ both latency-oriented and throughput-oriented cores
- 3D packaging for more memory bandwidth

Blue Waters Computing System

Operational at Illinois since 3/2013

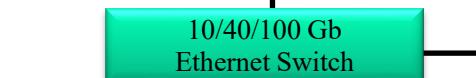
49,504 CPUs -- 4,224 GPUs



**12.5 PF
1.6 PB DRAM
\$250M**



WAN



120+ Gb/sec



Spectra Logic: 300 PBs



>1 TB/sec



Sonexion: 26 PBs

Cray XK7 Compute Node

XK7 Compute Node Characteristics

AMD Series 6200 (Interlagos)

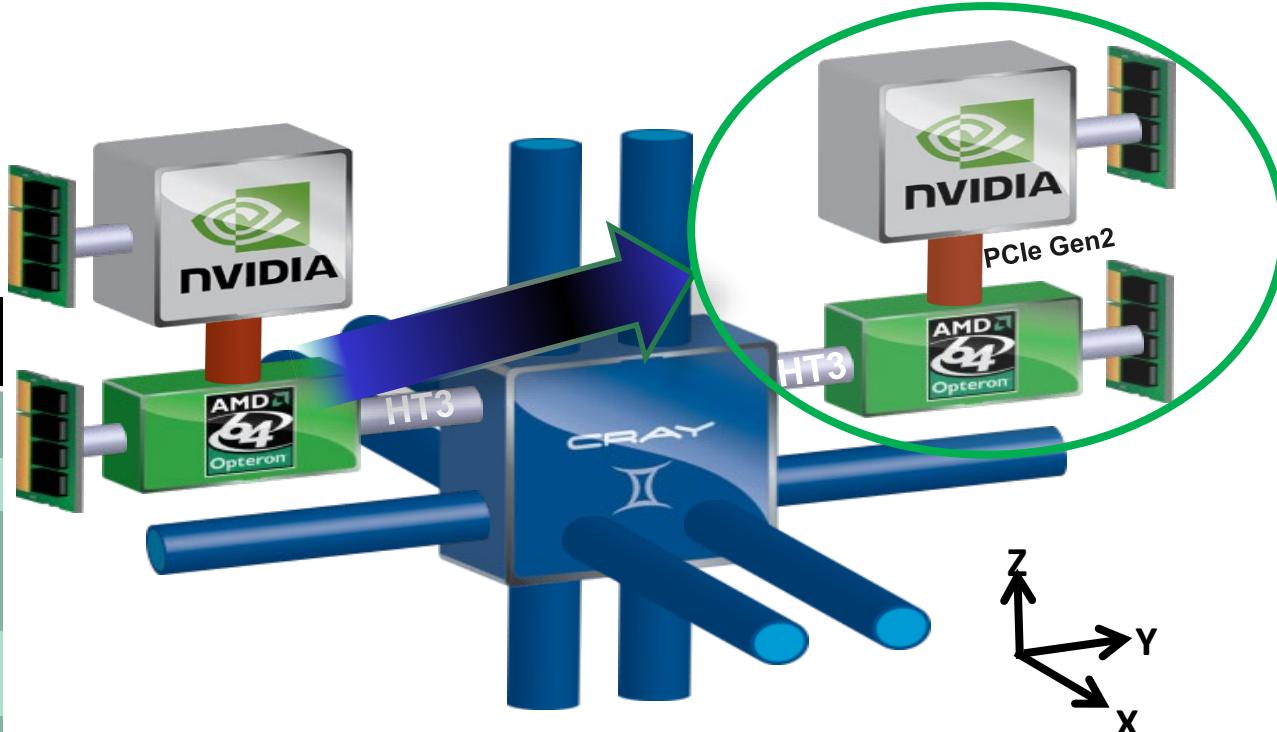
NVIDIA Kepler

Host Memory
32GB
1600 MT/s DDR3

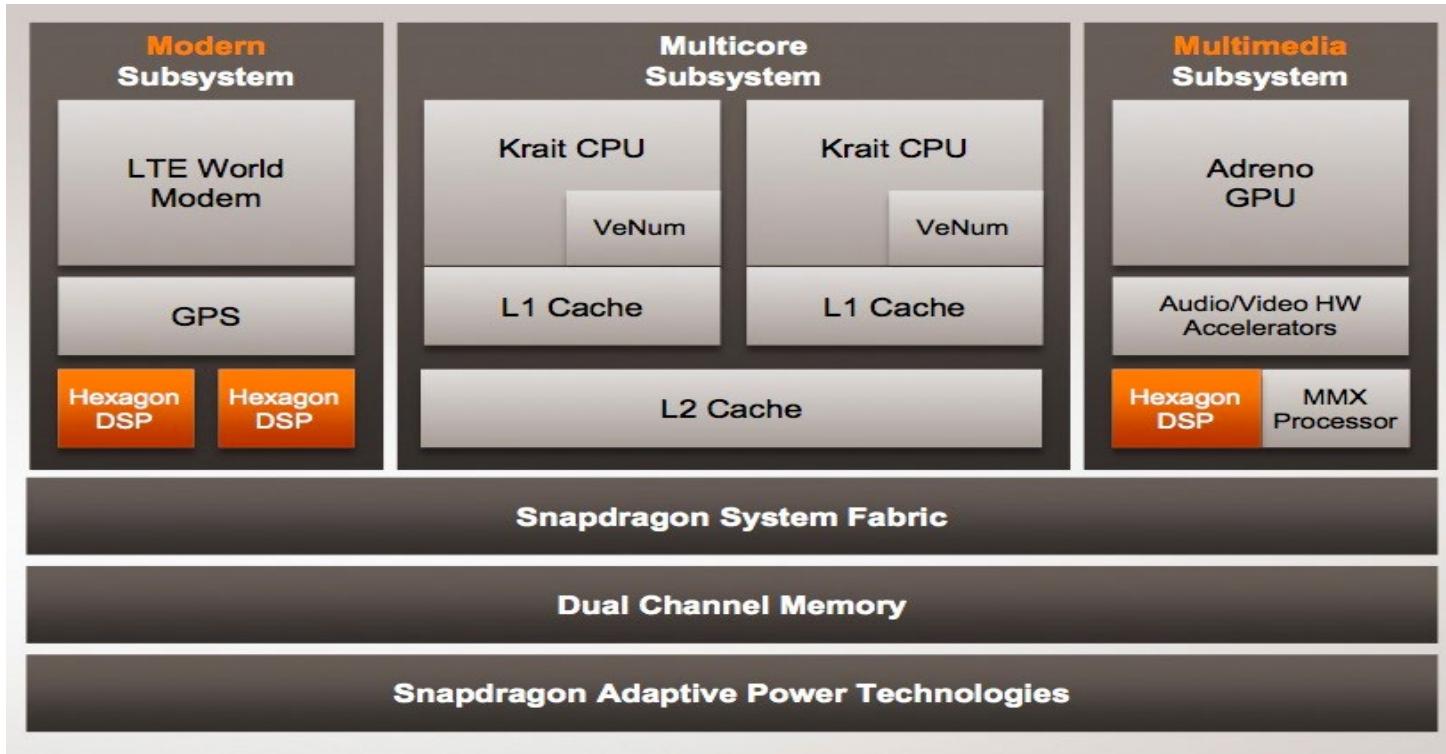
NVIDIA Tesla X2090 Memory
6GB GDDR5 capacity

Gemini High Speed Interconnect

Keplers in final installation

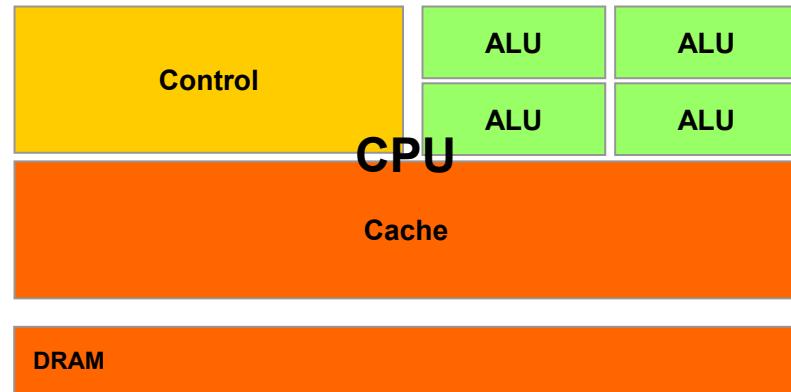


Qualcomm SoC for Mobile



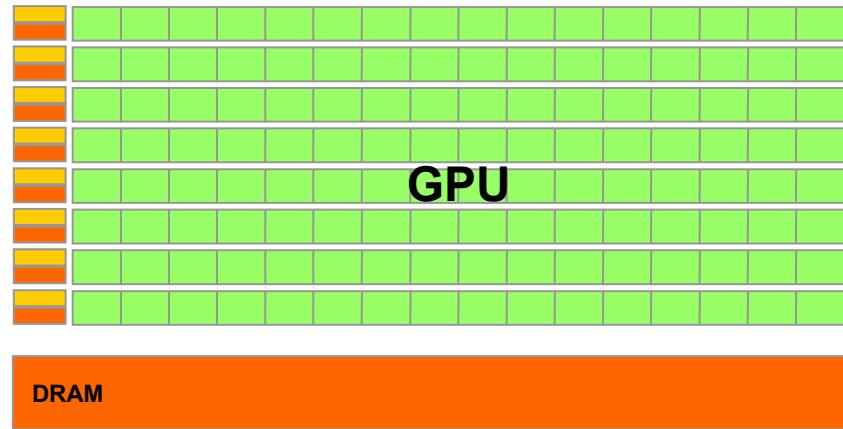
CPUs: Latency Oriented Design

- High clock frequency
- Large caches
 - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
 - Branch prediction for reduced branch latency
 - Data forwarding for reduced data latency
- Powerful ALU
 - Reduced operation latency



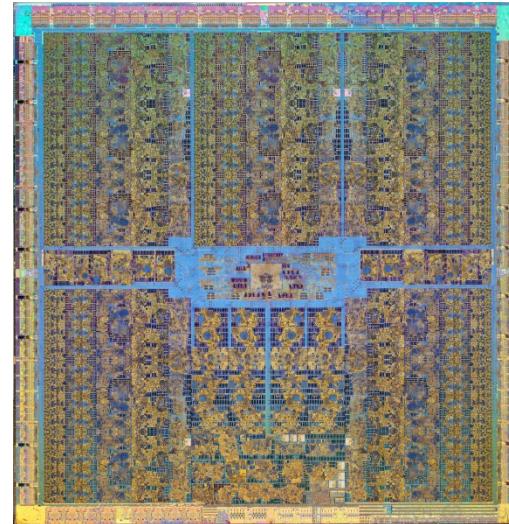
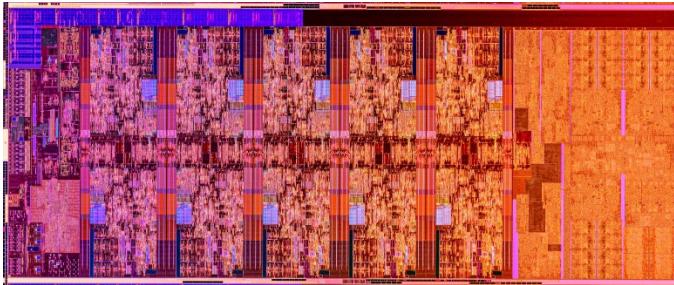
GPUs: Throughput Oriented Design

- Moderate clock frequency
- Small caches
 - To boost memory throughput
- Simple control
 - No branch prediction
 - No data forwarding
- Energy efficient ALUs
 - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies



CPU vs GPU

- 10th Gen Intel Core processor
 - 10 cores silicon
 - 14 nm process
- NVIDIA GK110
 - 2,880 CUDA cores
 - 28 nm process



Winning Strategies Use Both CPU & GPU

- CPUs for sequential parts where latency hurts
 - CPUs can be 10+X faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
 - GPUs can be 10+X faster than CPUs for parallel code

Heterogeneous Parallel Computing Applications

Financial Analysis

Scientific Simulation

Engineering Simulation

Data Intensive Analytics

Medical Imaging

Digital Audio Processing

Digital Video Processing

Computer Vision

Machine Learning

Electronic Design Automation

Biomedical Informatics

Statistical Modeling

Ray Tracing Rendering

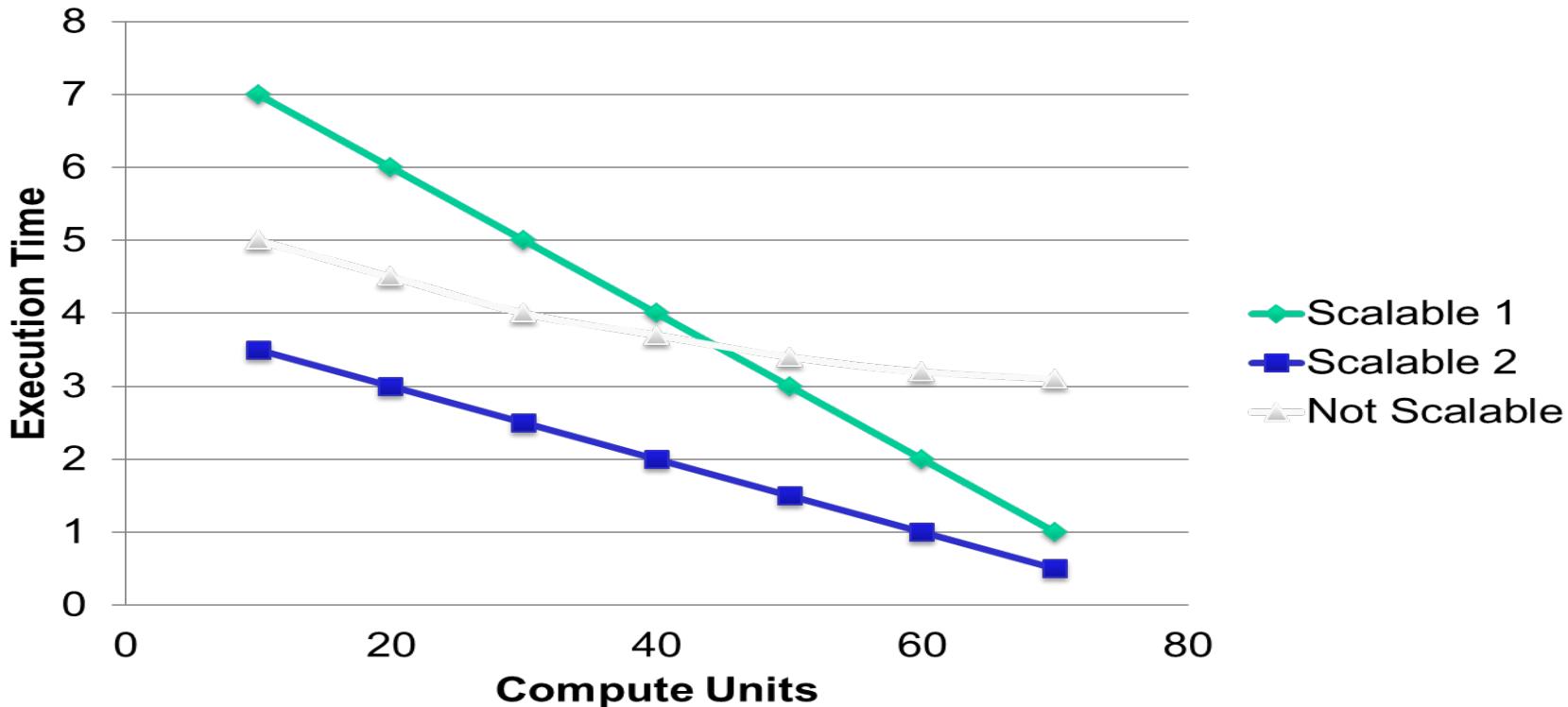
Interactive Physics

Numerical Methods

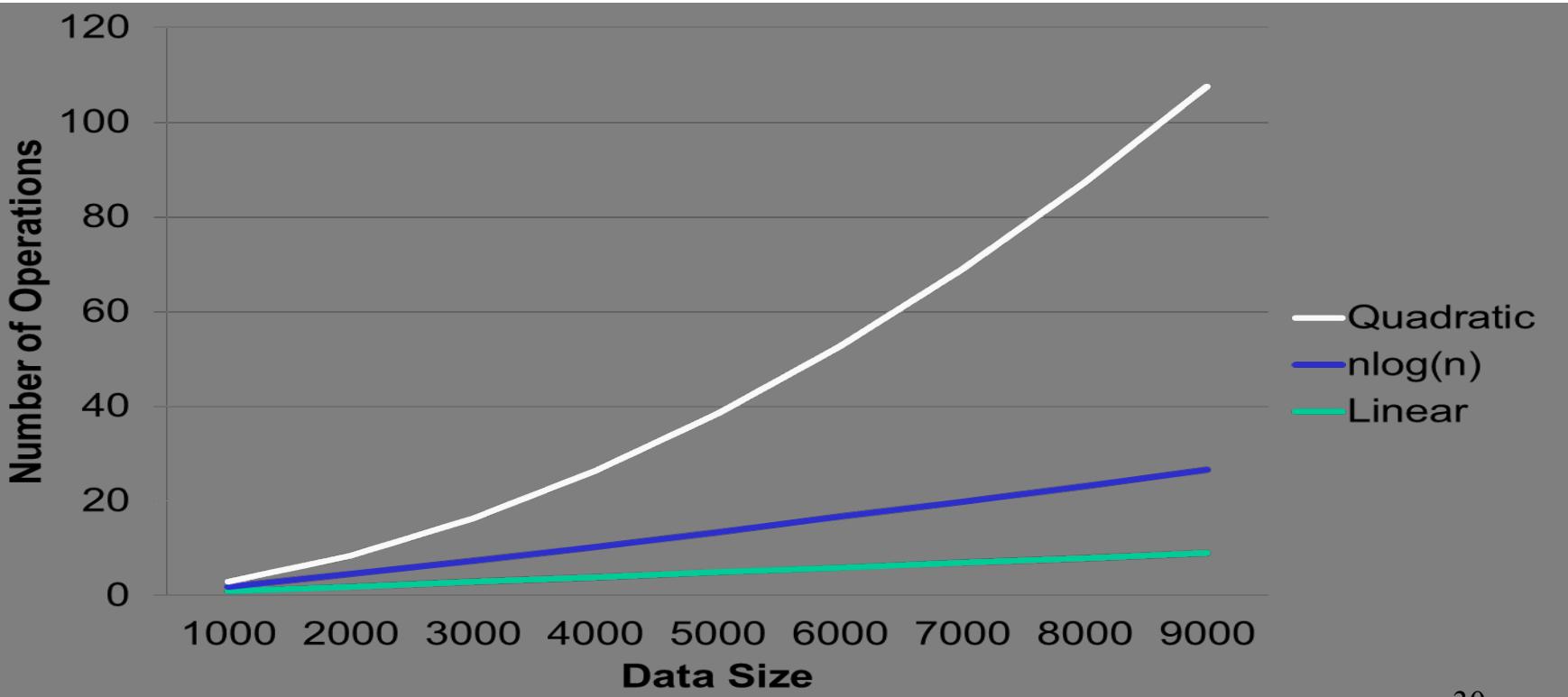
Parallel Programming Workflow

- Identify compute intensive parts of an application
- Adopt/create scalable algorithms
- Optimize data arrangements to maximize locality
- Performance Tuning
- Pay attention to code **portability, scalability, and maintainability**

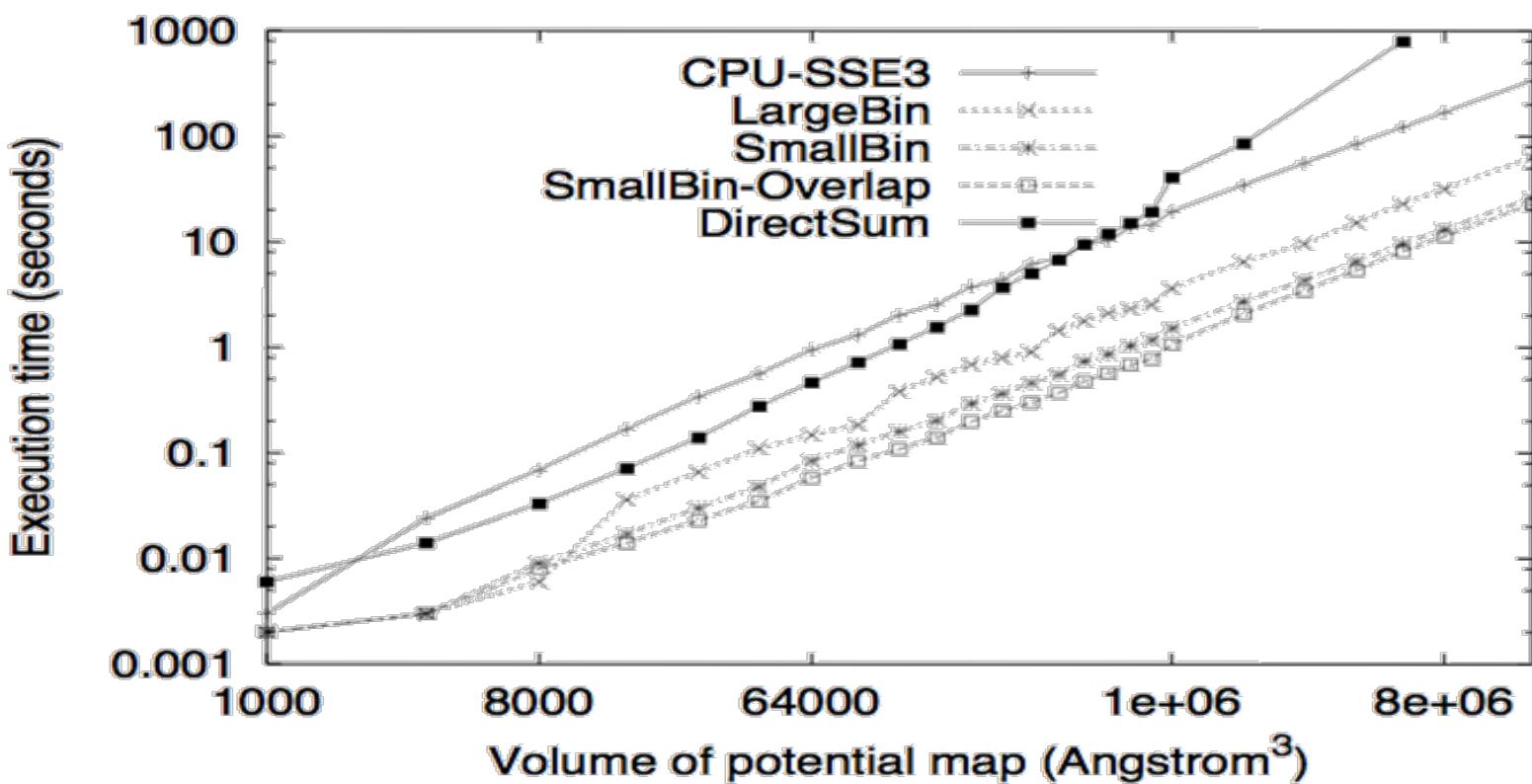
Parallelism Scalability



Algorithm Complexity and Data Scalability



A Real Example of Data Scalability Particle-Mesh Algorithms



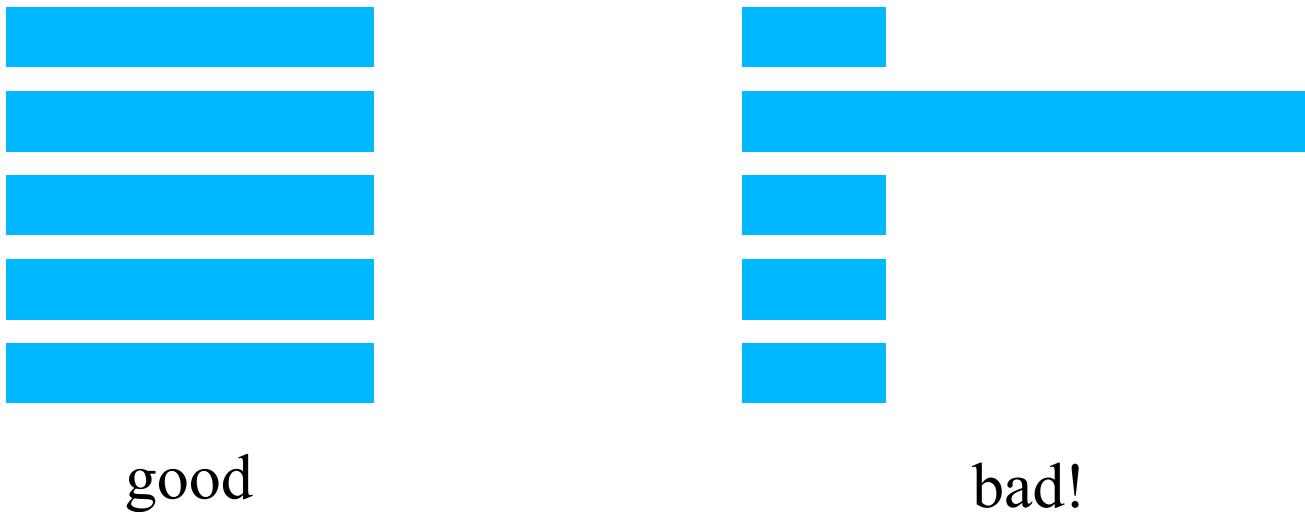


Massive Parallelism - Regularity



Load Balance

- The total amount of time to complete a parallel job is limited by the thread that takes the longest to finish



Global Memory Bandwidth

Ideal



Reality



Conflicting Data Accesses Cause Serialization and Delays

- Massively parallel execution cannot afford serialization



- Contentions in accessing critical data causes serialization

What is the stake?

- Scalable and portable software lasts through many hardware generations

*Scalable algorithms and libraries can be
the best legacy we can leave behind from
this era*



ANY MORE QUESTIONS?



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 27:
Generalizing Parallelism
and
Course Retrospective

Course Reminders

- Labs
 - All are done, all graded
- Project
 - PM3 is due this Friday
 - Final competition is due next week
- Midterm Exam 2
 - Tuesday, May 2nd 7:00pm-8:30pm US Central time
 - Via Canvas, similar to MT1
 - **See wiki for details**

Objective

- To learn terminology and concepts from the broader high-performance computing community
- To generalize some of the techniques illustrated in class for use with future codes
- To see the impact parallelism is making on the world around us

ECE 408 Retrospective: What did we do this semester?

- Elementary Computational Patterns
 - Matrix Multiply, Convolution, Reduction, Scan, Histogram, Sparse Representations
- Parallel Optimization
 - Threading, Memory Management, Coalescing, Thread Divergence, Task Management, Profiling
- Programming Systems
 - CUDA, OpenACC, (OpenCL, DPC++, Hip, OpenMP,...)

Speedup Measures the Success of Parallelization

- Let's start by defining **parallel speedup** (usually just called speedup).
- Let's say that
 - when I run my program in parallel
 - it finishes **X** times faster
 - than when I run it sequentially.
- Specifically,
 - **$X = T(\text{sequential}) / T(\text{parallel})$** , and
 - **X is the speedup** of my parallel code.
- Note that speedup assumes a fixed problem size.

Speedup Depends on the Best Sequential Code

- We have $T(\text{sequential}) / T(\text{parallel})$.
- **But how do we find $T(\text{sequential})$?**
- $T(\text{sequential})$ **should measure** the
 - **best algorithm** for a sequential machine
(may/may not be the algorithm parallelized),
 - **optimized** for a sequential machine, with
 - **no parallelism support** remnants
(no parallel overhead).

Find (Don't Write) a Competitive Baseline

- **Sequential code is** what we in Engineering call
 - the **baseline design**,
 - the alternative against which
 - we demonstrate improvements.
- As Prof. Hwu once pointed out,
 - **no one will believe** that **you** worked hard
 - to **optimize your baseline...**
 - even if you did!
- If possible, **compare someone else's best work.**

Efficiency Measures Effective Use of Resources

- Next is **parallel efficiency** (or just efficiency).
- Efficiency measures how well a code uses parallel resources.
- When executing **on P processors**,
 - **efficiency = speedup on P processors / P.**

Efficiency is Often Below 1, But Should Not be Tiny

What value should efficiency have?

- According to those paying for the machines, 1.
- According to most real applications,
 - **something non-negligible, near 1**
 - but not 1,
 - as other bottlenecks come into play.

Efficiency is Rarely Above 1

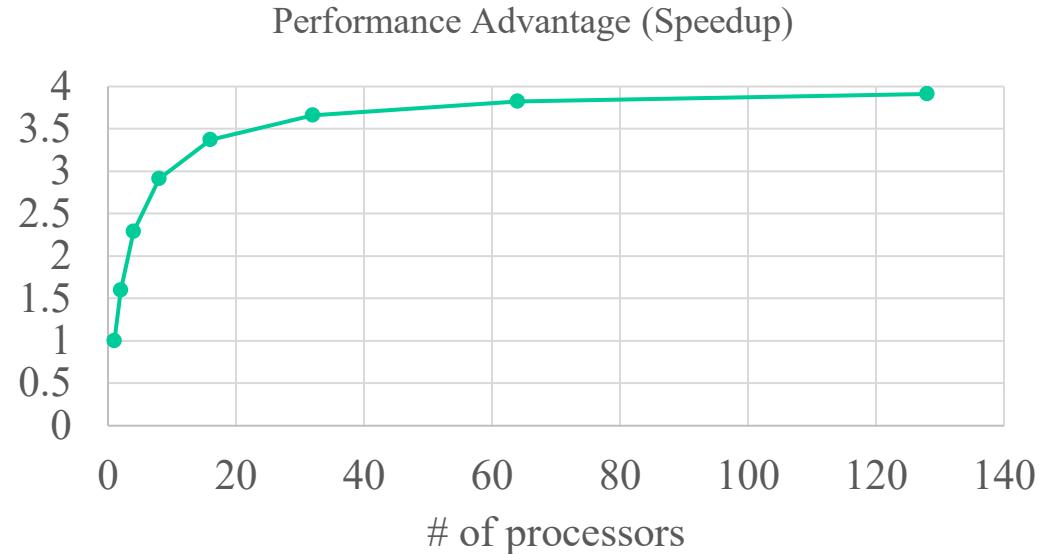
Can efficiency be >1 ?

- Rarely—called **superlinear speedup**.
- possible causes:
 - certain types of extra resources (such as caches)
 - luck (parallel search happens to find an answer more quickly).

Scalability Measures Effect of Parallel Overheads

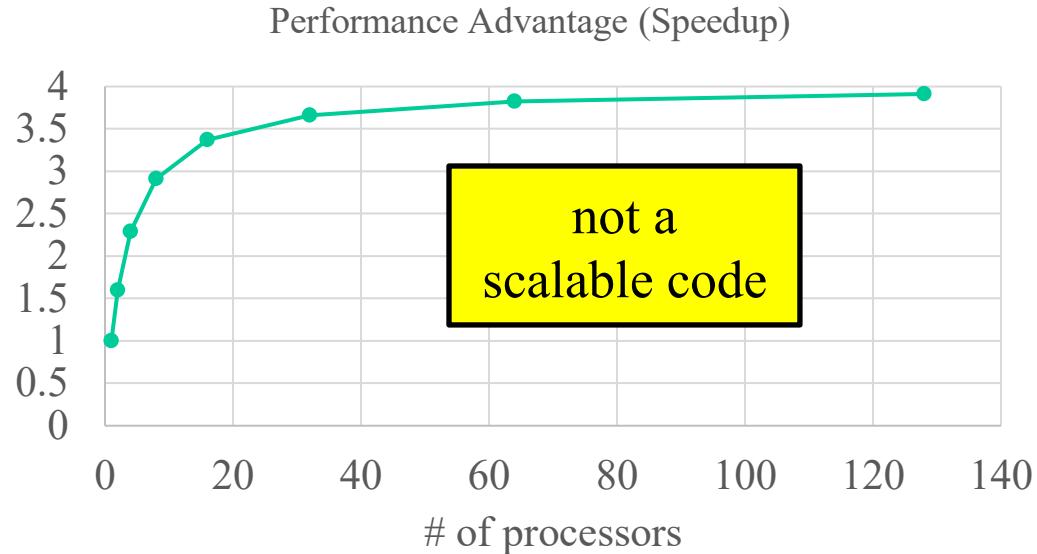
- Next, **scalability**:
 - **for how many processors is speedup linear**, or is efficiency flat?

- At some **P**, with fixed problem size, speedup will flatten out.



Good Scalability Requires Minimal Parallel Overhead

- For larger values of **P**, speedup starts to drop (unless one leaves processors idle).
- **Good scalability** means
 - **no falloff** on your machine
 - **for maximum measurable value of P.**



Efficiency Not So Meaningful When Cores Vary Widely

- **But what is P for a single GPU?**
 - 1?
 - Number of SMs?
 - Number of PEs (total)?
- We can still measure speedup,
 - but for a single GPU,
 - we **estimate efficiency**
 - **by comparing** resource **use**
 - **with** the GPU's **peak values**.
(As we've done in our class already.)

Speedup Measures Improvement for an Input Set

- Again, **speedup assumes a fixed problem size.**
 - For many applications, that's reasonable.
 - Users care about their input sets, not about hypothetical inputs.
- But that's **not always the best assumption.**

For Other Situations, We Need Different Metrics

- **Sometimes we care about throughput:**
 - frames per second for video / game quality,
 - transactions per second for databases, or
 - user operations per second for datacenters.
- And **sometimes input size**
 - **is limited** by memory
 - or by feasible runtime,
 - as in many supercomputing applications.

Scaling Problem Size with P Good for Science Apps

- Other **variants of speedup on P processors***:
- **scaled speedup**:
 - problem size is linear in **P**
 - (good scaled speedup is **1**)
- **memory-constrained speedup**:
 - biggest problem that fits in memory (which scales with **P**)
 - only works for **O(N)** algorithms

*J. P. Singh, J. L. Hennessy, A. Gupta, “Scaling Parallel Programs for Multiprocessors: Methodology and Examples,” *IEEE Computer*, 26(7):42-50, July 1993.

Problem Size Sometimes Chosen Through Practical Means

- Other **variants of speedup on P processors***:
- **time-constrained speedup**:
 - biggest problem that finishes by the time I return from lunch
 - sometimes reasonable...
 - ...but we could wait overnight for a grand challenge application?

*J. P. Singh, J. L. Hennessy, A. Gupta, “Scaling Parallel Programs for Multiprocessors: Methodology and Examples,” *IEEE Computer*, 26(7):42-50, July 1993.

Parallel Grain Size is the Work Done per Thread

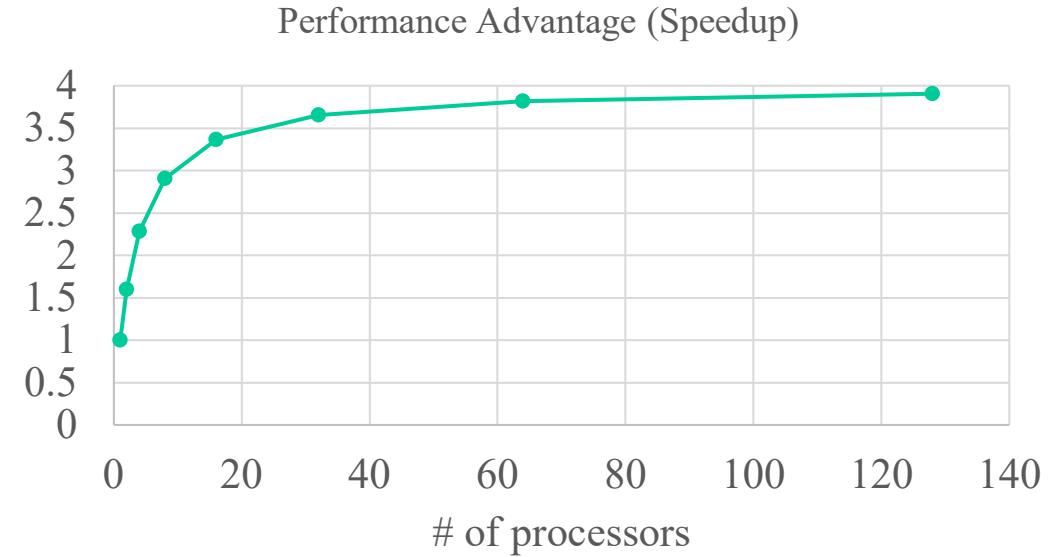
- **Parallel grain size** is work per thread (task).
 - Remember discussing what to parallelize?
 - Output elements, input elements, ...
- **Each source** of parallelism has **a natural grain size**:
 - loop body,
 - objects in a container,
 - rows/columns/blocks/elements in a matrix,
 - graph nodes/connected components.

Consider Different Sources of Parallelism

- Some sources exhibit higher work variance (and branch divergence) than others
 - conditionals/inner loops in loop body
 - complex per-object methods
 - rows in upper/lower diagonal matrix
 - matrix elements usually roughly constant
 - degree of nodes, size of connected components.
- **Be sure to consider the alternatives!**

Amdahl's Law Helps Set Expectations

- Amdahl's Law says
 - speedup is bounded above
 - by $1 / (\text{sequential fraction})$.
- For example, if you parallelize code that takes 75% of the time, you can't get more than $4 \times$ speedup.



Evaluate Your Work Intelligently and Meaningfully

- But, again, for fixed input.
- There are other ‘laws’ as well that view the problem differently.
- **So what matters most?**
 - Some apps today are missing/simplified due to resource limits.
 - Some apps become possible/more useful with bigger problem sizes.

**Fit evaluation of utility to your app,
not your app to an evaluation metric.**

A Few Useful Concepts

- Now, I'd like to go over a few useful ideas from high-performance computing.
- Most you've seen before, so I'll tie them in to what you've seen and done in our class.

Bulk Synchronous Execution Dominates Fast Computing

- The **bulk synchronous** style
 - dominates HPC and CUDA applications.
 - **Barriers separate** temporal **regions of code**
 - usually $O(100)$ lines long
 - interleaving / data **sharing occurs only within regions** (called phases).
- Why?
 - Simpler to debug regions than whole programs.
 - (similar to Stroustrup's view of classes' value).
- Bulk synchronous execution **does tend to correlate resource usage**, which is bad.

Necessary/Good Sources of Parallel Overhead

- Good ways to waste time in parallel;
 - push bits around (**communicate**)—a necessary overhead in most parallel codes
 - **do some extra work** (to avoid communicating)
 - for example, do pooling after convolution in a CNN kernel to reduce shared-to-global memory traffic
 - another: do extra adds to reduce the number of barriers, as in a Kogge-Stone scan
 - bicker about priority (**contend for shared resources**)

Bad Sources of Parallel Overhead

- Bad ways to waste time in parallel;
 - twiddle your thumbs (**wait for long-latency events**)
 - **watch others work**
 - example: branch divergence in a GPU
 - example: poor scheduling decisions
 - line up single file (**unnecessary serialization**)
 - example: coarse synchronization, lack of privatization
 - example: temporally correlated accesses to shared hardware resource
 - example: use one CUDA stream

Dynamic Load Balancing Sometimes Needed

- In our class, we have generally
 - assigned fixed work per thread.
 - Usually, this is the simplest approach, but it may lead to load imbalance.
- One common solution—**load balancing**:
 - dynamic mapping of work to threads using
 - one or more queues of work
 - pull chunk of work from a queue, do it, repeat
 - start with bigger chunks, later grab smaller
 - if queue is empty, **steal work** from another.

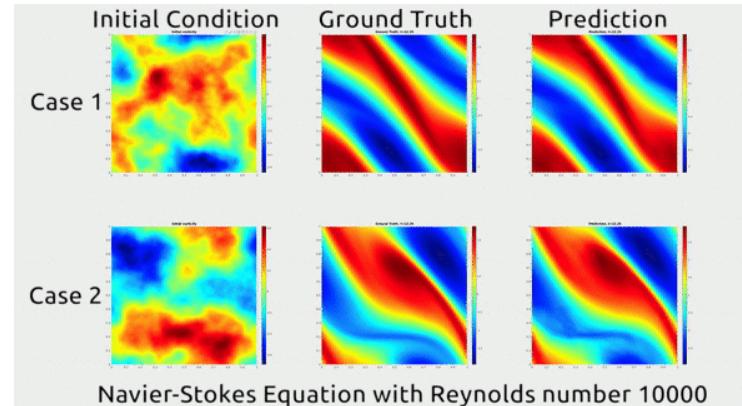
CUDA Scheduling May Need to Become More Expressive

- One last question: kernel/block scheduling.
- Most OS schedulers use **time-sharing**: try to be fair to all of the running programs.
- But if you have many processors, why pay parallel overhead?
- Use **space-sharing** instead!
- Lots of supercomputers and datacenters do.
- How are thread blocks within CUDA kernels scheduled?

Recent news from the real world (1/3)

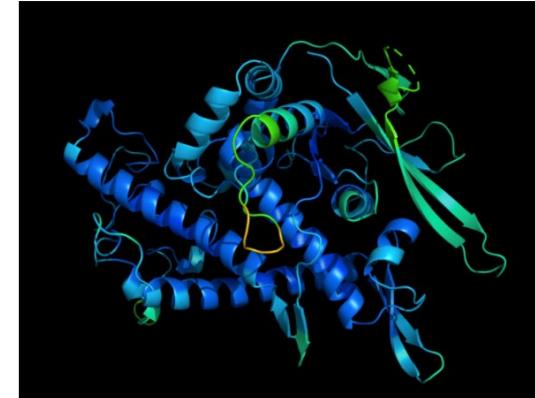
AI has cracked a key mathematical puzzle for understanding our world

Partial differential equations can describe everything from planetary motion to plate tectonics, but they're notoriously hard to solve.



'It will change everything': DeepMind's AI makes gigantic leap in solving protein structures

Google's deep-learning program for determining the 3D shapes of proteins stands to transform biology, say scientists.



A protein's function is determined by its 3D shape. Credit: DeepMind

Recent news from the real world (2/3)

Meet GPT-3. It Has Learned to Code (and Blog and Argue).

The latest natural-language system generates tweets, pens poetry, summarizes emails, answers trivia questions, translates languages and even writes its own computer programs.

One of his experiments involved a pop psychologist, Scott Barry Kaufman. The system took in Mr. Kaufman's name and a topic for discussion: creativity. Then, when asked "How do we become more creative?" GPT-3 responded instantly:

I think creative expression is a natural byproduct of growing up in a diverse world. The more diverse the world is, the more you get exposed to different people, to different opportunities, to different places and to different challenges. And the more diverse that is, the more likely you'll be to be able to put the dots together to form something new. And in many ways, I think if you want to be creative, you have to go for it. If you want to be a writer, you have to write, if you want to be a musician, you have to create music, if you want to be a comedian, you have to create comedy, and the more you create, the more likely it is that you'll stumble onto some interesting stuff.

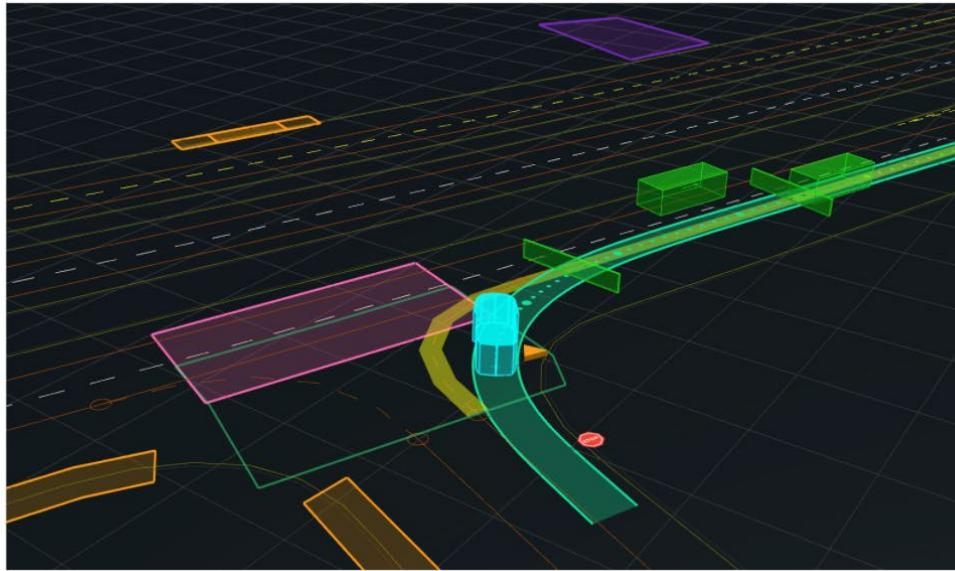
Later, when Mr. Wrigley [posted the paragraph on Twitter](#), somebody looped in the real Scott Barry Kaufman. He was stunned. "[It definitely sounds like something I would say](#)," the real Mr. Kaufman tweeted, later adding, "[Crazy accurate A.I.](#)"

Recent news from the real world (3/3)

April 28, 2020

Off road, but not offline: How simulation helps advance our Waymo Driver

TECHNOLOGY



Gaining 100+ years of experience in one day

Simulation is vital in the advancement of self-driving technology. At Waymo, one day in simulation is like driving more than 100 years in the real world. In simulation, we drive around 20 million miles a day, expanding the scale and complexity of our experience. True to our Alphabet heritage, our team has world-class expertise in applying cloud technology at massive scale to advance and grow our simulation efforts. To date, we have driven over 15 billion miles in simulation, and we continue to increase the velocity of our learning. In simulation, we can keep learning from each of our 20 million autonomous miles on public roads, prepare for rare edge cases, explore new ideas, validate and test new software, and continue improving our rider experience.

If this is exciting to you...

- Courses in Advanced Computing: ECE 508, ECE 511, CS 533
- Computational Science: CSE 401
- Topical Courses: Bioinformatics, Machine Learning / AI, Scientific Computing, Material Science



THAT'S ALL, FOLKS!

GOOD LUCK ON THE EXAM!



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 23: Alternatives to CUDA

Accelerated Computing is no longer a question



GPU vendors include:
Nvidia
AMD
Intel
Samsung
Apple
Qualcomm
ARM
Etc....

CUDA is just one model for Compute Acceleration

OpenGL (1992)

DirectX (1995)

GPGPU (2002)

CUDA (2007)

OpenCL (2008)

OpenACC (2012)

C++AMP (2013)

RenderScript (2013)

OpenMP (2016)

Metal (2014)

SYCL (2014)

Vulkan (2016)

ROCM HIP (2016)

Existing frameworks such as MPI, TBB, OpenCV adapted to provide support.
New frameworks such as Caffe, TensorFlow, R, PyCUDA natively support acceleration.

OpenCL, HIP, OpenACC, MPI

- OpenCL: An Open Standard Acceleration API
- Heterogeneous-Computing Interface for Portability (HIP)
- OpenACC: A “Low-Code” Acceleration API
- MPI: A Large Scale, Multi-Node Parallel API

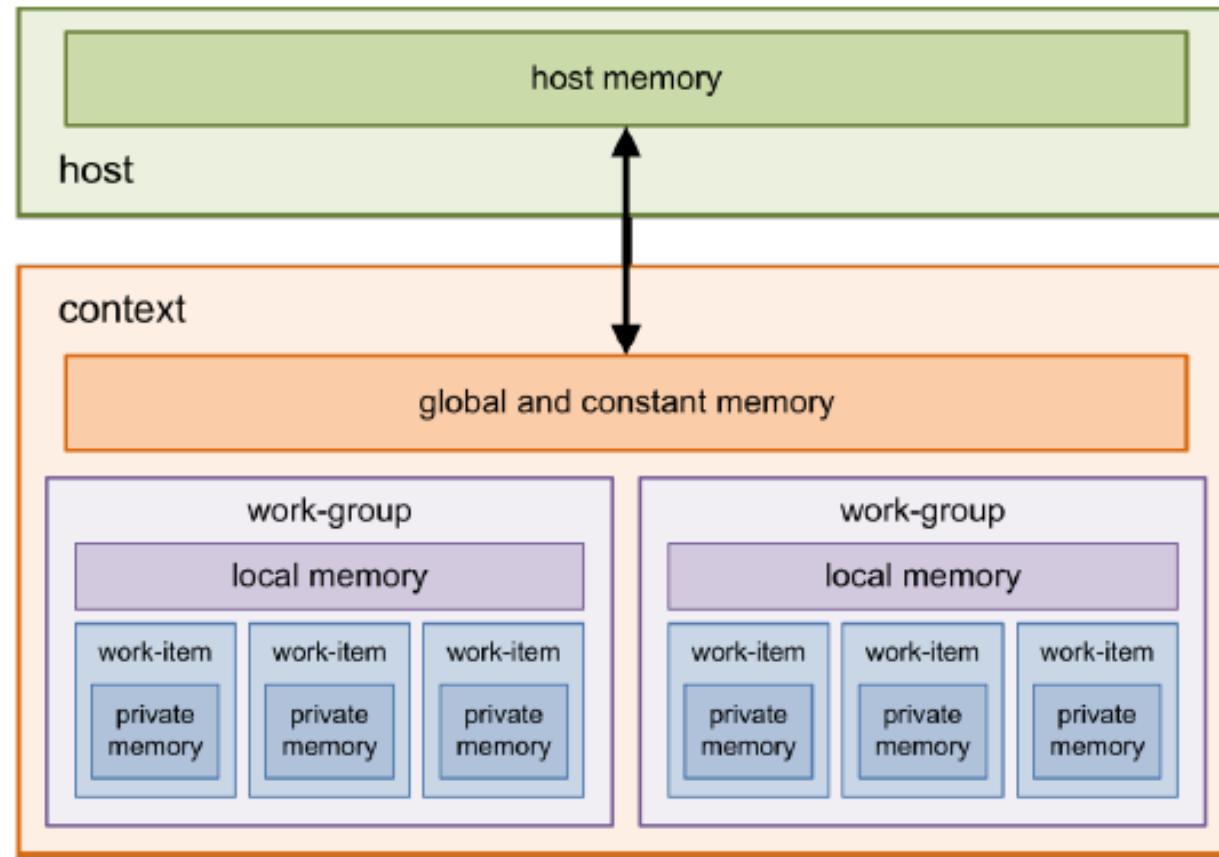
Common Traits for Acceleration APIs

- HARDWARE
 - Hierarchy of lightweight cores
 - Local scratchpad memories
 - Lack of HW coherence
 - Slow global atomics
 - Threading
- SOFTWARE
 - Kernel oriented acceleration
 - Device memory vs. Host memory
 - Software managed memory
 - Grids, Blocks, Threads
 - Bulk Synchronous Parallelism

OpenCL

- Framework for CPUs, GPUs, DSPs, FPGAs, etc. (not just Nvidia GPUs)
- Initially developed by Apple with support from AMD, IBM, Qualcomm, Intel, and Nvidia. OpenCL 1.0 launched in 2008.
- OpenCL 2.2 launched in May 2017
- Apple announces dropping of OpenCL in 2018

OpenCL Memory Model



OpenCL MatMult

- Notice similarity to CUDA
- WorkGroup similar to Block
- WorkItem similar to Thread
- local similar to shared

```
1.// Tiled and coalesced version
2._kernel void myGEMM2(int M, int N, int K, __global float* A, __global float* B, __global float* C) {
3.
4.    // Thread identifiers
5.    const int row = get_local_id(0); // Local row ID (max: TS)
6.    const int col = get_local_id(1); // Local col ID (max: TS)
7.    const int globalRow = TS*get_group_id(0) + row; // Row ID of C (0..M)
8.    const int globalCol = TS*get_group_id(1) + col; // Col ID of C (0..N)
9.
10.   // Local memory to fit a tile of TS*TS elements of A and B
11.   __local float Asub[TS][TS];
12.   __local float Bsub[TS][TS];
13.
14.   // Initialise the accumulation register
15.   float acc = 0.0f;
16.   // Loop over all tiles
17.   const int numTiles = K/TS;
18.   for (int t=0; t<numTiles; t++) {
19.
20.       // Load one tile of A and B into local memory
21.       const int tiledRow = TS*t + row;
22.       const int tiledCol = TS*t + col;
23.       Asub[col][row] = A[tiledCol*M + globalRow];
24.       Bsub[col][row] = B[globalCol*K + tiledRow];
25.
26.       // Synchronise to make sure the tile is loaded
27.       barrier(CLK_LOCAL_MEM_FENCE);
28.
29.       // Perform the computation for a single tile
30.       for (int k=0; k<TS; k++)
31.           acc += Asub[k][row] * Bsub[col][k];
32.
33.       // Synchronise before loading the next tile
34.       barrier(CLK_LOCAL_MEM_FENCE);
35.   }
36.
37.   // Store the final result in C
38.   C[globalCol*M + globalRow] = acc;
39. }
```

HIP

- Heterogeneous-Computing Interface for Portability (HIP)
 - C++ dialect designed to ease conversion of CUDA applications to portable C++ code.
 - Provides a C-style API and a C++ kernel language.
 - The C++ interface can use templates and classes across the host/kernel boundary.
- HIP code can run on AMD hardware (through the HCC compiler) or NVIDIA hardware (through the NVCC compiler).
- The HIPify tool automates much of the conversion work by performing a source-to-source transformation from CUDA to HIP.

vectorAdd with HIP

```
__global__ void vecAdd(double *a, double *b, double *c, int n) {
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    if (id < n) c[id] = a[id] + b[id];
}

...
hipMalloc(&d_a, nbytes);
hipMalloc(&d_b, nbytes);
hipMalloc(&d_c, nbytes);

hipMemcpy(d_a, h_a, bytes, hipMemcpyHostToDevice);
hipMemcpy(d_b, h_b, bytes, hipMemcpyHostToDevice);

blockSize = 1024;
gridSize = (int)ceil((float)n/blockSize);

hipLaunchKernelGGL(vecAdd, dim3(gridSize), dim3(blockSize), 0, 0, d_a, d_b, d_c, n);
hipDeviceSynchronize( );

hipMemcpy(h_c, d_c, bytes, hipMemcpyDeviceToHost);
...
```

OpenACC

The OpenACC Application Programming Interface (API) provides a set of

- compiler directives (pragmas),
- library routines, and
- environment variables

that enable

- FORTRAN, C and C++ programs
- to execute on accelerator devices
- including GPUs and CPUs.

Pragmas Provide Extra Information

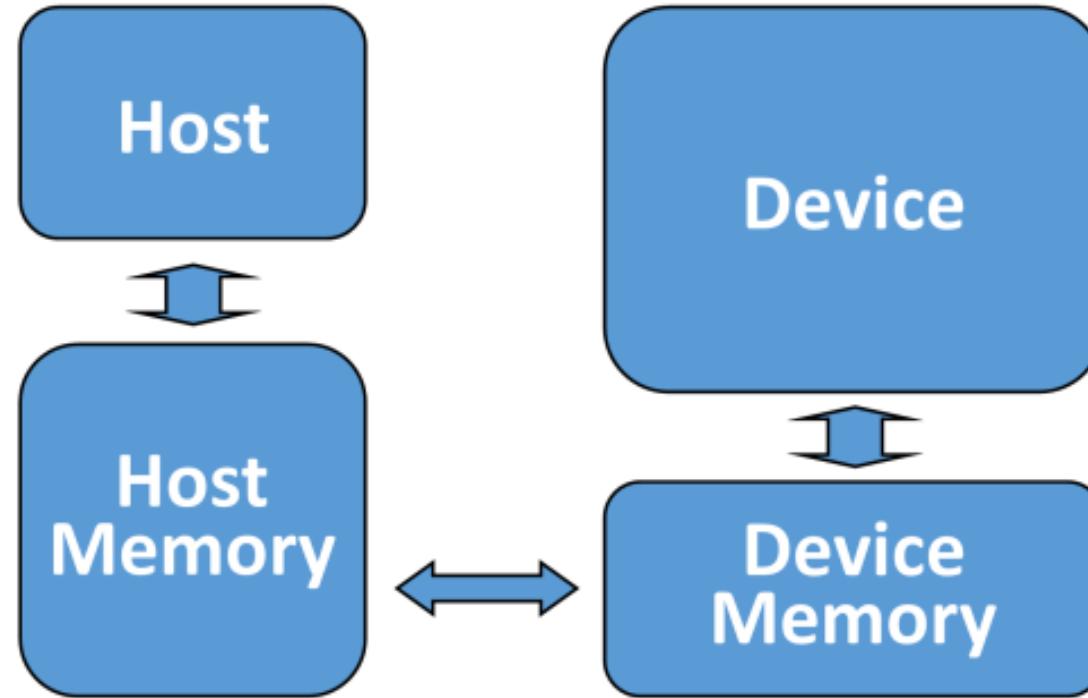
In C and C++,

- the `#pragma` directive
- provides the compiler with
- information not specified in the language.

For OpenACC, they look like this:

#pragma acc [the information goes here]

The OpenACC Abstract Machine Model



The OpenACC Directives

The diagram illustrates the mapping of three OpenACC concepts to specific directives in a C-like pseudocode:

- Manage Data Movement** points to the directive `#pragma acc data copyin(x,y) copyout(z)`.
- Initiate Parallel Execution** points to the directives `#pragma acc parallel` and `#pragma acc loop gang vector`, along with the parallel loop body.
- Optimize Loop Mappings** points to the directive `#pragma acc loop映射` (note: the text in the image shows '映射' instead of 'mappings') and the nested loop structure.

```
#pragma acc data copyin(x,y) copyout(z)
{
    ...
#pragma acc parallel
    {
        #pragma acc loop gang vector
        for (i = 0; i < n; ++i) {
            z[i] = x[i] + y[i];
            ...
        }
    }
    ...
}
```

Simple Matrix-Matrix Multiplication in OpenACC

```
1 void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2 {
3
4 #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw]) copyout(P[0:Mh*Nw])
5 for (int i=0; i<Mh; i++) {
6     #pragma acc loop
7     for (int j=0; j<Nw; j++) {
8         float sum = 0;
9         for (int k=0; k<Mw; k++) {
10             float a = M[i*Mw+k];
11             float b = N[k*Nw+j];
12             sum += a*b;
13         }
14         P[i*Nw+j] = sum;
15     }
16 }
17 }
```

Add Pragmas to Sequential Code

The **code** is

- **identical to the sequential version**
- **except for the two pragmas**
- at lines 2 and 4.

OpenACC uses the compiler directive mechanism to extend the base language.

Simple Matrix-Matrix Multiplication in OpenACC

```
1 void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw) {  
2 #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw]) copyout(P[0:Mh*Nw])  
3 for (int i=0; i<Mh; i++) {  
4     #pragma acc loop  
5     for (int j=0; j<Nw; j++) {  
6         float sum = 0;  
7         for (int k=0; k<Mw; k++) {  
8             float a = M[i*Mw+k];  
9             float b = N[k*Nw+j];  
10            sum += a*b;  
11        }  
12        P[i*Nw+j] = sum;  
13    }  
14 }  
15 }
```

tells compiler

- to execute ‘i’ loop
- (lines 3 through 14)
- in parallel on accelerator.

copyin/copyout specify

- how matrix data
- should be transferred between memories.

Simple Matrix-Matrix Multiplication in OpenACC

```
1 void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw) {  
2 #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw]) copyout(P[0:Mh*Nw])  
3 for (int i=0; i<Mh; i++) {  
4     #pragma acc loop  
5     for (int j=0; j<Nw; j++) {  
6         float sum = 0;  
7         for (int k=0; k<Mw; k++) {  
8             float a = M[i*Mw+k];  
9             float b = N[k*Nw+j];  
10            sum += a*b;  
11        }  
12        P[i*Nw+j] = sum;  
13    }  
14}  
15}
```

tells compiler

- to map ‘j’ loop
- (lines 5 through 13)
- to second level
- of parallelism on accelerator.

Motivating Goal: One Version of Code

OpenACC programmers

- can often start with a sequential version,
- then annotate their program with directives,
- leaving most kernel details and data transfers
- to the OpenACC compiler.

OpenACC code can be compiled by non-OpenACC compilers by ignoring the pragmas.

Reality is More Complicated

Reality check:

- can be **difficult to write code**
- that works **correctly and well**
- **with and without pragmas.**

Some OpenACC programs

- behave differently or even incorrectly
- if pragmas are ignored.

Pitfall: Strong Dependence on Compiler

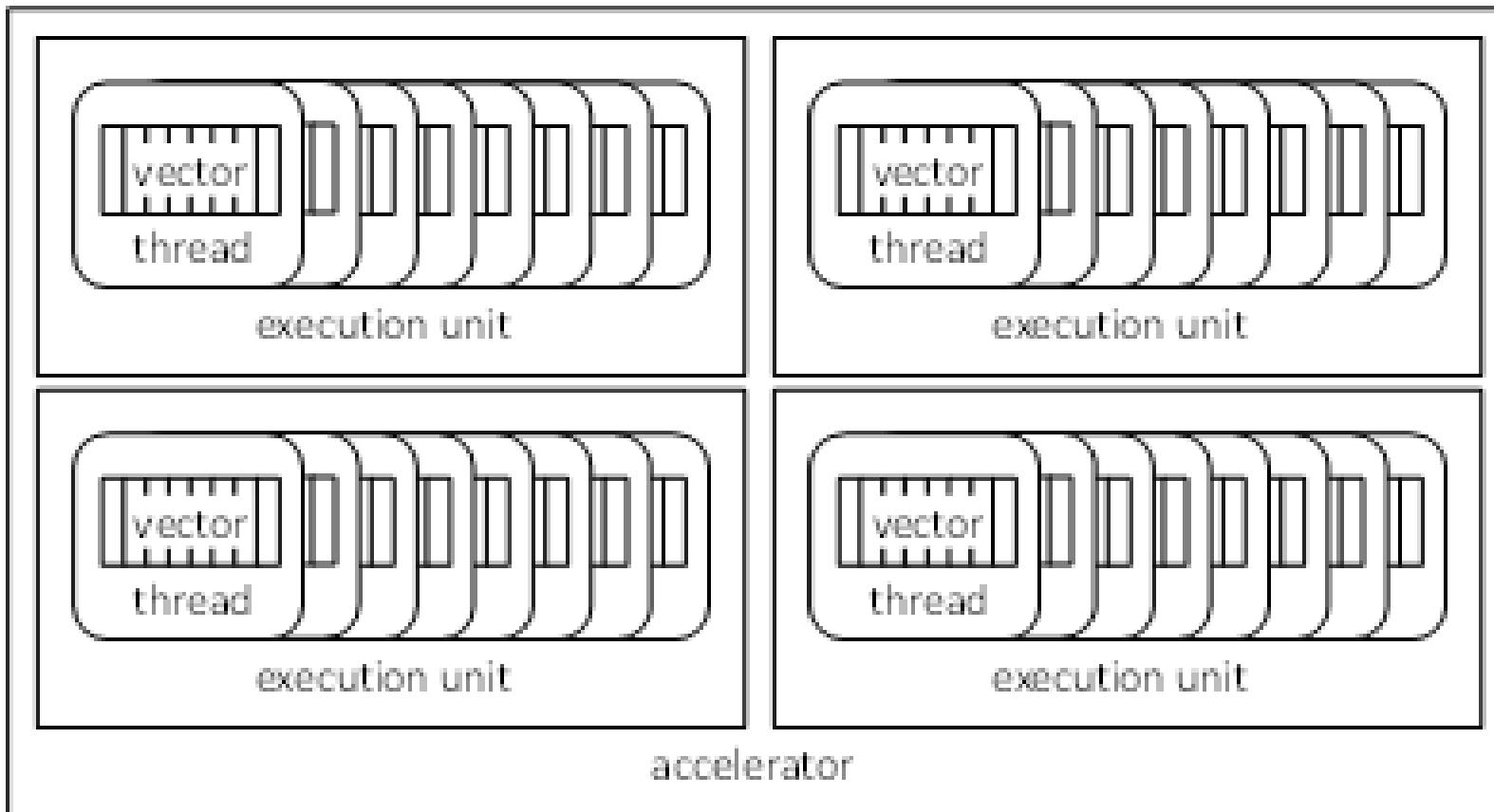
Some OpenACC pragmas

- are hints to the OpenACC compiler,
- which may or may not be able to act accordingly

Performance depends heavily

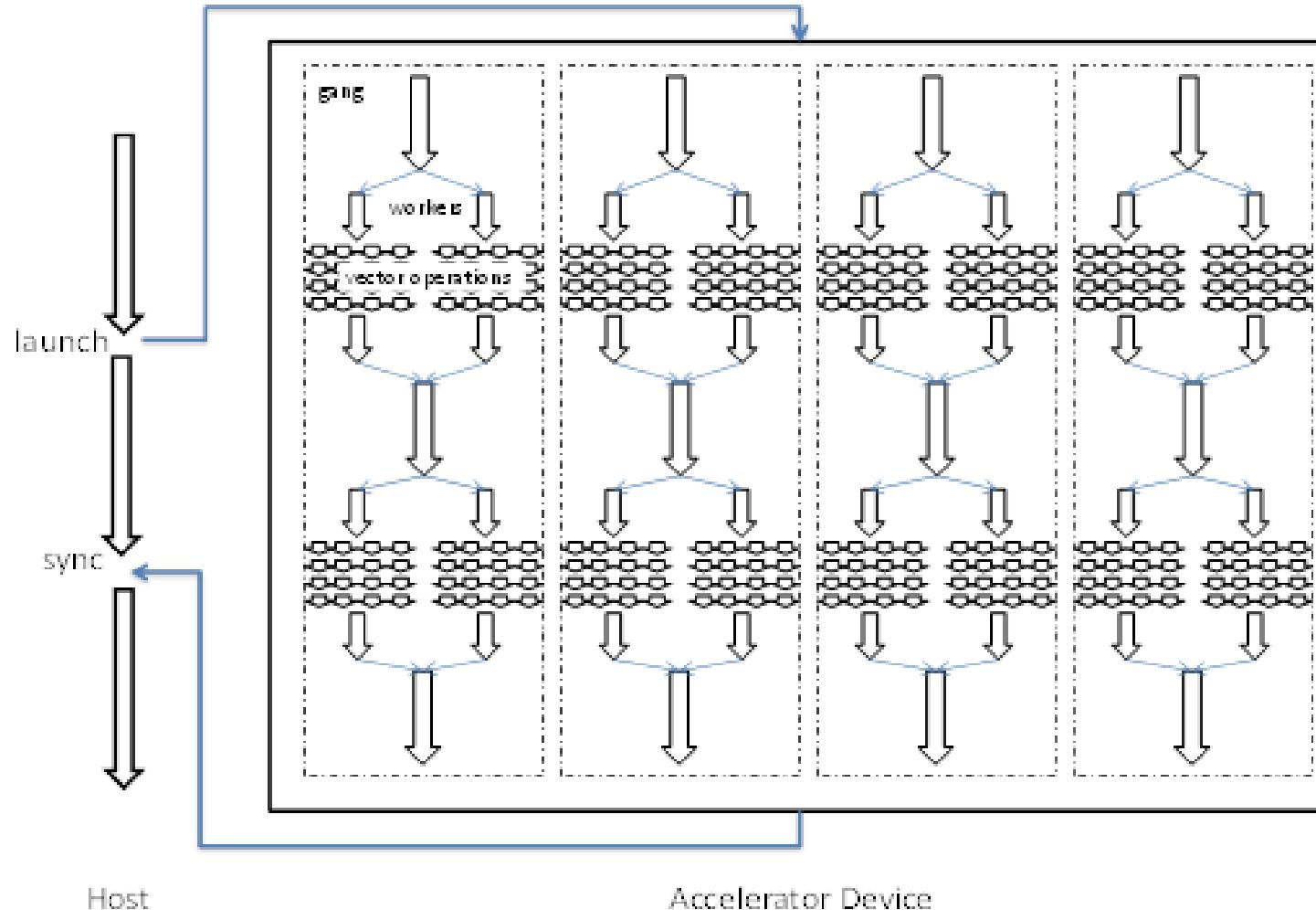
- **on the quality of the compiler**
- (more so than with CUDA or OpenCL).

OpenACC Device Model



Currently OpenACC does not allow user-specified synchronization across threads.

OpenACC Execution Model (Terminology: Gangs and Works)



Parallel vs. Loop Constructs

```
#pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw]) copyout(P[0:Mh*Nw])
for (int i=0; i<Mh; i++) {
    ...
}
```

is equivalent to:

```
#pragma acc parallel copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw]) copyout(P[0:Mh*Nw])
{
    #pragma acc loop
    for (int i=0; i<Mh; i++) {
        ...
    }
}
```

(a parallel region that consists of just a loop)

Parallel Construct

- A parallel construct is executed on an accelerator
- One can specify the number of gangs and number of works in each gang
- Programmer's directive

```
#pragma acc parallel copyout(a) num_gangs(1024) num_workers(32)
{
    a = 23;
}
```

1024*32 workers will be created. a=23 will be executed
redundantly by all 1024 gang leads

What does each “Gang Loop” do?

```
#pragma acc parallel num_gangs(1024)
{
    for (int i=0; i<2048; i++) {
        ...
    }
}
```

The for-loop will be redundantly executed by 1024 gangs

```
#pragma acc parallel num_gangs(1024)
{
    #pragma acc loop gang
    for (int i=0; i<2048; i++) {
        ...
    }
}
```

The 2048 iterations of the for-loop will be divided among 1024 gangs for execution

Worker Loop

```
#pragma acc parallel num_gangs(1024) num_workers(32)
{
    #pragma acc loop gang
    for (int i=0; i<2048; i++) {
        #pragma acc loop worker
        for (int j=0; j<512; j++) {
            foo(i,j);
        }
    }
}
```

1024*32=32K workers will be created, each executing $1M/32K = 32$ instance of foo()

A More Complex Example

```
#pragma acc parallel num_gangs(32)
```

```
{
```

```
    Statement 1; Statement 2;
```

```
#pragma acc loop gang
```

```
for (int i=0; i<n; i++) {
```

```
    Statement 3; Statement 4;
```

```
}
```

```
    Statement 5; Statement 6;
```

```
#pragma acc loop gang
```

```
for (int i=0; i<m; i++) {
```

```
    Statement 7; Statement 8;
```

```
}
```

```
    Statement 9;
```

```
    if (condition)
```

```
        Statement 10;
```

```
}
```

- Statements 1 and 2 are redundantly executed by 32 gangs
- The n for-loop iterations are distributed to 32 gangs

Kernel Regions

```
#pragma acc kernels
{
    #pragma acc loop num_gangs(1024)
    for (int i=0; i<2048; i++) {
        a[i] = b[i];
    }

    #pragma acc loop num_gangs(512)
    for (int j=0; j<2048; j++) {
        c[j] = a[j]*2;
    }

    for (int k=0; k<2048; k++) {
        d[k] = c[k];
    }
}
```

- Kernel constructs are descriptive of programmer intentions (suggestions)

Reduction

```
#pragma acc parallel loop
reduction(+:sum)
for(int i=0;i<n;i++) {
    sum +=
        xcoefs[i]*ycoefs[i];
}
```

- Because each iteration of the loop adds to the variable sum, we must declare a reduction.
- A parallel reduction may return a slightly different result than a sequential addition due to floating point limitations.

C/C++ vs. FORTRAN

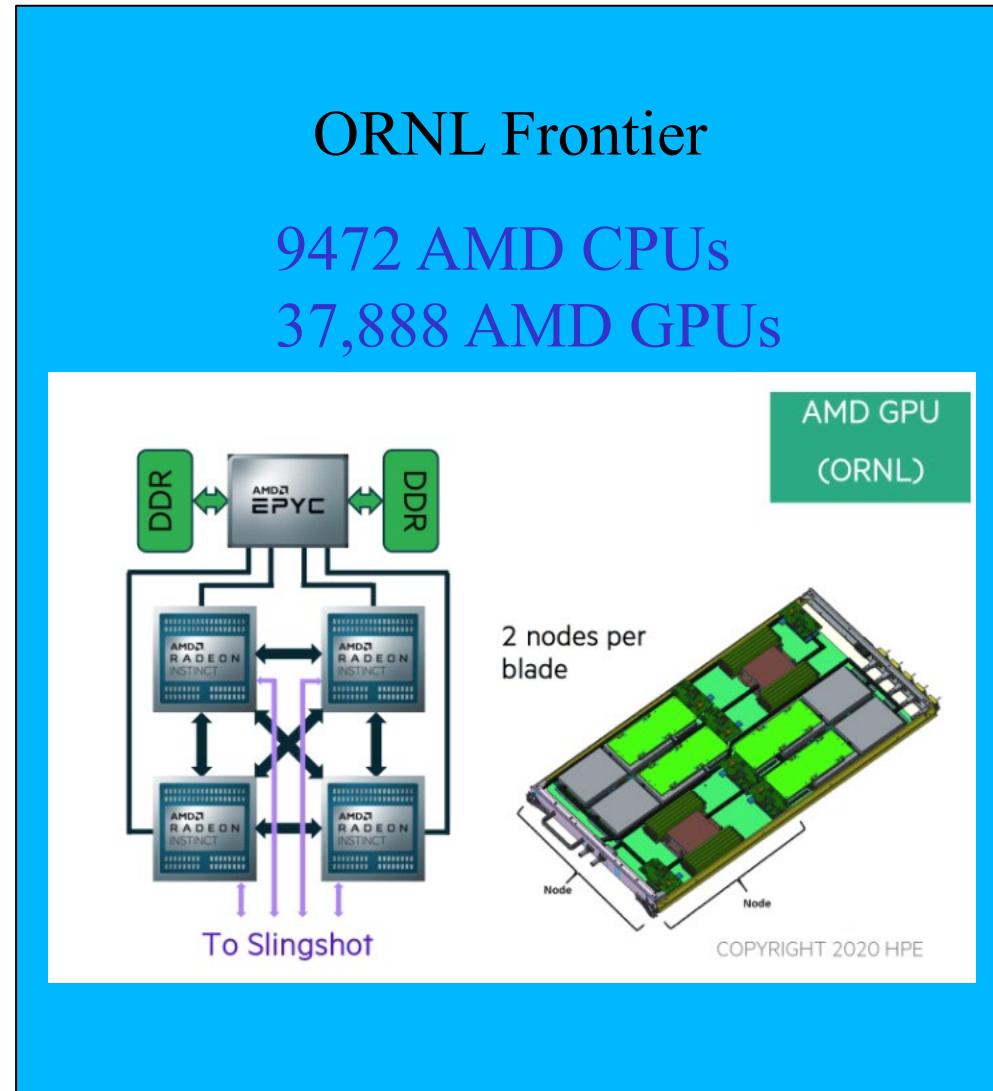
```
// C or C++
#pragma acc <directive> <clauses>
{ ... }

! Fortran
!$acc <directive> <clauses>

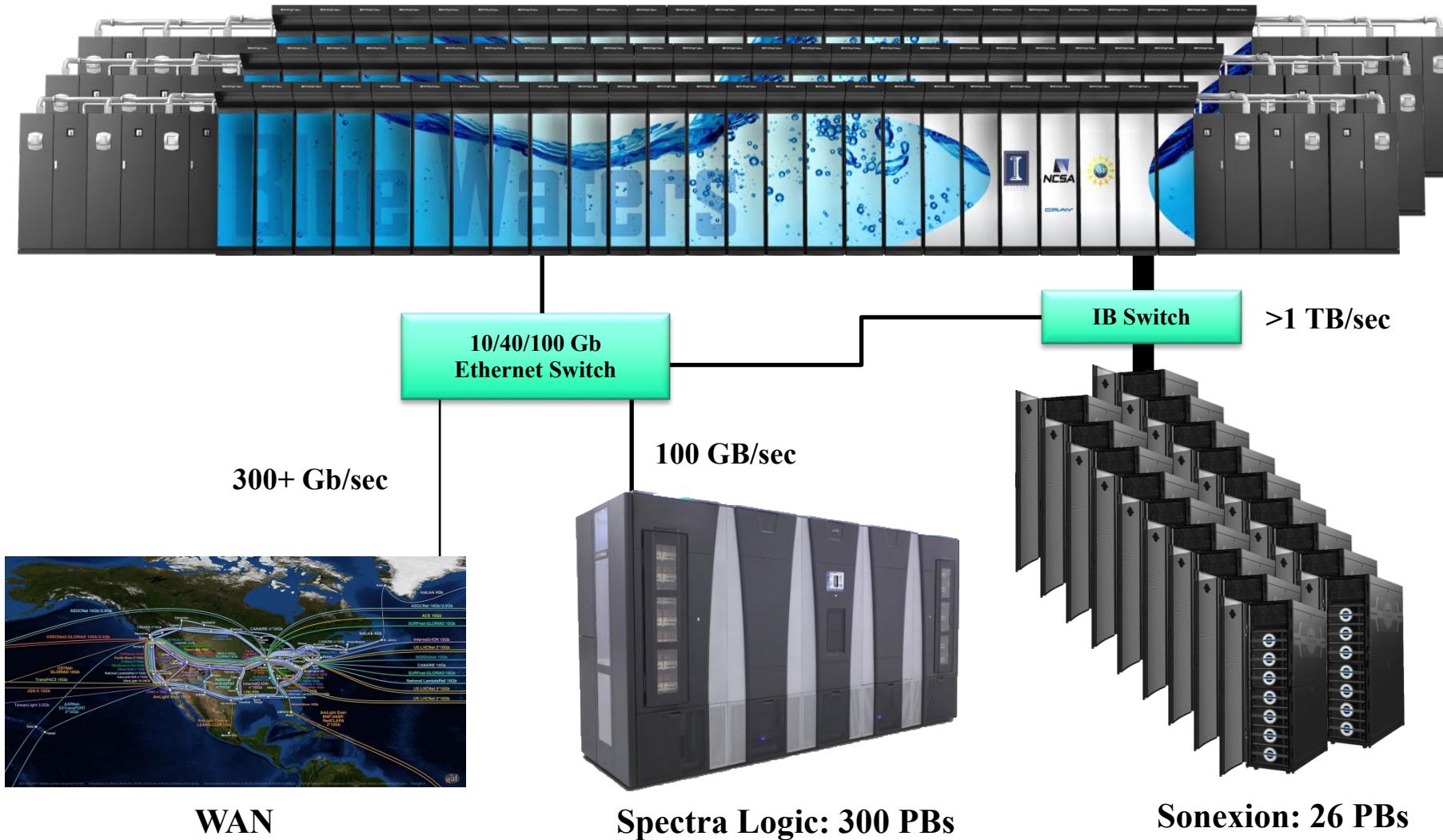
...
!$acc end <directive>
```

Top 5 Supercomputers (Fall 2022)

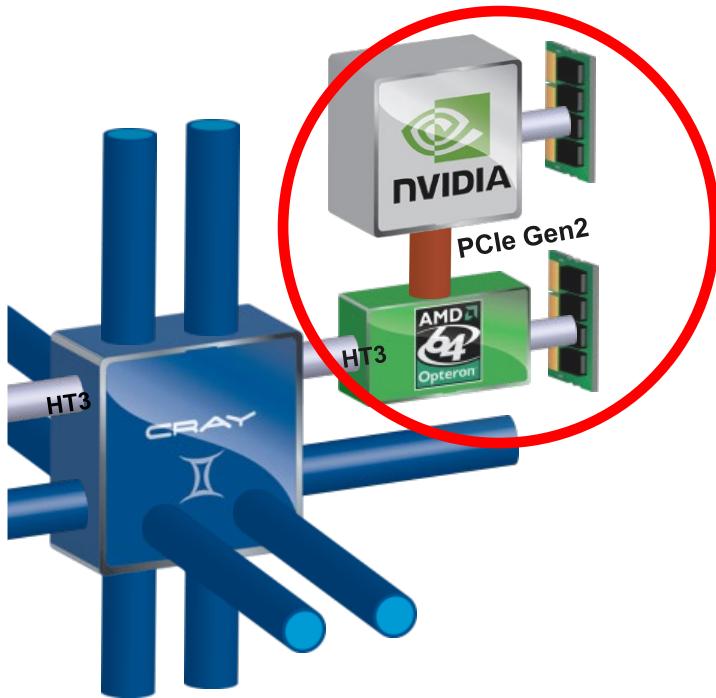
Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016
4	Leonardo - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 40 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,463,616	174.70	255.75	5,610
5	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096



Blue Waters @ UIUC (2013-2021)



Cray XK7 Nodes

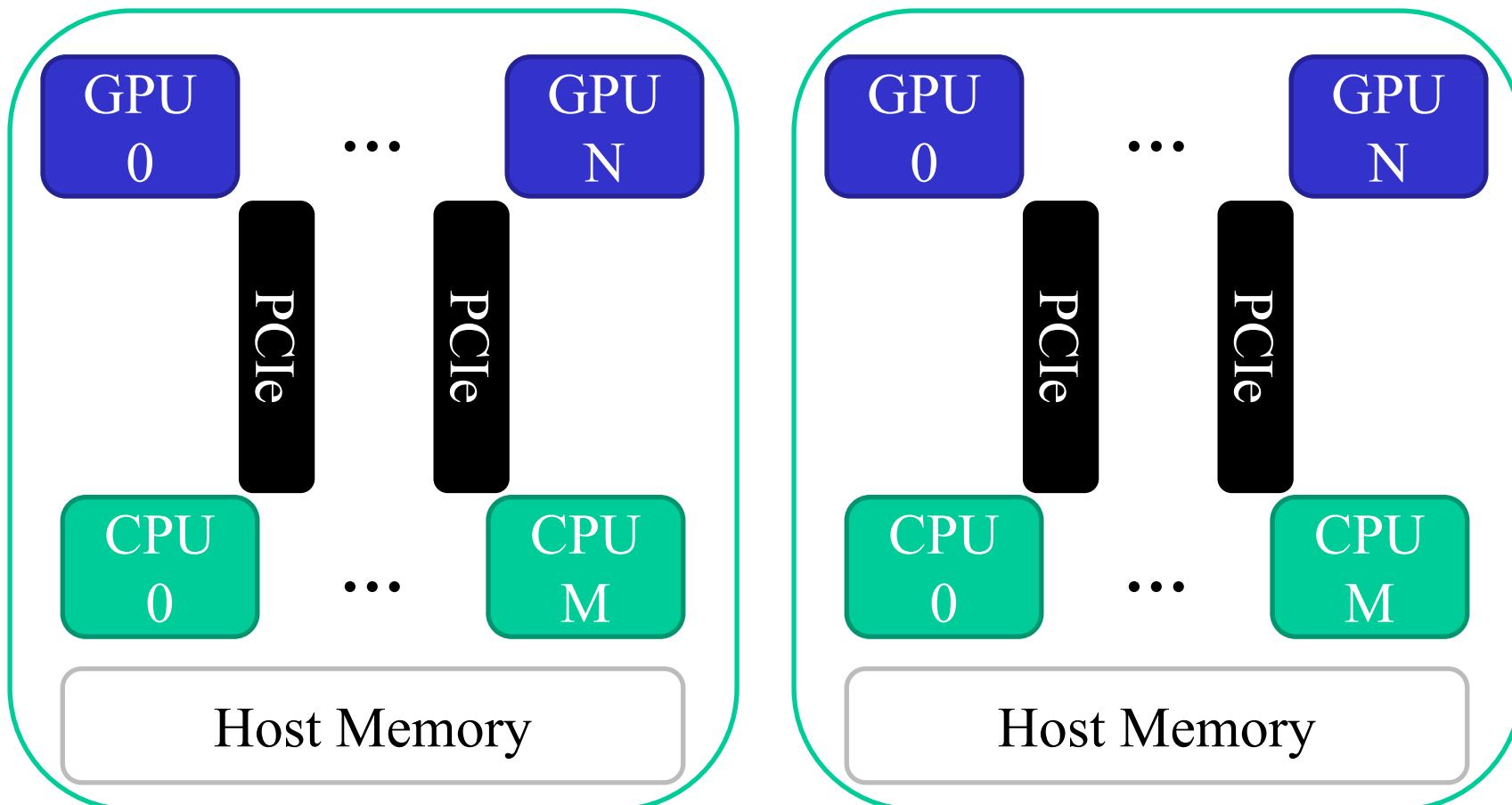


- Dual-socket Node
 - One AMD Interlagos chip
 - 8 core modules, 32 threads
 - 156.5 GFs peak performance
 - 32 GBs memory
 - 51 GB/s bandwidth
 - One NVIDIA Kepler chip
 - 1.3 TFs peak performance
 - 6 GBs GDDR5 memory
 - 250 GB/sec bandwidth
- Gemini Interconnect
 - Same as XE6 nodes

**Blue Waters contains 4,224
Cray XK7 compute nodes.**

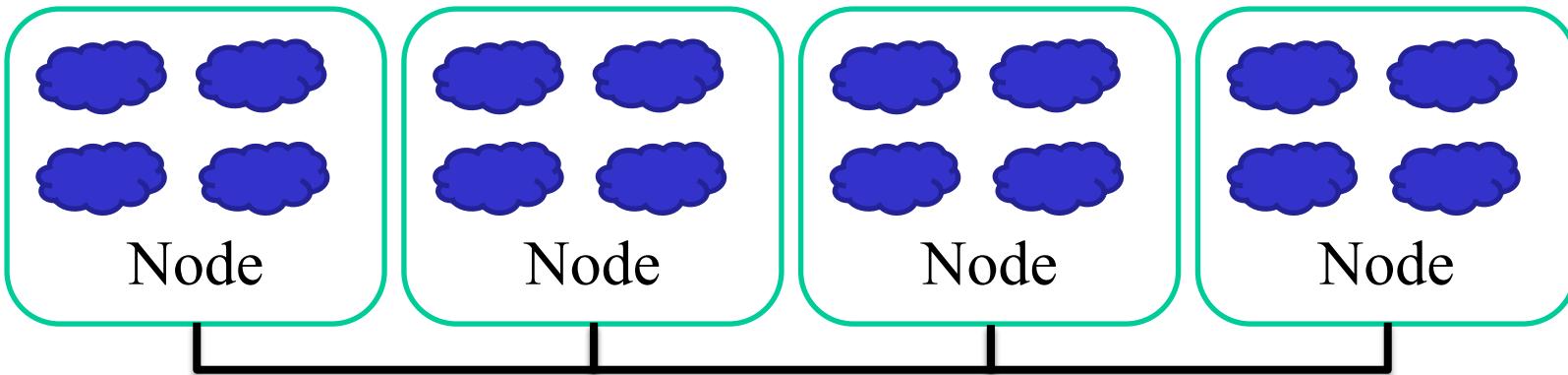
Abstract CUDA-based Node

- Each node contains N GPUs



MPI Model

- Many processes distributed in a cluster



- Each process computes part of the output
- Processes communicate with each other through message passing (not global memory)
- Processes can synchronize through messages

MPI Initialization, Info

- User launches an MPI job with X processes by executing in the command shell
 - MPIrun -np X
- `int MPI_Init(int *argc, char ***argv)`
 - Initialize MPI
- `MPI_COMM_WORLD`
 - MPI group formed with all allocated nodes
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
 - Rank of the calling process in group of comm
- `int MPI_Comm_size(MPI_Comm comm, int *size)`
 - Number of processes in the group of comm

Vector Addition: Main Process

```
int main(int argc, char *argv[]) {
    int vector_size = 1024 * 1024 * 1024;
    int pid=-1, np=-1;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    if(np < 3) {
        if(0 == pid) printf("Nedded 3 or more processes.\n");
        MPI_Abort( MPI_COMM_WORLD, 1 ); return 1;
    }
    if(pid < np - 1)
        compute_node(vector_size / (np - 1));
    else
        data_server(vector_size);

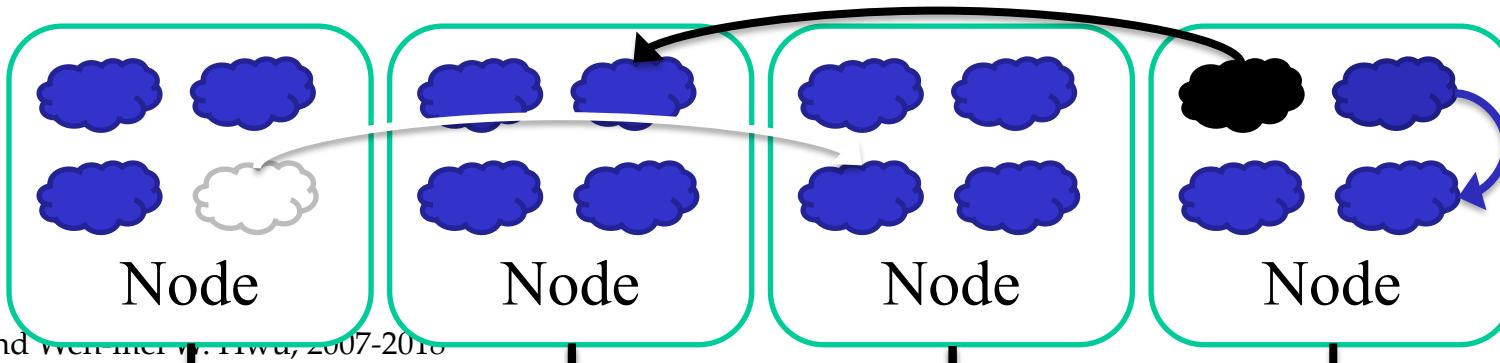
    MPI_Finalize();
    return 0;
}
```

MPI Sending Data

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
 - `buf`: Starting address of send buffer
 - `count`: Number of elements in send buffer (nonnegative integer)
 - `datatype`: Datatype of each send buffer element
 - `dest`: Rank of destination (integer)
 - `tag`: Message tag (integer)
 - `comm`: Communicator (handle)

MPI Sending Data

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
 - `Buf`: Initial address of send buffer
 - `Count`: Number of elements in send buffer (nonnegative integer)
 - `Datatype`: Datatype of each send buffer element
 - `Dest`: Rank of destination (integer)
 - `Tag`: Message tag (integer)
 - `Comm`: Communicator (handle)

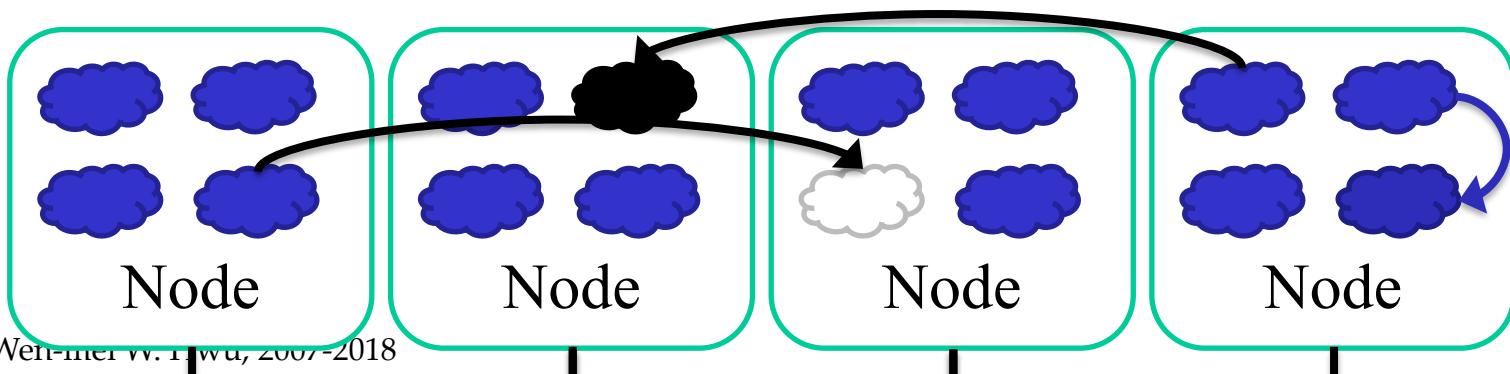


MPI Receiving Data

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
 - **Buf**: Starting address of receive buffer
 - **Count**: Maximum number of elements in receive buffer (non-negative integer)
 - **Datatype**: Datatype of each receive buffer element
 - **Source**: Rank of source (integer)
 - **Tag**: Message tag (integer)
 - **Comm**: Communicator (handle)
 - **Status**: Status object

MPI Receiving Data

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
 - `Buf`: Initial address of receive buffer
 - `Count`: Maximum number of elements in receive buffer (non-negative integer)
 - `Datatype`: Datatype of each receive buffer element
 - `Source`: Rank of source (integer)
 - `Tag`: Message tag (integer)
 - `Comm`: Communicator (handle)
 - `Status`: Status object (Status)



Vector Addition: Server Process (I)

```
void data_server(unsigned int vector_size) {
    int np, num_nodes = np - 1, first_node = 0, last_node = np - 2;
    unsigned int num_bytes = vector_size * sizeof(float);
    float *input_a = 0, *input_b = 0, *output = 0;

    /* Set MPI Communication Size */
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    /* Allocate input data */
    input_a = (float *)malloc(num_bytes);
    input_b = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);
    if(input_a == NULL || input_b == NULL || output == NULL) {
        printf("Server couldn't allocate memory\n");
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }
    /* Initialize input data */
    random_data(input_a, vector_size , 1, 10);
    random_data(input_b, vector_size ,
```

Vector Addition: Server Process (II)

```
/* Send data to compute nodes */
float *ptr_a = input_a;
float *ptr_b = input_b;

for(int process = 1; process < last_node; process++) {
    MPI_Send(ptr_a, vector_size / num_nodes, MPI_FLOAT,
             process, DATA_DISTRIBUTE, MPI_COMM_WORLD);
    ptr_a += vector_size / num_nodes;

    MPI_Send(ptr_b, vector_size / num_nodes, MPI_FLOAT,
             process, DATA_DISTRIBUTE, MPI_COMM_WORLD);
    ptr_b += vector_size / num_nodes;
}
```

Vector Addition: Server Process (III)

```
/* Wait for compute to complete*/
MPI_Barrier(MPI_COMM_WORLD);

/* Collect output data */
MPI_Status status;
for(int process = 0; process < num_nodes; process++) {
    MPI_Recv(output + process * num_points / num_nodes,
              num_points / num_comp_nodes, MPI_REAL, process,
              DATA_COLLECT, MPI_COMM_WORLD, &status );
}

/* Store output data */
store_output(output, dimx, dimy, dimz);

/* Release resources */
free(input);
free(output);
}
```

Vector Addition: Compute Process (I)

```
void compute_node(unsigned int vector_size ) {
    int np;
    unsigned int num_bytes = vector_size * sizeof(float);
    float *input_a, *input_b, *output;
    MPI_Status status;

    MPI_Comm_size(MPI_COMM_WORLD, &np);
    int server_process = np - 1;

    /* Alloc host memory */
    input_a = (float *)malloc(num_bytes);
    input_b = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);

    /* Get the input data from server process */
    MPI_Recv(input_a, vector_size, MPI_FLOAT, server_process,
              DATA_DISTRIBUTE, MPI_COMM_WORLD, &status);
    MPI_Recv(input_b, vector_size, MPI_FLOAT, server_process,
              DATA_DISTRIBUTE, MPI_COMM_WORLD, &status);
```

Vector Addition: Compute Process (II)

```
/* Compute the partial vector addition */
for(int i = 0; i < vector_size; ++i) {
    output[i] = input_a[i] + input_b[i];
}

/* Or, can offload to GPU here */
/* cudaMalloc(), cudaMemcpy(), kernel launch, etc. */

MPI_Barrier(MPI_COMM_WORLD);

/* Send the output */
MPI_Send(output, vector_size, MPI_FLOAT,
         server_process, DATA_COLLECT, MPI_COMM_WORLD);

/* Release memory */
free(input_a);
free(input_b);
free(output);
}
```



**ANY MORE QUESTIONS?
READ CHAPTER 15**

Also see <https://developer.nvidia.com/intro-to-openacc-course-2016>



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 22:

Data Transfer and CUDA Streams

(Task Parallelism)

Course Reminders

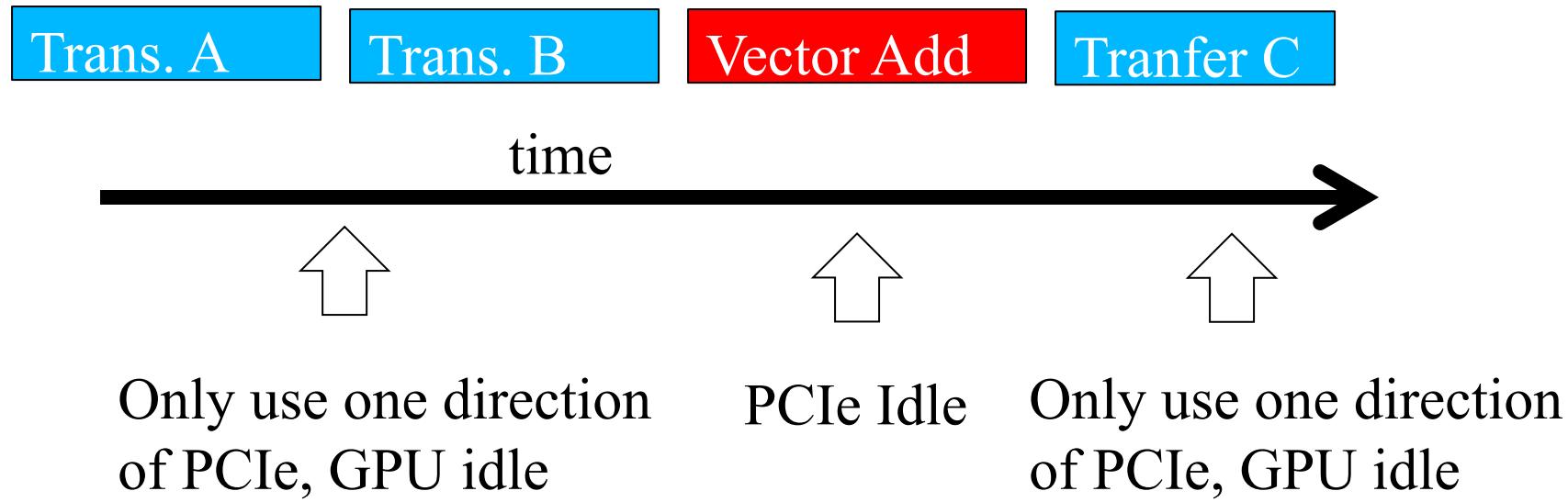
- Lab 6 is due this Friday
- Lab 7 is due next Friday
- PM3 should be out by now
 - This is a much more involved PM, start early
 - Do not wait to submit until last minute!
 - Competition description is out too, but first you need to finish PM3

Objective

- To learn more advanced features of the CUDA APIs for data transfer and kernel launch
 - Task parallelism for overlapping data transfer with kernel computation
 - CUDA streams

Serialized Data Transfer

- So far, the way we use cudaMemcpy serializes data transfer and GPU computation



Device Overlap

- Most CUDA devices support *device overlap*
 - *Simultaneously execute a kernel while performing a copy between device and host memory*

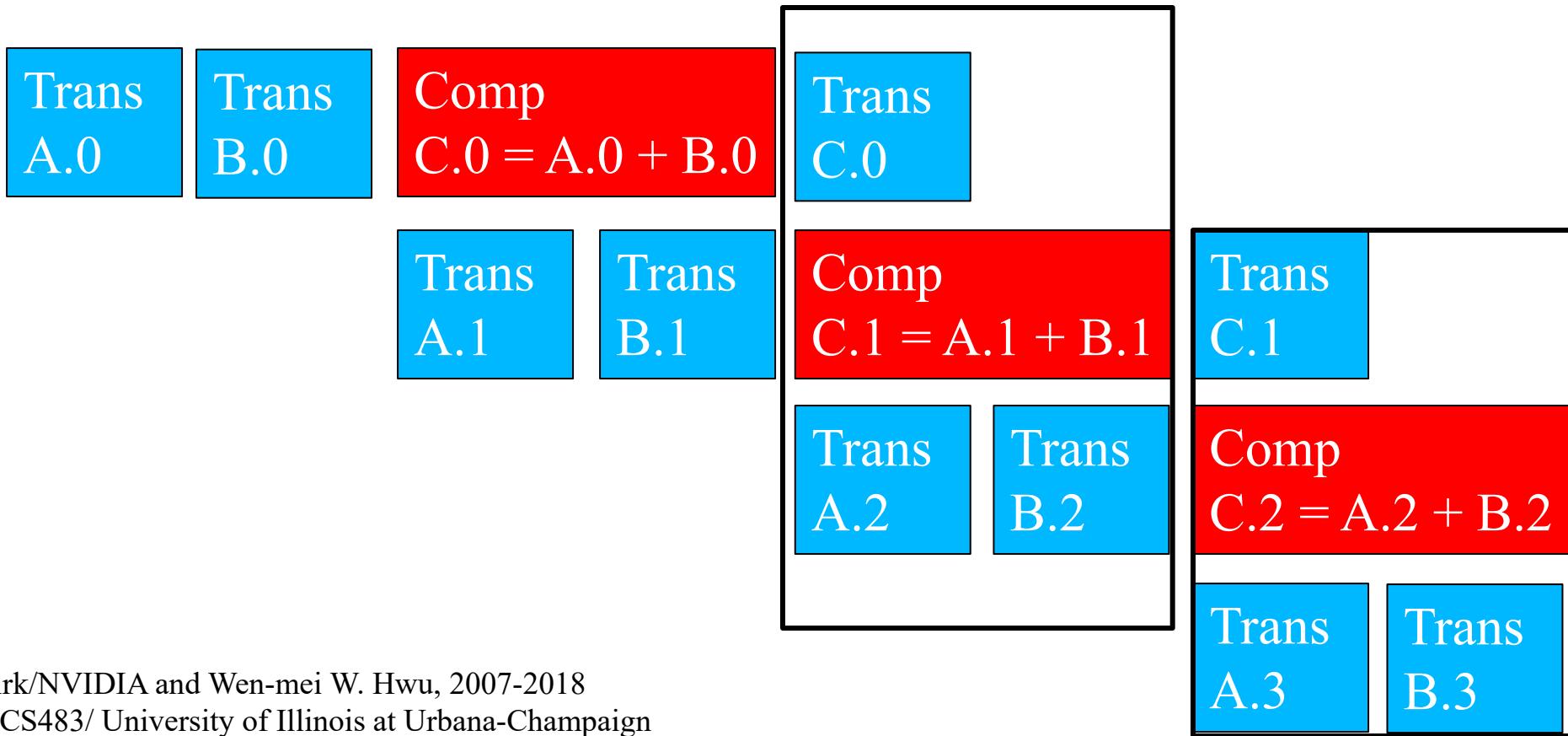
```
int dev_count;
cudaDeviceProp prop;

cudaGetDeviceCount( &dev_count);
for (int i = 0; i < dev_count; i++) {
    cudaGetDeviceProperties(&prop, i);

    if (prop.deviceOverlap) ...
```

Overlapped (Pipelined) Timing

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments

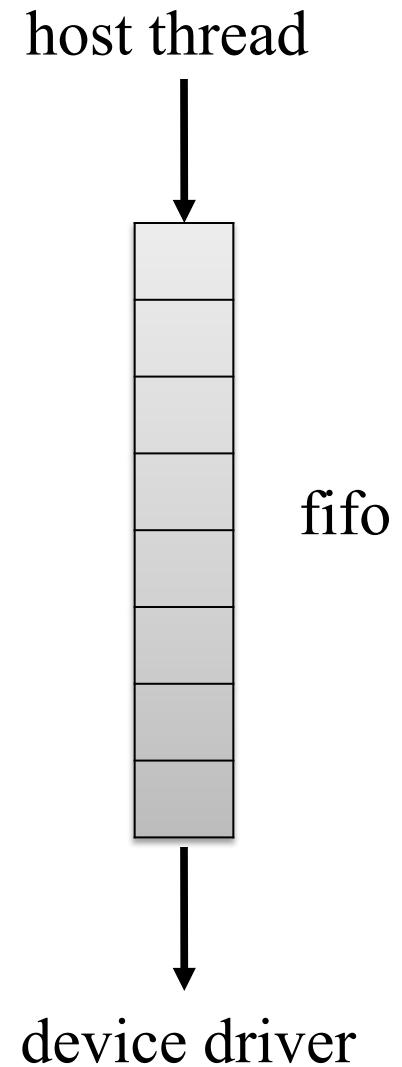


Using CUDA Streams and Asynchronous Memcpy

- CUDA supports parallel execution of kernels and cudaMemcpy with **streams**
- Each stream **is a queue of operations** (kernel launches and cudaMemcpy's)
- Operations (tasks) in different streams
 - can execute in parallel
 - a version of **task parallelism**

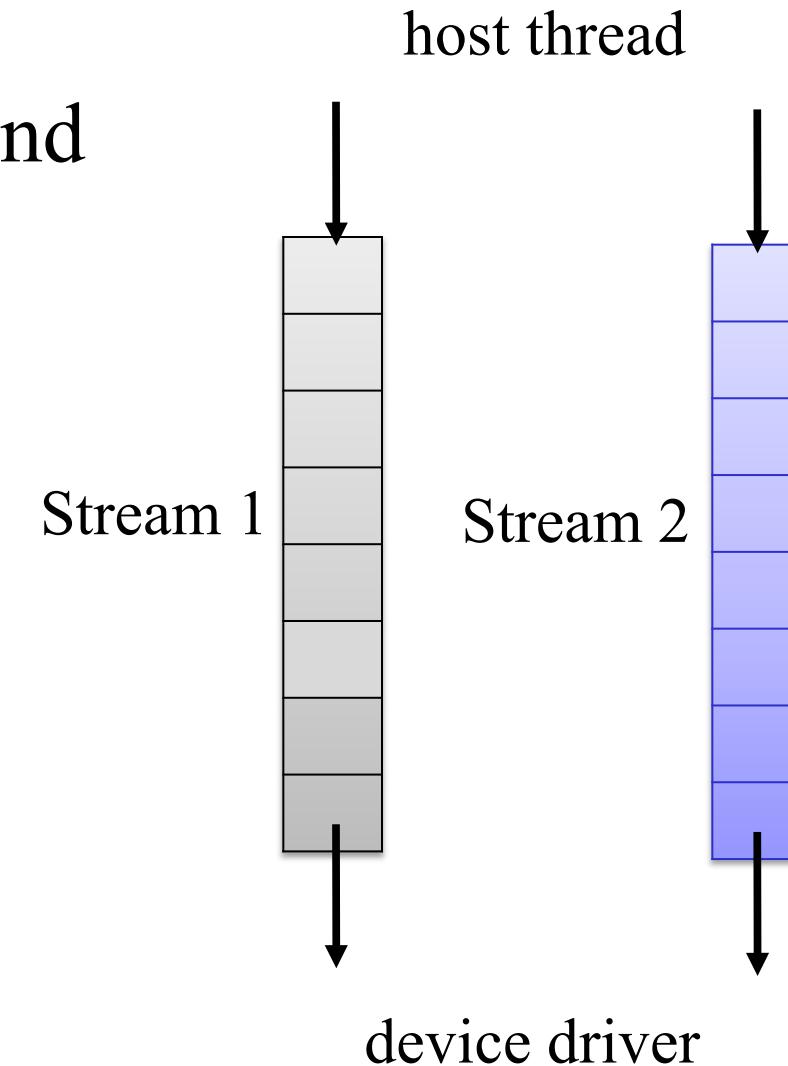
Streams

- Device requests made from the host code are put into a queue
 - Queue processed asynchronously by the driver and device. Called a “Stream”
 - Driver ensures that commands in the queue are processed strictly in sequence. Memory copies end before kernel launch, etc.

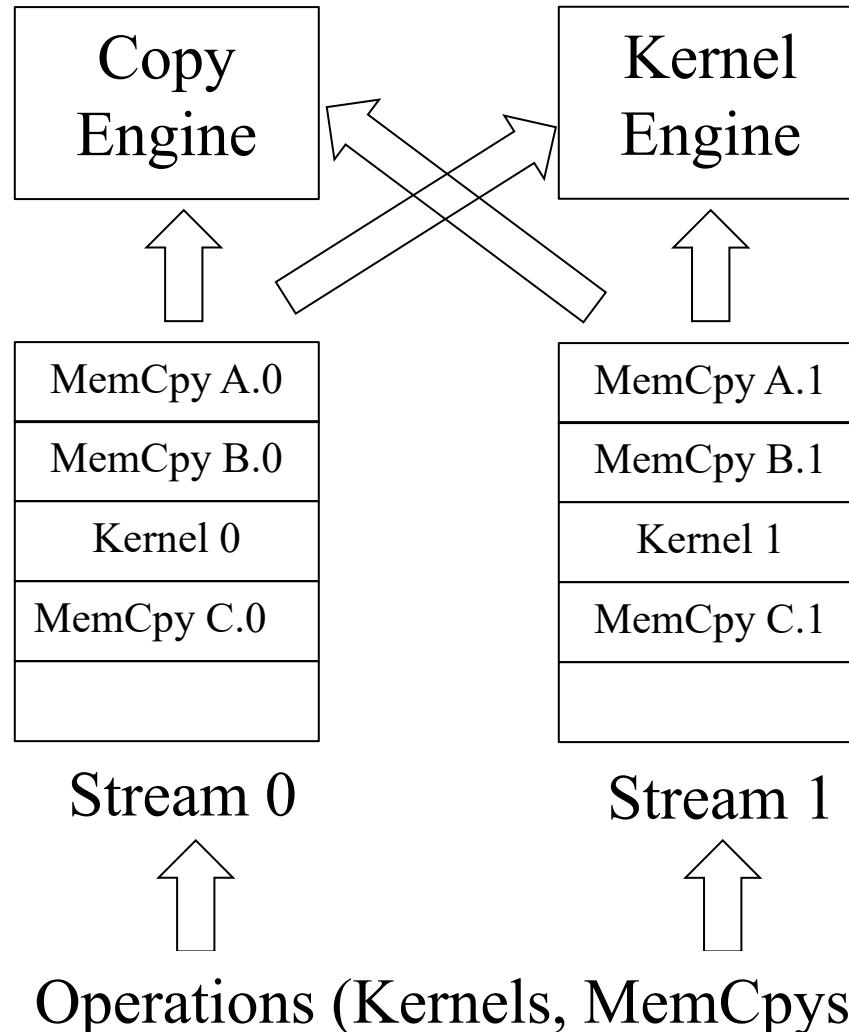
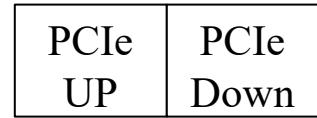


Streams cont.

- To allow concurrent copying and kernel execution, multiple queues are required



Conceptual View of Streams



A Simple Multi-Stream Host Code

```
cudaStream_t      stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);
float *d_A0, *d_B0, *d_C0;      // device memory for stream 0
float *d_A1, *d_B1, *d_C1;      // device memory for stream 1

// cudaMalloc for d_A0, d_B0, d_C0, d_A1, d_B1, d_C1 go here

for (int i=0; i<n; i+=SegSize*2) {
    // copy data in stream0
    // lunch kernel in stream0
    // copy results in stream0
    // copy data in stream1
    // lunch kernel in stream1
    // copy results in stream1
}
```

A Simple Multi-Stream Host Code

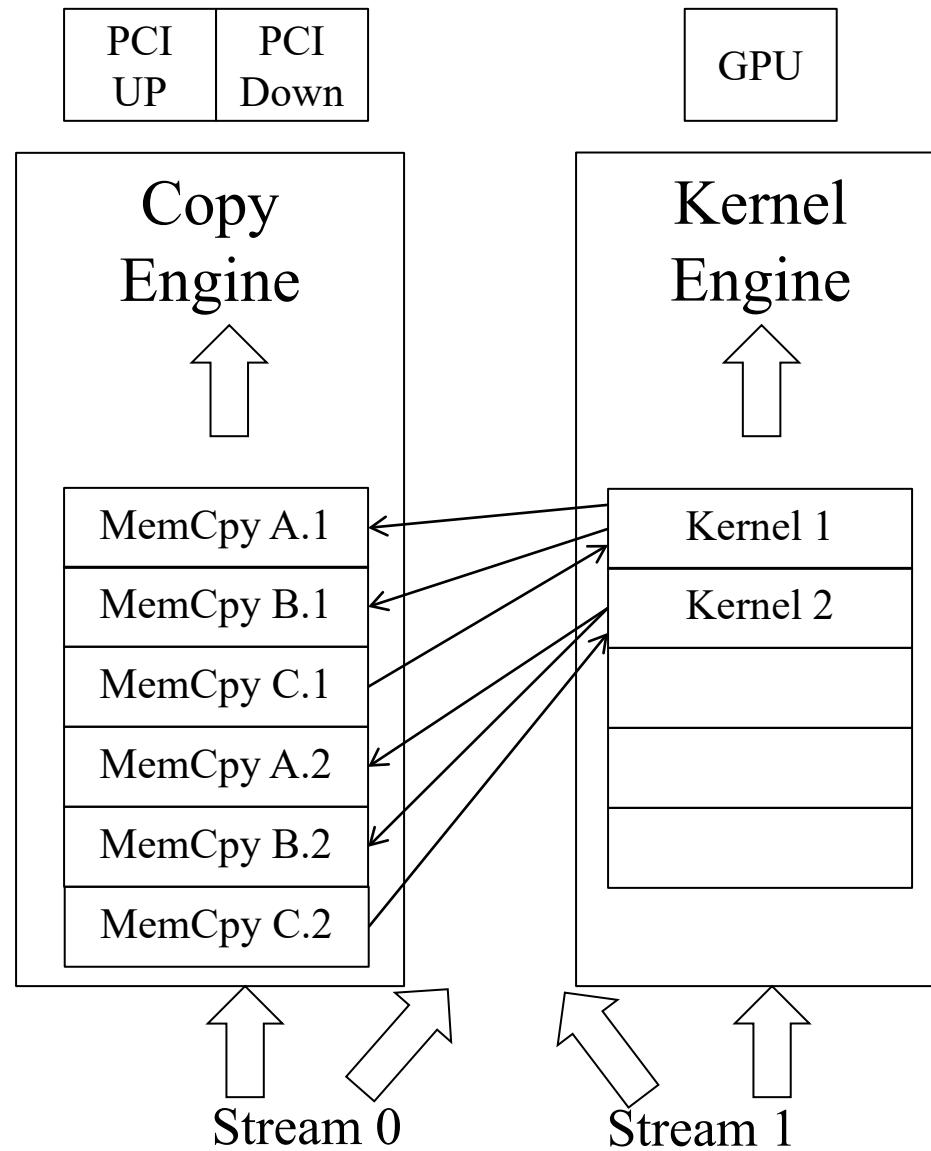
```
for (int i=0; i<n; i+=SegSize*2)
{
    cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),..., stream0);
    cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float),..., stream0);
    vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, ...);
    cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),..., stream0);

    cudaMemcpyAsync(d_A1, h_A+i+SegSize,
                    SegSize*sizeof(float),..., stream1);
    cudaMemcpyAsync(d_B1, h_B+i+SegSize,
                    SegSize*sizeof(float),..., stream1);
    vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, ...);
    cudaMemcpyAsync(h_C+i+SegSize, d_C1,
                    SegSize*sizeof(float),..., stream1);
}
```

Older GPUs Support Streams in Software

Task at head of queue waits for dependencies (arcs) before executing.

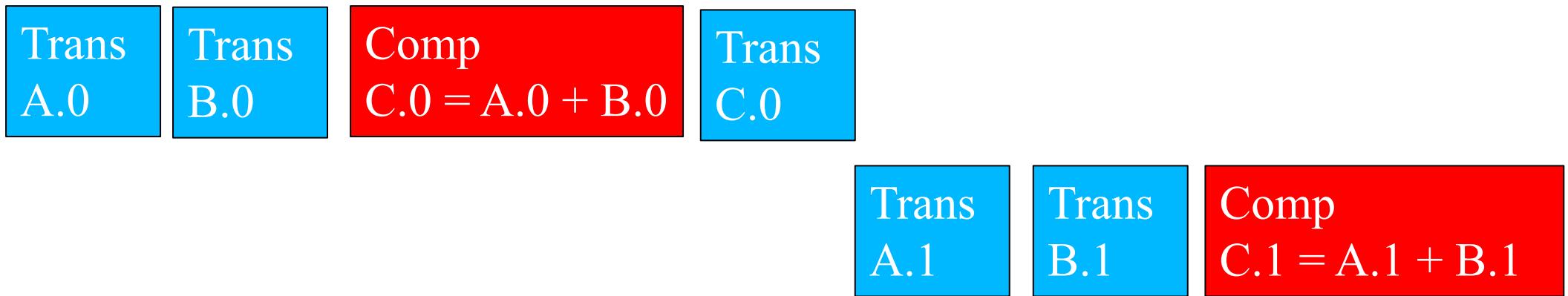
For example,
Kernel 1 waits for
Memcpy A.1 and
Memcpy B.1 to
finish.



Operations (Kernels, MemCpys)

Not quite the overlap we want

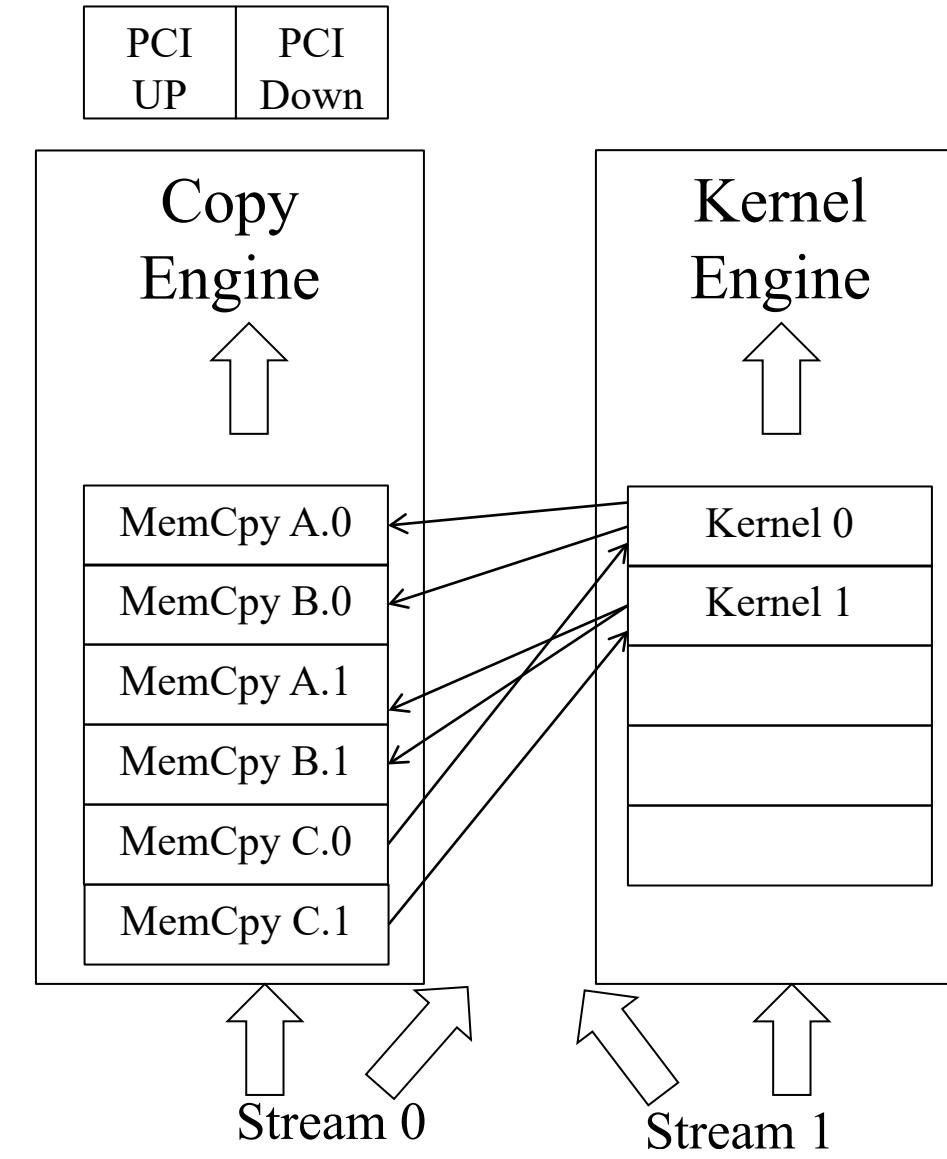
- C.0 blocks A.1 and B.1 in the copy engine queue



A Better Multi-Stream Host Code

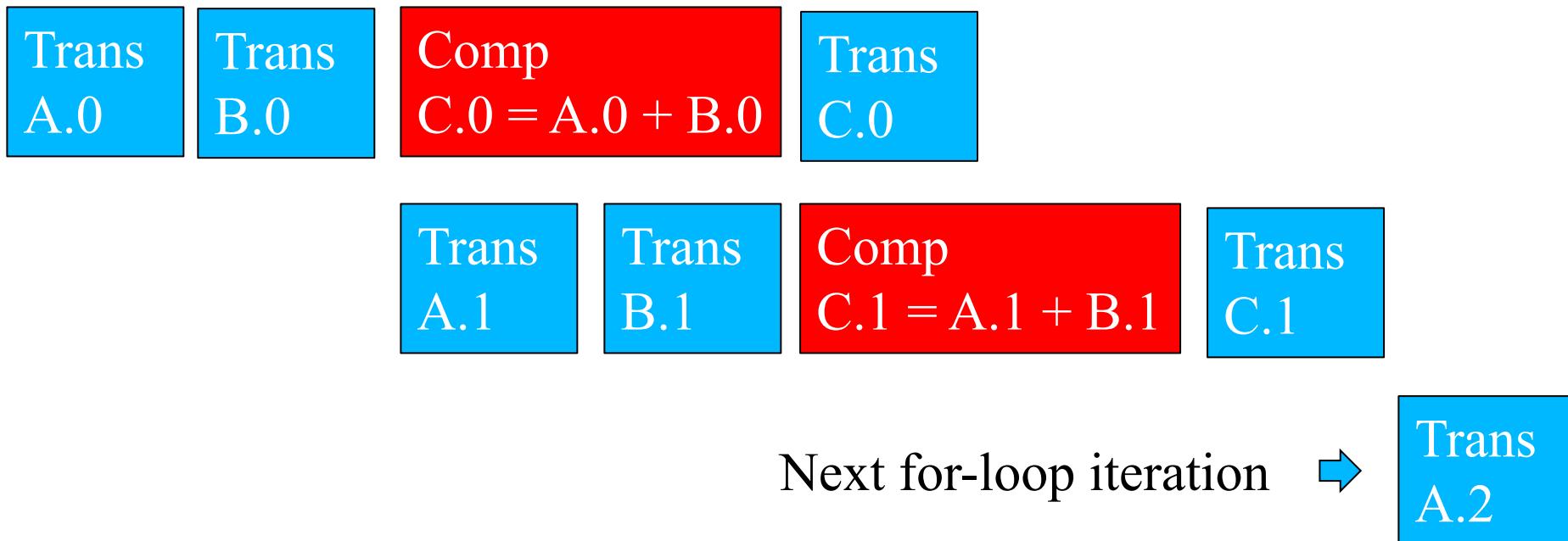
```
for (int i=0; i<n; i+=SegSize*2) {  
  
    cudaMemcpyAsync(d_A0, h_A+i; SegSize*sizeof(float),..., stream0);  
    cudaMemcpyAsync(d_B0, h_B+i; SegSize*sizeof(float),..., stream0);  
    cudaMemcpyAsync(d_A1, h_A+i+SegSize;  
                    SegSize*sizeof(float),..., stream1);  
    cudaMemcpyAsync(d_B1, h_B+i+SegSize;  
                    SegSize*sizeof(float),..., stream1);  
  
    vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, ...);  
    vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, ...);  
    cudaMemcpyAsync(d_C0, h_C+i; SegSize*sizeof(float),..., stream0);  
    cudaMemcpyAsync(d_C1, h_C+i+SegSize;  
                    SegSize*sizeof(float),..., stream1);  
}  
}
```

A View Closer to Reality



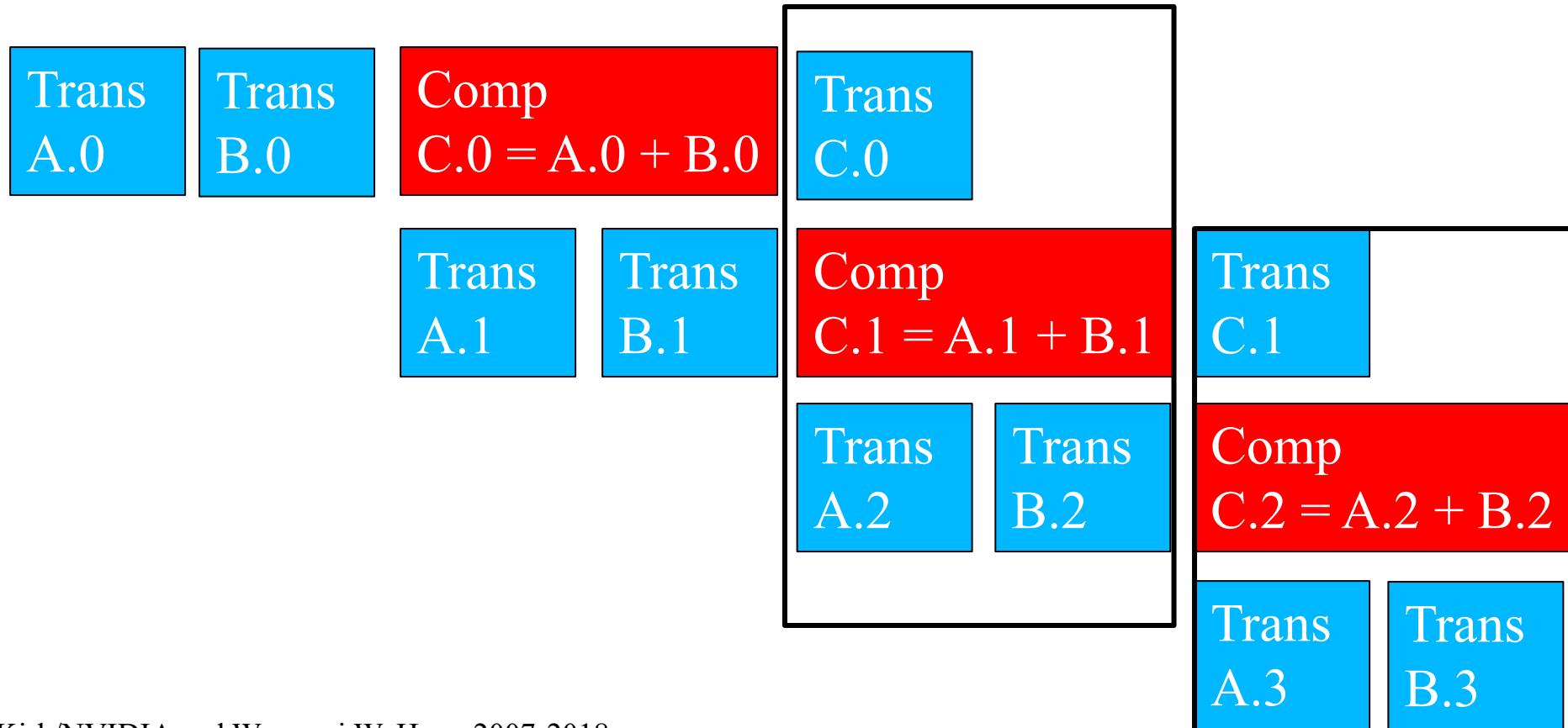
Better Overlap with Two Streams

- C.0 no longer blocks A.1 and B.1 in the copy engine queue
- However, C.1 still blocks A.2 and B.2 from the next iteration – PCIe used for only one direction



Three streams needed for continuous pipelining

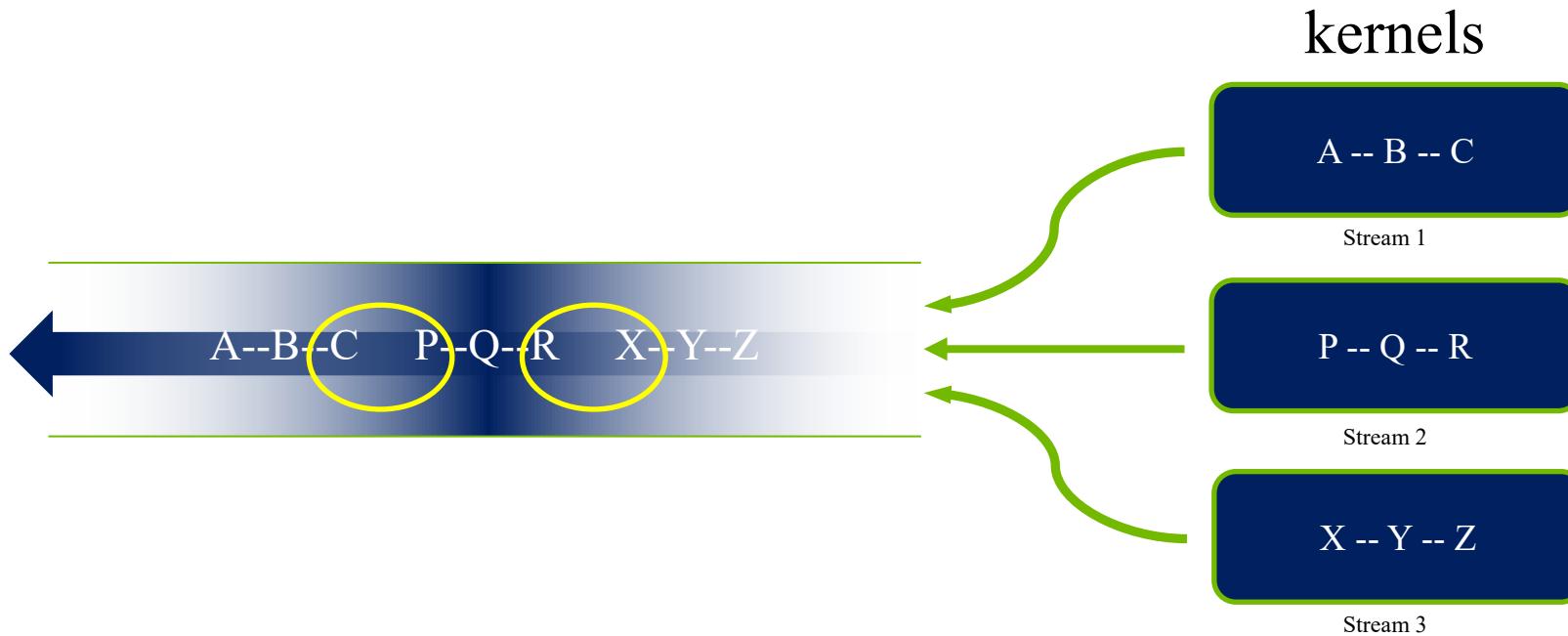
- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments



Hyper Queue

- Provide multiple real stream queues for each engine
- Allow more concurrency by allowing some streams to make progress for an engine while others are blocked

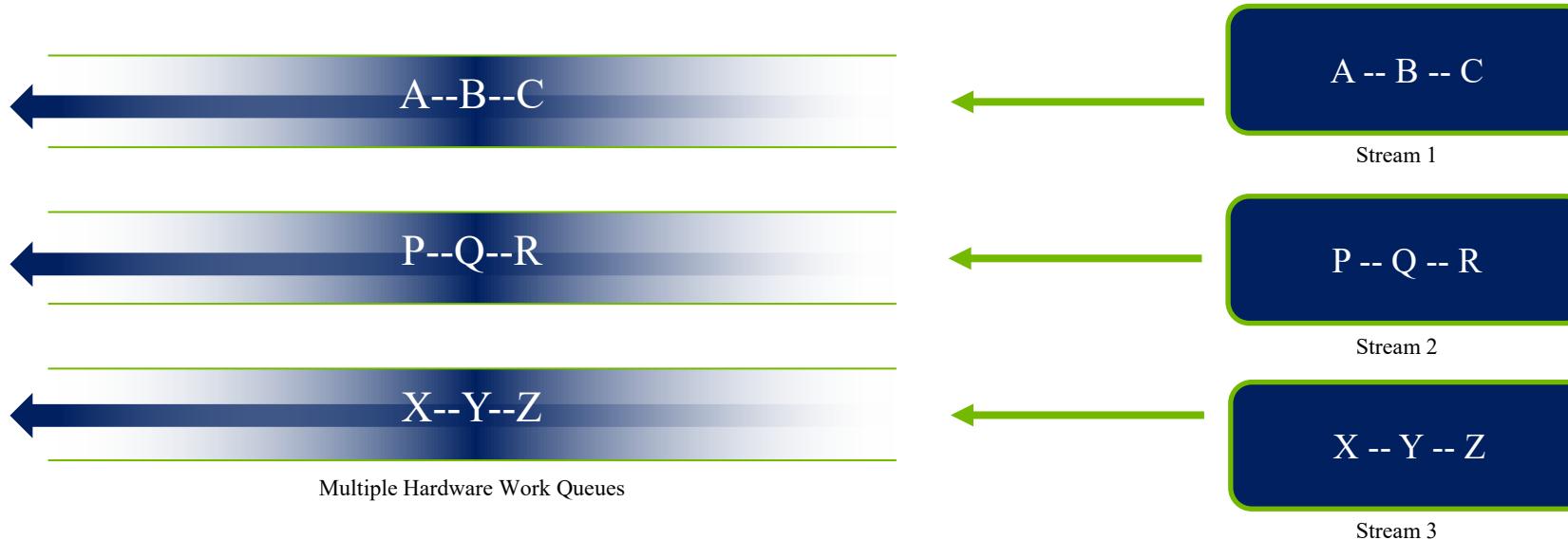
Fermi (and older) Concurrency



Fermi allows 16-way concurrency

- Up to 16 grids can run at once
- But kernels from CUDA streams multiplex into a single queue
- Overlap only at stream edges

Kepler Improved Concurrency



Kepler allows 32-way concurrency

- One work queue per stream
- Concurrency at full-stream level
- No inter-stream dependencies

Smaller Segments Reduce Boundary Effects

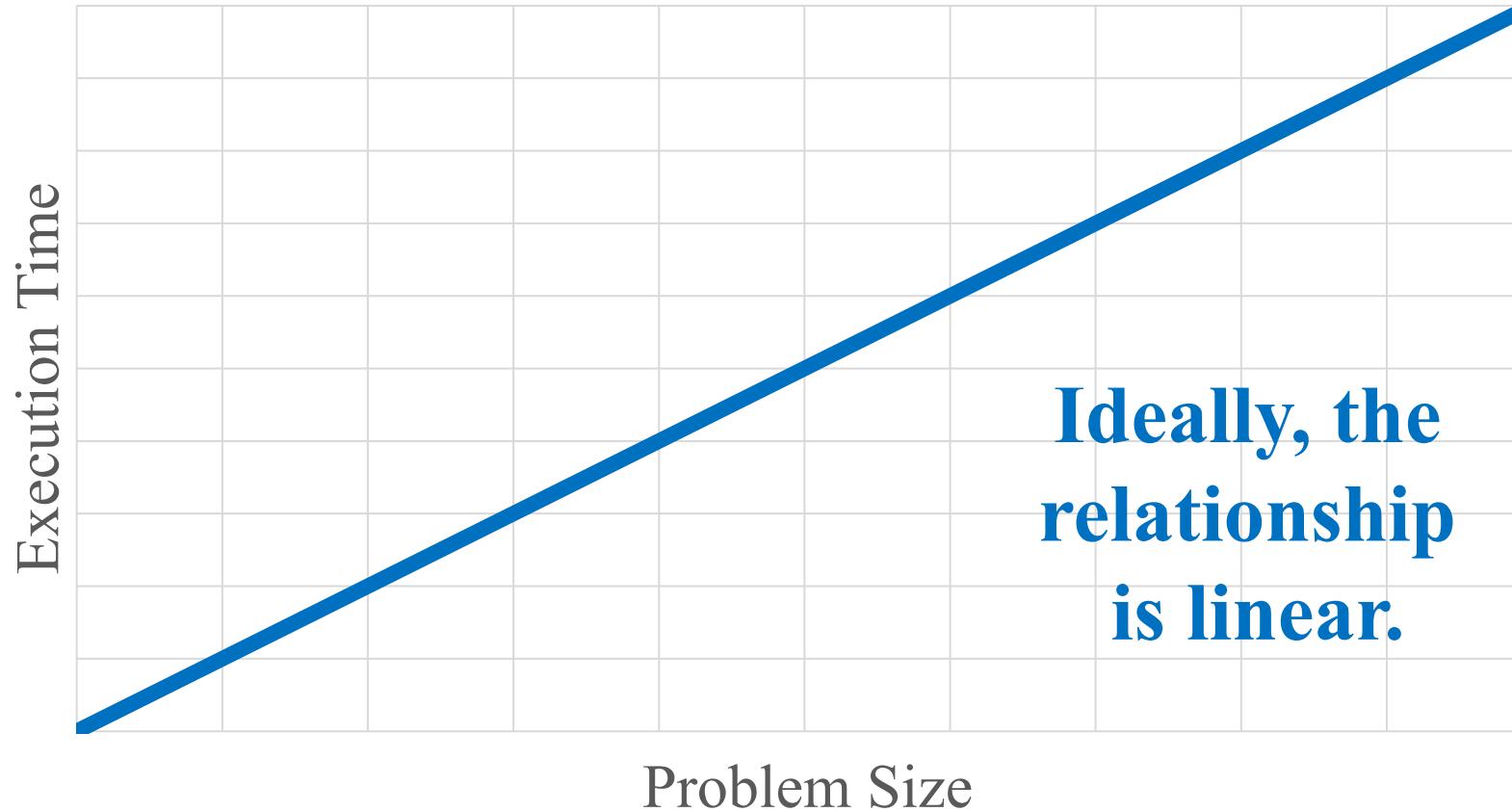
How small should segments be?

- If we **overlap**
 - **transfer of** segment N's **inputs**,
 - **computation** of segment N – 1, **and**
 - **transfer** of segment N – 2's **results**,
- we **still have non-overlapping work** at the beginning and the end.

So segments should be really small?

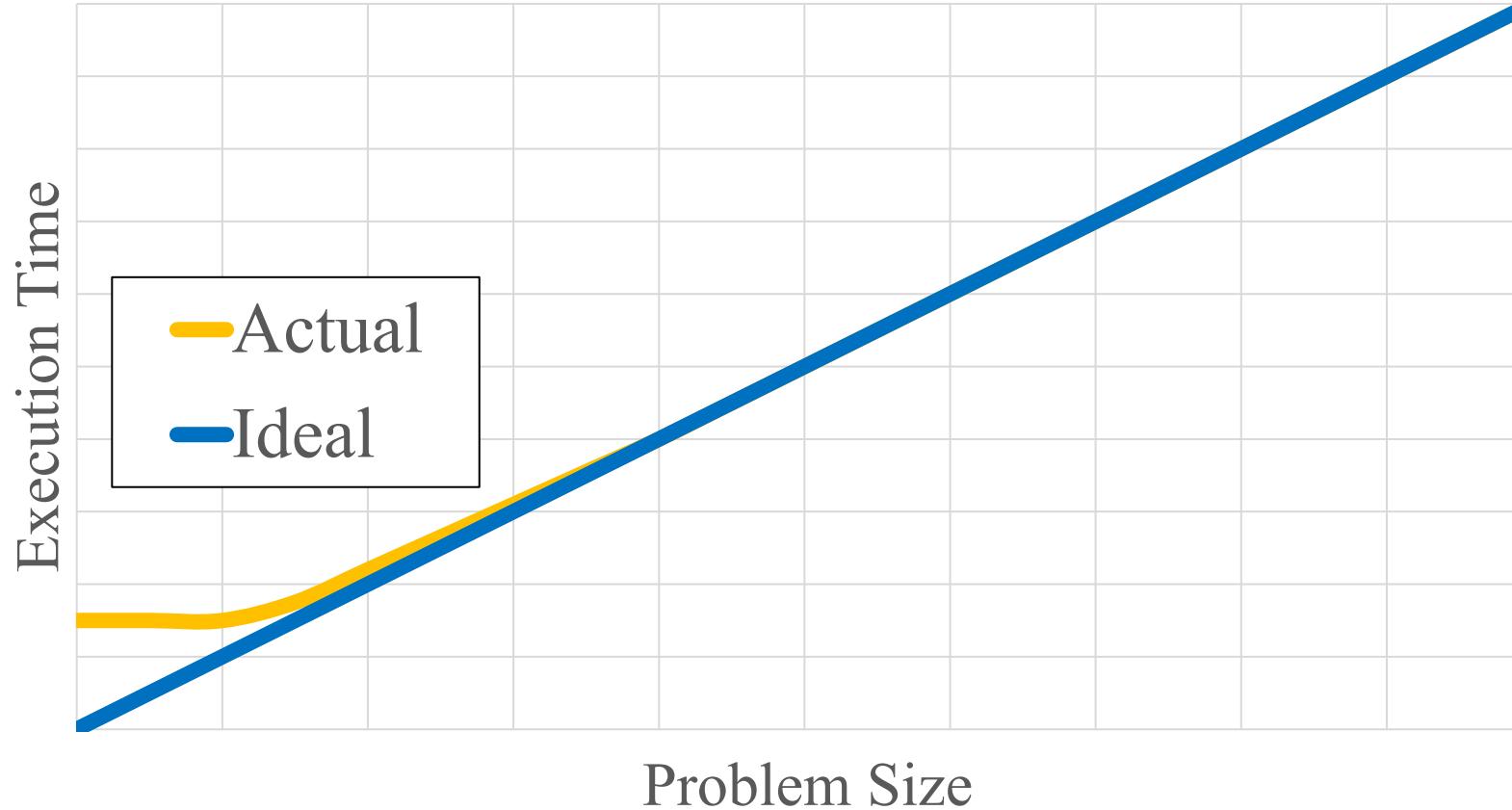
Execution Time is Ideally Linear in Size

Think about execution time as a function of segment size.



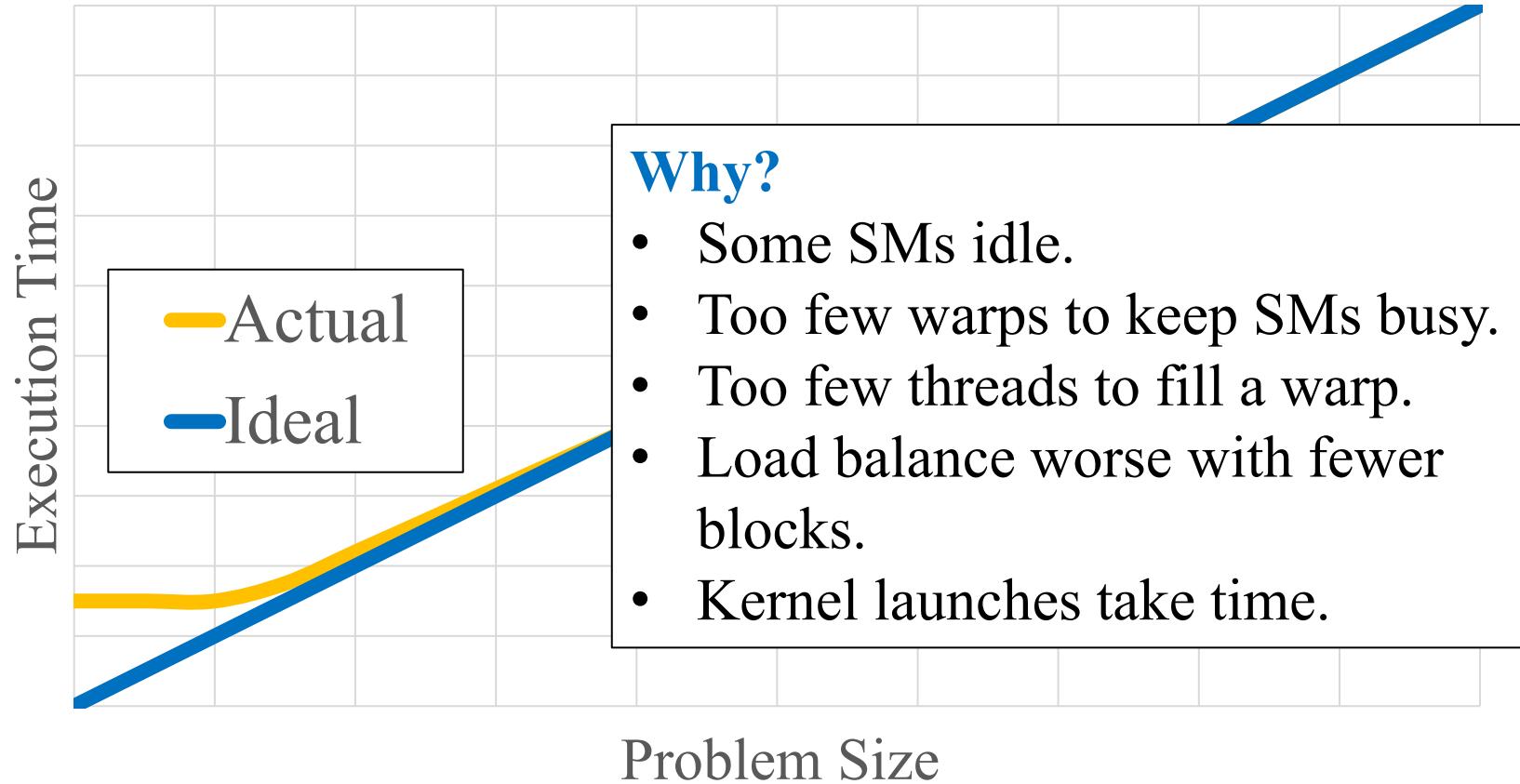
Execution Time Never Reaches Zero

But real execution time has a minimum.



Execution Time Never Reaches Zero

But real execution time has a minimum.



Use Moderate Segment Size and Device Query

Data transfers

- **have similar non-linearities** for small sizes
- due to startup costs on host and DMA.

So how small should segments be?

Moderately sized.

Best size likely to **depend on GPU**.



**ANY MORE QUESTIONS
READ CHAPTER 13**

Problem Solving

- Q: You are tasked with performing some operations on a very long vector. Upon profiling your kernel code with nsys, you realize that a significant portion of the execution time is taken up by serial memory transfers to and from the device. You decide to use CUDA streams to overlap the memory transfers with computation to decrease the total execution time. There is a constant overhead for starting a kernel computation on the GPU, which is independent of the size of the data that the kernel is processing. (Note that kernel launch cannot overlap with kernel execution.) The following diagram illustrates the timeline for a single stream execution (not to scale).

Input Transfer A (Host to Device)	Kernel Launch Overhead	Compute Time	Output Transfer B (Device to Host)
--------------------------------------	---------------------------	--------------	---------------------------------------

The times for each of the section shown above is as follows:

- Total Input Transfer time: 10 sec
- Constant Kernel Launch overhead: 1 sec
- Total Compute time: 10 sec
- Total Output Transfer Time: 10 sec
- Assume that you have access to 3 hardware streams that enable continuous pipelining. How many segments should the input vector be divided into so as to minimize the total execution time where continuous pipelining is possible?
- A:



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 21

GPU as part of the PC Architecture

Course Reminders

- MP 6
 - due next week
- MP 7
 - due the week after
- Project PM 2
 - due this Friday
- Project PM 3
 - due in 3 weeks

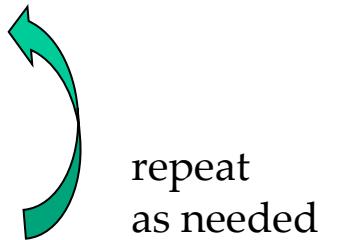
11	Lecture 20	Apr. 4	Sparse Matrix II
	Lecture 21	Apr. 6	GPU as a part of the PC Architecture
	PM 2	Apr. 7	Baseline GPU Convolution Kernel (GitHub, report)
	PM3 Released		GPU Convolution Kernel Optimizations
12	Lecture 22	Apr. 11	Task parallelism and asynchronous data transfer
		Apr. 13	Guest Lecture: TBD
	Lab 6	Apr. 14	Histogramming (Github, quiz)
13	Lecture 23	Apr. 18	Other acceleration APIs: OpenACC, OpenCL; MPI for distributed memory systems with GPUs
	Lecture 24	Apr. 20	Guest Lecture: TBD
	Lab 7	Apr. 21	Sparse Matrix Multiply (Github, quiz)
14	Lecture 25	Apr. 25	Guest Lecture: TBD
	Lecture 26	Apr. 27	Generalizing Parallelism and Course Retrospective
	PM 3	Apr. 28	GPU Convolution Kernel Optimizations
15	Exam 2	May 2	Midterm 2
	Final Competition (Optional)	May 5	Kernel Performance Competition (GitHub)

Objectives

- To understand the impact of data transfers on performance when using a GPU as a co-processor
 - speeds and feeds of traditional CPU
 - speeds and feeds when employing a GPU
- To develop a knowledge base for performance tuning for modern GPUs

Review: Canonical CUDA Program Structure

- Global variables declaration
- Kernel functions
 - `__global__ void kernelOne(...)`
- Main () // host code
 - allocate memory space on the device – `cudaMalloc(&d_GlbVarPtr, bytes)`
 - transfer data from host to device – `cudaMemcpy(d_GlbVarPtr, h_Gl...)`
 - execution configuration setup
 - kernel call – `kernelOne<<<execution configuration>>>(args...);`
 - transfer results from device to host – `cudaMemcpy(h_GlbVarPtr,...)`
 - optional: compare against golden (host computed) solution



Bandwidth:

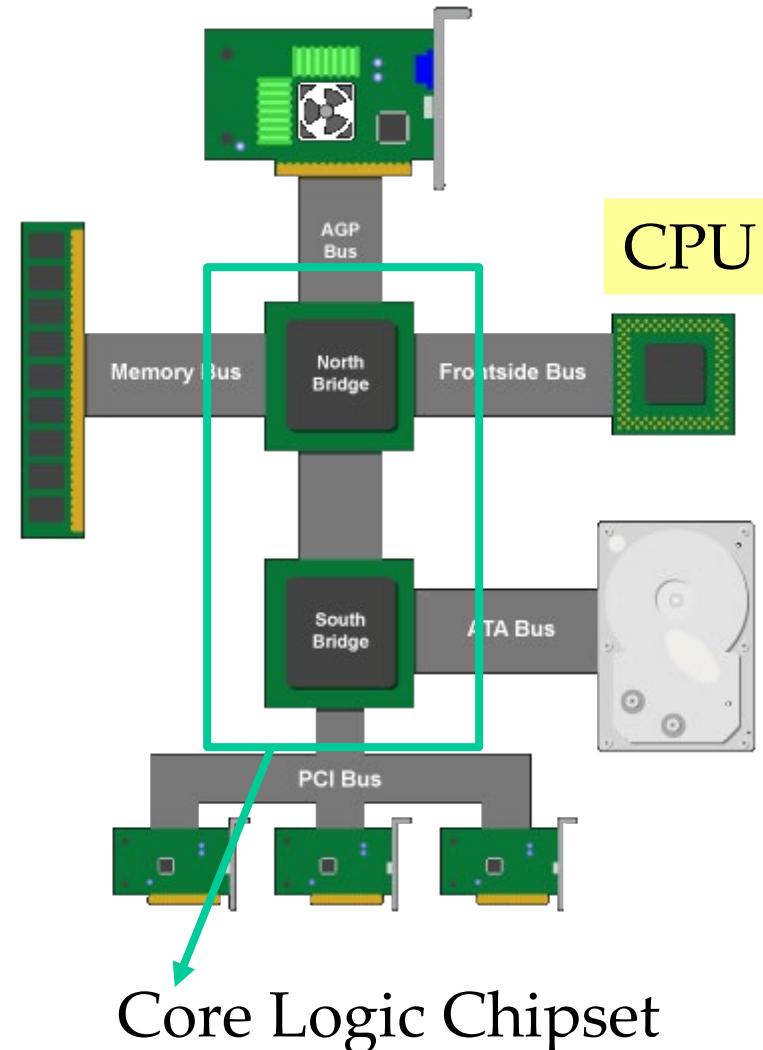
The Gravity of Modern Computer Systems

Bandwidth between key components ultimately **dictates system performance**

- **Especially for GPUs** processing large amounts of data.
- Tricks like buffering, reordering, caching can temporarily defy the rules in some cases.
- Ultimately, performance falls back to what the “speeds and feeds” dictate.

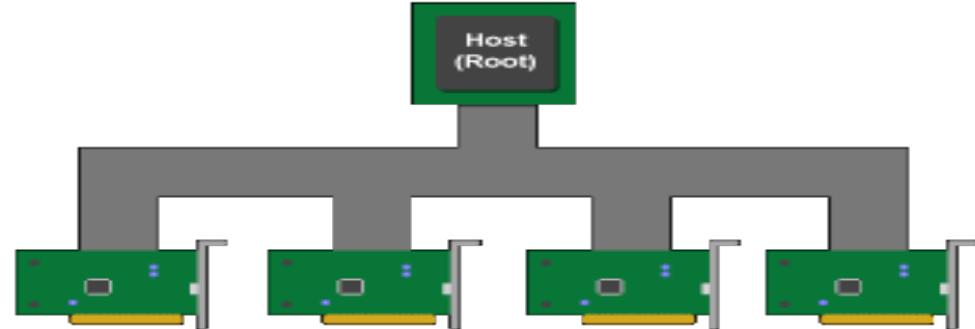
Classic (Historical) PC Architecture

- Northbridge connects 3 components that must communicate at high speed
 - CPU, DRAM, video
 - Video needs first-class access to DRAM
 - Previous NVIDIA cards are connected to AGP, up to 2 GB/s transfers
- Southbridge serves as a concentrator for slower I/O devices



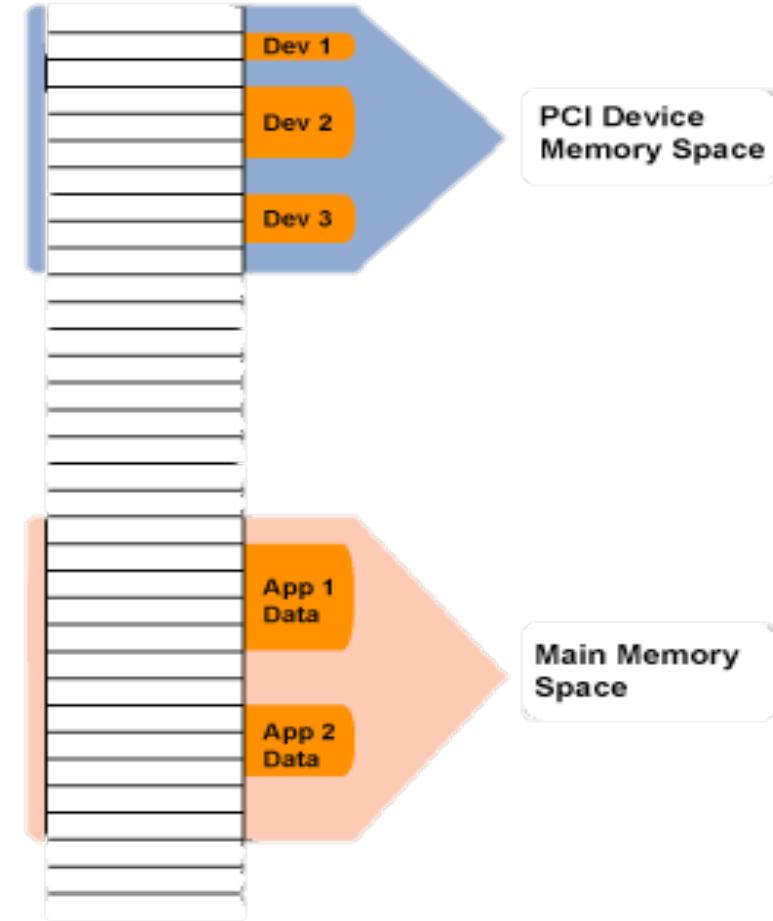
(Original) PCI Bus Specification

- Connected to the South Bridge
 - Originally 33 MHz, 32-bit wide, 132 MB/second peak transfer rate
 - Later, 66 MHz, 64-bit, 528 MB/second peak
 - Upstream bandwidth remain slow for device (~256MB/s peak)
 - **Shared bus with arbitration**
 - Winner of arbitration becomes bus master and can connect to CPU or DRAM through the southbridge and northbridge



PCI as Memory Mapped I/O

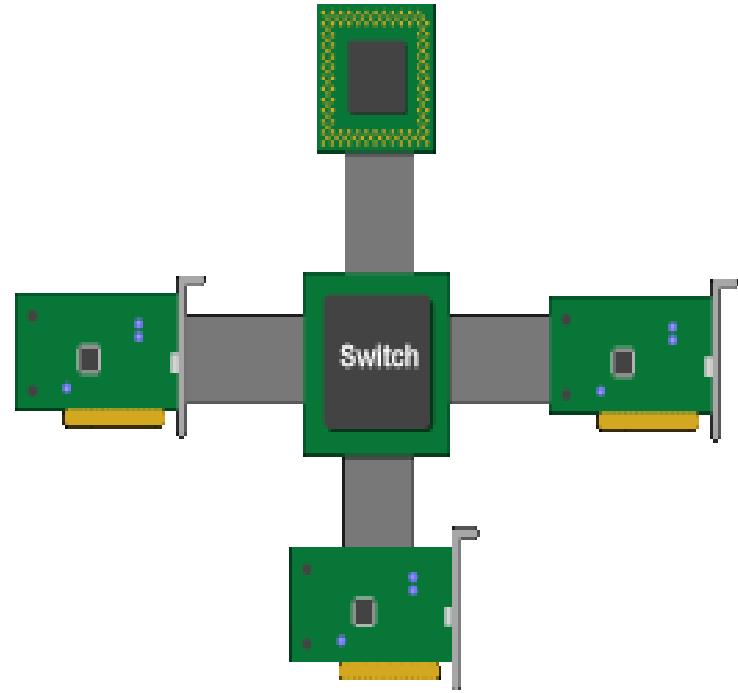
- PCI device registers are mapped into the CPU's physical address space
 - Accessed through loads/stores (kernel mode)
- Addresses are assigned to the PCI devices at boot time
 - All devices listen for their addresses



PCI Express (PCIe)

switched, point-to-point connection

- each card has dedicated “link” to the central switch, with no arbitration
- packet switches: messages form virtual channel
- prioritized packets for QoS (such as for real-time video streaming)

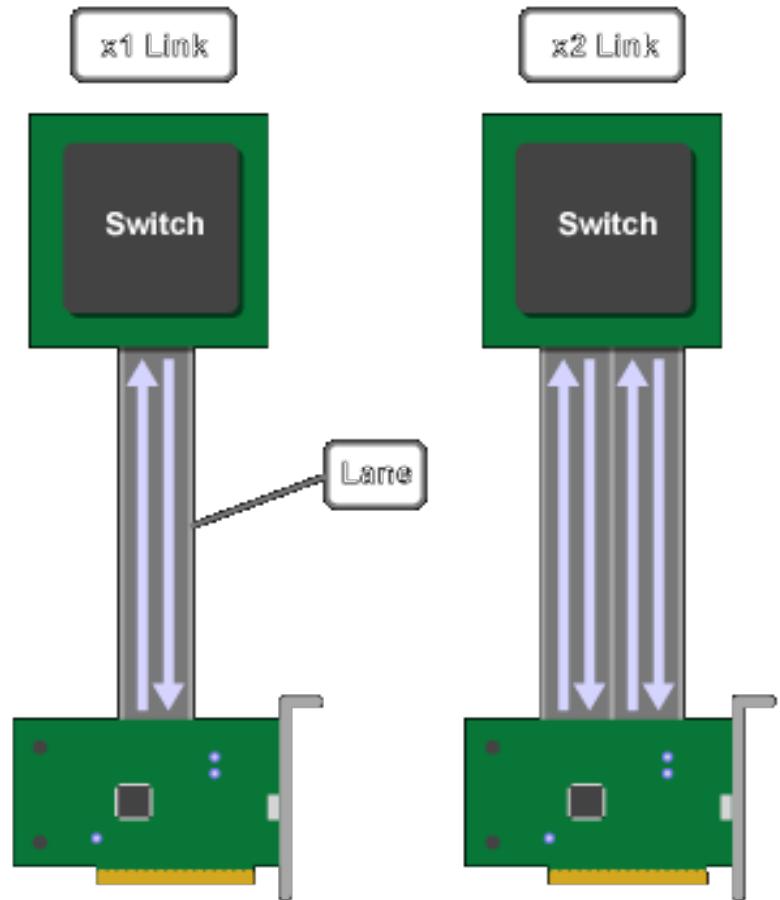


PCIe Generations

- Within a generation, number of lanes in a link can be scaled
 - using distinct physical channels (more bits / wider transfers)
 - $\times 1, \times 2, \times 4, \times 8, \times 16, \times 32, \dots$
- Each new generation aims to double the speed
 - Current generation is PCIe 5.0, however it is supported only on a very limited set of systems, e.g., IBM Power10
 - 32GT/s
 - PCIe 4.0 is supported on modern AMD, Intel, and IBM systems
 - However, PCIe Gen. 3 is still very widely used

PCIe Gen 3 Links and Lanes

- Each link consists of one or more lanes
 - Each lane is 1-bit wide (4 wires, each 2-wire pair can transmit 8Gb/s in one direction)
 - 2-wire pair is used for differential signaling
 - Upstream and downstream simultaneous and symmetric
 - Each Link can combine 1, 2, 4, 8, 12, 16 lanes- x1, x2, etc.
- Each byte data is **128b/130b** encoded into 130 bits with equal number of 1's and 0's; net data rate 7.8768 GB/s per lane each way.
 - Thus, the net data rates are 985 MB/s (x1) 1.97 GB/s (x2), 3.94 GB/s (x4), 7.9 GB/s (x8), 15.8 GB/s (x16), each way



Foundation: 8/10 bit encoding

- Goal is to maintain DC balance while have sufficient state transition for clock recovery
- The difference of 1s and 0s in a 20-bit stream should be ≤ 2
- There should be no more than 5 consecutive 1s or 0s in any stream
- 00000000, 00000111, 11000001 bad
- 01010101, 11001100 good
- Find 256 good patterns among 1024 total patterns of 10 bits to encode an 8-bit datum
- a 20% overhead

Current: 128/130 bit encoding

- Same goal: maintain DC balance while have sufficient state transition for clock recovery
- 1.5% overhead instead of 20%
- Scrambler function: long runs of 0s, 1s vanishingly small
- Instead of guaranteed run length of 8/10b
- At least one bit shift every 66 bits

Patterns Contain Many 0s and 1s

A question for fun:

- if we need 2^{128} code words
- chosen from all 2^{130} 130-bit patterns
- how many 0s/1s must we consider including?

Answer: 63-67 (of either type)

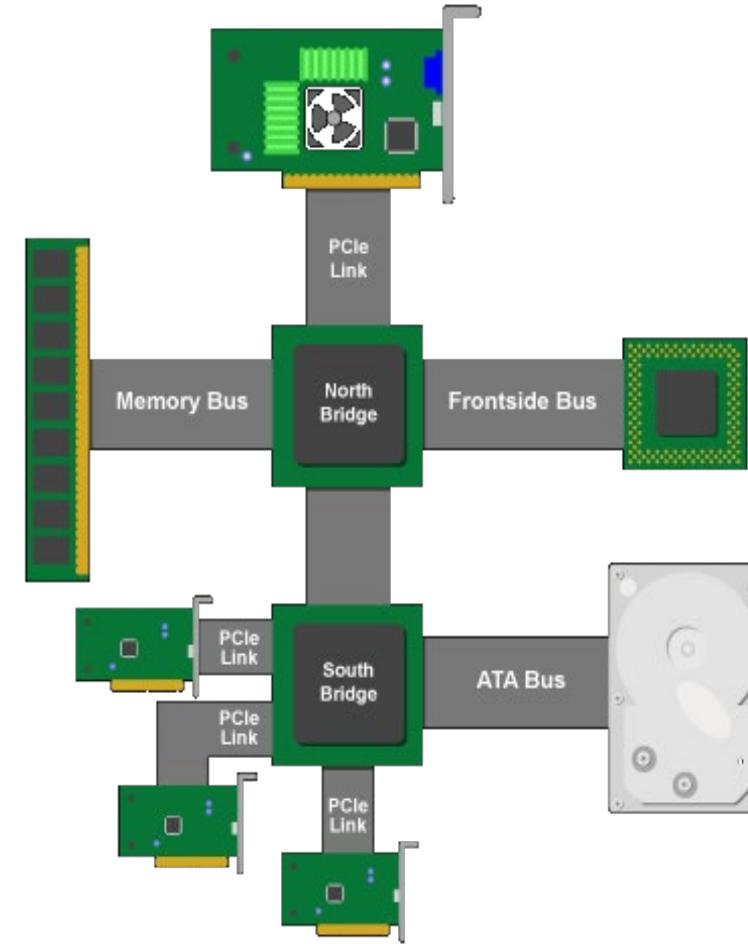
Thus 128b/130b code words are pretty well-balanced, and have lots of 0-1 transitions (for clock recovery).

Recent PCIe PC Architecture

PCIe forms the interconnect backbone within PC.

Northbridge and Southbridge are PCIe switches.

Source: Jon Stokes, PCI Express: An Overview
(<http://arstechnica.com/articles/paedia/hardware/pcie.ars>)



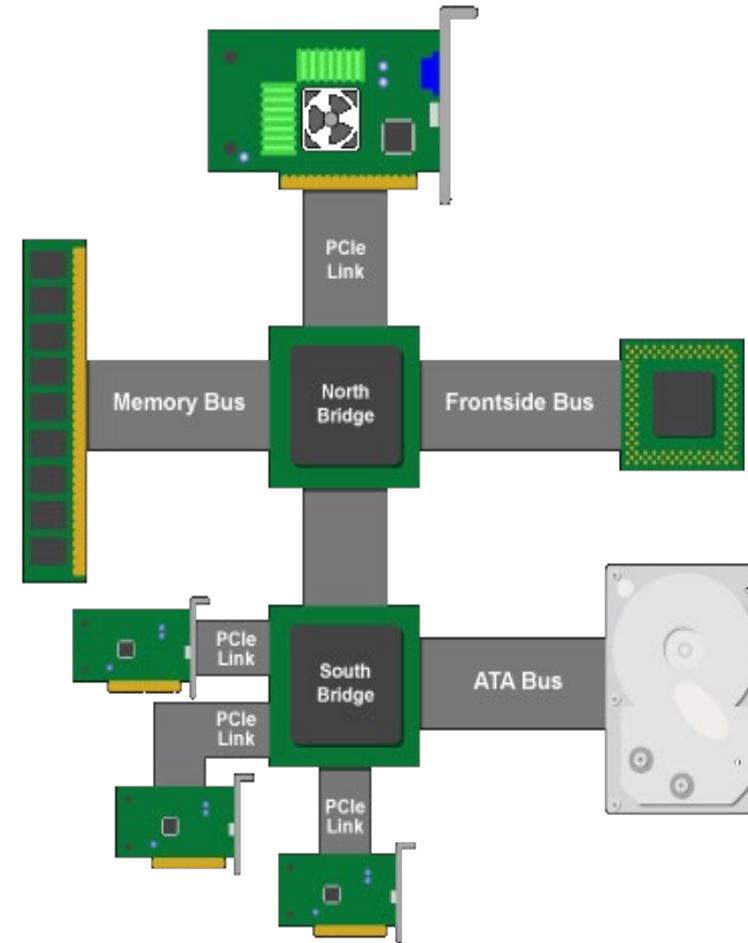
Recent PCIe PC Architecture

How is PCI supported?

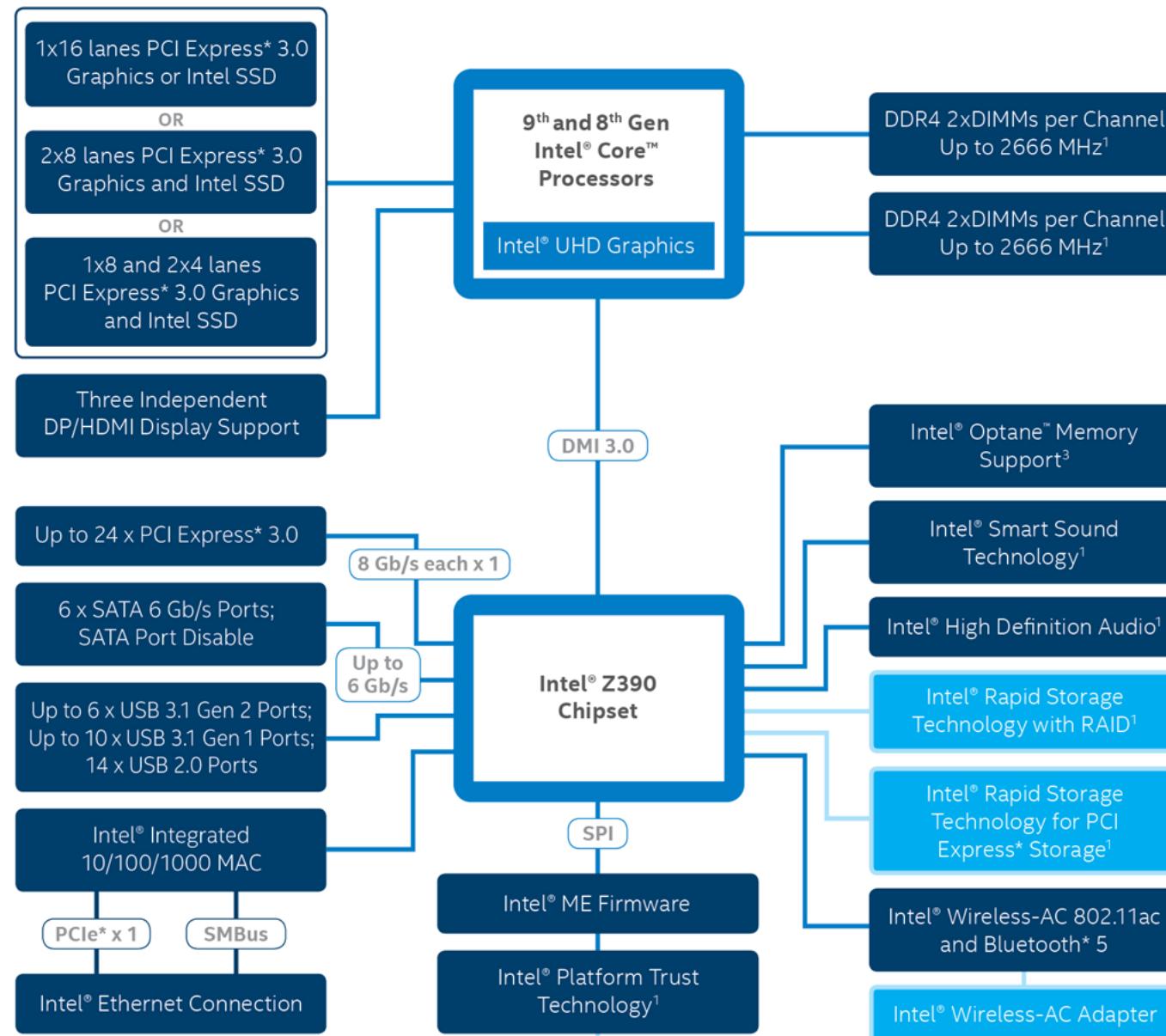
- Need a PCI-PCIe bridge, which is
- sometimes included as part of Southbridge, or
- can add as a separate PCIe I/O card.

Current systems integrate PCIe controllers directly on chip with CPU.

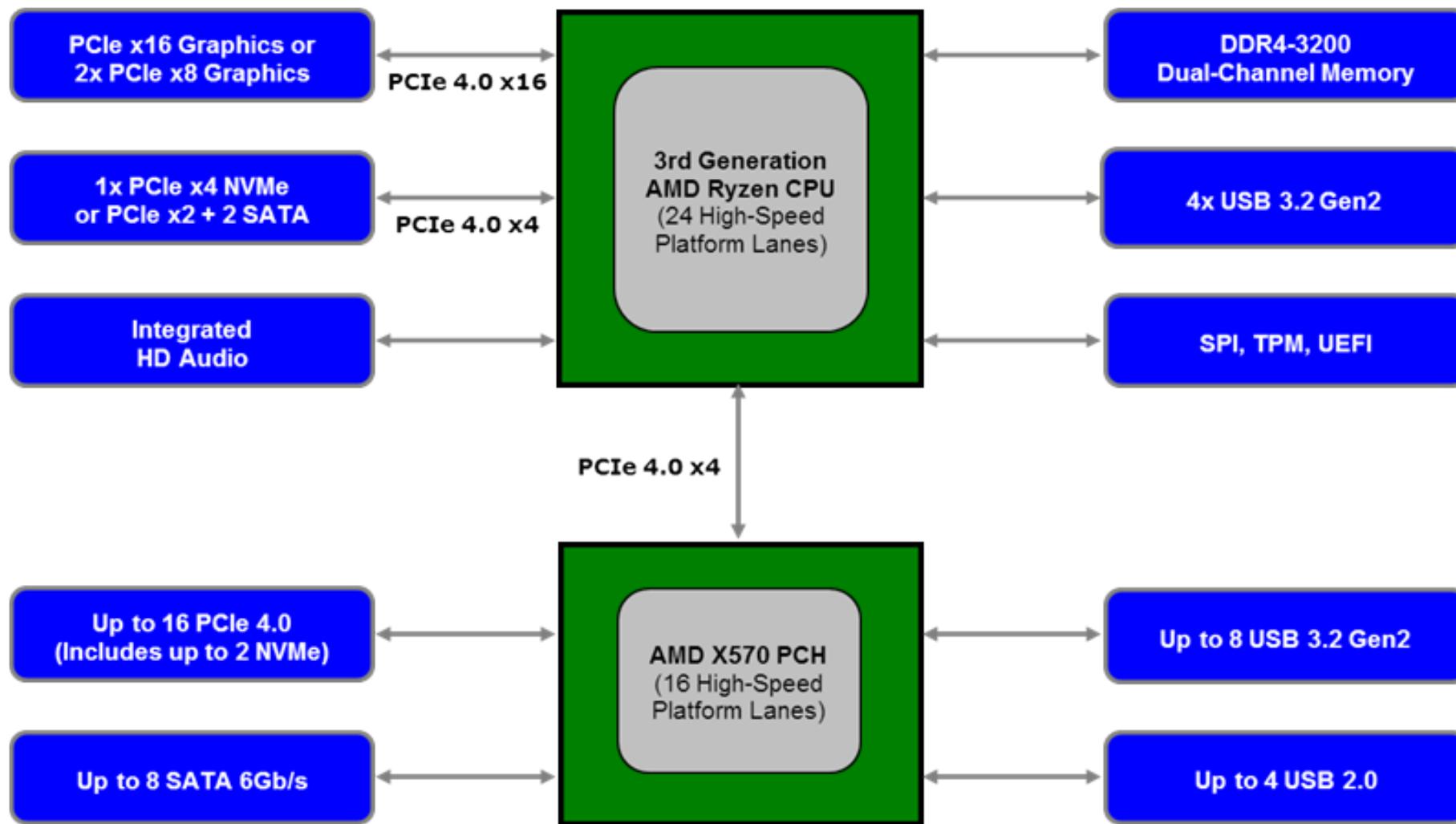
Source: Jon Stokes, PCI Express: An Overview
(<http://arstechnica.com/articles/paedia/hardware/pcie.ars>)



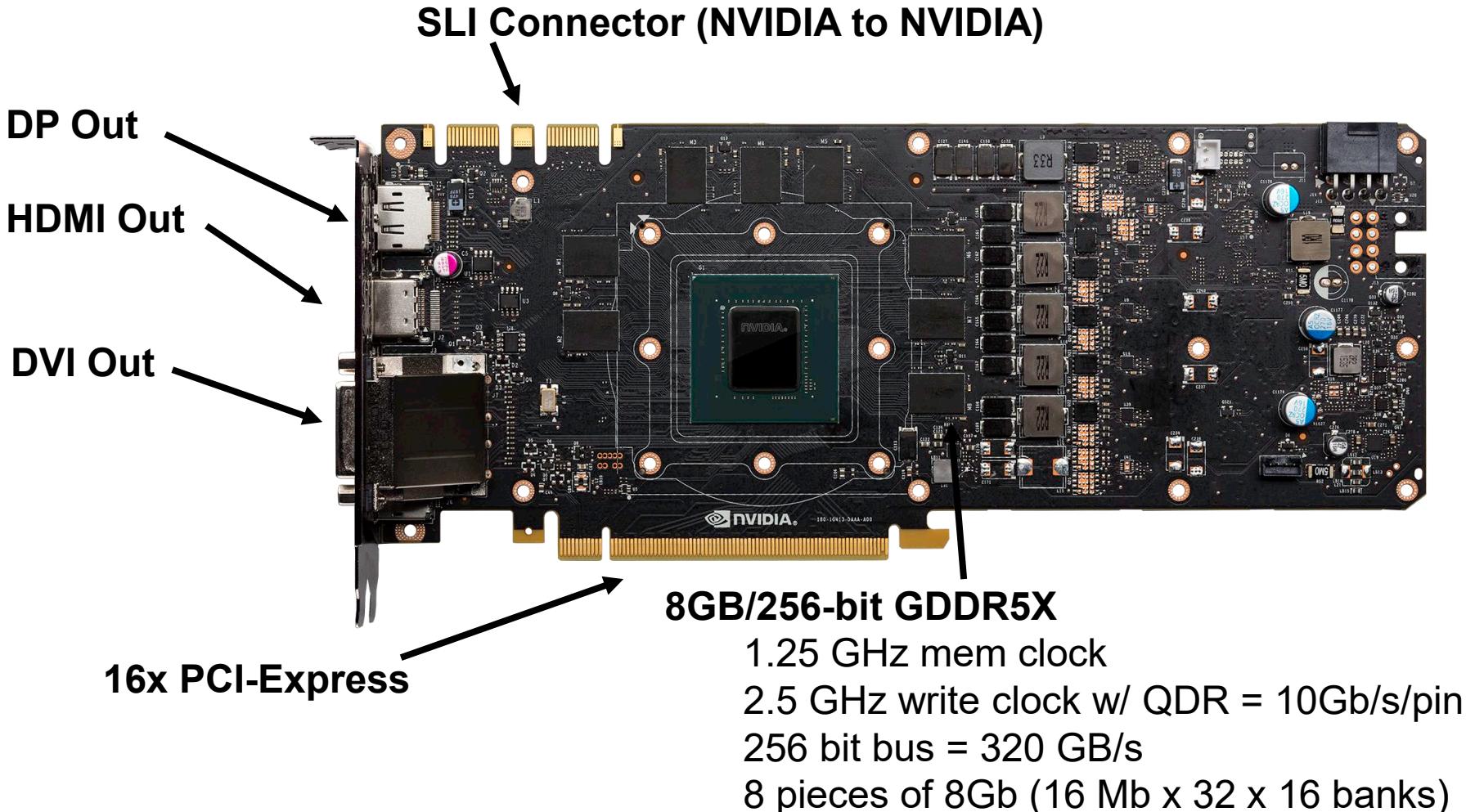
Modern Intel PCIe PC Architecture



Modern AMD PCIe PC Architecture



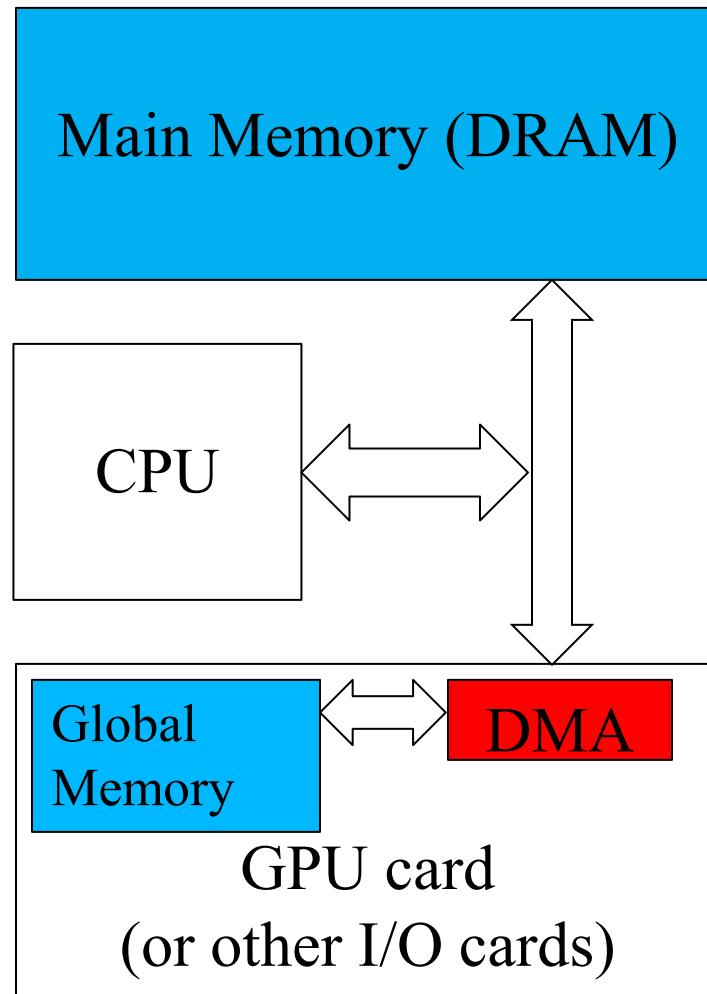
GeForce GTX 1080 (Pascal) GPU Consumer Card Details



PCIe Data Transfer using DMA

DMA (Direct Memory Access) is used to fully utilize the bandwidth of an I/O bus

- DMA uses physical address for source and destination
- Transfers a number of bytes requested by OS
- Needs pinned memory



Pinned Memory

- DMA uses physical addresses
- The OS could accidentally page out the data that is being read or written by a DMA and page in another virtual page into the same location
- Pinned memory cannot be paged out
- If a source or destination of a cudaMemcpy in the host memory is not pinned, it needs to be first copied to a pinned memory – extra overhead
- cudaMemcpy is much faster with pinned host memory source or destination

Allocate/Free Pinned Memory (a.k.a. Page Locked Memory)

- `cudaHostAlloc()`
 - Three parameters
 - Address of pointer to the allocated memory
 - Size of the allocated memory in bytes
 - Option – use `cudaHostAllocDefault` for now
- `cudaFreeHost()`
 - One parameter
 - Pointer to the memory to be freed

Using Pinned Memory

- Use the allocated memory and its pointer the same way those returned by malloc();
- The only difference is that the allocated memory cannot be paged by the OS
- The cudaMemcpy function should be about 2x faster with pinned memory
- Pinned memory is a limited resource whose over-subscription can have serious consequences

Important Trends

- Knowing yesterday, today, and tomorrow
 - The PC world is becoming flatter
 - CPU and GPU are being fused together
 - Outsourcing of computation is becoming easier...

ANY MORE QUESTIONS

Problem Solving

- Q: The following table gives the SM limits for compute capability (CC) 1.3. Nvidia defines *occupancy* as the ratio of the number of warps per SM for a given kernel configuration divided by the device limit. In a GPU with CC 1.3, which kernel achieves the maximum occupancy?
- A:

	CC 1.3
Max warps per SM	32
Max blocks per SM	8
Shared memory per SM	16K
Registers per SM	16K
Max block size	512

kernel	blockDim	gridDim	Shared memory per block	Registers per block
K1	160	4096	7K	1K
K2	224	2048	8K	6K
K3	288	1024	10K	9K
K4	96	8192	4K	2K



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 20

Parallel Sparse Methods II

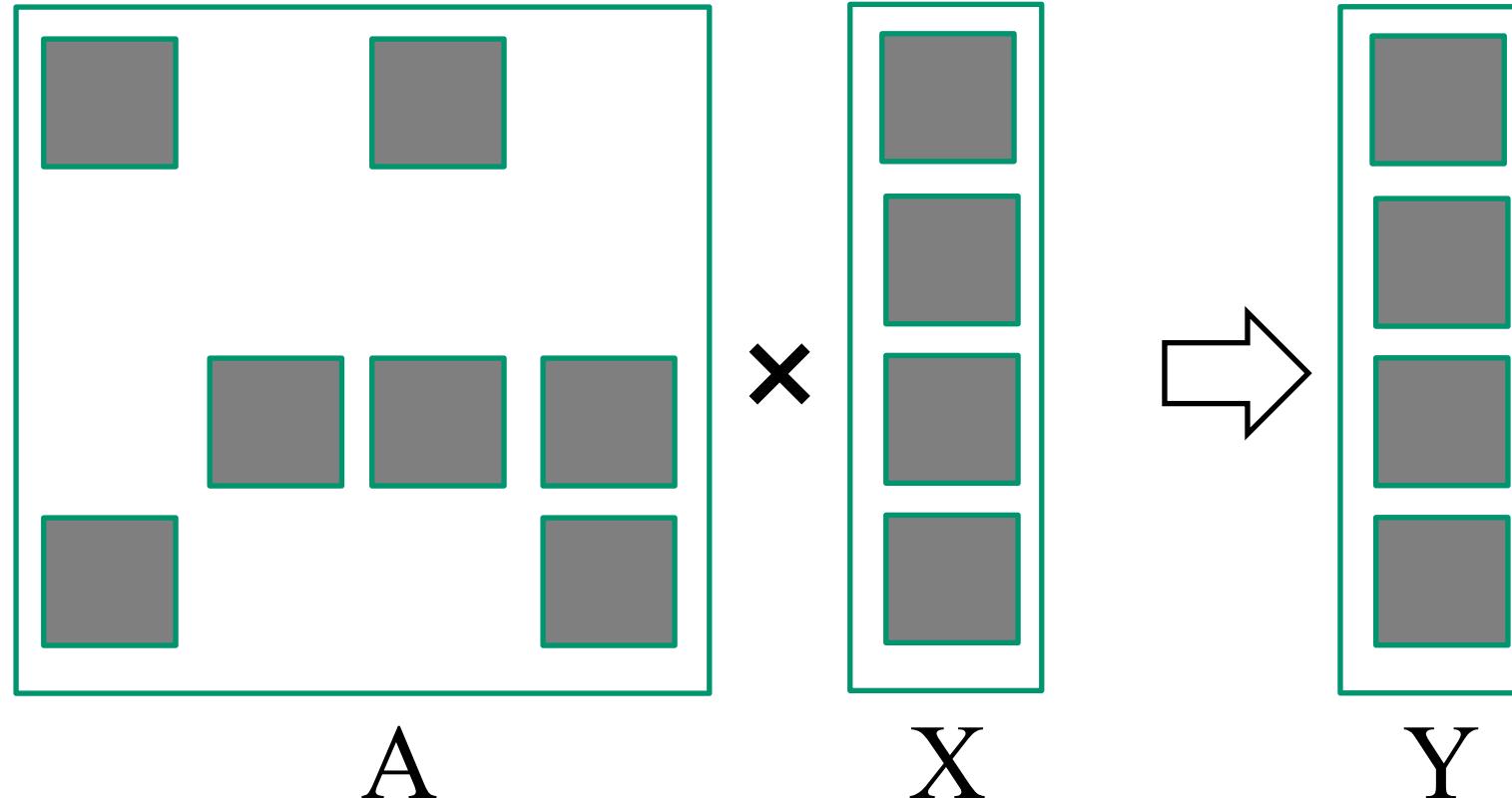
Course Reminders

- Assignments status
 - PM2 is due this Friday
 - PM3 will be released next Monday
 - MP5.1 is graded; we are grading MP5.2 now
- About PM2/3
 - We only have 8 GPUs in our Rai cluster, 6 of which will be (eventually) in the profiling (exclusive) queue
 - When you submit to the exclusive queue, nobody else can use the GPU
 - Running profiling with a batch of 10K can easily take 10 minutes or more
 - Therefore, develop and debug with a much smaller batch, e.g., of 100 or 1000, and profile only when your code is fully tested
 - Expect very long wait time towards the submission deadline
 - Main point: start early! Do not wait until the last hour to do your work, you will not be able to get the profiling done.

Objective

- To learn to regularize irregular data with
 - Limiting variations with clamping
 - Sorting
 - Transposition
- To learn to write a high-performance SpMV kernel based on JDS transposed format

Sparse Matrix-Vector Multiplication (SpMV)



Compressed Sparse Row (CSR) Format

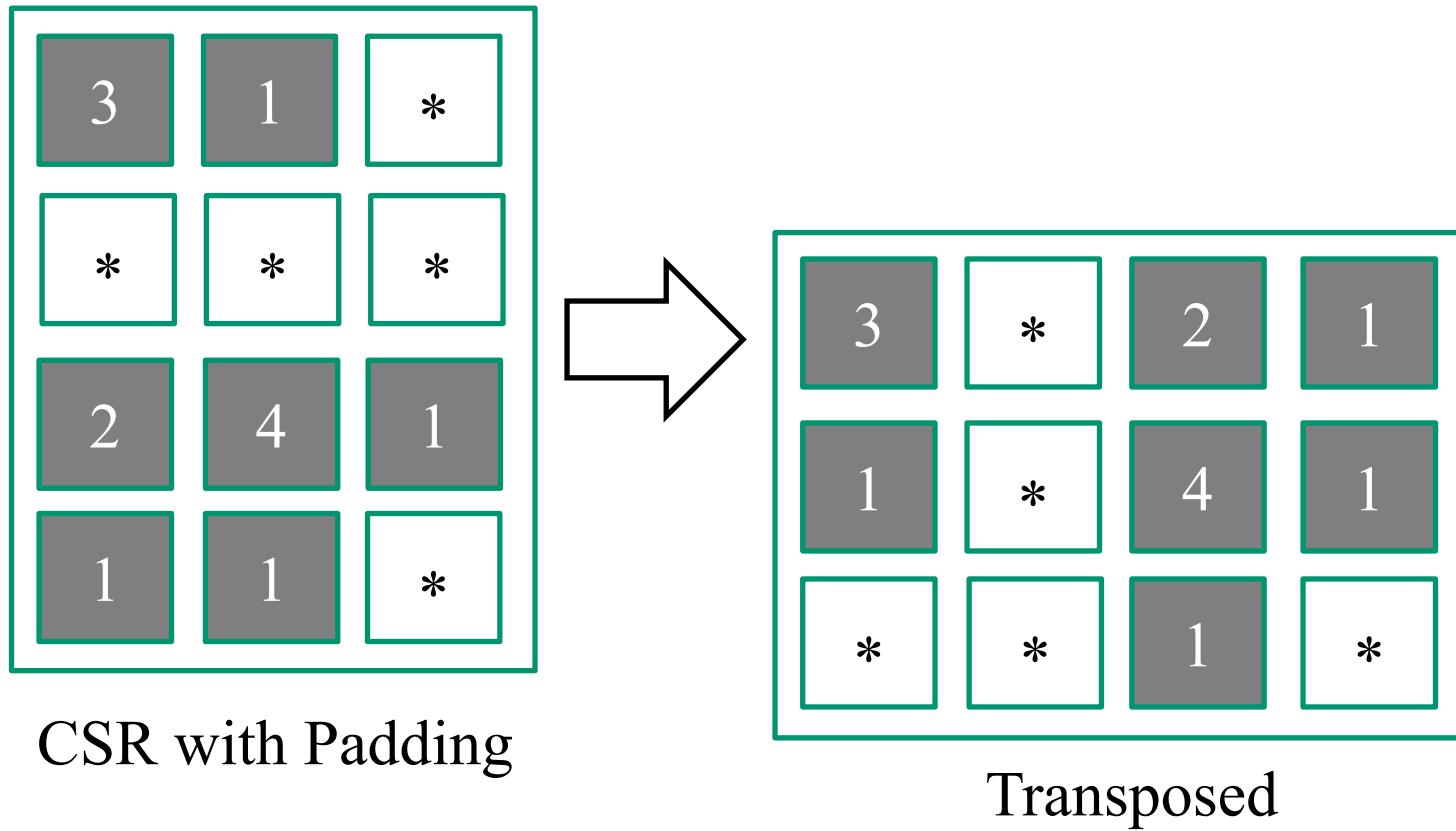
CSR Representation

		Row 0	Row 2	Row 3
Nonzero values	data [7]	{ 3, 1, 2, 4, 1, 1, 1 }		
Column indices	col_index [7]	{ 0, 2, 1, 2, 3, 0, 3 }		
Row Pointers	row_ptr [5]	{ 0, 2, 2, 5, 7 }		

Dense representation

Row 0	3	0	1	0	Thread 0
Row 1	0	0	0	0	Thread 1
Row 2	0	2	4	1	Thread 2
Row 3	1	0	0	1	Thread 3

Regularizing SpMV with ELL(PACK) Format



- Pad all rows to the same length
 - Inefficient if a few rows are much longer than others
- Transpose (Column Major) for DRAM efficiency
- Both data and col_index padded/transposed

Coordinate (COO) format

- Explicitly list the column & row indices for every non-zero element

		Row 0	Row 2	Row 3
Nonzero values	<code>data[7]</code>	{ 3, 1,	2, 4, 1,	1, 1 }
Column indices	<code>col_index[7]</code>	{ 0, 2,	1, 2, 3,	0, 3 }
Row indices	<code>row_index[7]</code>	{ 0, 0,	2, 2, 2,	3, 3 }

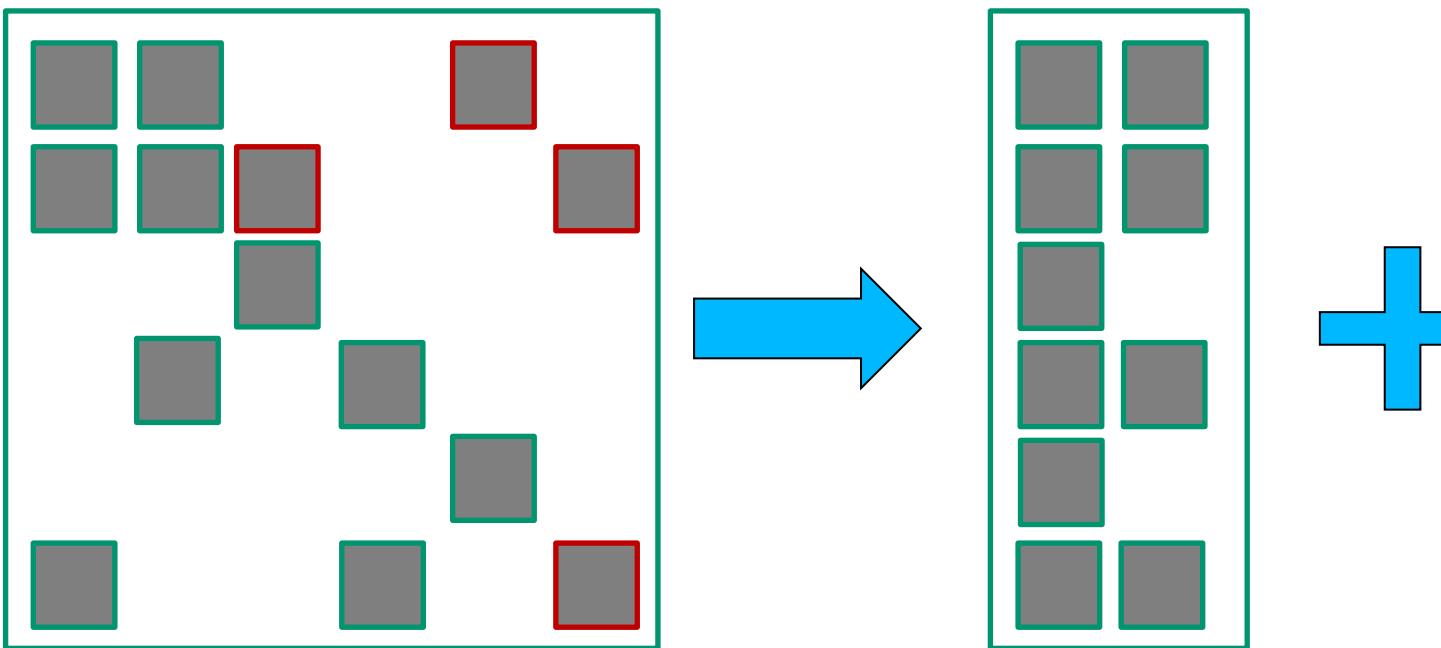
COO Allows Reordering of Elements

		Row 0	Row 2	Row 3
Nonzero values	data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
Column indices	col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
Row indices	row_index[7]	{ 0, 0,	2, 2, 2,	3, 3 }

Nonzero values	data[7]	{ 1 1, 2, 4, 3, 1 1 }
Column indices	col_index[7]	{ 0 2, 1, 2, 0, 3, 3 }
Row indices	row_index[7]	{ 3 0, 2, 2, 0, 2, 3 }

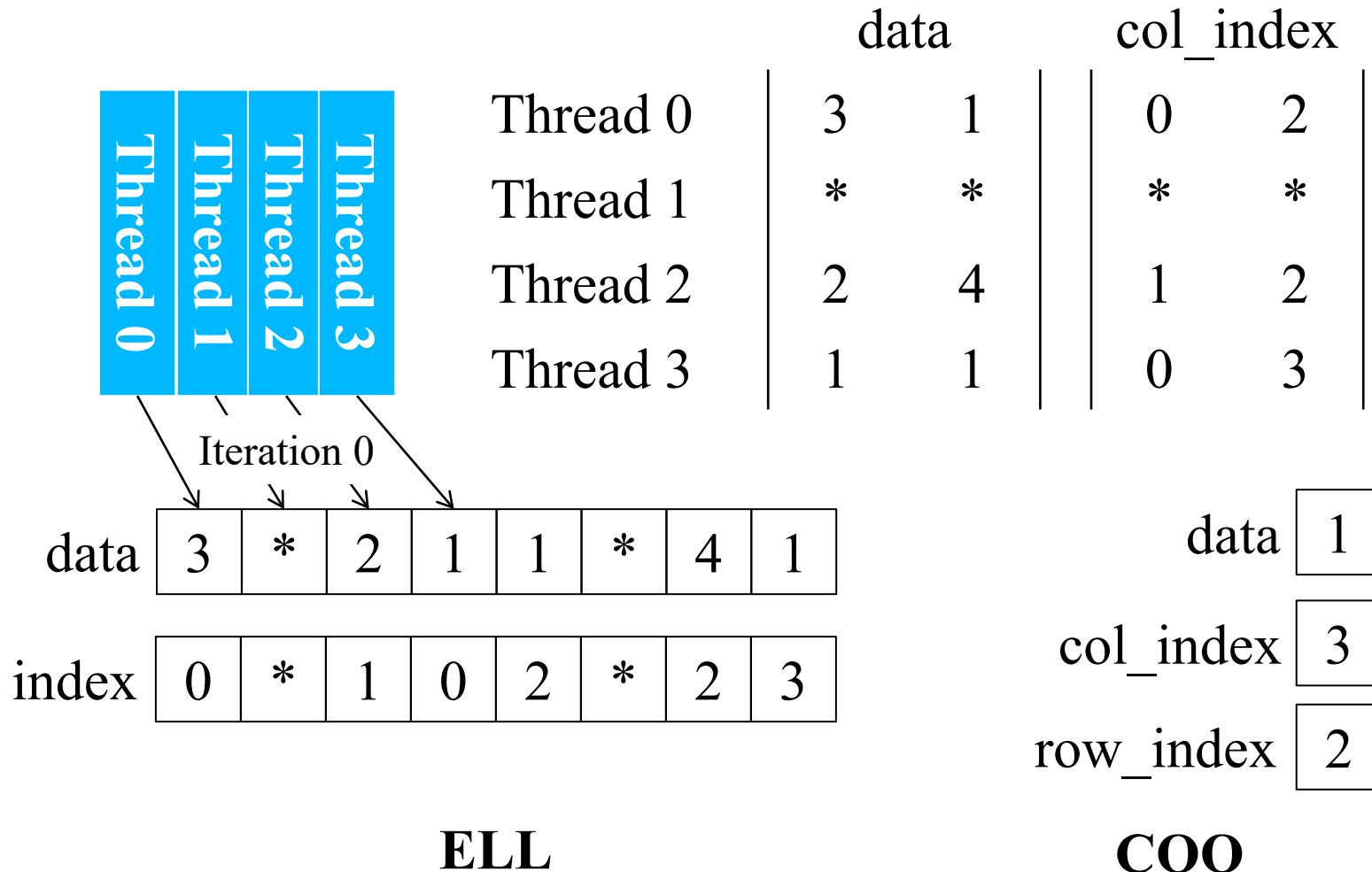
Hybrid Format (ELL + COO)

- ELL handles *typical* entries
- COO handles *exceptional* entries
 - Implemented with segmented reduction

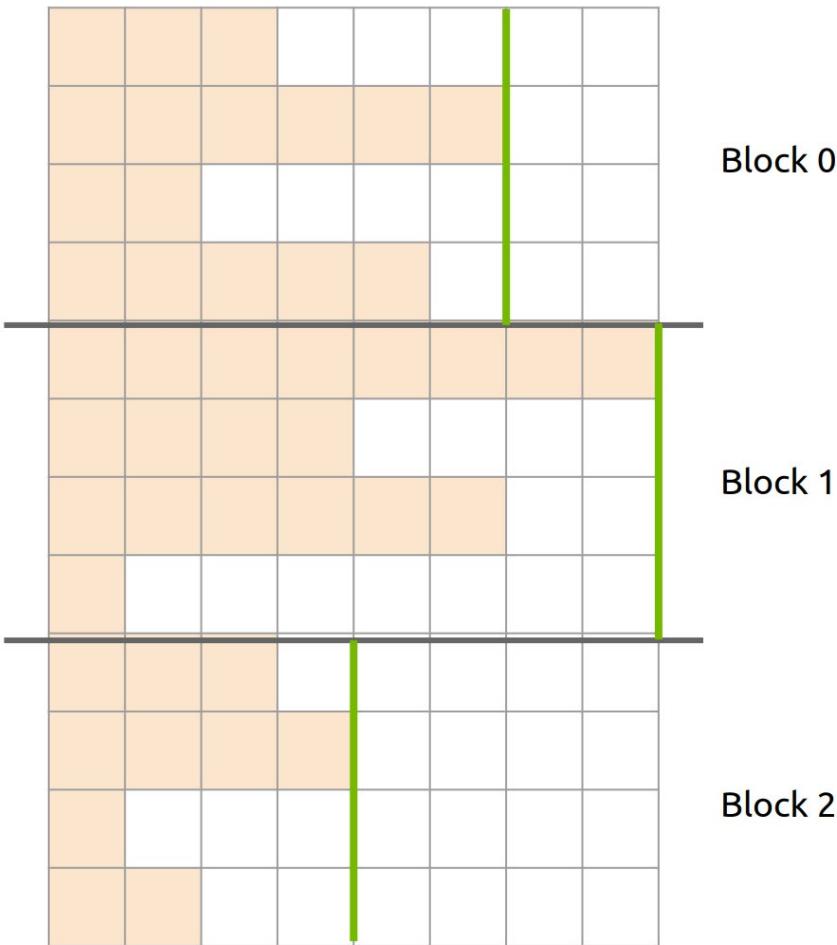


Often implemented in sequential host code in practice

Reduced Padding with Hybrid Format

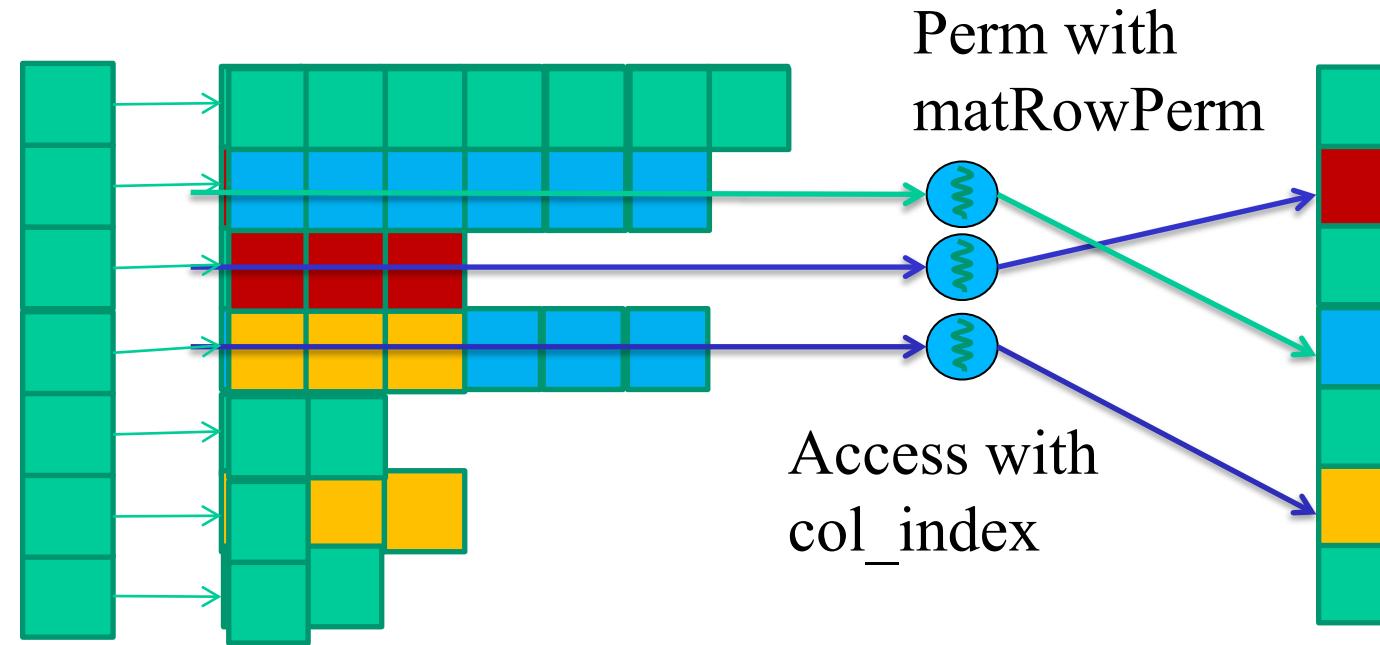


CSR Run-time



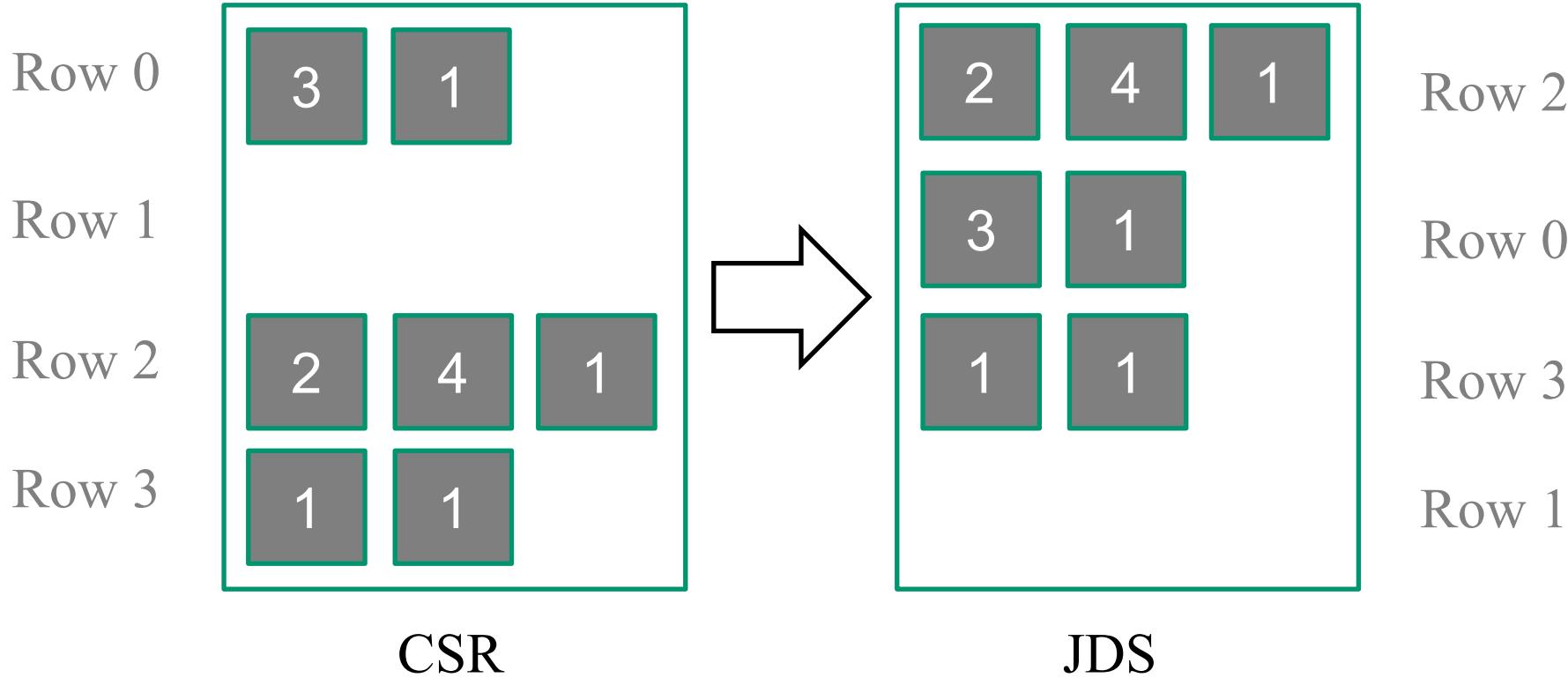
Block performance is determined by longest row

JDS (Jagged Diagonal Sparse) Kernel Design for Load Balancing



Sort rows into descending order according to number of non-zero. Keep track of the original row numbers so that the output vector can be generated correctly.

Sorting Rows According to Length (Regularization)



CSR to JDS Conversion

		Row 0	Row 2	Row 3
Nonzero values	data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
Column indices	col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
Row Pointers	row_ptr[5]	{ 0, 2, 2,		5, 7 }
			Row 2	Row 0 Row 3
Nonzero values	data[7]	{ 2, 4, 1,	3, 1,	1 1 }
Column indices	col_index[7]	{ 1, 2, 3,	0, 2,	0, 3 }
JDS Row Pointers	jds_row_ptr[5]	{ 0,	3,	5, 7, 7 }
JDS Row Indices	jds_row_perm[4]	{ 2,	0,	3, 1 }

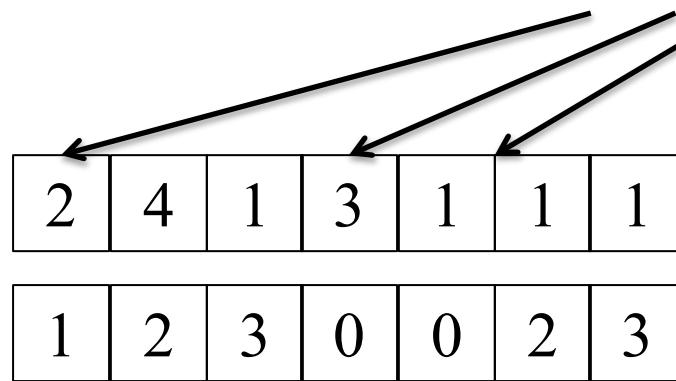
JDS Summary

Nonzero values `data[7]` { 2, 4, 1, 3, 1, 1, 1 }

Column indices `jds_col_index[7]` { 1, 2, 3, 0, 2, 0, 3 }

JDS row indices `jds_row_perm[4]` { 2, 0, 3, 1 }

JDS Row Ptrs `jds_row_ptr[5]` { 0, 3, 5, 7, 7 }



A Parallel SpMV/JDS Kernel

```
1. __global__ void SpMV_JDS(int num_rows, float *data, int *col_index,
   int *jds_row_ptr, int *jds_row_perm, float *x, float *y) {
2.     int row = blockIdx.x * blockDim.x + threadIdx.x;
3.     if (row < num_rows) {
4.         float dot = 0;
5.         int row_start = jds_row_ptr[row];
6.         int row_end = jds_row_ptr[row+1];
7.         for (int elem = row_start; elem < row_end; elem++) {
8.             dot += data[elem] * x[col_index[elem]];
9.         }
10.        y[jds_row_perm[row]] = dot;
11.    }
12. }
```

Nonzero values data[7]

Row 2	Row 0	Row 3
{ 2, 4, 1, }	{ 3, 1, }	{ 1, 1 }

Column indices col_index[7]

{ 1, 2, 3, }	{ 0, 2, }	{ 0, 3 }
--------------	-----------	----------

JDS Row Pointers jds_row_ptr[5]

{ 0, }	{ 3, }	{ 5, }	{ 7, 7 }
--------	--------	--------	----------

JDS Row Indices jds_row_perm[4]

{ 2, }	{ 0, }	{ 3, }	{ 1 }
--------	--------	--------	-------

JDS vs. CSR - Control Divergence

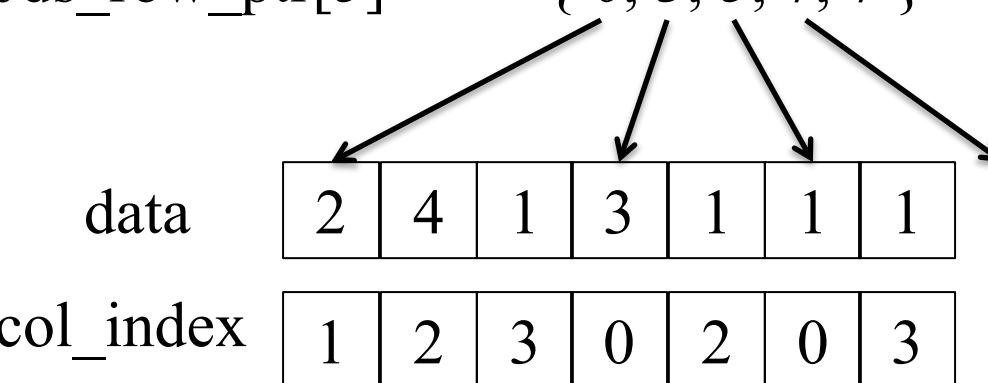
- Threads still execute different number of iterations in the JDS kernel for-loop
 - However, neighboring threads tend to execute similar number of iterations because of sorting.
 - Better thread utilization, less control divergence

Nonzero values `data[7]` $\{ 2, 4, 1, 3, 1, 1, 1 \}$

Column indices `col_index[7]` $\{ 1, 2, 3, 0, 2, 0, 3 \}$

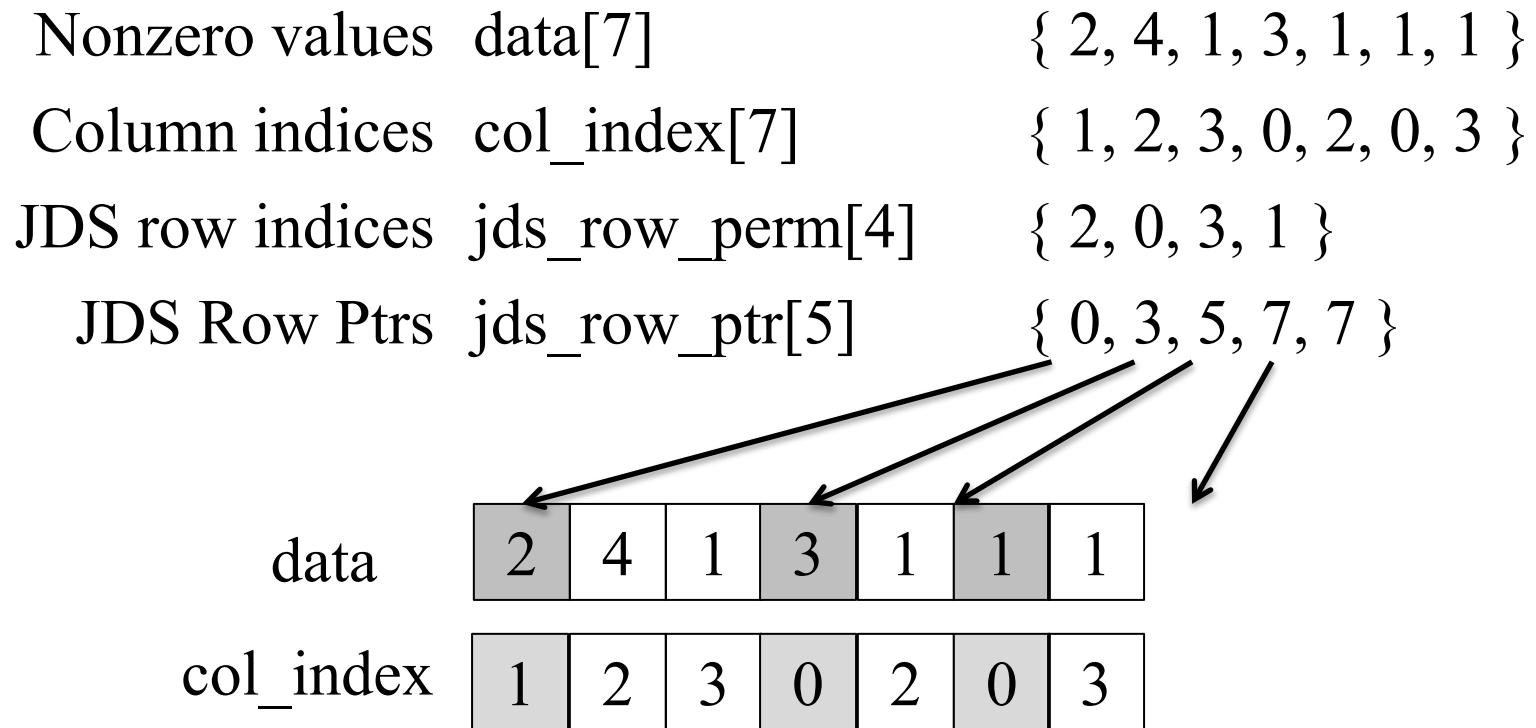
JDS row indices `Jds_row_perm[4]` $\{ 2, 0, 3, 1 \}$

JDS Row Ptrs `Jds_row_ptr[5]` $\{ 0, 3, 5, 7, 7 \}$

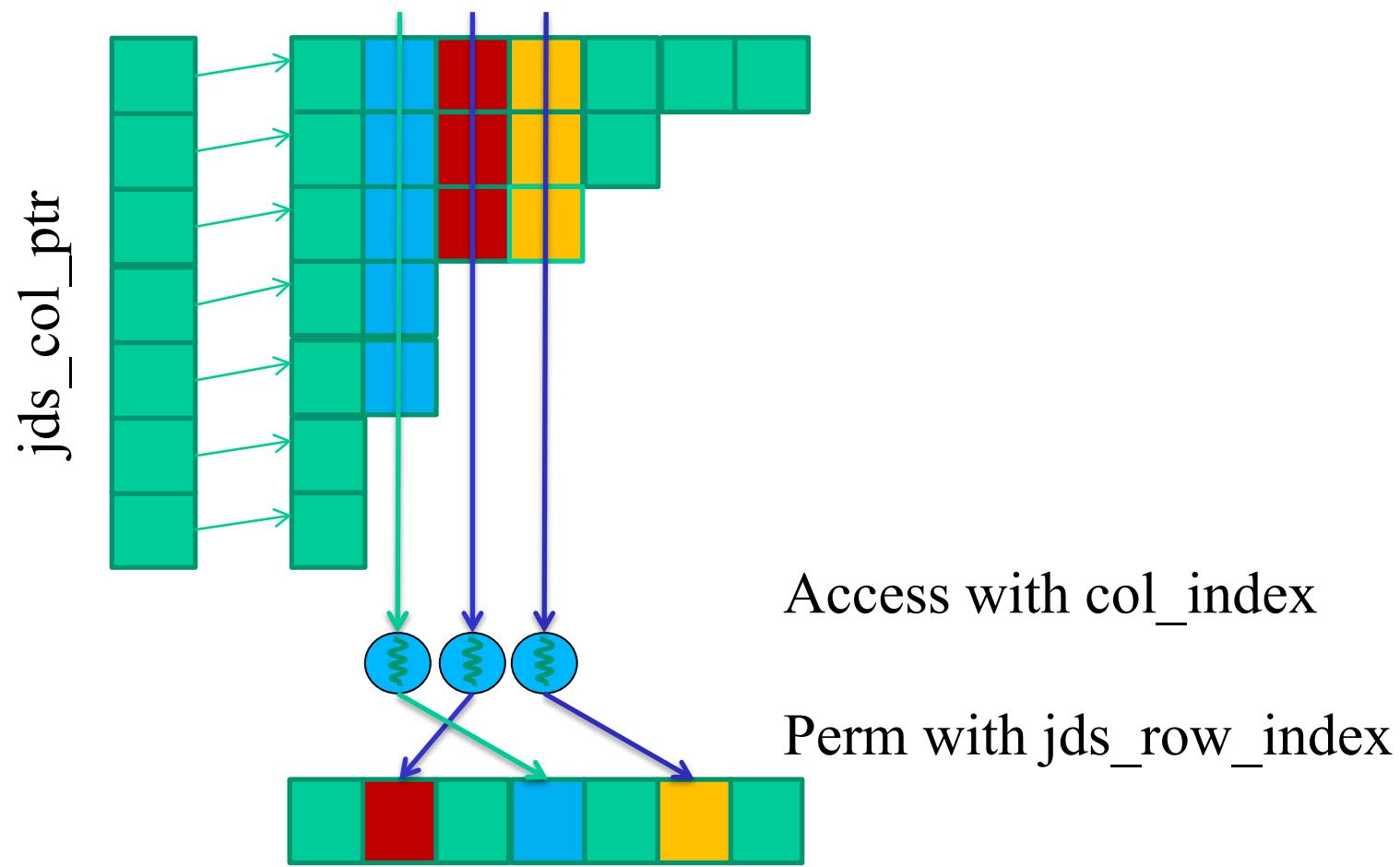


JDS vs. CSR Memory Divergence

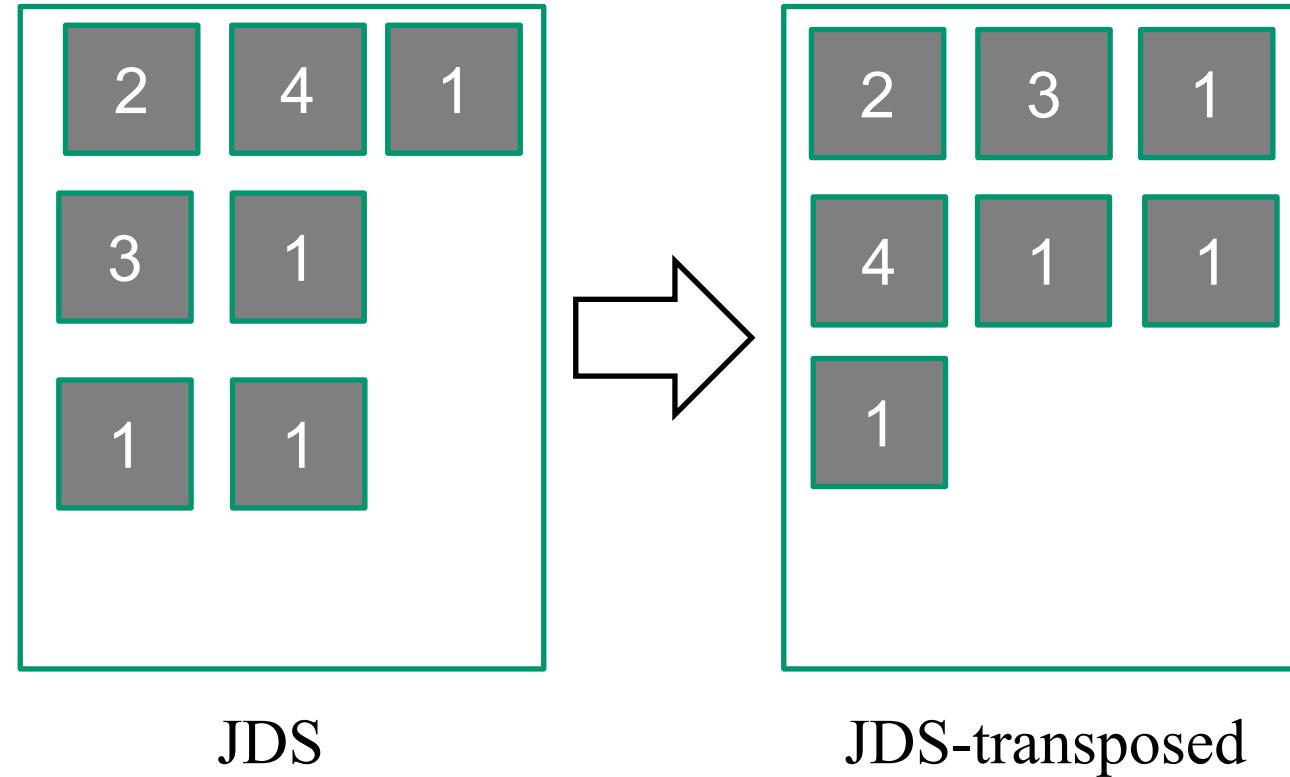
- Adjacent threads still access non-adjacent memory locations



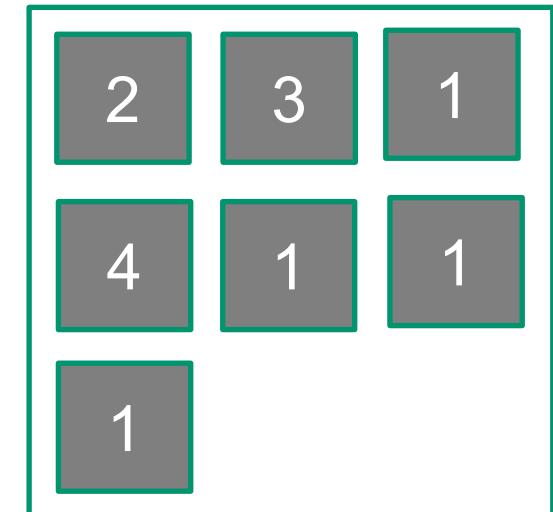
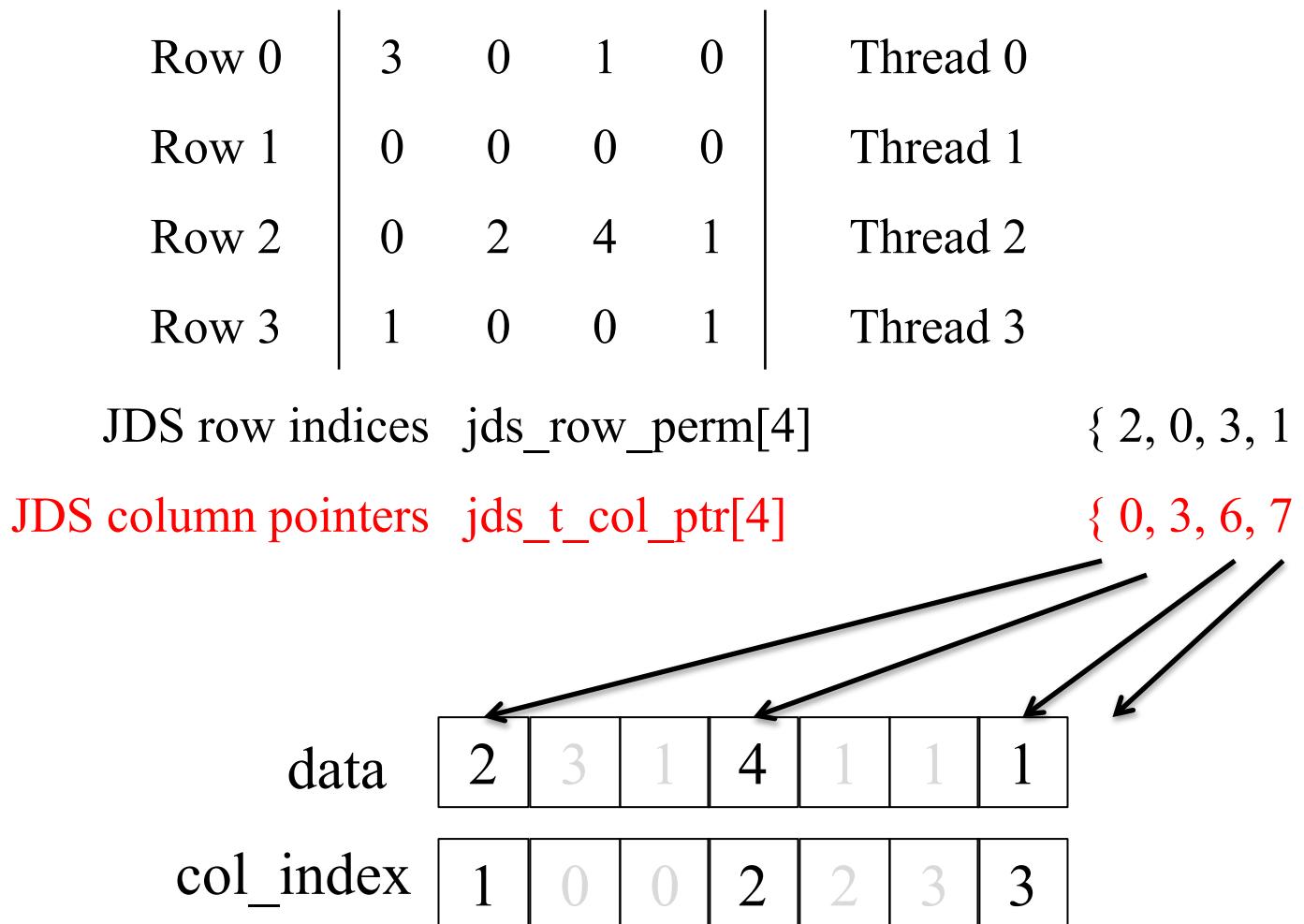
JDS with Transposition



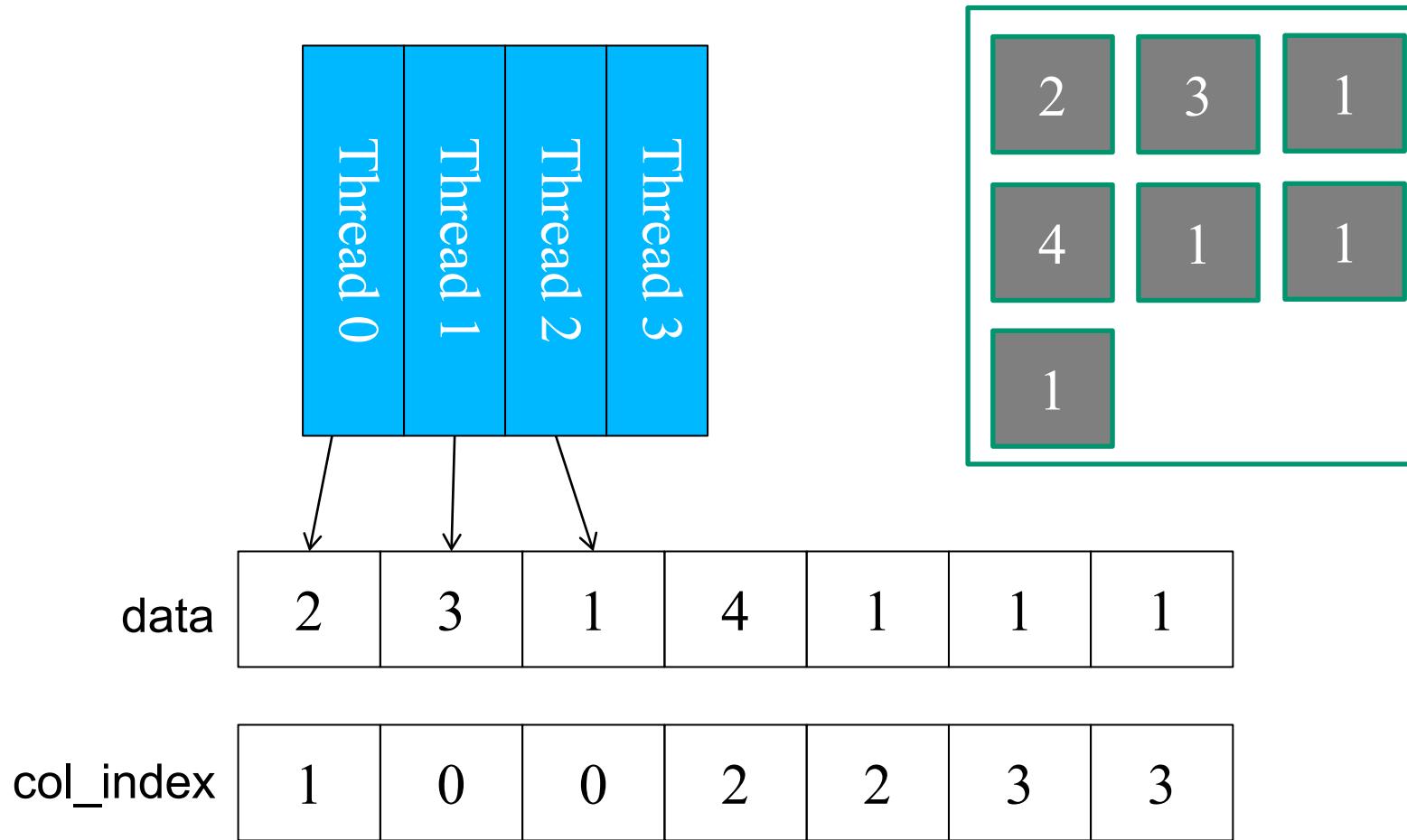
Transposition for Memory Coalescing



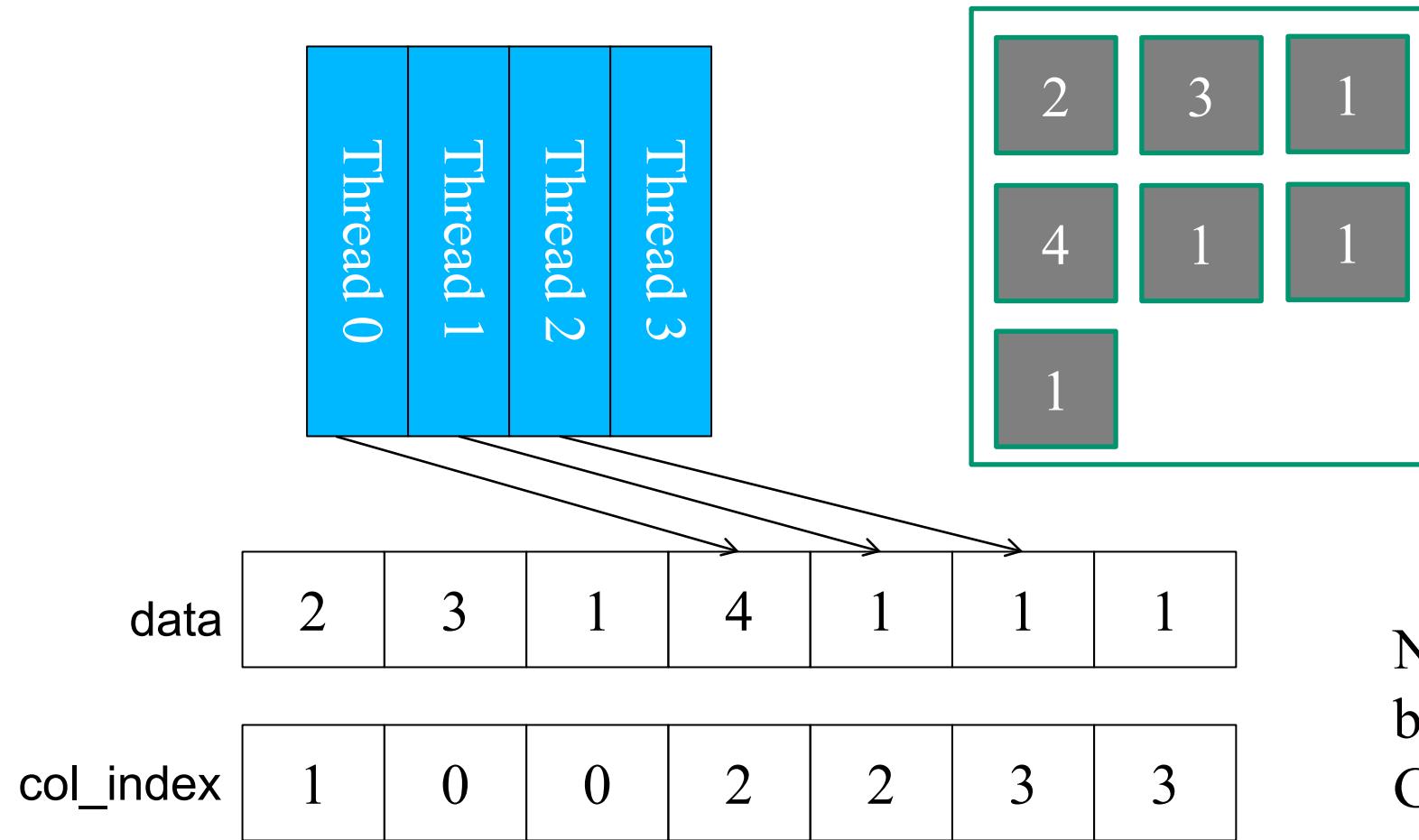
JDS Format with Transposed Layout



JDS with Transposition: Memory Coalescing

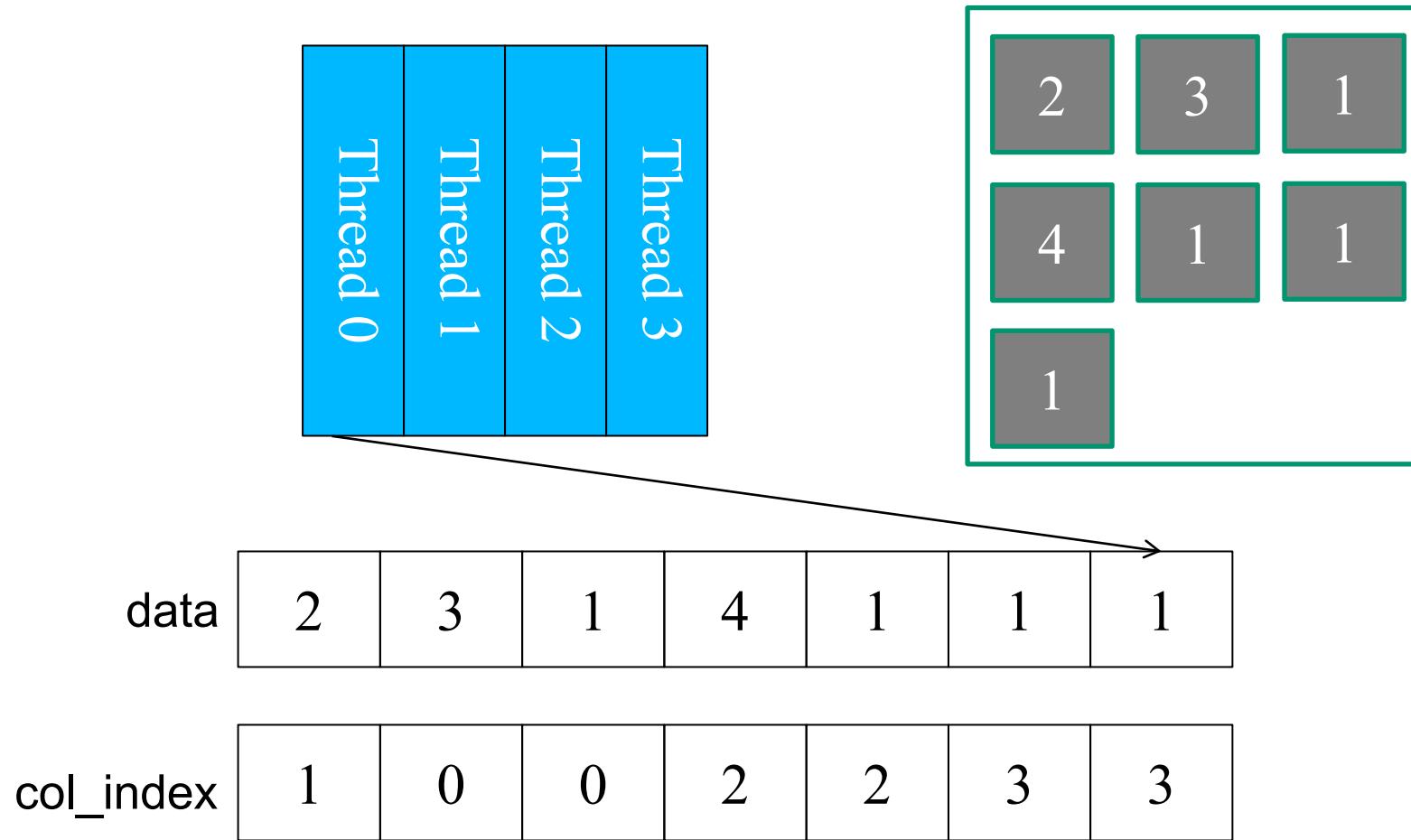


JDS with Transposition: Memory Coalescing



Not aligned with DRAM
bursts but OK with recent
GPUs

JDS with Transposition: Memory Coalescing



A Parallel SpMV/JDS_T Kernel

```
1. __global__ void SpMV_JDS_T(int num_rows, float *data, int *col_index,
   int *jds_t_col_ptr, int *jds_row_perm, float *x, float *y) {
2.     int row = blockIdx.x * blockDim.x + threadIdx.x;
3.     if (row < num_rows) {
4.         float dot = 0;
5.         unsigned int sec = 0;
6.         while (jds_t_col_ptr[sec+1] - jds_t_col_ptr[sec] > row) {
7.             dot += data[jds_t_col_ptr[sec]+row] *
8.                 x[col_index[jds_t_col_ptr[sec]+row]];
9.             sec++;
10.        }
11.        y[jds_row_perm[row]] = dot;
12.    }
13. }
```

Nonzero values data[7]

Sec 0

{ 2, 3, 1,

Sec 1

{ 4, 1, 1 }

Sec 2

{ 1 }

Column indices col_index[7]

{ 1, 0, 0,

{ 2, 2, 3 }

{ 3 }

JDS_T Column Pointers jds_t_col_ptr[5]

{ 0,

{ 3,

{ 6,

{ 7,7 }

JDS Row Indices jds_row_perm[4]

{ 2,

{ 0,

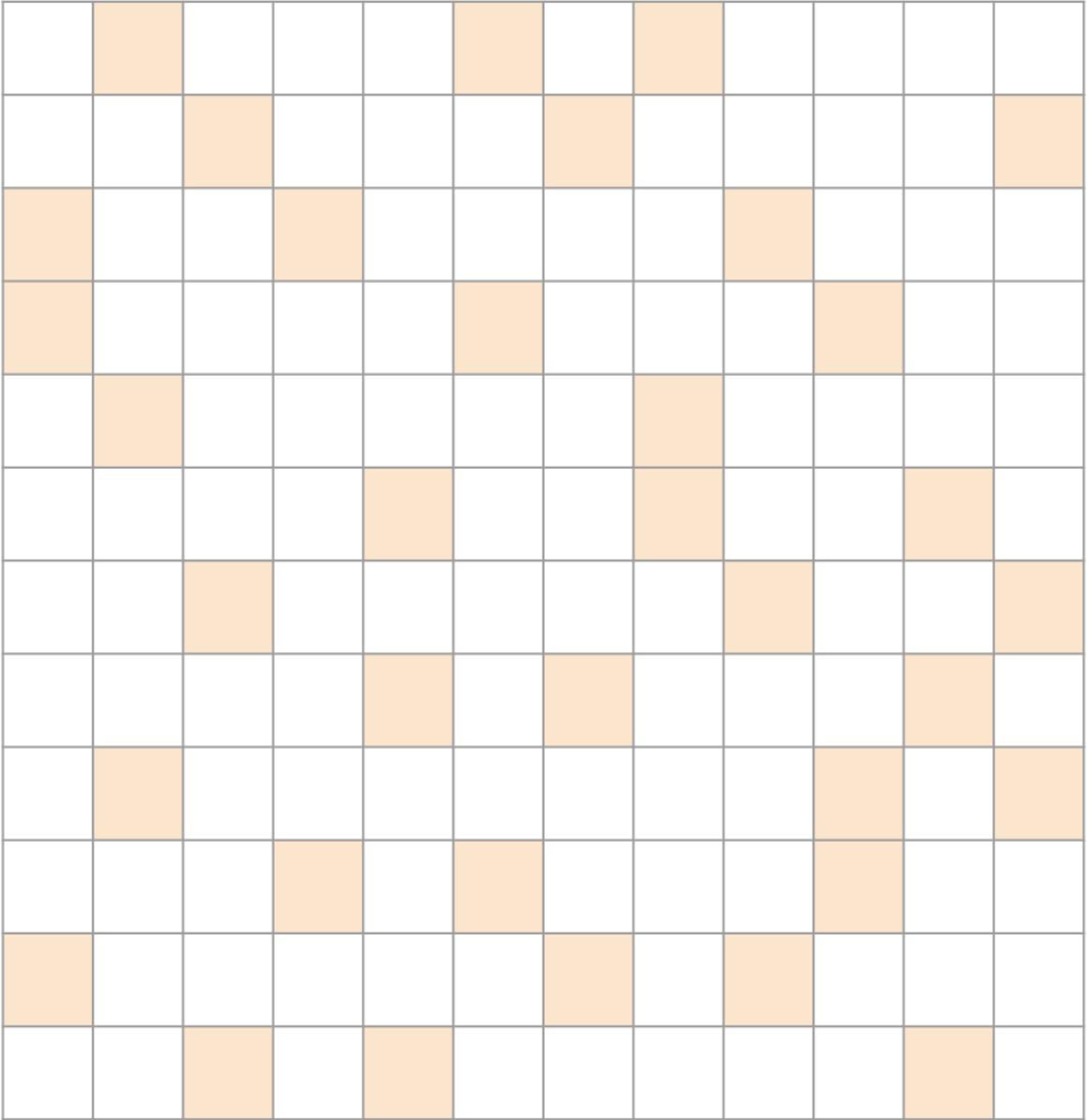
{ 3,

{ 1 }

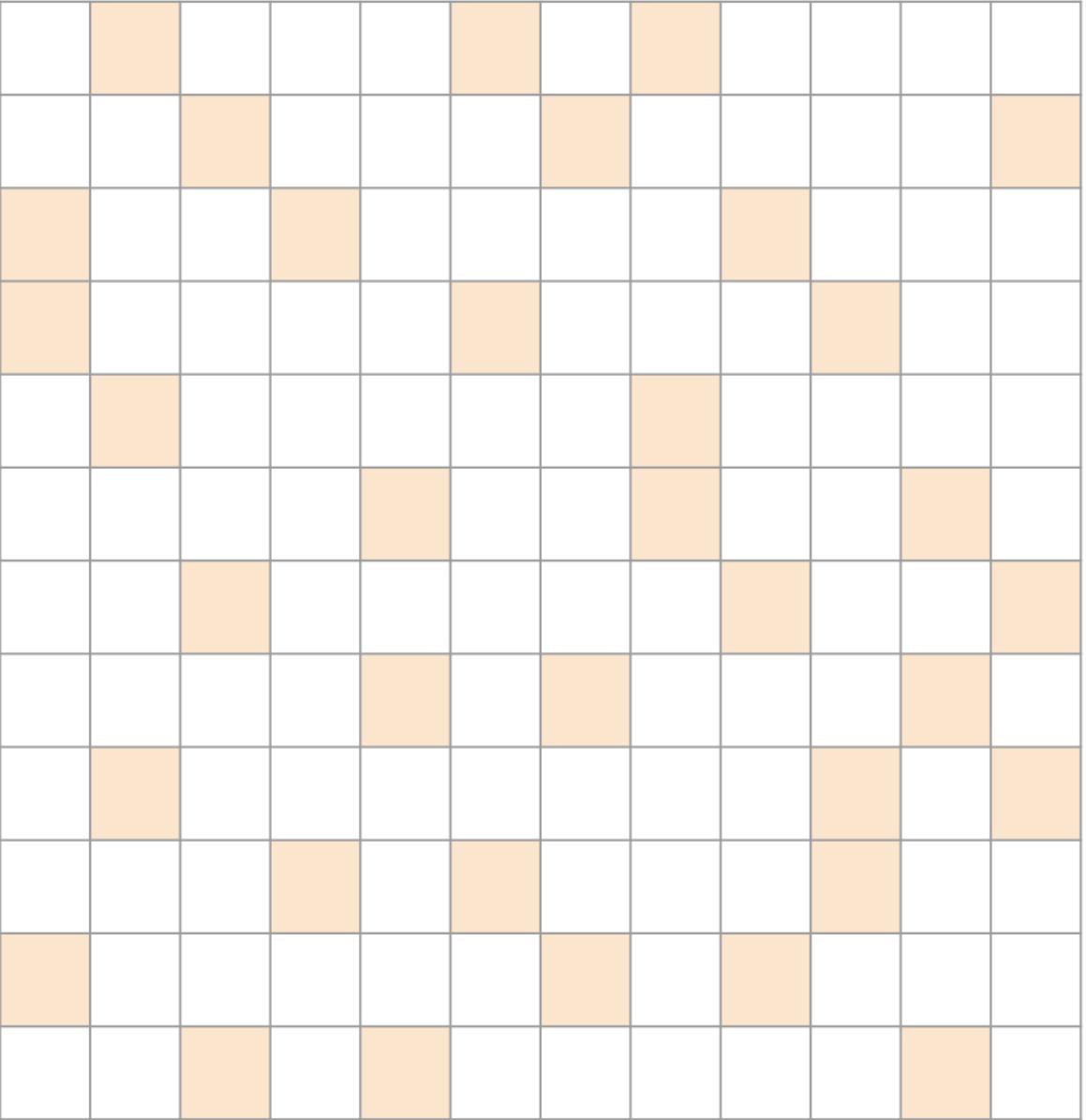
Lab 7 Variable Names

JDS_T Length of Cols matRows[4] {3, 2, 2, 0 }

		Sec 0	Sec 1	Sec 2
Nonzero values	matData[7]	{ 2, 3, 1,	4, 1, 1	1 }
Column indices	matCols[7]	{ 1, 0, 0,	2, 2, 3	3 }
JDS_T Column Pointers	matColStart[4]	{ 0, 3,	6,	7 }
JDS Row Indices	matRowPerm[4]	{ 2, 0,	3,	1 }



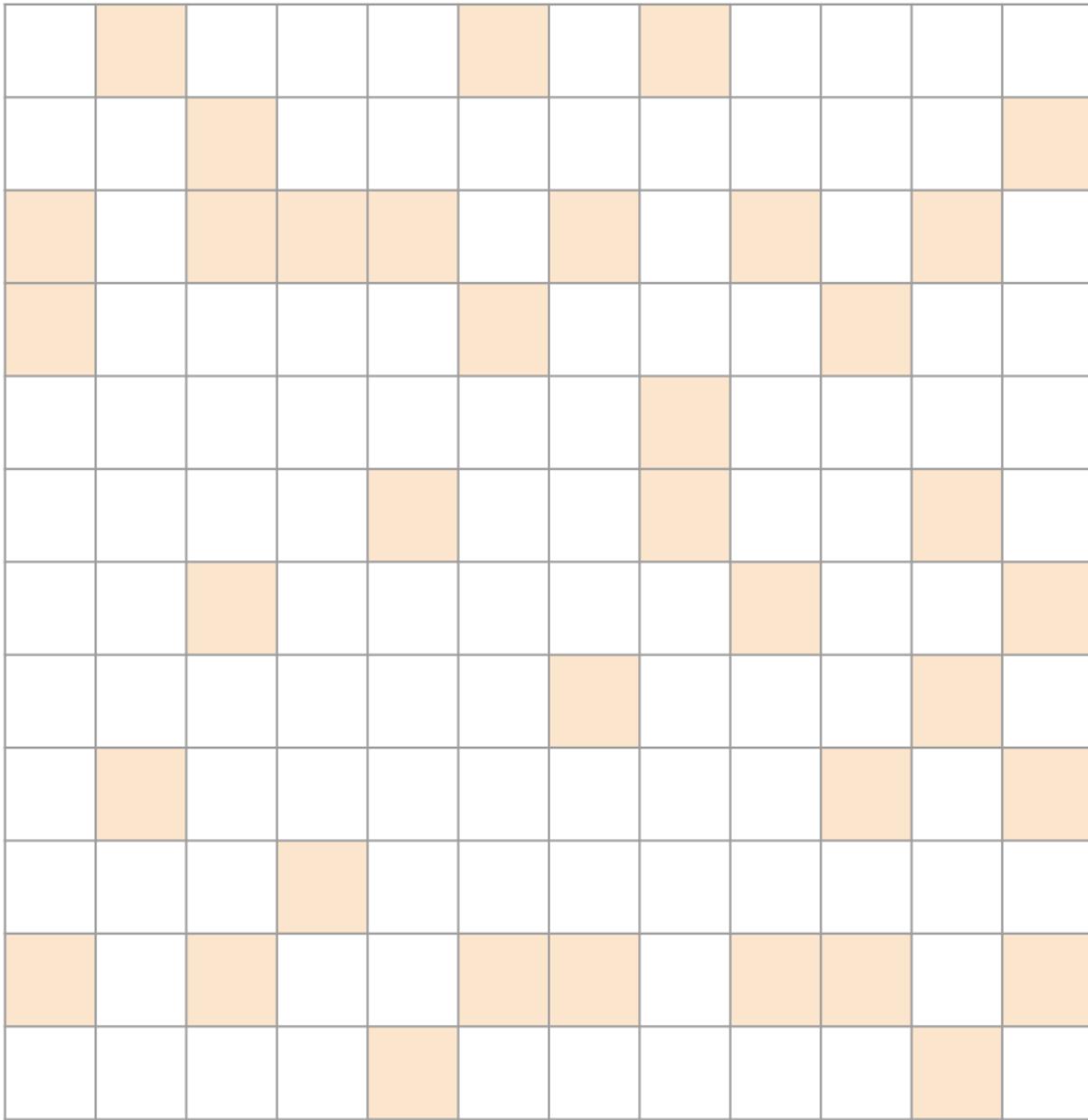
Roughly Random...



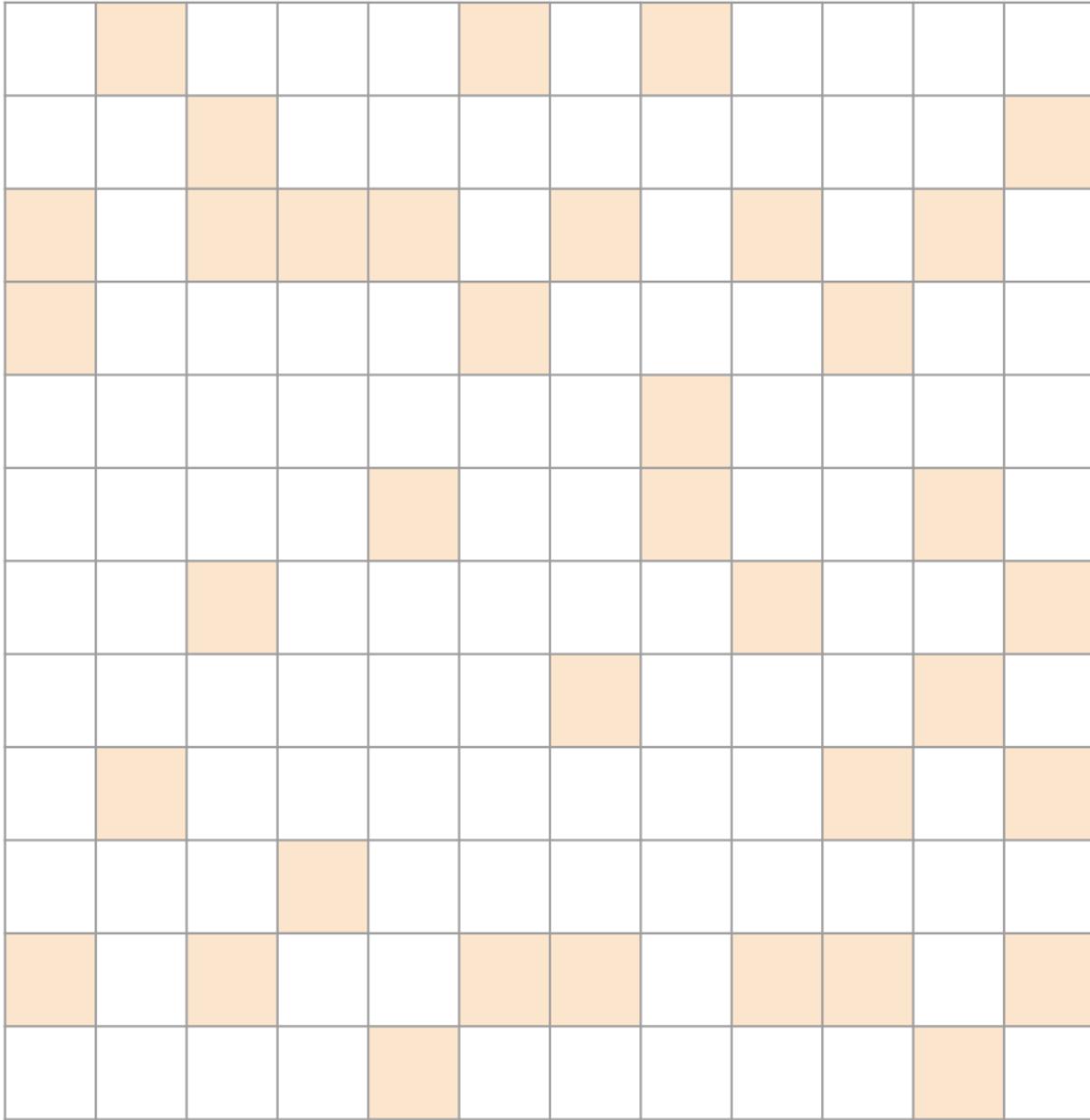
Roughly Random...

Probably best with ELL.

- Padding will be uniformly distributed
- Sparse representation will be uniform



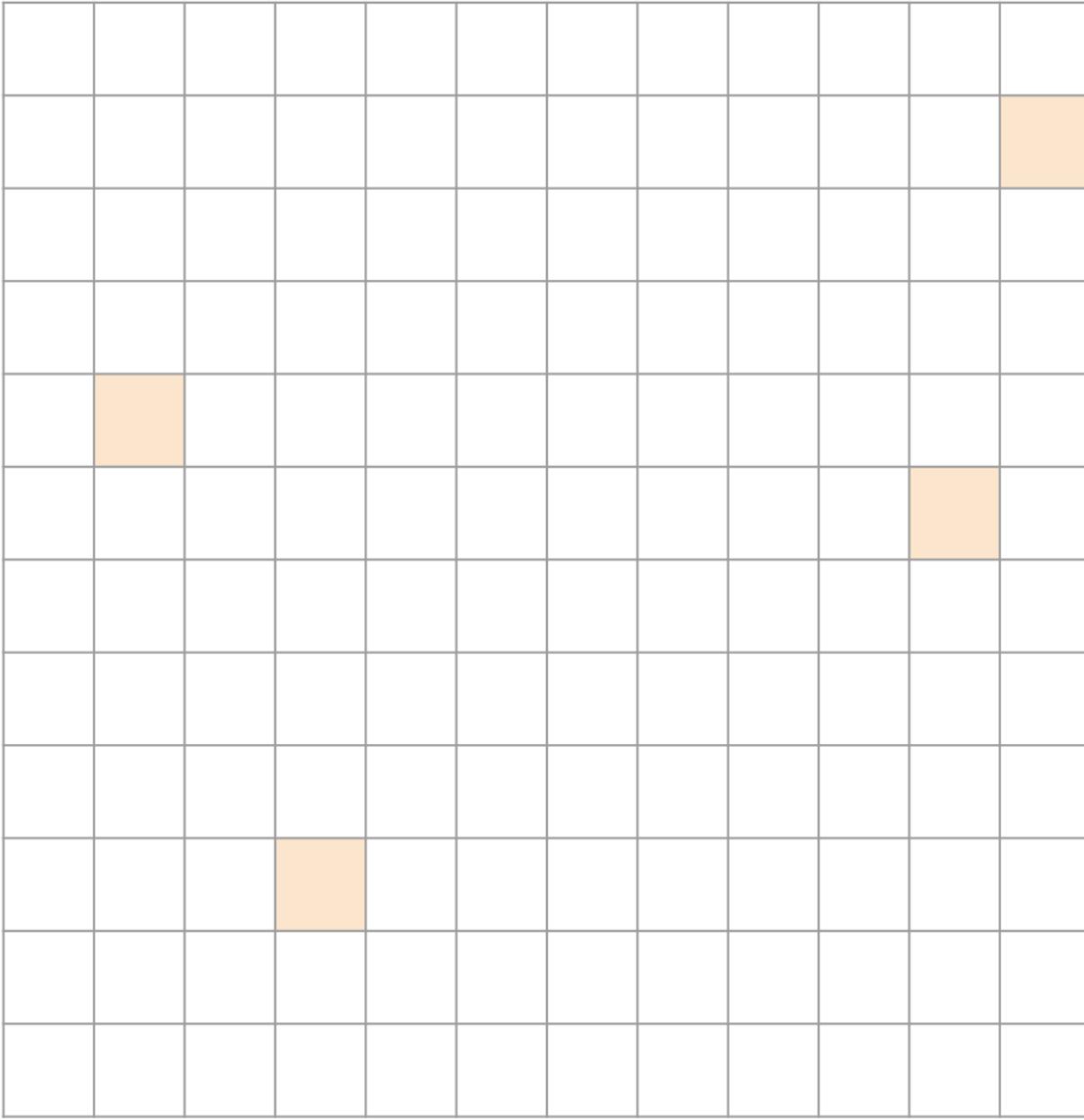
High variance in rows...



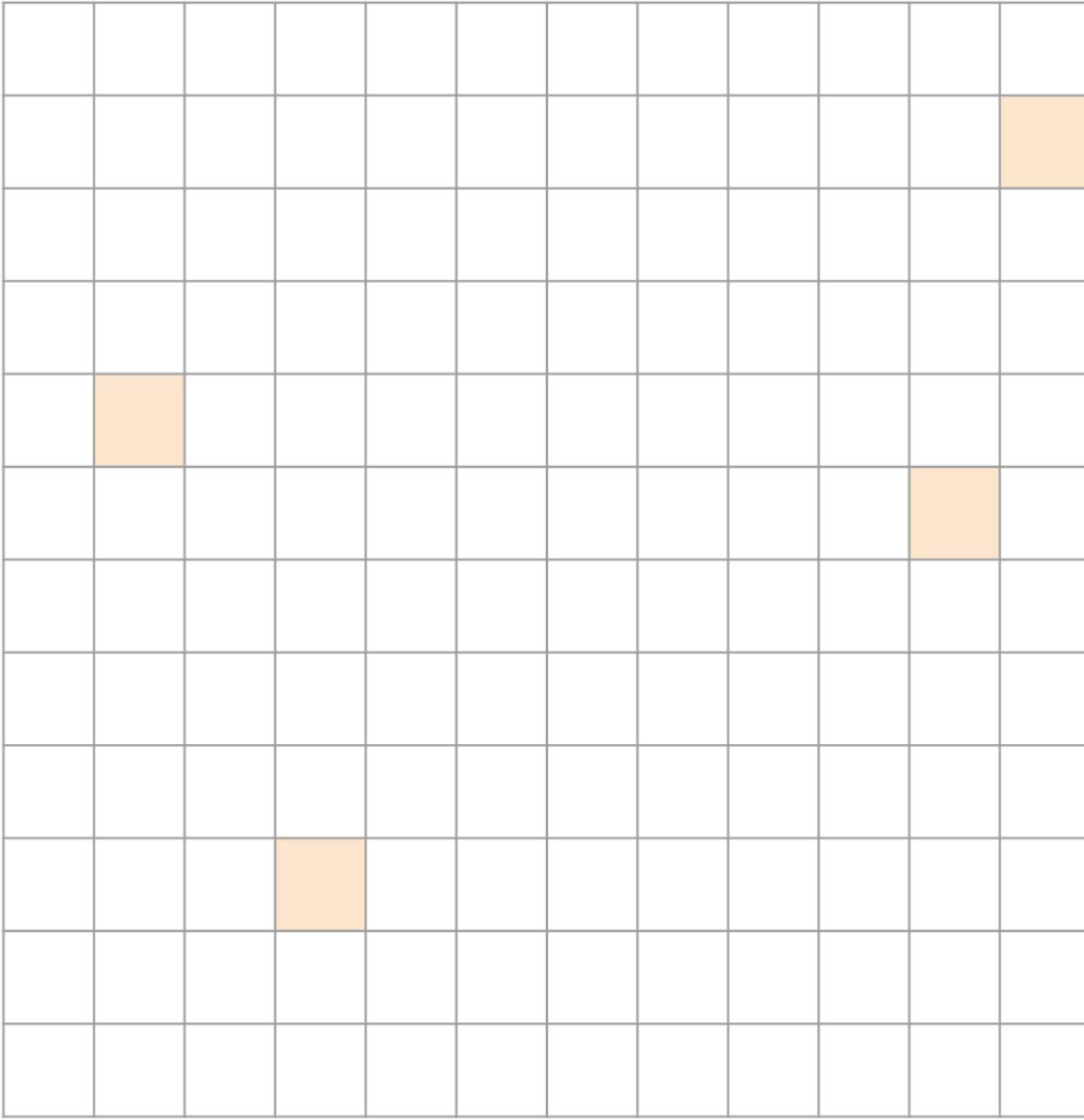
High variance in rows

Probably best with ELL/COO

- Benefit of ELL for most cases
- Outliers are captured with COO



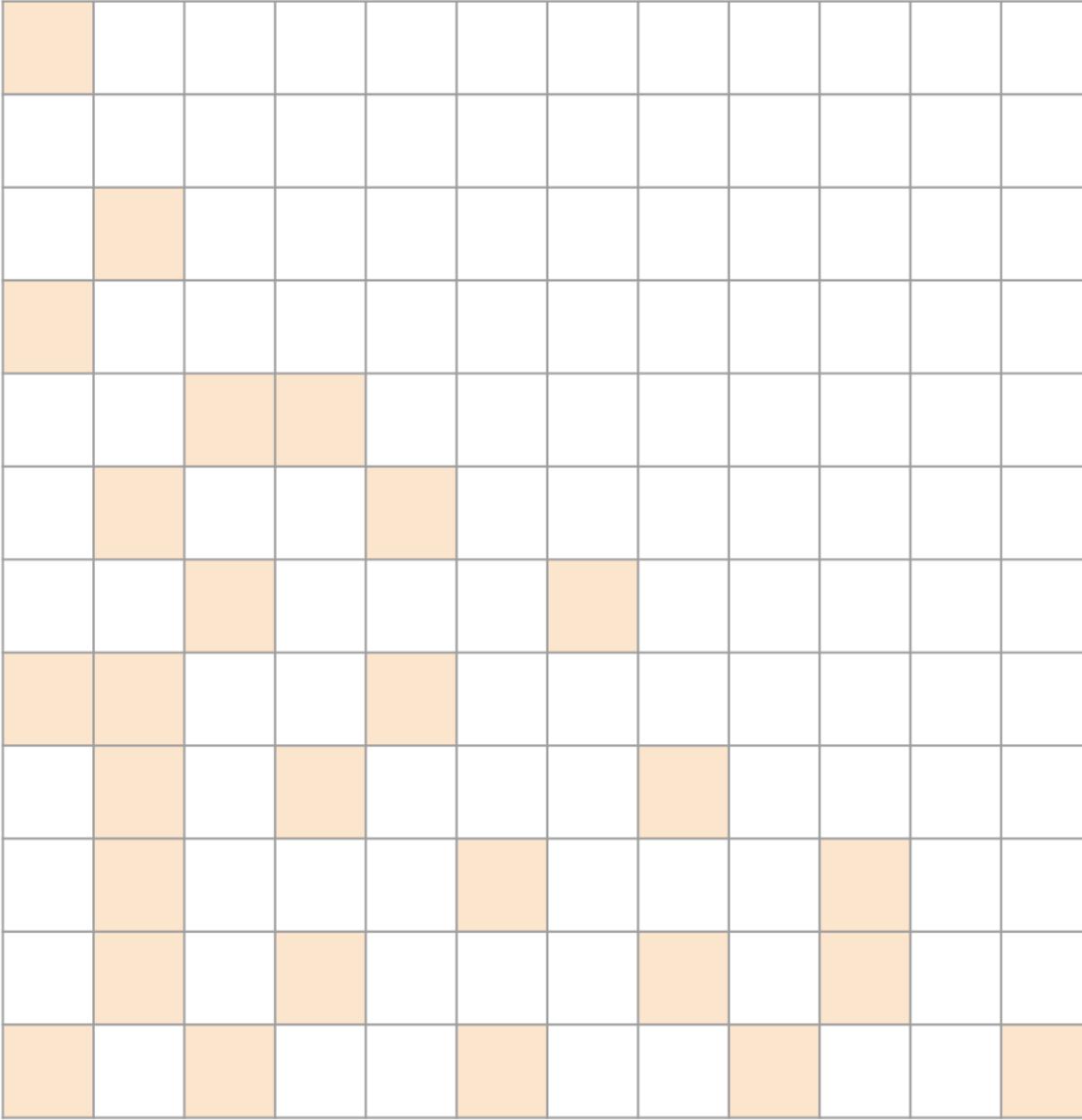
Very sparse...



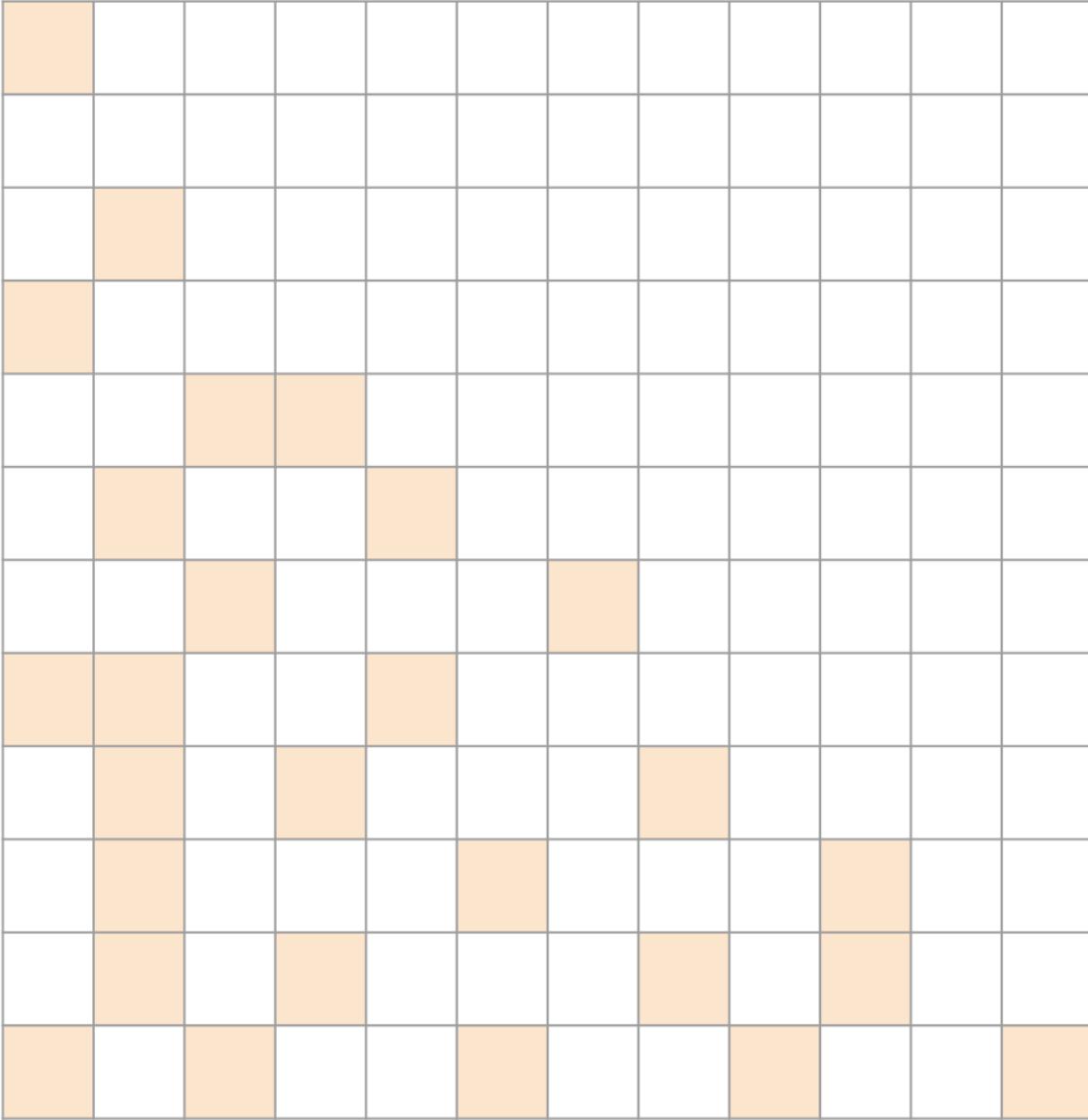
Very sparse

Probably best with COO

- Not a lot of data, compute is sparse



Roughly triangular...

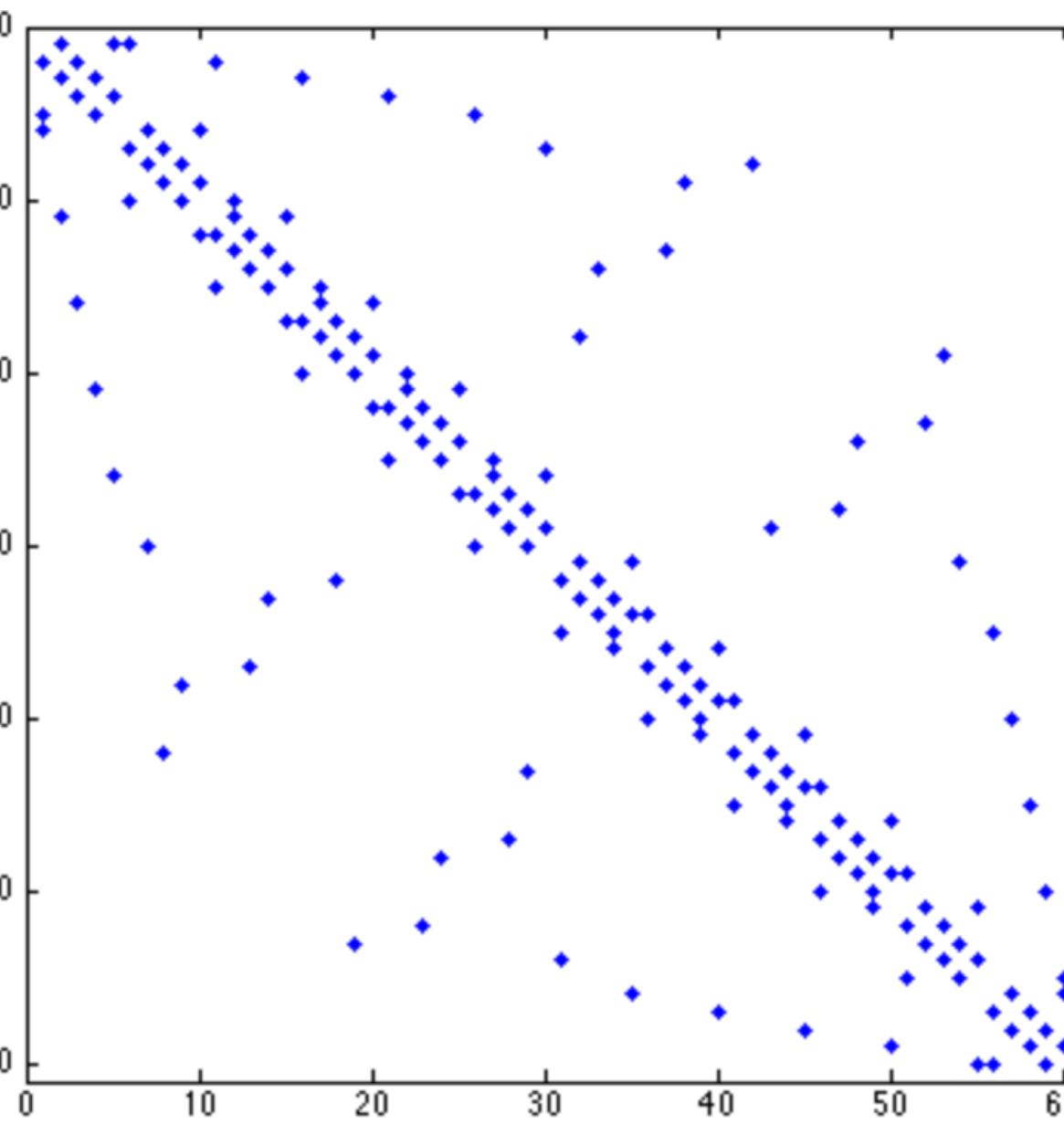


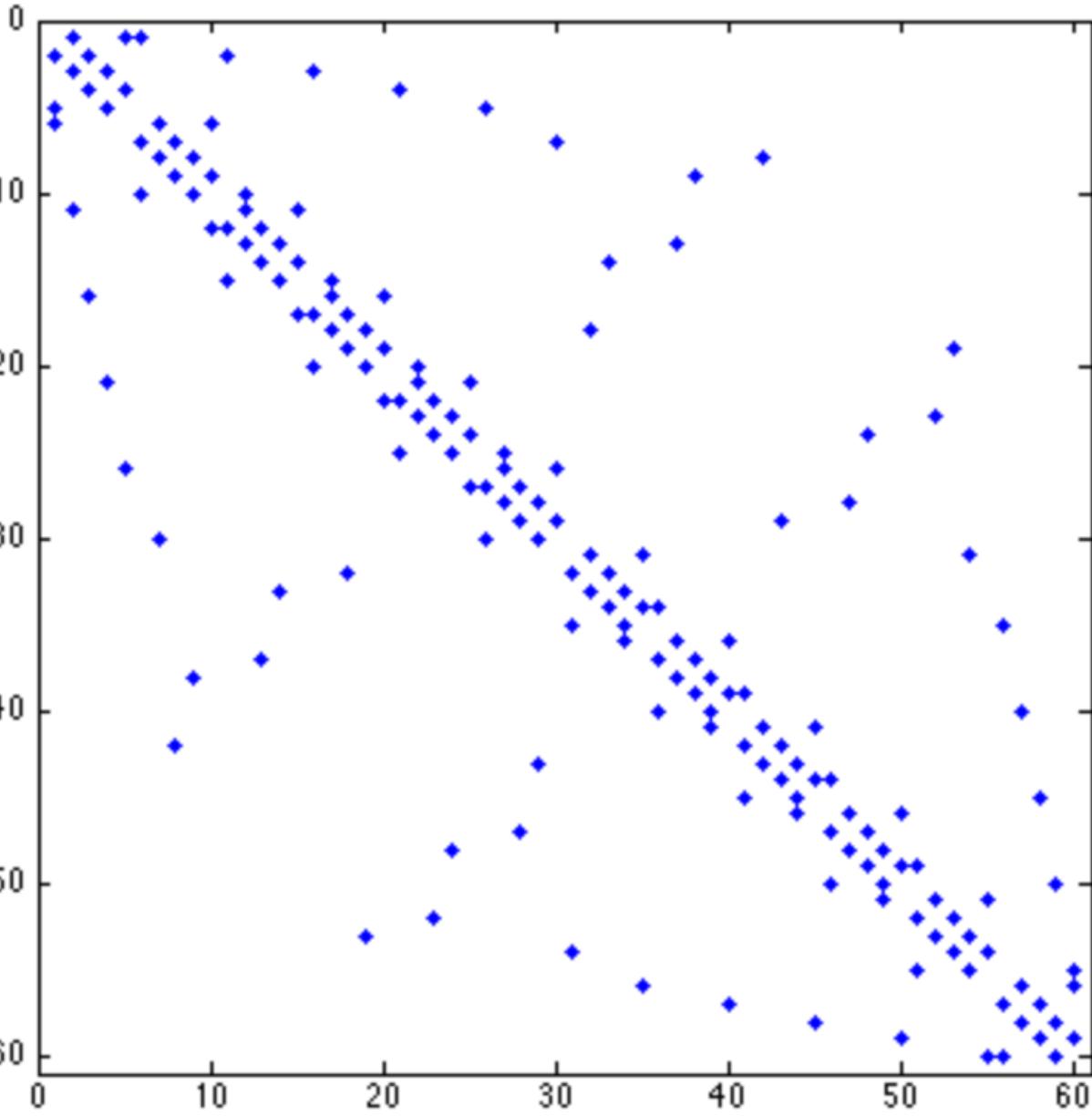
Roughly triangular...

Probably best with JDS

- Takes advantage of sparsity structure

Banded Matrix...

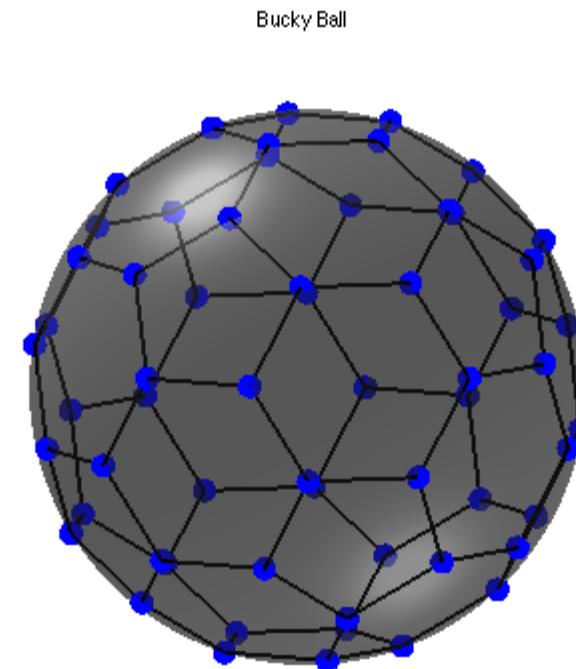
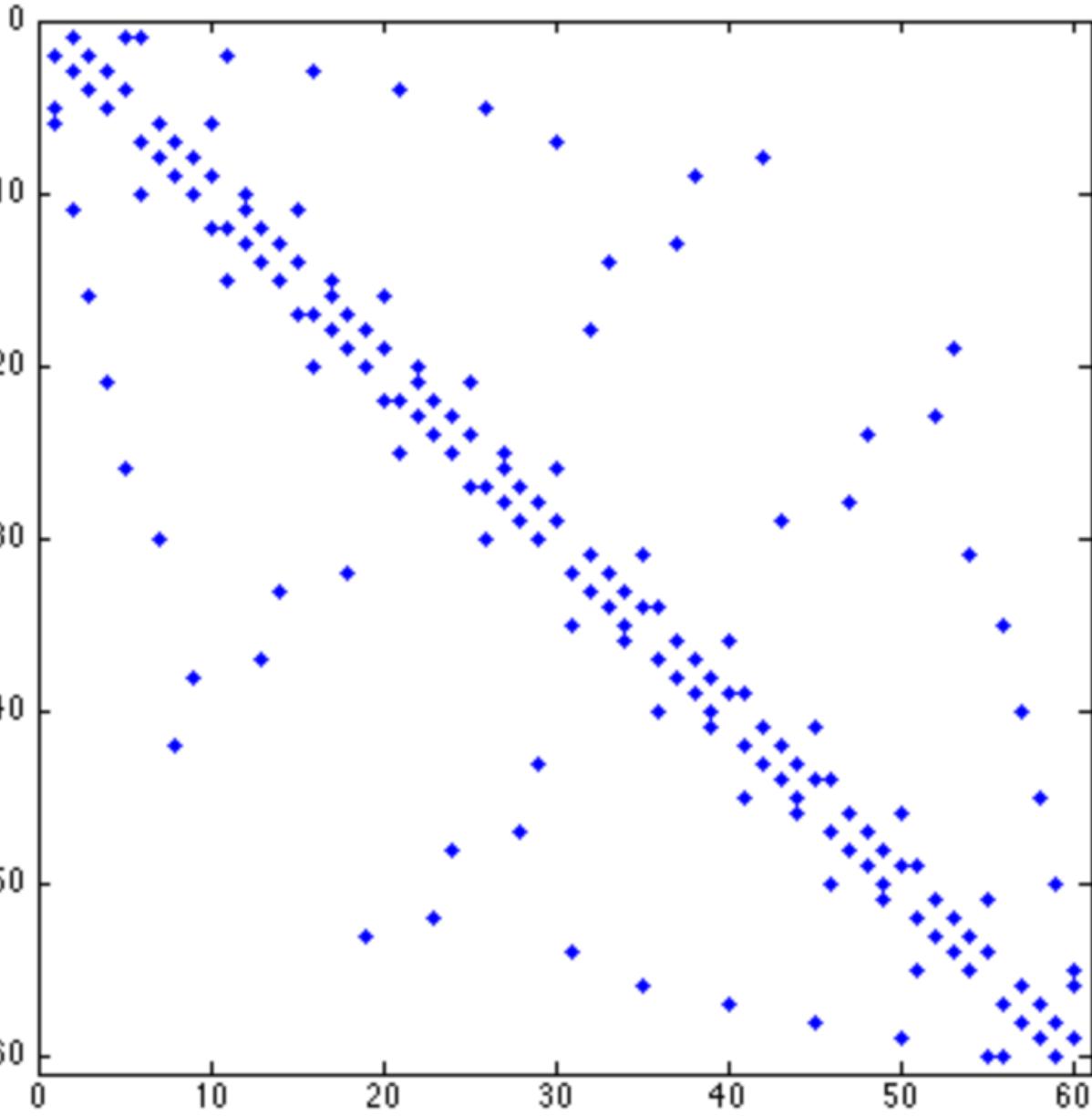




Banded Matrix...

Probably best with ELL

- Small amount of variance in rows



Other formats

- Diagonal (DIA): for strictly banded/diagonal matrices
- Packet (PKT): create diagonal submatrices by reordering rows/cols
- Dictionary of Keys (DOK): map of (row/col) to data
- Compressed Sparse Column (CSC): when to use over CSR?
- Blocked CSR: useful for block sparse matrices
- Hybrids of these...

Sparse Matrices as Foundation for Advanced Algorithm Techniques

- Graphs are often represented as sparse adjacency matrices
 - Used extensively in social network analytics, natural language processing, etc.
 - Sparse Matrix-Matrix multiplication (SpMM) is a fundamental operator in GNNs, which performs a multiplication between a sparse matrix and a dense matrix.
- Binning techniques often use sparse matrices for data compaction
 - Used extensively in ray tracing, particle-based fluid dynamics methods, and games
- These will be covered in ECE508/CS508



**ANY MORE QUESTIONS
READ CHAPTER 10**

Problem Solving

- Q: Consider the following sparse Matrix:
- For each of the following data layouts in memory, select the option that best matches all the sparse matrix formats that can store the data in memory as depicted.
- A:
 - 1) CSR, COO
 - 2) ??
 - 3) JDS, COO
 - 4) COO
 - 5) JDS-Transposed, COO

$$\begin{bmatrix} 1 & 0 & 4 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 7 & 0 & 9 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 6 & 5 & 0 & 0 & 8 \end{bmatrix}$$

Layout 1:

1	4	2	7	9	3	6	5	8
---	---	---	---	---	---	---	---	---

Layout 2:

1	2	7	6	4	0	9	5	0	0	3	8
---	---	---	---	---	---	---	---	---	---	---	---

Layout 3:

7	9	3	6	5	8	1	4	2
---	---	---	---	---	---	---	---	---

Layout 4:

9	7	1	2	4	3	5	8	6
---	---	---	---	---	---	---	---	---

Layout 5:

7	6	1	2	9	5	4	3	8
---	---	---	---	---	---	---	---	---



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 19
Parallel Sparse Methods

Course Reminders

- MP5.2 is due this week
- MP 6 is out
 - Due on April 14th
- Project Milestone 2: Baseline Convolution Kernel
 - Due next week
- Take a note of the day/time of midterm 2
 - May 2nd evening

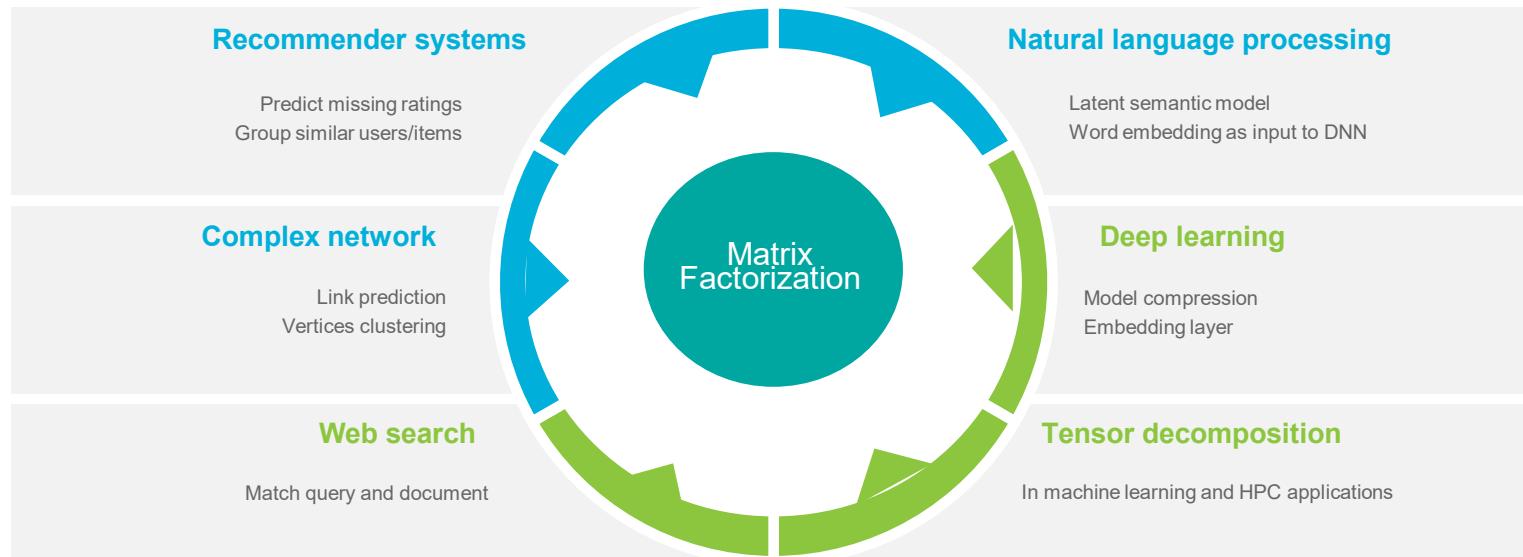
Objective

- To learn the key techniques for compacting input data in parallel sparse methods for reduced consumption of memory bandwidth
 - better utilization of on-chip memory
 - fewer bytes transferred to on-chip memory
 - Better utilization of global memory
 - Challenge: retaining regularity

Sparse Matrix

- Many real-world systems are sparse in nature
 - Linear systems described as sparse matrices
- Solving sparse linear systems
 - Iterative Conjugate Gradient solvers based on sparse matrix-vector multiplication is a common method
- Solution of PDE systems can be formulated into linear operations expressed as sparse matrix-vector multiplication

Sparse Matrix in Analytics and AI



A small image of a Netflix movie rating matrix, showing a grid of user IDs and movie titles with numerical ratings.

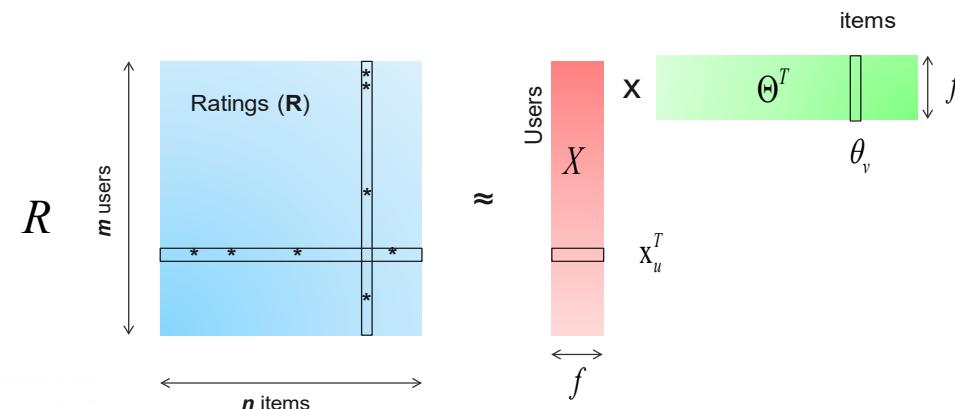
	2			4		5
	5	3		1		
		4		2		
	1	3		3		4
				2		4
	1	3		5		

NETFLIX



amazon.com Quora

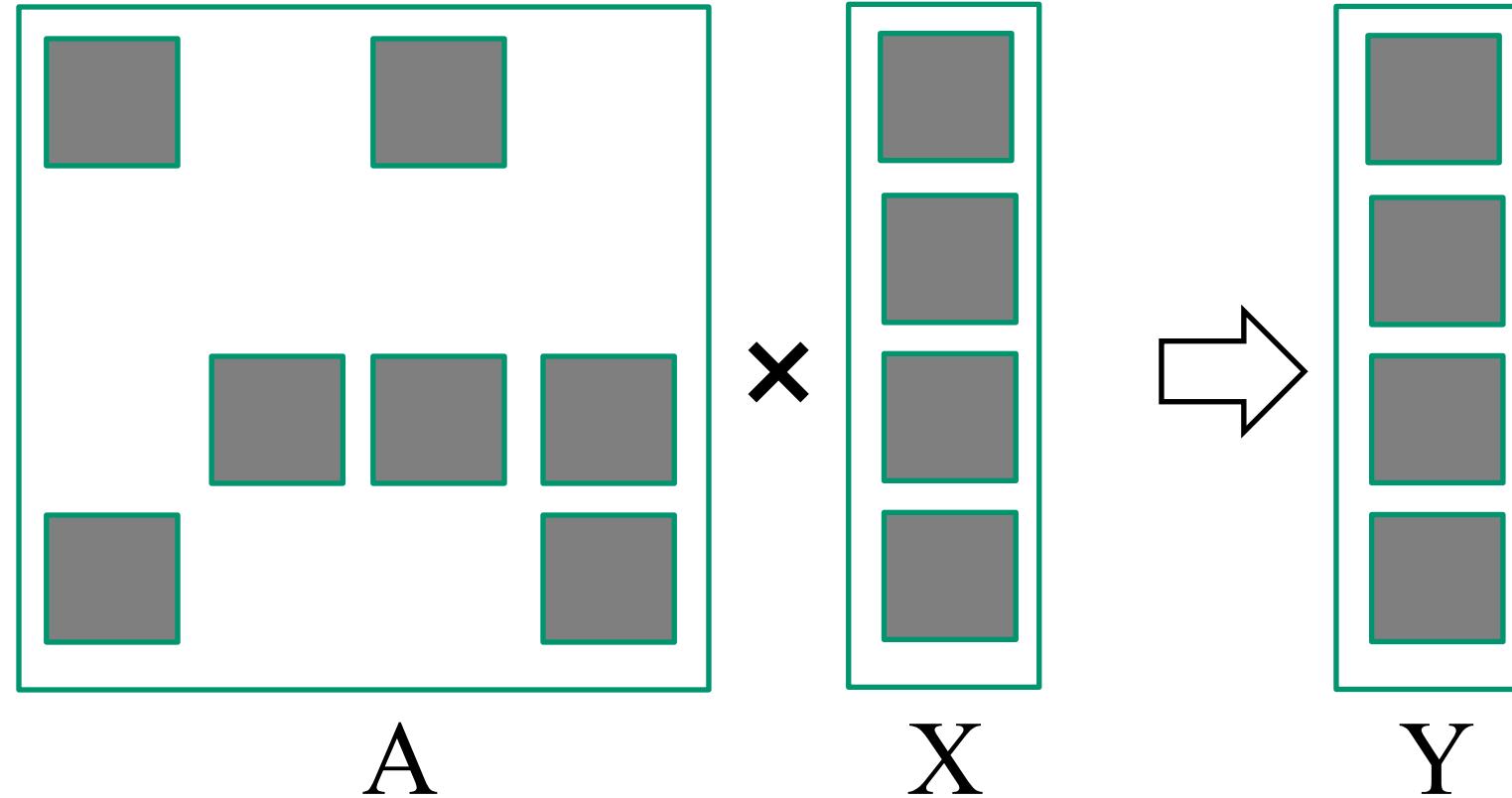
Apple Music



Sparse Matrix in Scientific Computing

Science Area	Number of Teams	Codes	Struct Grids	Unstruct Grids	Dense Matrix	Sparse Matrix	N-Body	Monte Carlo	FFT	PIC	Sig I/O
Climate and Weather	3	CESM, GCRM, CM1/WRF, HOMME	X	X		X		X			X
Plasmas/Magnetosphere	2	H3D(M), VPIC, OSIRIS, Magtail/UPIC	X				X		X		X
Stellar Atmospheres and Supernovae	5	PPM, MAESTRO, CASTRO, SEDONA, ChaNGa, MS-FLUKSS	X			X	X	X		X	X
Cosmology	2	Enzo, pGADGET	X			X	X				
Combustion/Turbulence	2	PSDNS, DISTUF	X						X		
General Relativity	2	Cactus, Harm3D, LazEV	X			X					
Molecular Dynamics	4	AMBER, Gromacs, NAMD, LAMMPS				X	X		X		
Quantum Chemistry	2	SIAL, GAMESS, NWChem			X	X	X	X			X
Material Science	3	NEMOS, OMEN, GW, QMCPACK			X	X	X	X			
Earthquakes/Seismology	2	AWP-ODC, HERCULES, PLSQR, SPECFEM3D	X	X			X				X
Quantum Chromo Dynamics	1	Chroma, MILC, USQCD	X		X	X					
Social Networks	1	EPISIMDEMICS									
Evolution	1	Eve									
Engineering/System of Systems	1	GRIPS, Revisit						X			
Computer Science	1			X	X	X			X		X

Sparse Matrix-Vector Multiplication (SpMV)



Challenges

- Compared to dense matrix multiplication, SpMV
 - Is irregular/unstructured
 - Has little input data reuse
 - Benefits little from compiler transformation tools
- Key to maximal performance
 - Maximize regularity (by reducing divergence and load imbalance)
 - Maximize DRAM burst utilization (layout arrangement)

A Simple Parallel SpMV

Row 0	3	0	1	0	Thread 0
Row 1	0	0	0	0	Thread 1
Row 2	0	2	4	1	Thread 2
Row 3	1	0	0	1	Thread 3

- Each thread processes one row

Compressed Sparse Row (CSR) Format

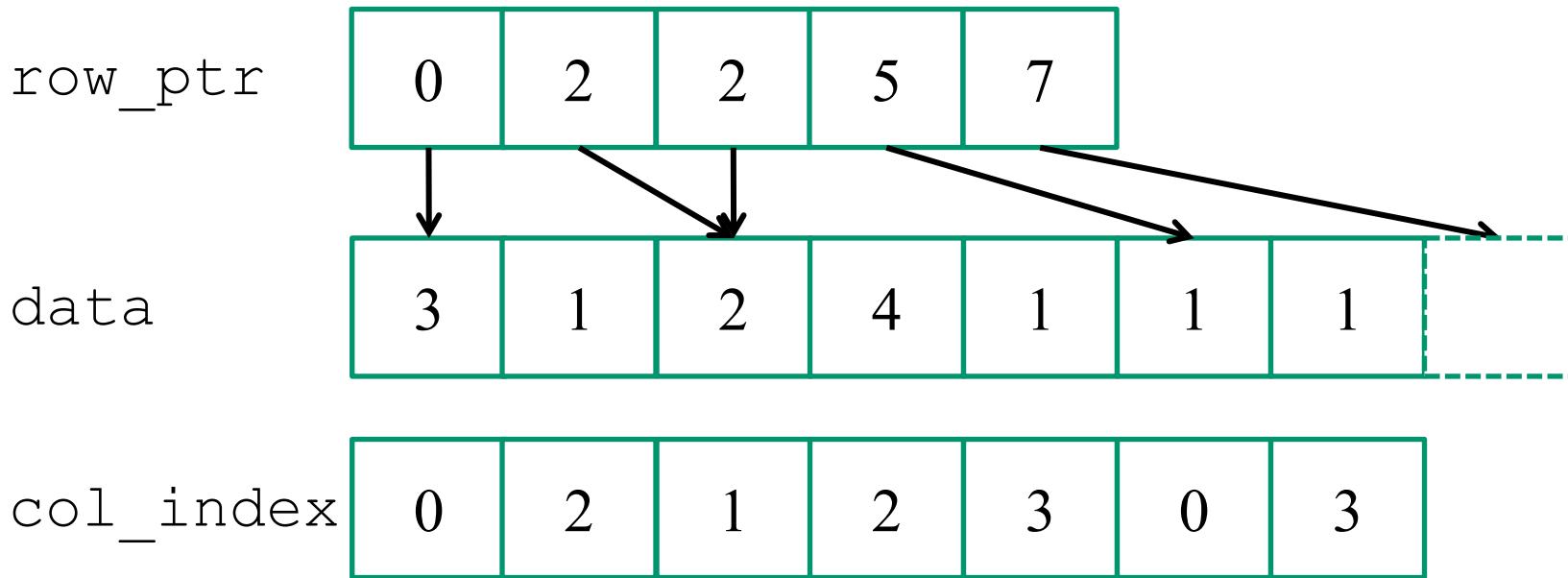
CSR Representation

		Row 0	Row 2	Row 3
Nonzero values	data [7]	{ 3, 1, 2, 4, 1, 1, 1 }		
Column indices	col_index [7]	{ 0, 2, 1, 2, 3, 0, 3 }		
Row Pointers	row_ptr [5]	{ 0, 2, 2, 5, 7 }		

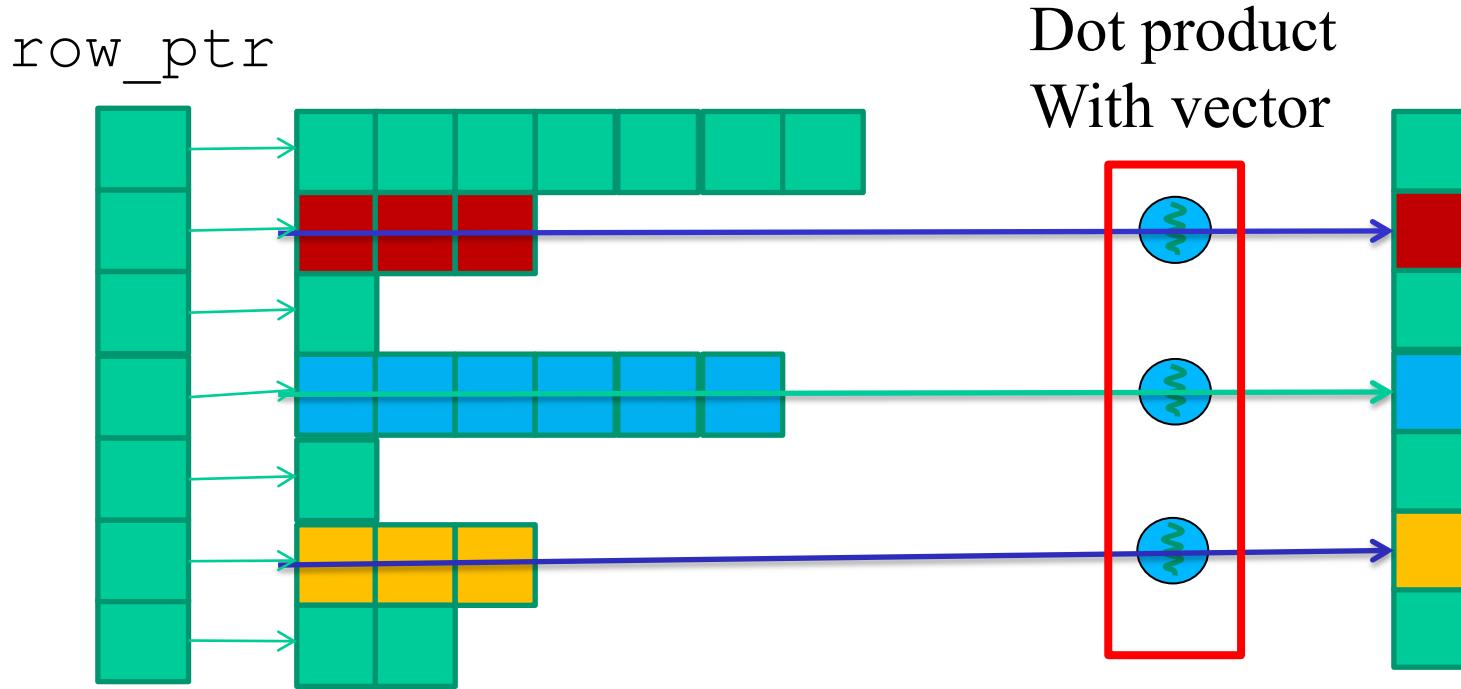
Dense representation

Row 0	3	0	1	0	Thread 0
Row 1	0	0	0	0	Thread 1
Row 2	0	2	4	1	Thread 2
Row 3	1	0	0	1	Thread 3

CSR Data Layout



CSR Kernel Design



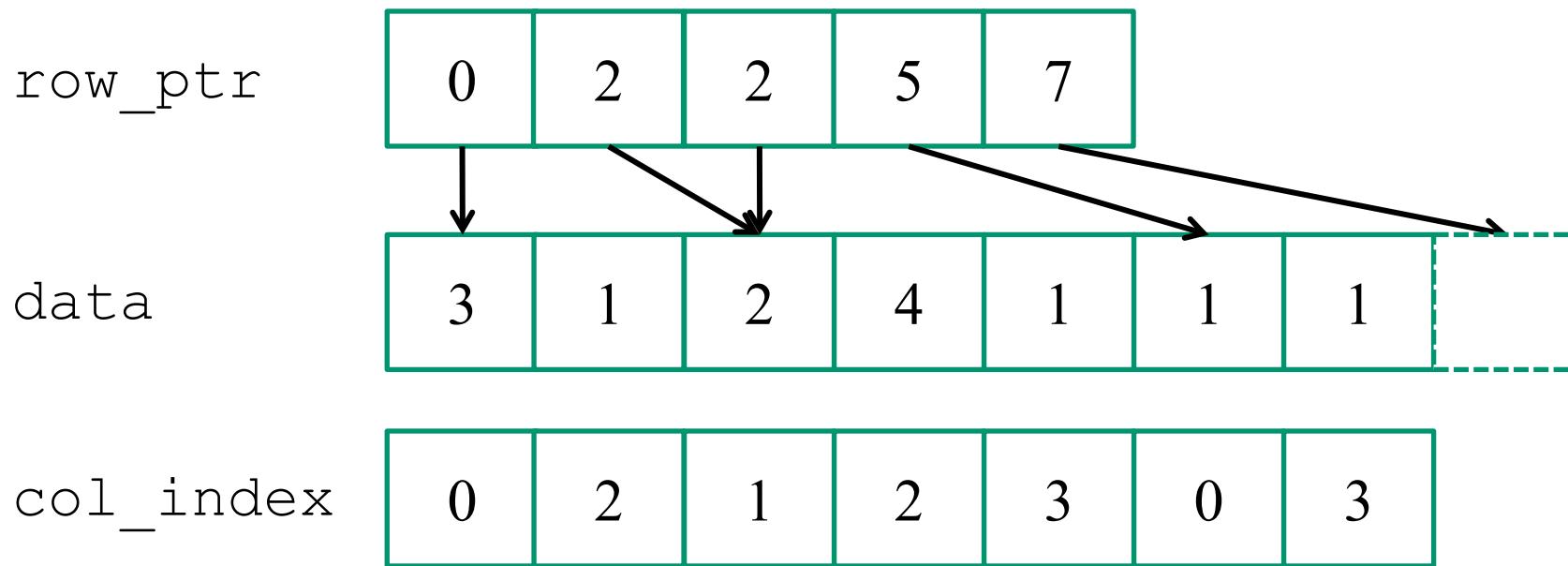
A Parallel SpMV/CSR Kernel (CUDA)

```
1. __global__ void SpMV_CSR(int num_rows, float *data, int
   *col_index, int *row_ptr, float *x, float *y)
{
2.     int row = blockIdx.x * blockDim.x + threadIdx.x;
3.     if (row < num_rows) {
4.         float dot = 0;
5.         int row_start = row_ptr[row];
6.         int row_end = row_ptr[row+1];
7.         for (int elem = row_start; elem < row_end; elem++)
8.             dot += data[elem] * x[col_index[elem]];
9.         y[row] = dot;
    }
}
```

		Row 0	Row 2	Row 3
Nonzero values	data[7]	{ 3, 1, 2, 4, 1, 1, 1 }		
Column indices	col_index[7]	{ 0, 2, 1, 2, 3, 0, 3 }		
Row Pointers	row_ptr[5]	{ 0, 2, 2, 5, 7 }		

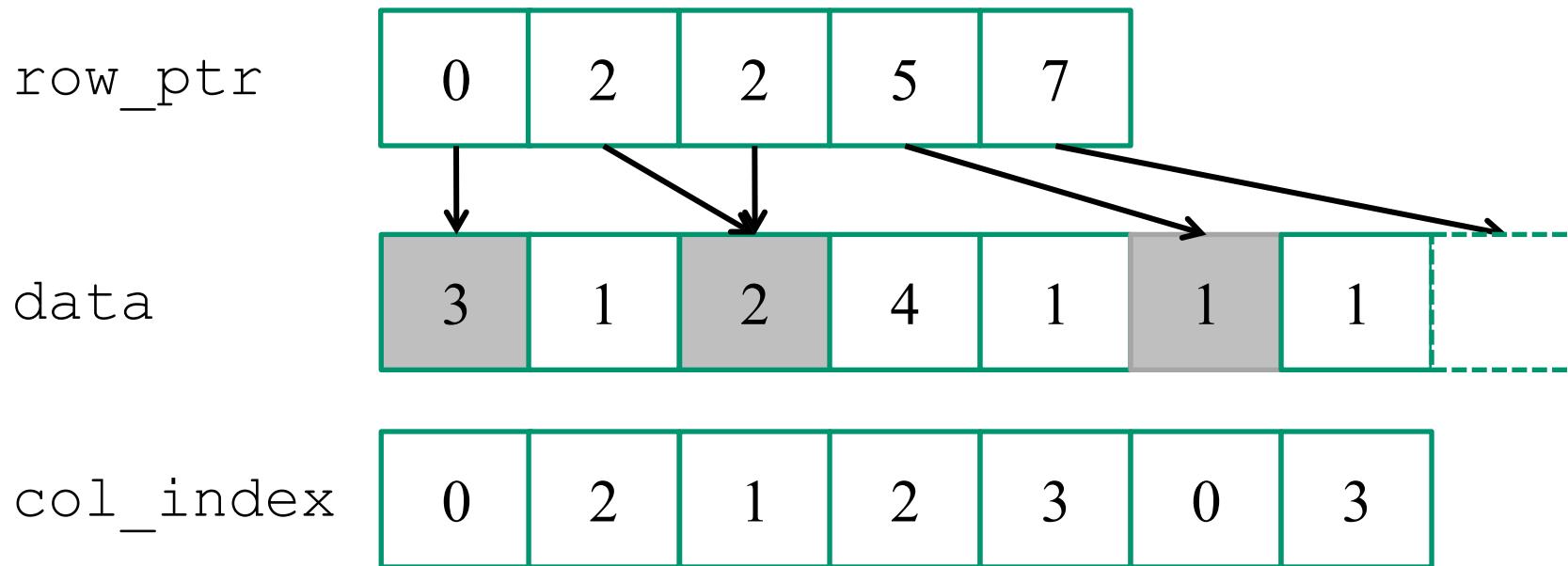
CSR Kernel Control Divergence

- Threads execute different number of iterations

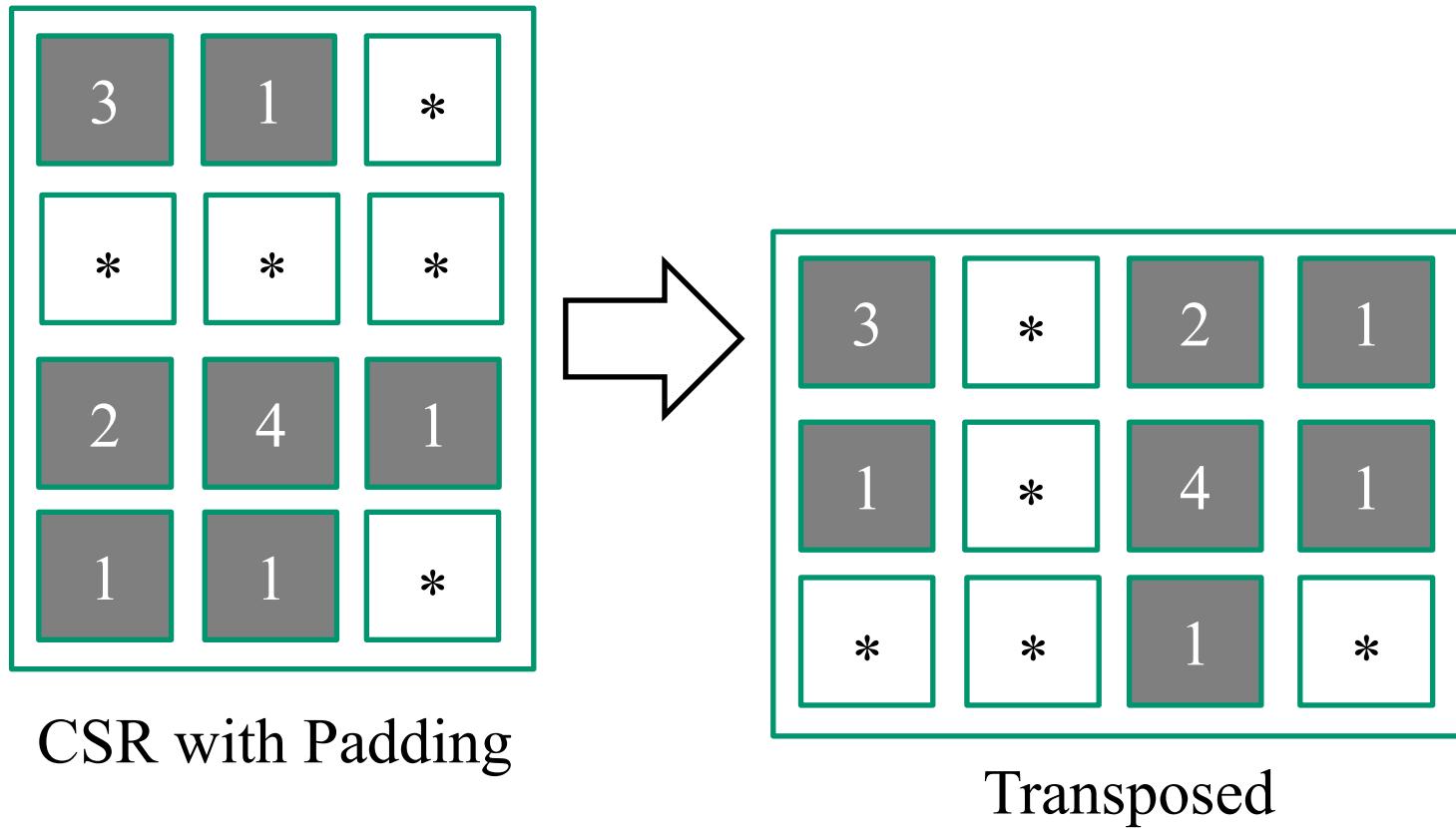


CSR Kernel Memory Divergence (Uncoalesced Accesses)

- Adjacent threads access non-adjacent memory locations
 - Grey elements are accessed by all threads in iteration 0

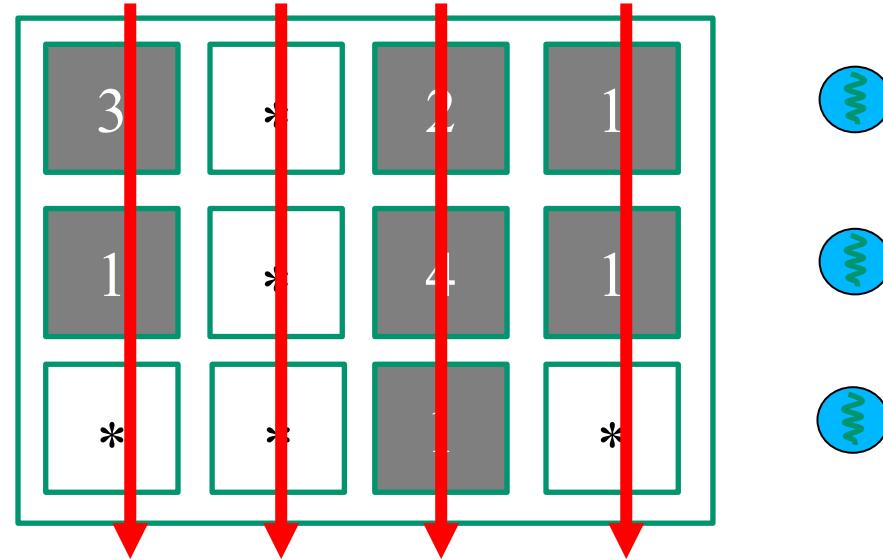


Regularizing SpMV with ELL(PACK) Format



- Pad all rows to the same length
 - Inefficient if a few rows are much longer than others
- Transpose (Column Major) for DRAM efficiency
- Both data and col_index padded/transposed

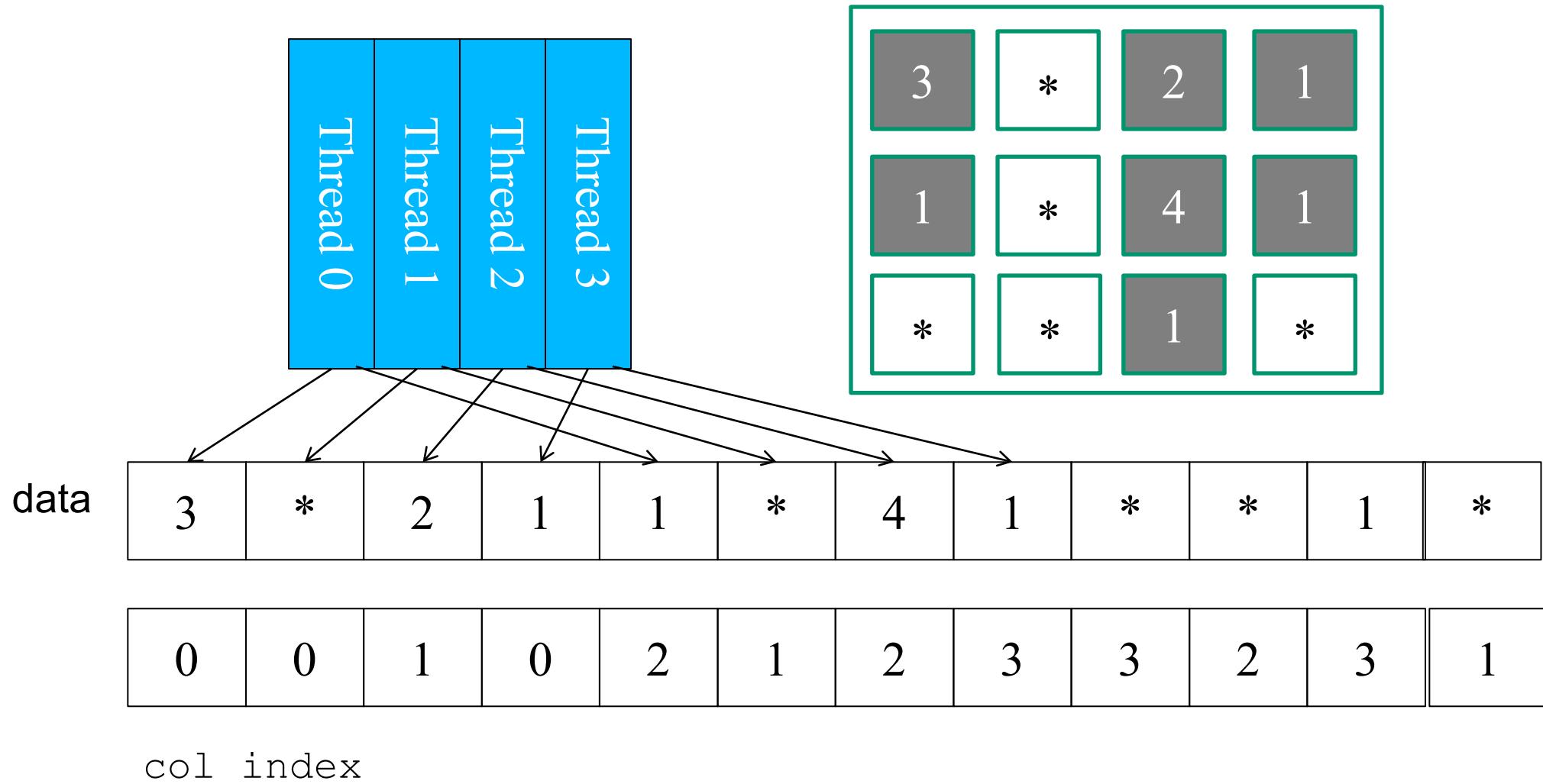
ELL Kernel Design



A parallel SpMV/ELL kernel

```
1. __global__ void SpMV_ELL(int num_rows, float *data,
   int *col_index, int num_elem, float *x, float *y)
{
2.     int row = blockIdx.x * blockDim.x + threadIdx.x;
3.     if (row < num_rows) {
4.         float dot = 0;
5.         for (int i = 0; i < num_elem; i++)
6.             dot += data[row+i*num_rows]*x[col_index[row+i*num_rows]];
7.         y[row] = dot;
}
}
```

Memory Coalescing with ELL



Coordinate (COO) format

- Explicitly list the column & row indices for every non-zero element

		Row 0	Row 2	Row 3
Nonzero values	<code>data[7]</code>	{ 3, 1,	2, 4, 1,	1, 1 }
Column indices	<code>col_index[7]</code>	{ 0, 2,	1, 2, 3,	0, 3 }
Row indices	<code>row_index[7]</code>	{ 0, 0,	2, 2, 2,	3, 3 }

COO Allows Reordering of Elements

		Row 0	Row 2	Row 3
Nonzero values	data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
Column indices	col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
Row indices	row_index[7]	{ 0, 0,	2, 2, 2,	3, 3 }

Nonzero values	data[7]	{ 1 1, 2, 4, 3, 1 1 }
Column indices	col_index[7]	{ 0 2, 1, 2, 0, 3, 3 }
Row indices	row_index[7]	{ 3 0, 2, 2, 0, 2, 3 }

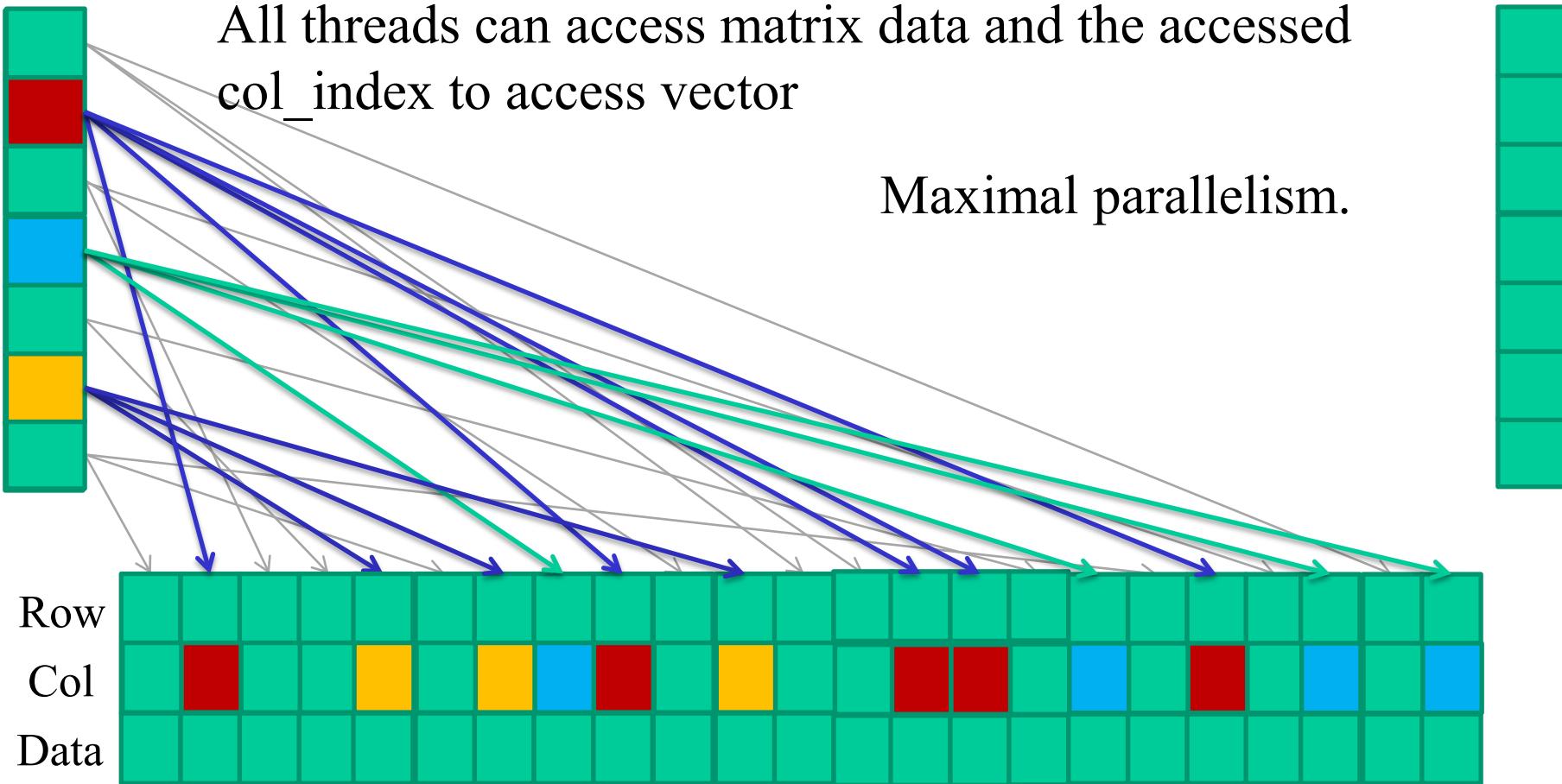
COO Kernel

```
1.     for (int i = 0; i < num_elem; i++)  
2.         y[row_index[i]] += data[i] * x[col_index[i]];
```

a sequential loop that implements SpMV/COO

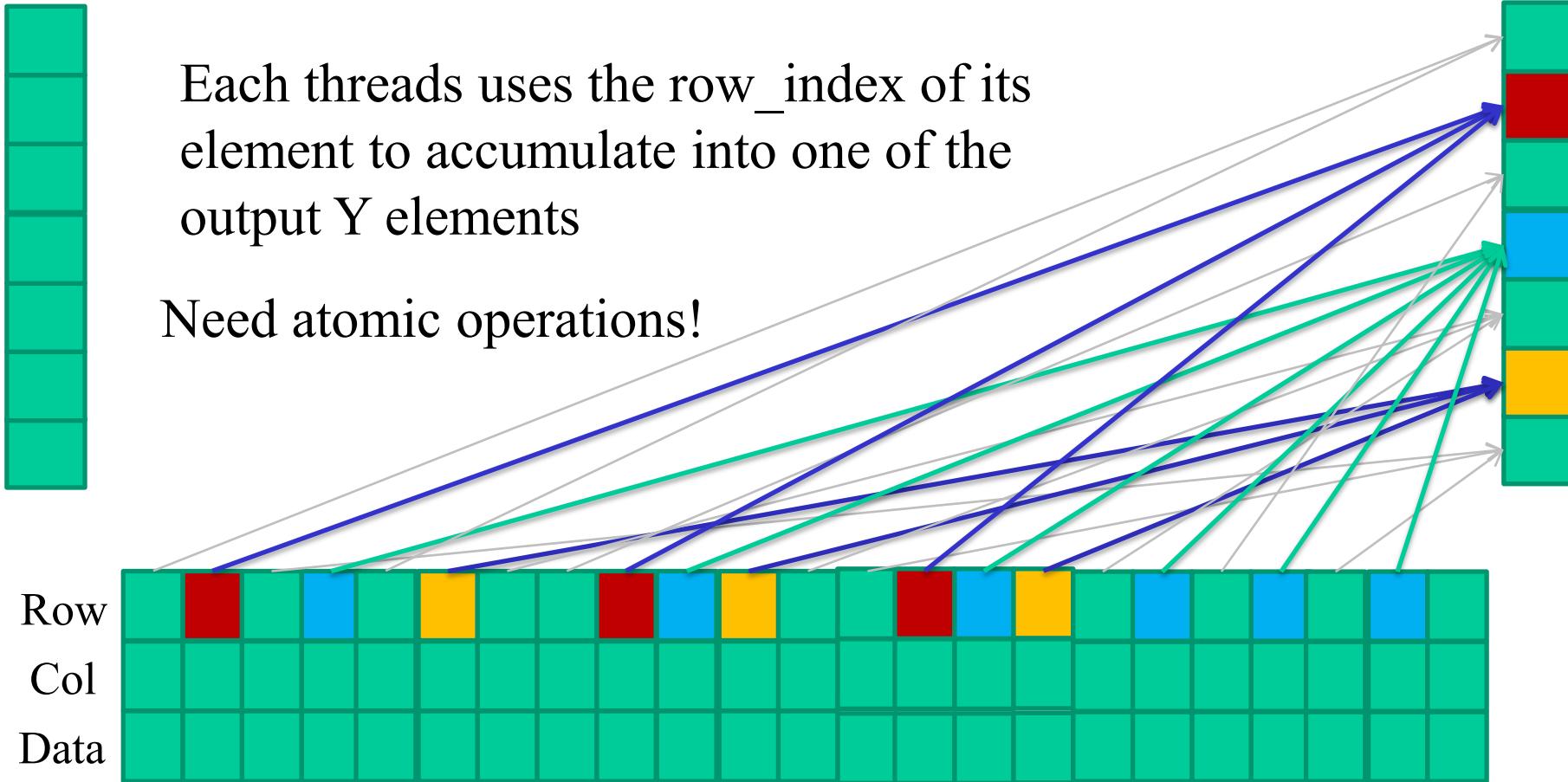
COO Kernel Design

Accessing Input Matrix and Vector



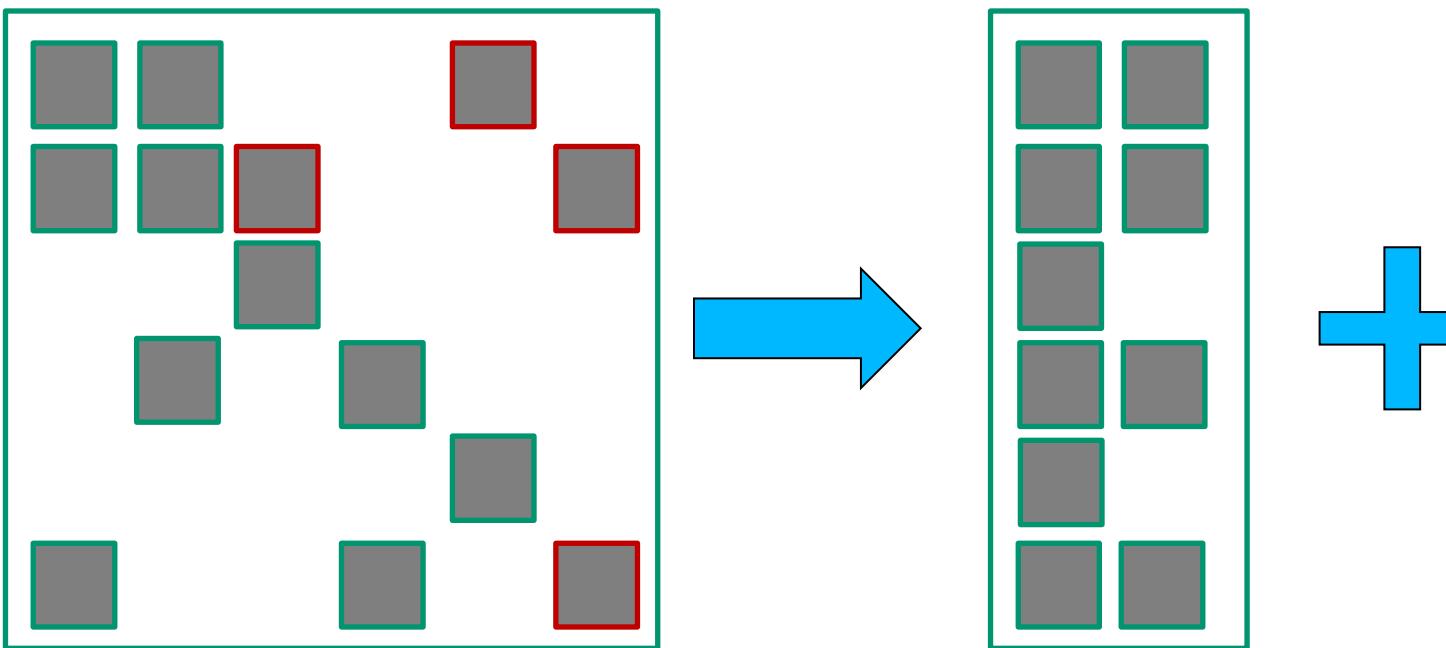
COO kernel Design

Accumulating into Output Vector



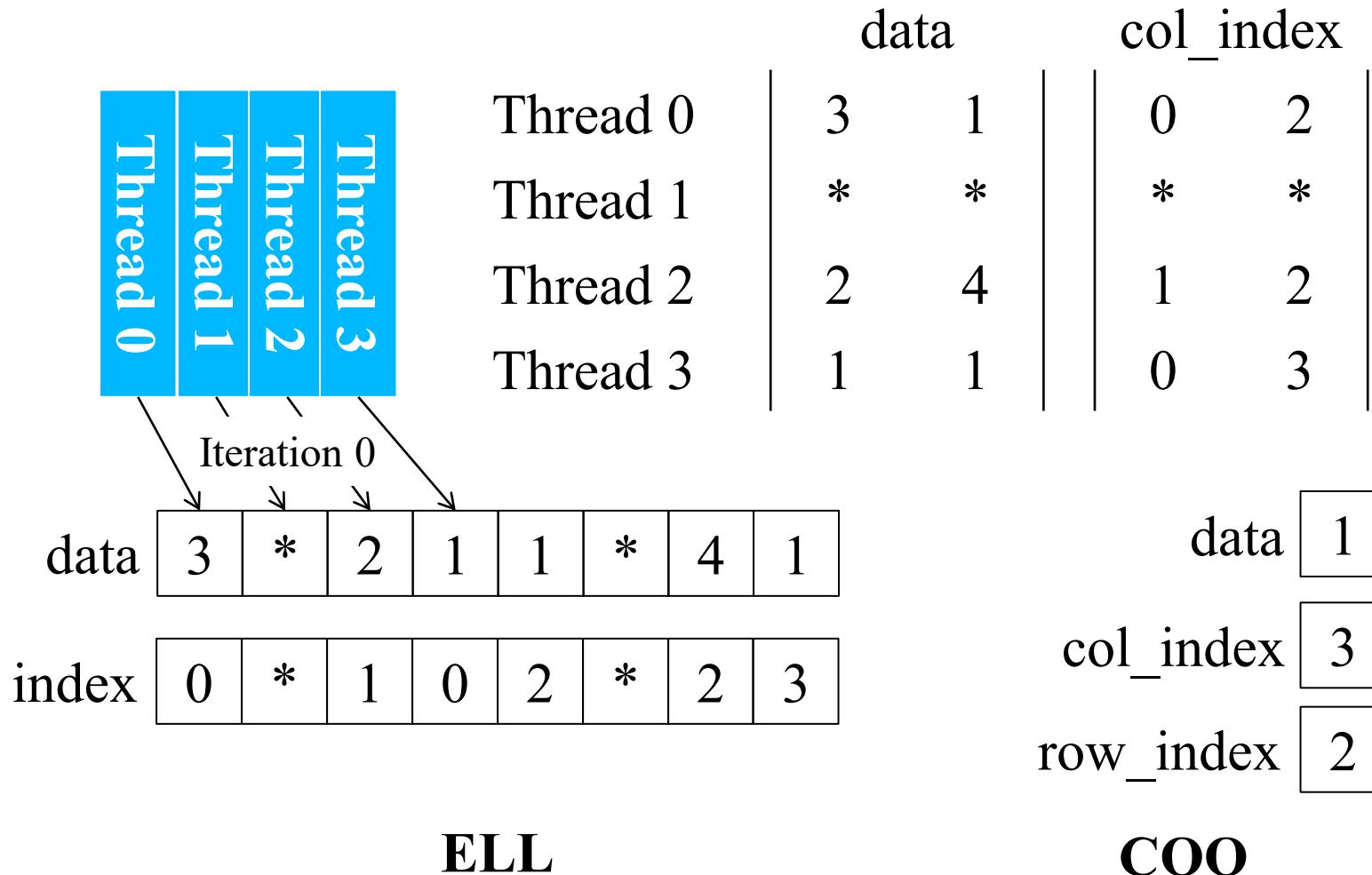
Hybrid Format (ELL + COO)

- ELL handles *typical* entries
- COO handles *exceptional* entries
 - Implemented with segmented reduction



Often implemented in
sequential host code in
practice

Reduced Padding with Hybrid Format





**ANY MORE QUESTIONS
READ CHAPTER 10**

Problem Solving

- Q: Given matrix A,
which of the following are correct?

$$A = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 0 & 0 & 1 \end{vmatrix}$$

CSR representation:

Data = [1,2,1,1,2,3,4,1,1]

Col_idx = [0,2,3,0,1,2,3,0,3]

Row_ptr = [0,1,3,7,9]

COO representation

Data = [1,2,1,1,2,3,4,1,1]

Col_idx = [0,2,3,0,1,2,3,0,3]

Row_idx = [0,1,1,3,3,3,3,7,7]

- A: only CSR



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 18
Atomic Operations and
Histogramming



Course Reminders

- MP5.2 is due this week
- Project Milestone 2: Baseline Convolution Kernel
 - Due next week

Objective

- To understand atomic operations
 - Read-modify-write in parallel computation
 - A primitive form of “critical regions” in parallel programs
 - Use of atomic operations in CUDA
 - Why atomic operations reduce memory system throughput
 - How to avoid atomic operations in some parallel algorithms
- To learn practical histogram programming techniques
 - Basic histogram algorithm using atomic operations
 - Atomic operation throughput
 - Privatization

A Common Collaboration Pattern

- Multiple bank tellers count the total amount of cash in the safe
- Each grab a pile and count
- Have a central display of the running total
- Whenever someone finishes counting a pile, add the subtotal of the pile to the running total
- A bad outcome
 - Some of the piles were not accounted for

A Common Arbitration Pattern

- Multiple customers booking air tickets
- Each
 - Brings up a flight seat map
 - Decides on a seat
 - Update the seat map, mark the seat as taken
- A bad outcome
 - Multiple passengers ended up booking the same seat

Read-Modify-Write Operations

thread1: $\text{Old} \leftarrow \text{Mem}[x]$
 $\text{New} \leftarrow \text{Old} + 1$
 $\text{Mem}[x] \leftarrow \text{New}$

thread2: $\text{Old} \leftarrow \text{Mem}[x]$
 $\text{New} \leftarrow \text{Old} + 1$
 $\text{Mem}[x] \leftarrow \text{New}$

If $\text{Mem}[x]$ was **initially 0**, what would the value of $\text{Mem}[x]$ be after threads 1 and 2 have completed?

- What does each thread get in their Old variable?

The answer may vary due to data races. To avoid data races, you should use atomic operations

Timing Scenario #1

Time	Thread 1	Thread 2
1	(0) Old \leftarrow Mem[x]	
2	(1) New \leftarrow Old + 1	
3	(1) Mem[x] \leftarrow New	
4		(1) Old \leftarrow Mem[x]
5		(2) New \leftarrow Old + 1
6		(2) Mem[x] \leftarrow New

- Thread 1 Old = 0
- Thread 2 Old = 1
- Mem[x] = 2 after the sequence

Timing Scenario #2

Time	Thread 1	Thread 2
1		(0) Old \leftarrow Mem[x]
2		(1) New \leftarrow Old + 1
3		(1) Mem[x] \leftarrow New
4	(1) Old \leftarrow Mem[x]	
5	(2) New \leftarrow Old + 1	
6	(2) Mem[x] \leftarrow New	

- Thread 1 Old = 1
- Thread 2 Old = 0
- Mem[x] = 2 after the sequence

Timing Scenario #3

Time	Thread 1	Thread 2
1	(0) Old \leftarrow Mem[x]	
2	(1) New \leftarrow Old + 1	
3		(0) Old \leftarrow Mem[x]
4	(1) Mem[x] \leftarrow New	
5		(1) New \leftarrow Old + 1
6		(1) Mem[x] \leftarrow New

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Timing Scenario #4

Time	Thread 1	Thread 2
1		(0) Old \leftarrow Mem[x]
2		(1) New \leftarrow Old + 1
3	(0) Old \leftarrow Mem[x]	
4		(1) Mem[x] \leftarrow New
5	(1) New \leftarrow Old + 1	
6	(1) Mem[x] \leftarrow New	

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Atomic Operations Prevent Interleaving

```
thread1: Old ← Mem[x]  
         New ← Old + 1  
         Mem[x] ← New
```

```
thread2: Old ← Mem[x]  
         New ← Old + 1  
         Mem[x] ← New
```

Or

```
thread1: Old ← Mem[x]  
         New ← Old + 1  
         Mem[x] ← New
```

```
thread2: Old ← Mem[x]  
         New ← Old + 1  
         Mem[x] ← New
```

Without Atomic Operations

Mem[x] initialized to 0

thread1: Old \leftarrow Mem[x]

New \leftarrow Old + 1

Mem[x] \leftarrow New

thread2: Old \leftarrow Mem[x]

New \leftarrow Old + 1

Mem[x] \leftarrow New

- Both threads receive 0
- Mem[x] becomes 1

Needed When Threads Write to Same Location

When **two threads**

- **may write to the same memory** location,
- the program may **need atomic operations**.

Sharing is **not always easy to recognize**...

- Do two insertions into a hash table share data?
- What about two graph node updates based on all of the nodes' neighbors?
- What if nodes are on same side of bipartite graph?

What Exactly is “Atomic?”

To a high-energy photon, atoms are not.

Atomicity is ALWAYS with respect to something.

Two **sections of code**

- **that execute atomically with respect to one another**
- **appear to the software as though**
- the programs’ **execution did not interleave** at all.

What Can Go Wrong?

Common failure mode:

- Programmer *thinks operations* are **independent**.
- Hasn't considered input data for which they are not.
- Or another programmer reuses code without understanding assumptions that imply independence.

Also: **atomicity does not constrain relative order**.

Implementing Atomic Operations

- Many ISAs offer synchronization primitives,
 - **instructions with one (or more) address operands**
 - **that execute atomically with respect to one another** when used on the same address.
- Mostly read, modify, write operations
 - Bit test and set
 - Compare and swap / exchange
 - Swap / exchange
 - Fetch and increment / add

Atomicity Enforced by Microarchitecture

When synchronization primitives execute,

- **hardware ensures** that **no other** thread
- **accesses** the location **until** the operation is **complete**.

Other threads that access the location

- are typically stalled or held in a queue until their turn.
- **Threads perform atomic operations serially.**

Atomic Operations in CUDA

- Function calls that are translated into single ISA instructions (a.k.a. *intrinsics*)
 - Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)
 - Read CUDA C programming Guide for more details
- Atomic Add

int atomicAdd(int address, int val);*

reads the 32-bit word **old** pointed to by **address** in global or shared memory, computes **(old + val)**, and stores the result back to memory at the same address. The function returns **old**.

More Atomic Adds in CUDA

- Unsigned 32-bit integer atomic add
unsigned int atomicAdd(unsigned int address, unsigned int val);*
- Unsigned 64-bit integer atomic add
unsigned long long int atomicAdd(unsigned long long int address, unsigned long long int val);*
- Single-precision floating-point atomic add
(capability > 2.0)
 - float atomicAdd(float* address, float val);

Building synchronization with atomics

- How would we build `__syncthreads()` for block?
- How would we create `__syncthreads()` for entire grid?
 - And why would this not be a good idea?
- How would we create a critical section? I.e., one thread per block executing a particular section of code?
- How would we create a critical section per grid?
 - Why doesn't this have the same issue as `__syncthreads()` for grid?

Atomic Compare and Swap (CAS)

```
Bool atomicCAS(int *address, int old, int new)
{
    if (*address != old)
        return false;

    *address = new;

    return true;
}
```

Atomic Add using CAS

```
int atomicAdd(int* address, int value)
{
    Bool done = false;

    while (!done) {
        old_v = *address;
        done = atomicCAS(address, old_v, old_v+value);
    }

    return old_v;
}
```

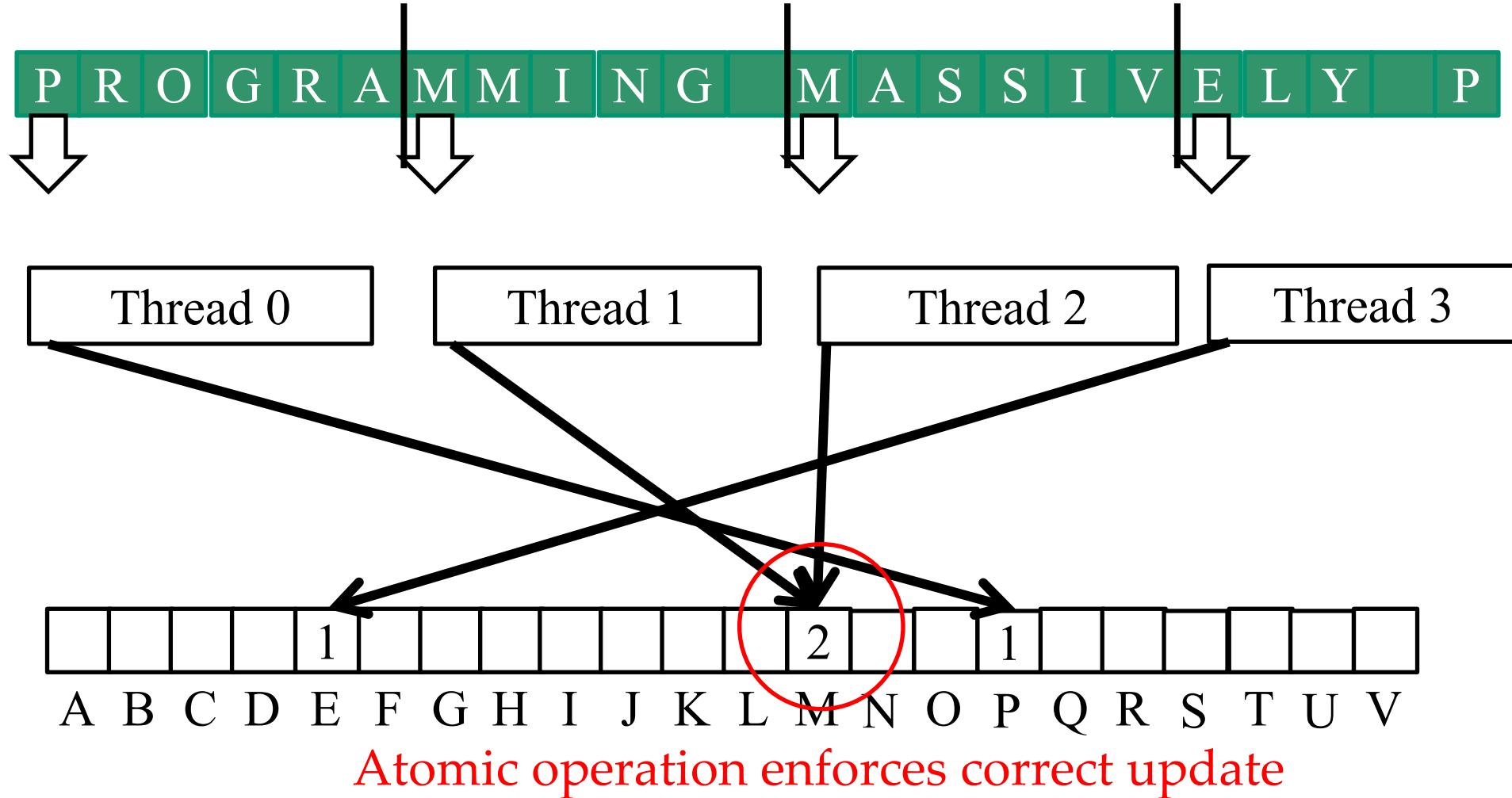
Histogramming

- A method for extracting notable features and patterns from large data sets
 - Feature extraction for object recognition in images
 - Fraud detection in credit card transactions
 - Correlating heavenly object movements in astrophysics
 - ...
- Basic histograms - for each element in the data set, use the value to identify a “bin” to increment

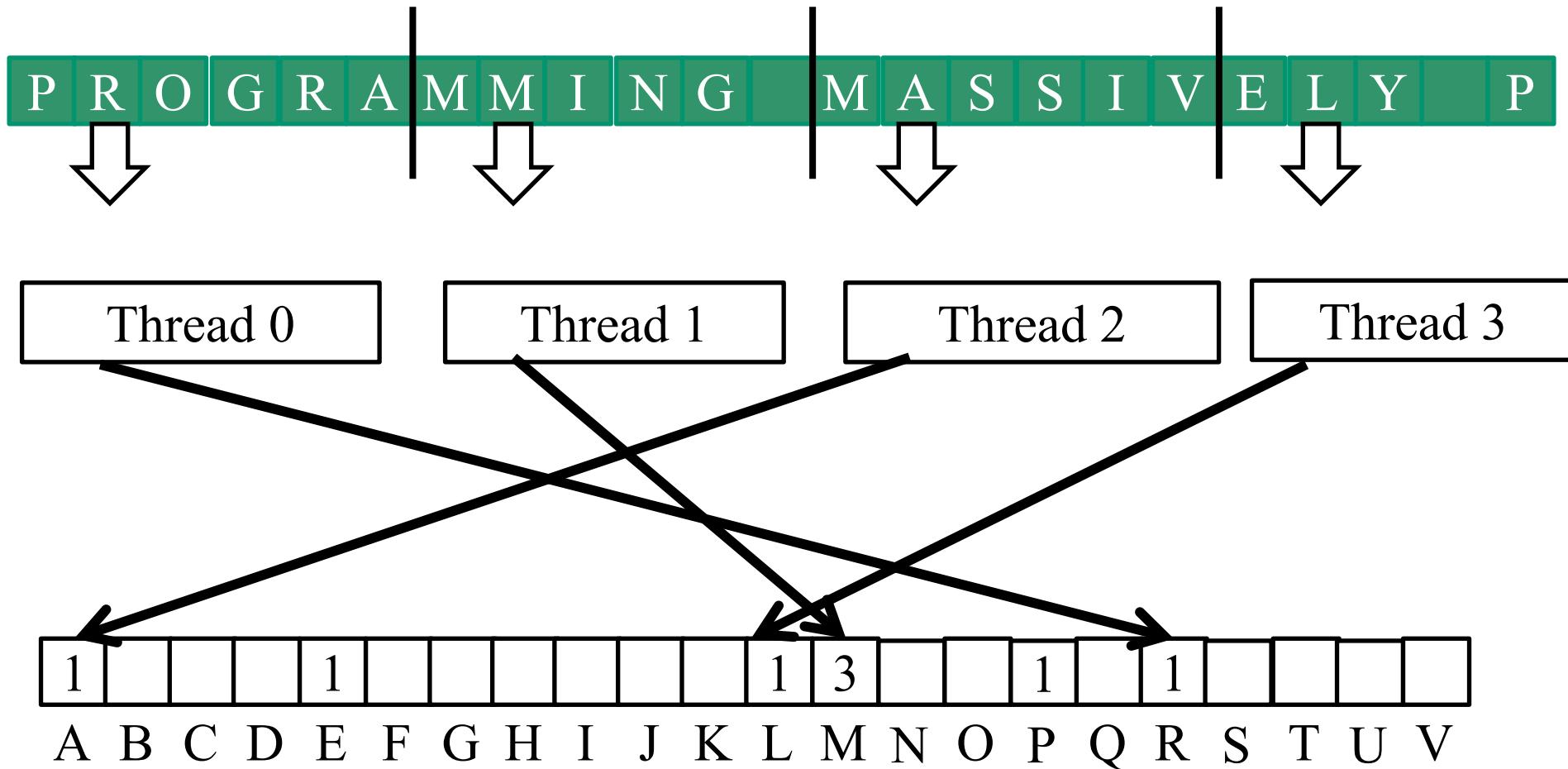
A Histogram Example

- In sentence “Programming Massively Parallel Processors” build a histogram of frequencies of each letter
- A(4), C(1), E(1), G(1), ...
- How do you do this in parallel?

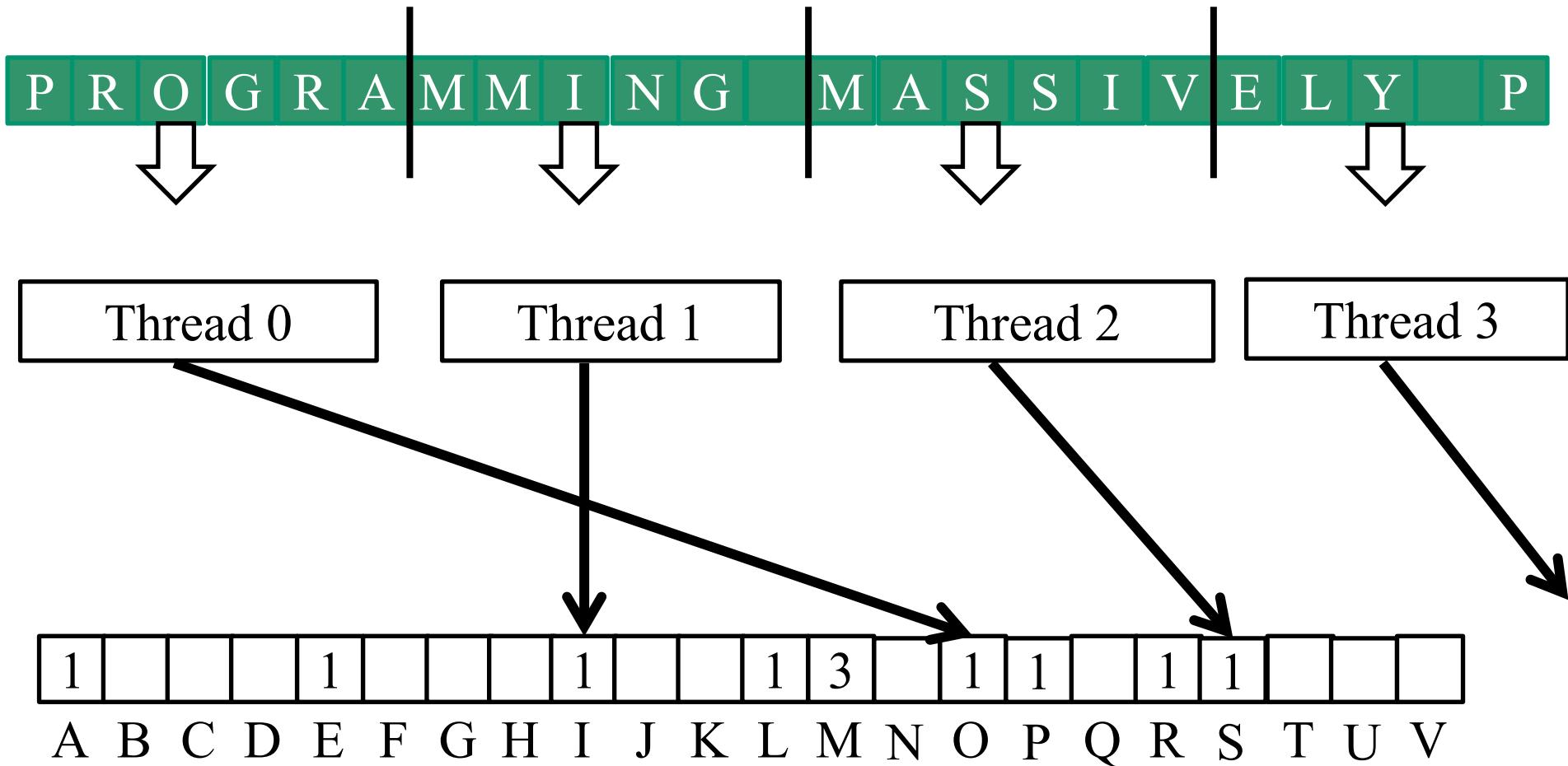
Iteration #1 – 1st letter in each section



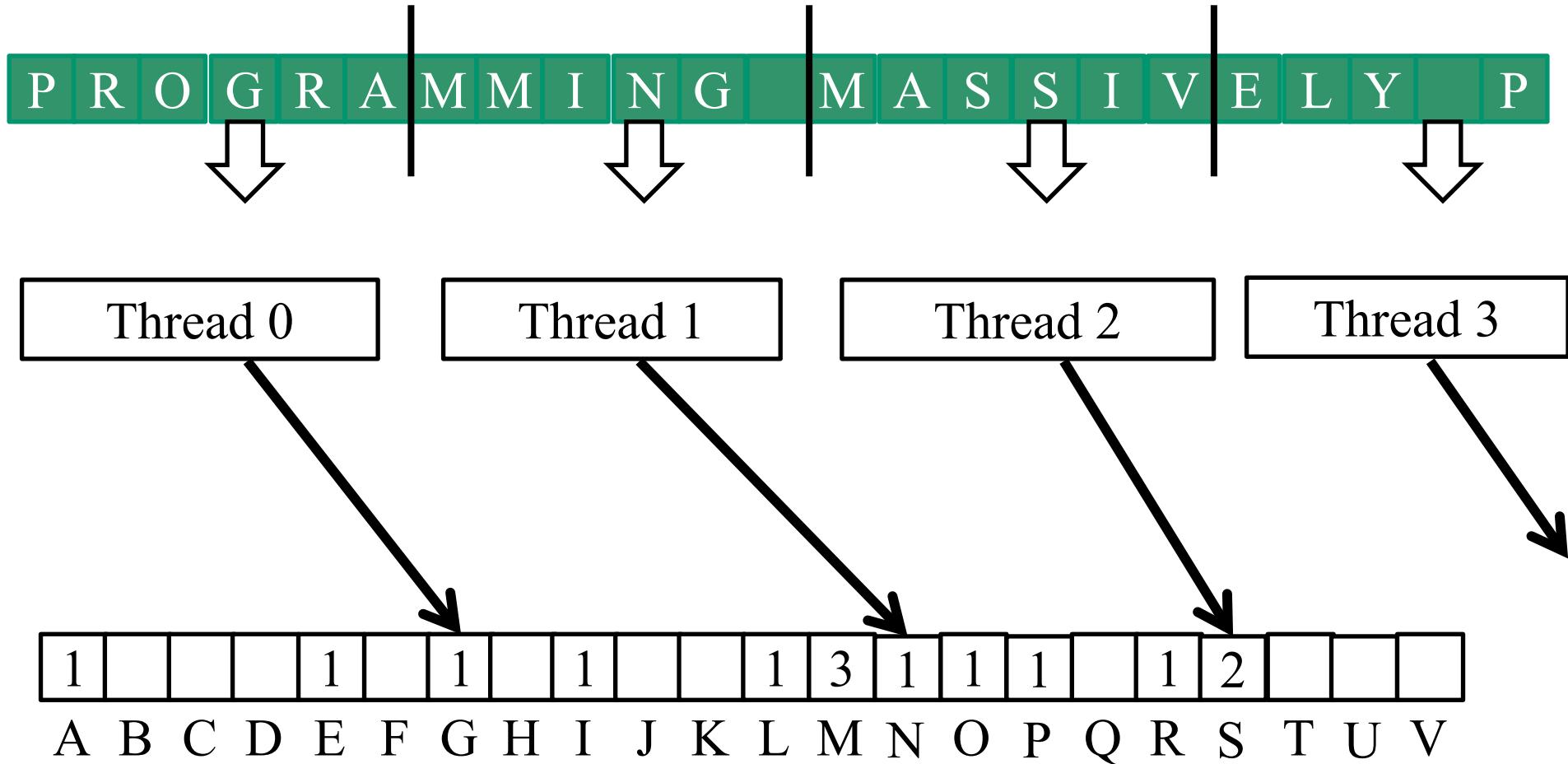
Iteration #2 – 2nd letter in each section



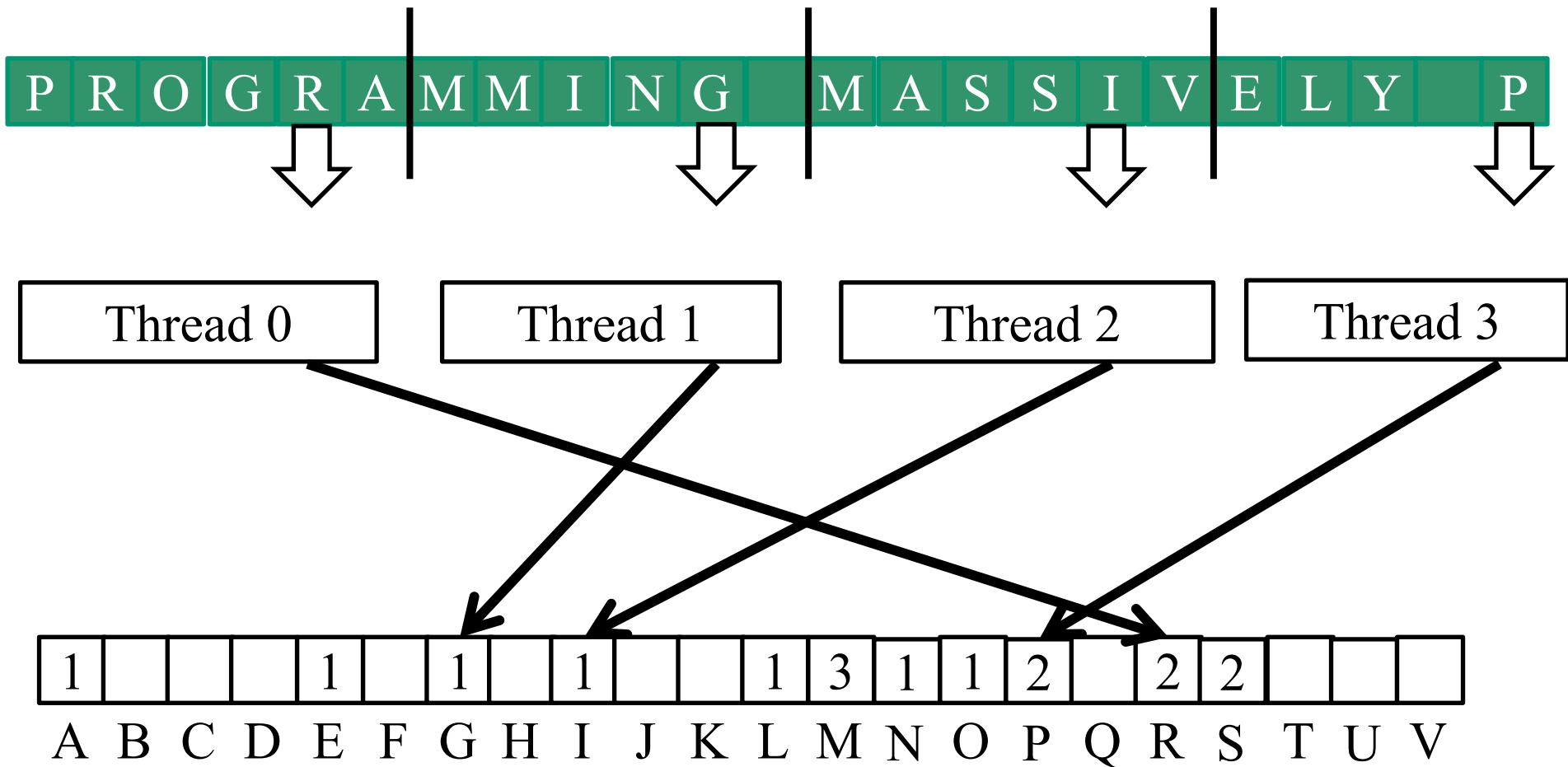
Iteration #3



Iteration #4

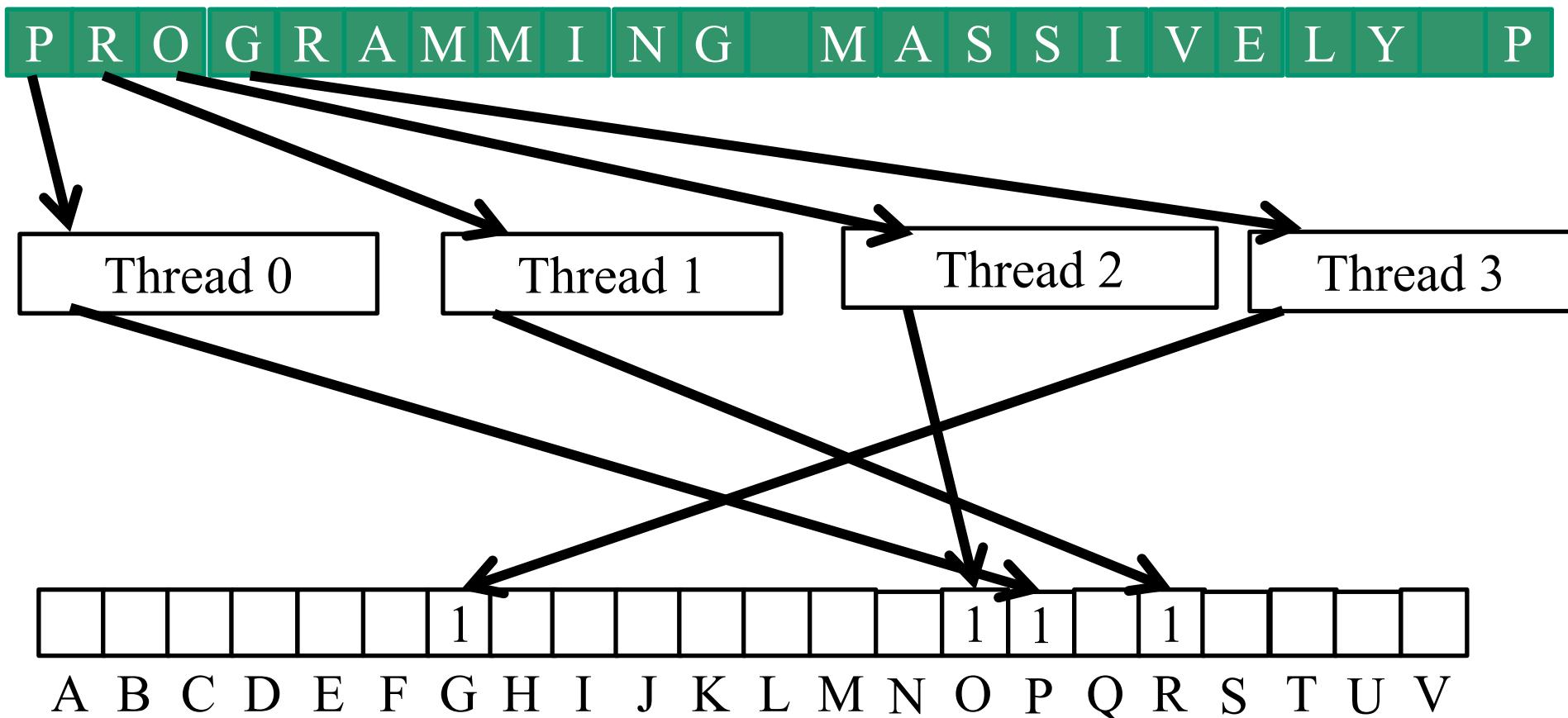


Iteration #5



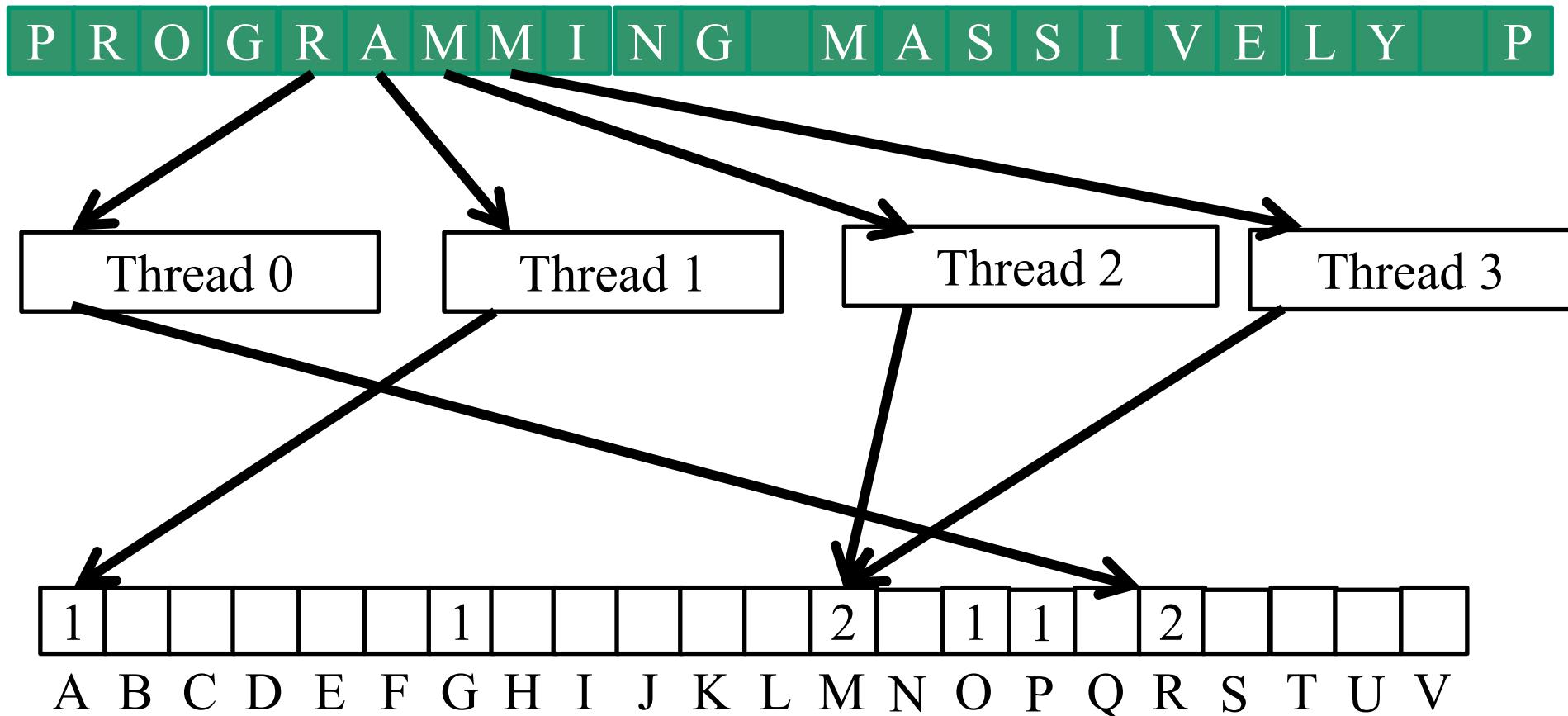
A better approach

- Reads from the input array are not coalesced
 - Assign inputs to each thread in a strided pattern
 - Adjacent threads process adjacent input letters



Iteration 2

- All threads move to the next section of input



A Histogram Kernel

- The kernel receives a pointer to the input buffer
- Each thread process the input in a strided pattern

```
__global__
void histo_kernel(unsigned char *buffer,
                  long size, unsigned int *histo) {

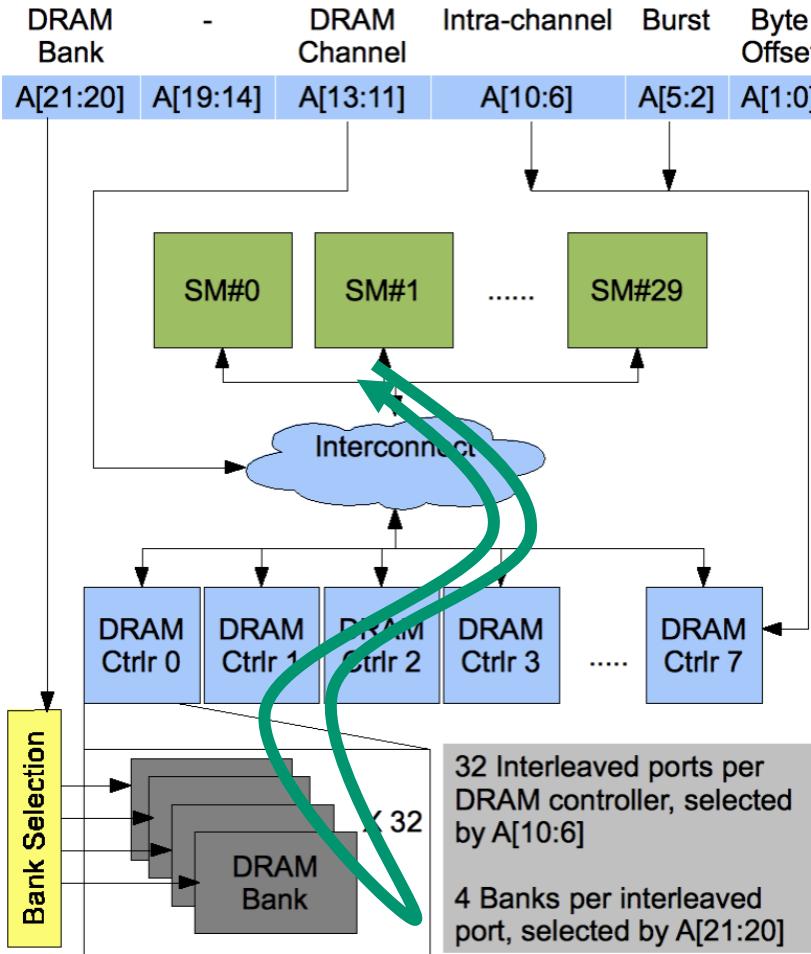
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;
```

More on the Histogram Kernel

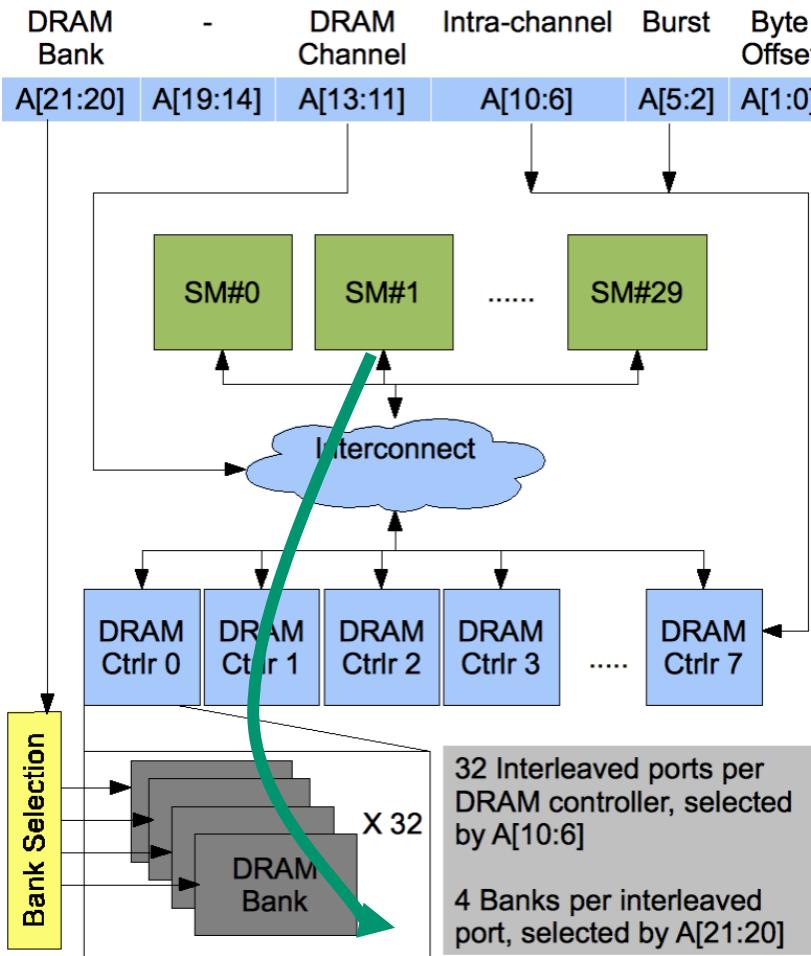
```
// All threads in the grid collectively handle  
// blockDim.x * gridDim.x consecutive elements  
  
while (i < size) {  
    atomicAdd( &(histo[buffer[i]]), 1);  
    i += stride;  
}  
}
```

Atomic Operations on DRAM



- An atomic operation starts with a read, with a latency of a few hundred cycles

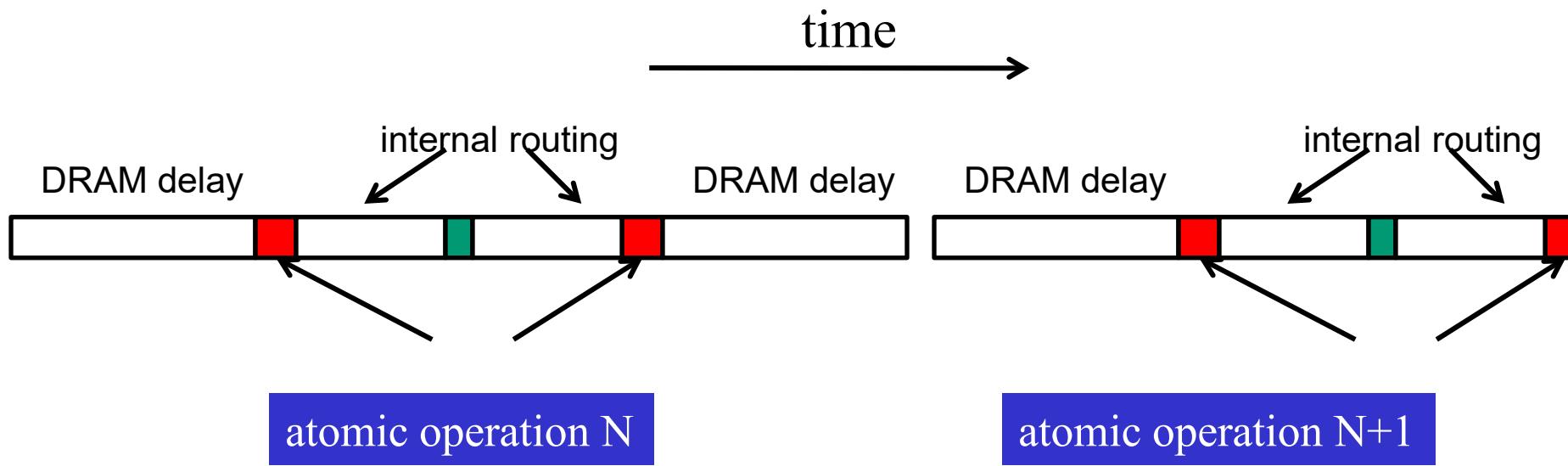
Atomic Operations on DRAM



- An atomic operation starts with a read, with a latency of a few hundred cycles
- The atomic operation ends with a write, with a latency of a few hundred cycles
- During this whole time, no one else can access the location

Atomic Operations on DRAM

- Each Load-Modify-Store has two full memory access delays
 - All atomic operations on the same variable (RAM location) are serialized



Latency determines throughput of atomic operations

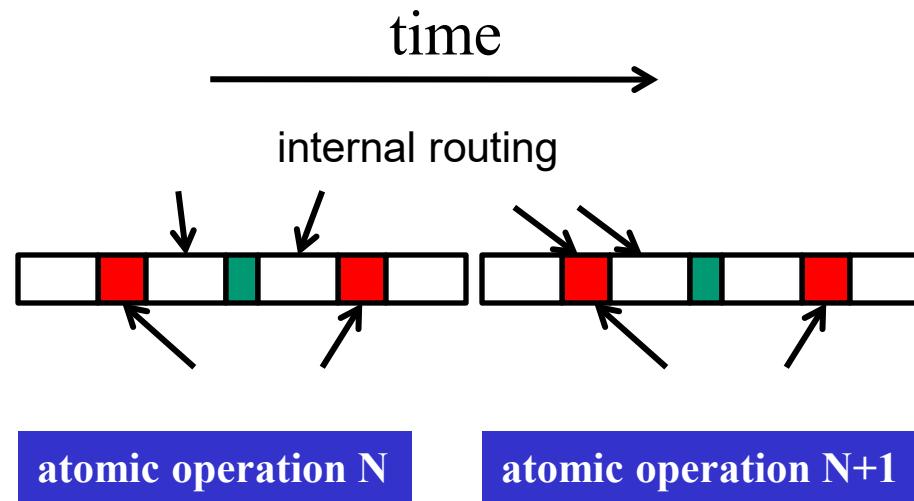
- Throughput of an atomic operation is the rate at which the application can execute an atomic operation on a particular location.
- The rate is limited by the total latency of the read-modify-write sequence, typically more than 1000 cycles for global memory (DRAM) locations.
- This means that if many threads attempt to do atomic operation on the same location (contention), the memory bandwidth is reduced to $< 1/1000!$

You may have a similar experience in supermarket checkout

- Some customers realize that they missed an item after they started to check out
- They run to the isle and get the item while the line waits
 - The rate of check is reduced due to the long latency of running to the isle and back.
- Imagine a store where every customer starts the check out before they even fetch any of the items
 - The rate of the checkout will be $1 / (\text{entire shopping time of each customer})$

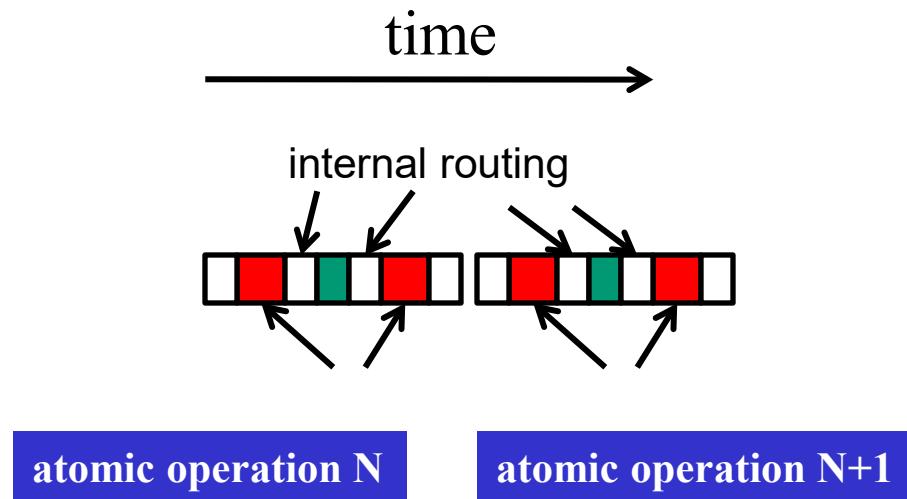
Hardware Improvements

- Atomic operations on L2 cache
 - medium latency, but still serialized
 - Global to all blocks
 - “Free improvement” on Global Memory atomics



Hardware Improvements

- Atomic operations on Shared Memory
 - Very short latency, but still serialized
 - Private to each thread block
 - Need algorithm work by programmers (more later)



Atomics in Shared Memory Requires Privatization

- Create private copies of the histo[] array for each thread block

```
__global__
```

```
void histo_kernel(unsigned char *buffer,
                  long size, unsigned int *histo) {
    __shared__ unsigned int histo_private[256];
    // warning: this will not work correctly if there are fewer than 256 threads!
    if (threadIdx.x < 256)
        histo_private[threadIdx.x] = 0;
    __syncthreads();
```

Build Private Histogram

- Use private copies of the histo[] array to compute

```
int i = threadIdx.x + blockIdx.x * blockDim.x;  
  
// stride is total number of threads  
int stride = blockDim.x * gridDim.x;  
  
while (i < size) {  
    atomicAdd( &(private_histo[buffer[i]]), 1);  
    i += stride;  
}
```

Build Final Histogram

- Copy from the histo[] arrays from each thread block to global memory

```
// wait for all other threads in the block to finish
__syncthreads();

if (threadIdx.x < 256)
    atomicAdd( &(histo[threadIdx.x]) ,
                private_histo[threadIdx.x] ) ;

}
```

More on Privatization

- Privatization is a powerful and frequently used techniques for parallelizing applications
- The operation needs to be associative and commutative
 - Histogram add operation is associative and commutative
- The histogram size needs to be small
 - Fits into shared memory
- What if the histogram is too large to privatize?



**ANY MORE QUESTIONS
READ CHAPTER 9**

Problem Solving

- Q: Suppose a processor supports atomic operations in L2 cache. Assume that each atomic operation takes 5ns to complete in L2 cache and 120ns to complete in DRAM. The kernel performs 20 floating-point operations per atomic operation, and a floating point operation takes 1ns. Assume the time of L2 atomic operations, DRAM atomic operations, and floating-point operations in each thread do not overlap. The floating-point throughput of the kernel execution is 0.2424 GFLOPS, and every thread in a block performs 5 atomic operations and 100 floating-point operations. What percent of the atomic operations happened in L2 cache?
- A: ??



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 17 Parallel Computation Patterns – Parallel Scan (Prefix Sum) – Part 2

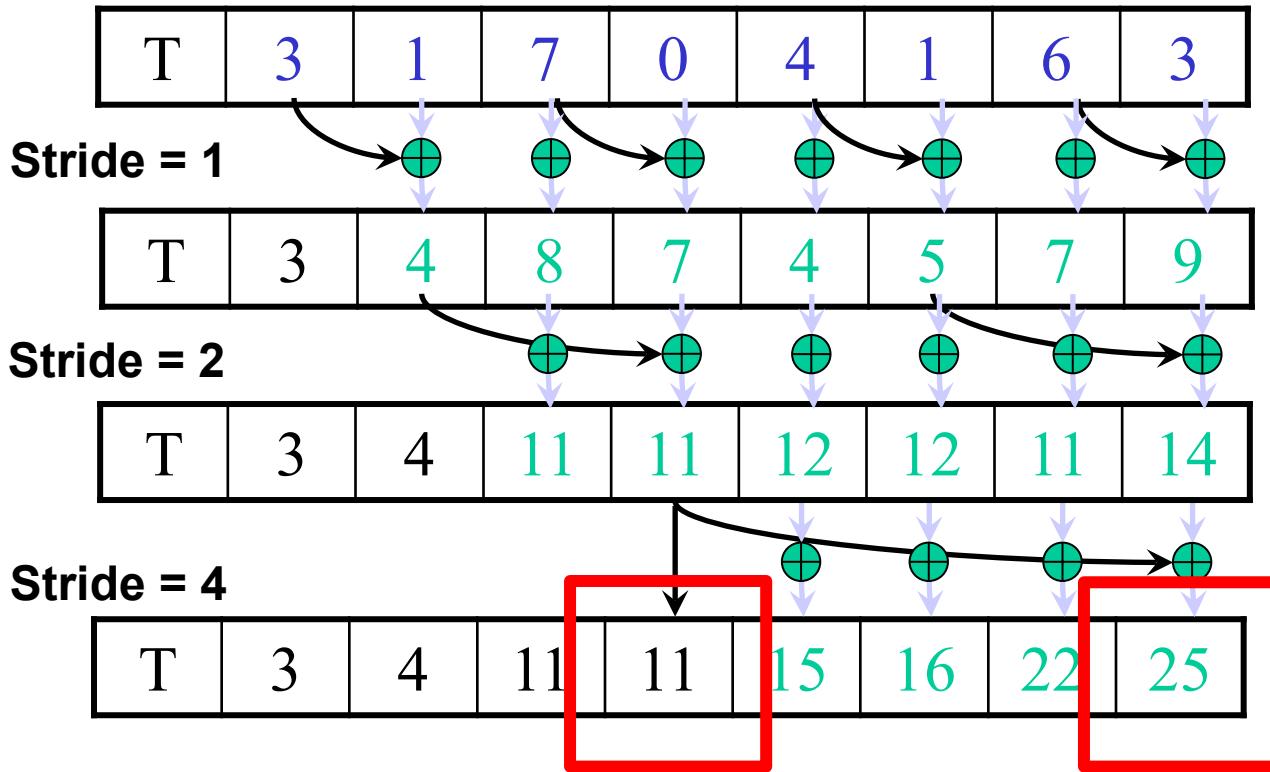
Course Reminders

- MP5.1 & MP5.2
 - MP5.1: Implement a kernel and associated host code that performs reduction of a 1D list stored in a C array. The reduction should give the sum of the list. You should implement the improved kernel discussed in the lecture. Your kernel should be able to handle input lists of arbitrary length.
 - MP5.2: Implement one or more kernels and their associated host code to perform parallel scan on a 1D list. The scan operator used will be addition. You should implement the work- efficient kernel discussed in lecture. Your kernel should be able to handle input lists of arbitrary length. However, for simplicity, you can assume that the input list will be at most 2,048 * 2,048 elements.
- Project Milestone 2: Baseline Convolution Kernel
 - Due 4/7/23

Objective

- To master parallel scan (prefix sum) algorithms
 - Work-efficiency vs. latency
 - Brent-Kung Tree Algorithm
 - Hierarchical algorithms

A Kogge-Stone Parallel Scan Algorithm



Work Efficiency Analysis

- A Kogge-Stone scan kernel executes $\log(n)$ parallel iterations
 - The steps do $(n-1), (n-2), (n-4), \dots, (n - n/2)$ add operations each
 - Total # of add operations: $n * \log(n) - (n-1) \rightarrow O(n * \log(n))$ work
- This scan algorithm is not very work efficient
 - Sequential scan algorithm does n adds
 - A factor of $\log(n)$ hurts: 20x for 1,000,000 elements!
 - Typically used within each block, where $n \leq 1,024$
- A parallel algorithm can be slow when execution resources are saturated due to low work efficiency

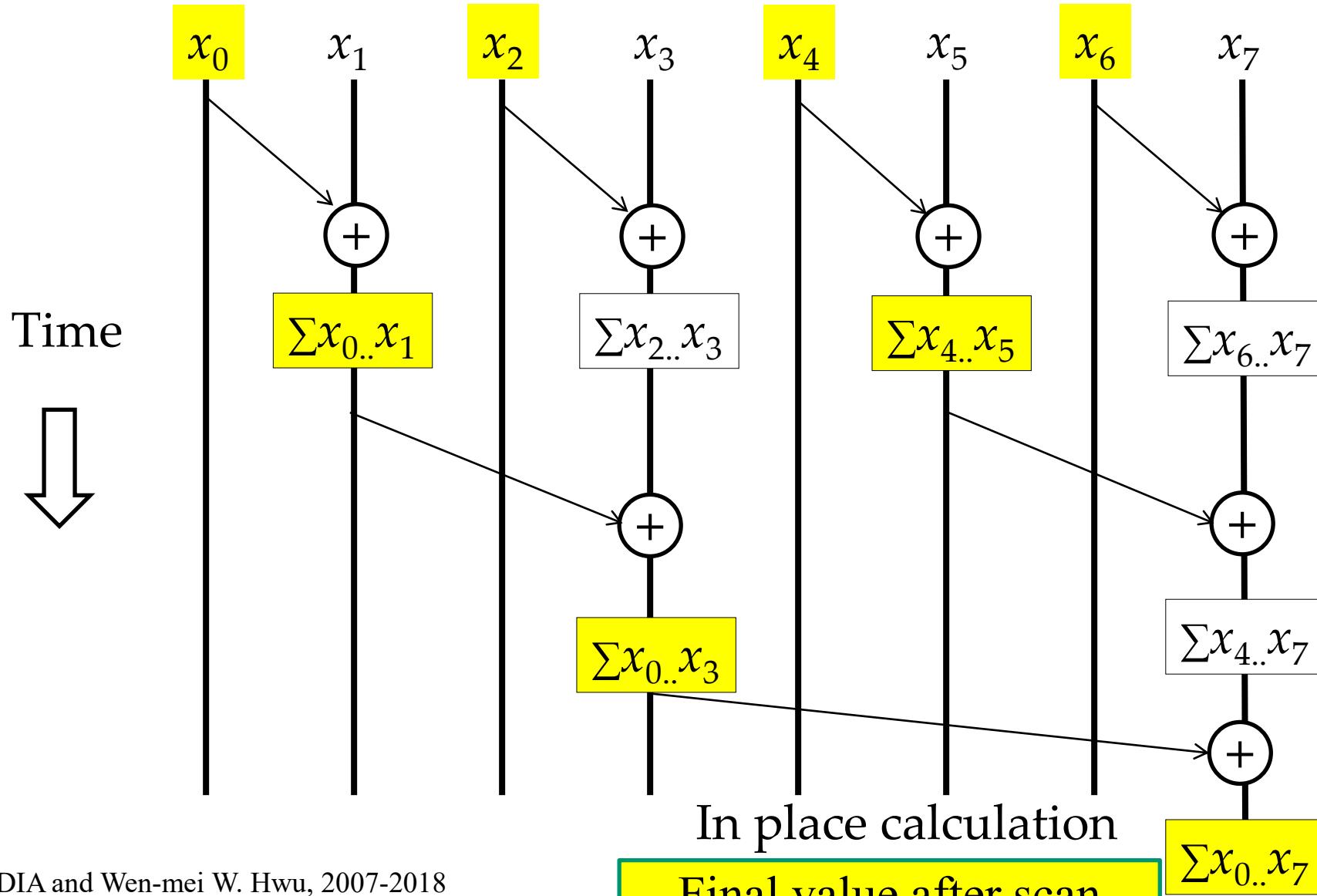
Improving Efficiency

- A common parallel algorithm pattern:

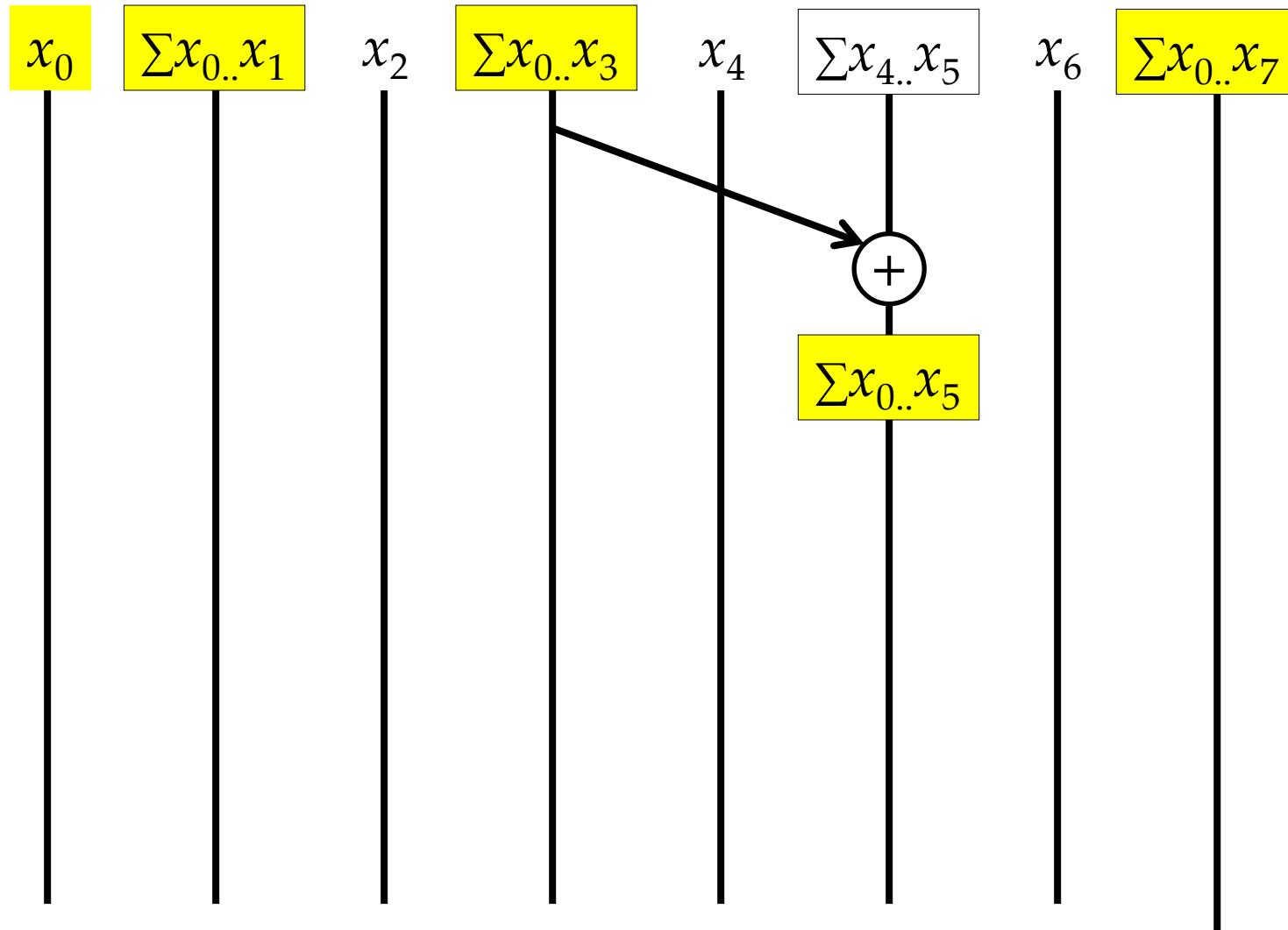
Balanced Trees

- Build a balanced binary tree on the input data and sweep it to and from the root
 - Tree is not an actual data structure, but a conceptual pattern
- For scan:
 - Traverse down from leaves to root building partial sums at internal nodes in the tree
 - Root holds sum of all leaves
 - Traverse back up the tree building the scan from the partial sums

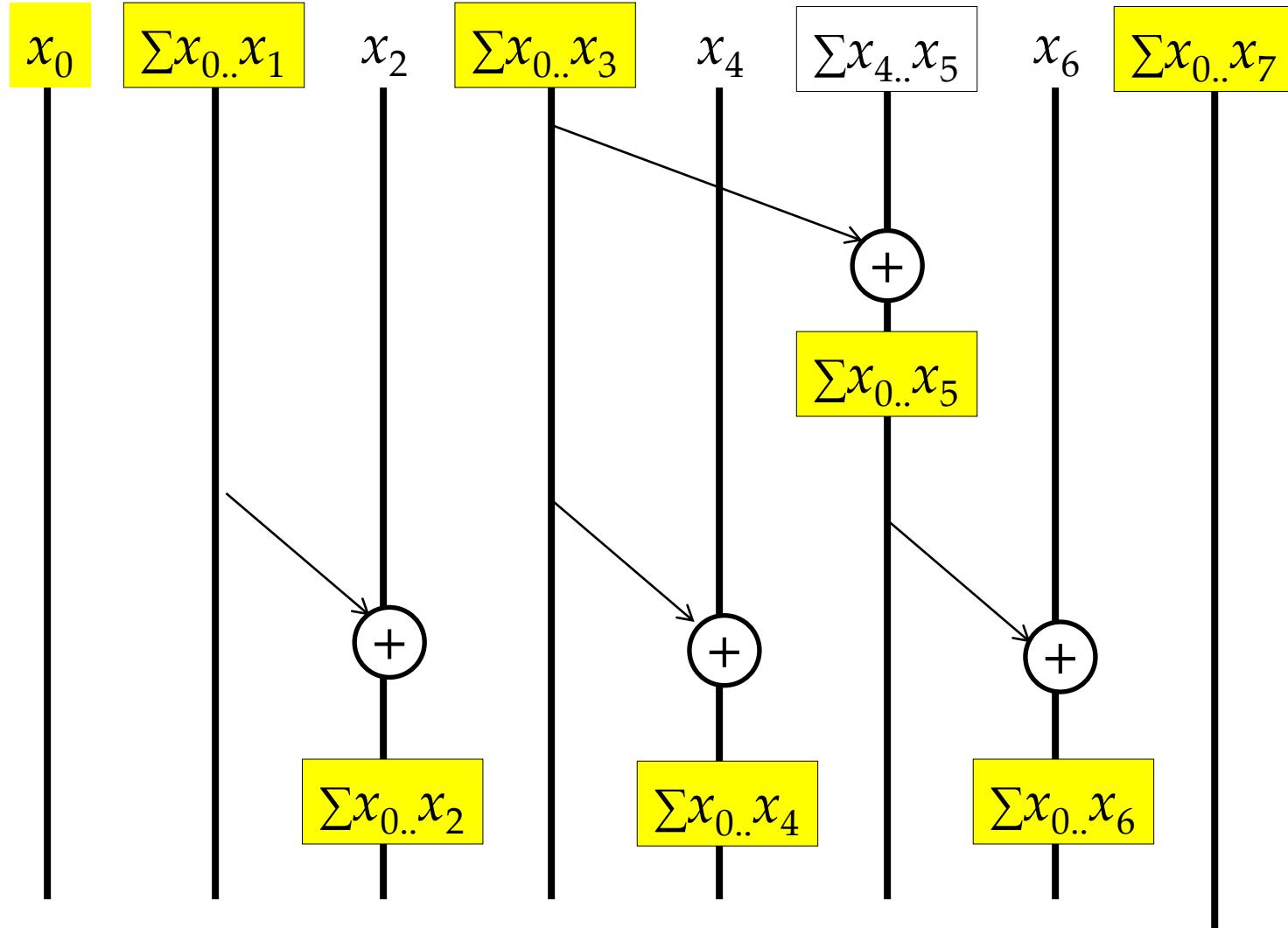
Brent-Kung Parallel Scan Step



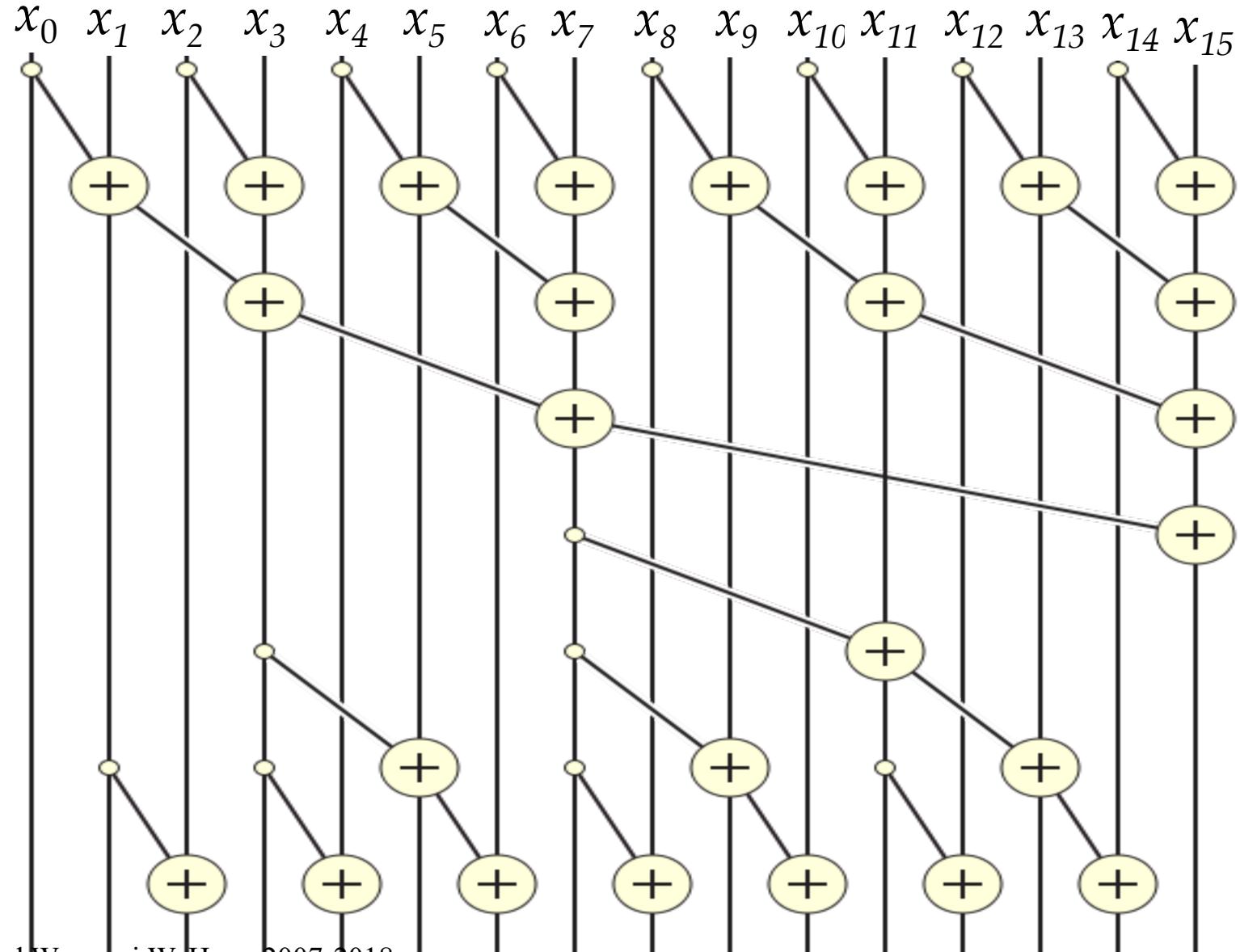
Inclusive Post-Scan Step



Inclusive Post Scan Step



Putting it Together (Data View)



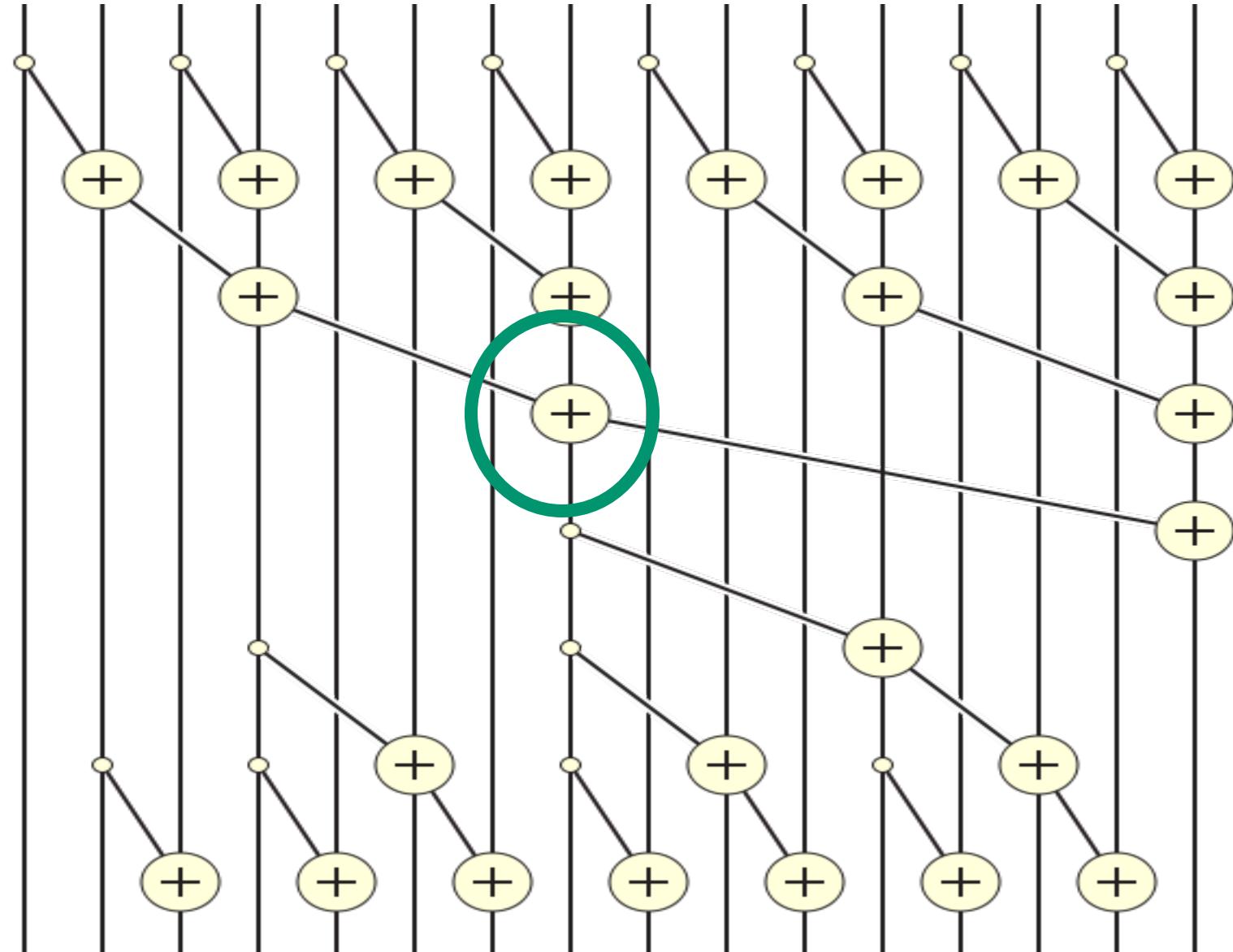
Reduction Step Kernel Code

```
// float T[2*BLOCK_SIZE] is in shared memory
// for previous slide, BLOCK_SIZE is 8

int stride = 1;
while(stride < 2*BLOCK_SIZE) {
    __syncthreads();
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < 2*BLOCK_SIZE && (index-stride) >= 0)
        T[index] += T[index-stride];
    stride = stride*2;
}

// In our example,
// threadIdx.x+1      = 1, 2, 3, 4, 5, 6, 7, 8
// stride = 1, index = 1, 3, 5, 7, 9, 11, 13, 15
```

Putting it Together



Post Scan Step (Distribution Tree)

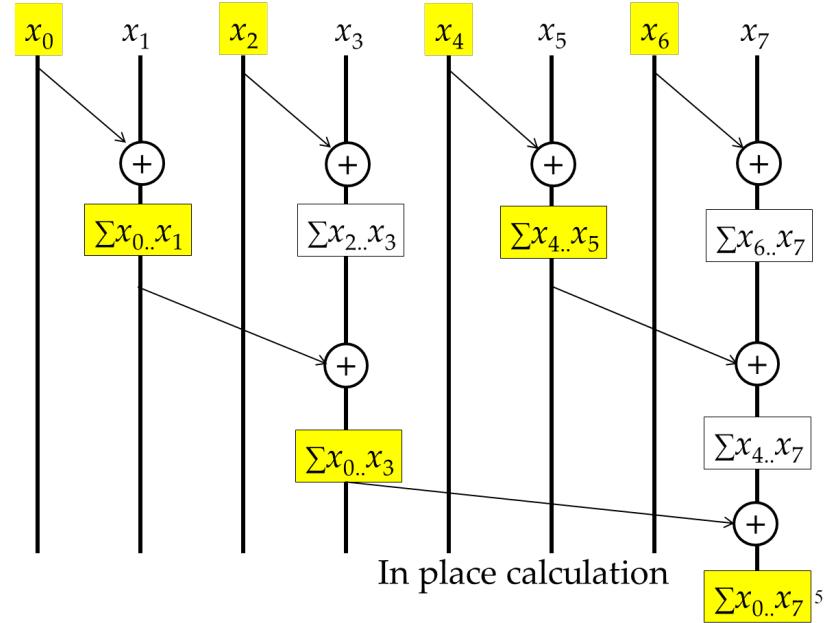
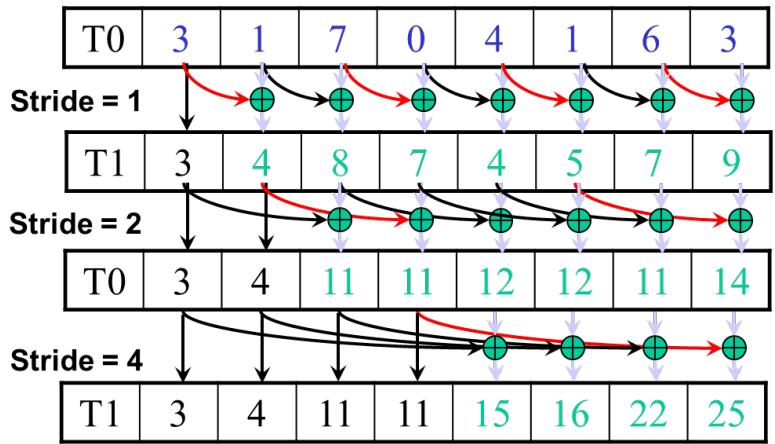
```
int stride = BLOCK_SIZE/2;
while(stride > 0) {
    __syncthreads();
    int index = (threadIdx.x+1)*stride*2 - 1;
    if ((index+stride) < 2*BLOCK_SIZE)
        T[index+stride] += T[index];
    stride = stride / 2;
}
```

```
// In our example,
// BLOCK_SIZE=8 stride=4, 2, 1
// for first iteration, active thread = 0 index = 7, stride = 11
```

Work Analysis

- The parallel Scan executes $2 * \log(n)$ parallel iterations
 - $\log(n)$ in reduction and $\log(n)$ in post scan
 - The iterations do $n/2, n/4,..1, (2-1), …, (n/4-1), (n/2-1)$ useful adds
 - In our example, $n = 16$, the number of useful adds is $16/2 + 16/4 + 16/8 + 16/16 + (16/8-1) + (16/4-1) + (16/2-1)$
 - Total adds: $(n-1) + (n-2) - (\log(n) - 1) = 2*(n-1) - \log(n) \rightarrow O(n)$ work
- The total number of adds is no more than twice of that done in the efficient sequential algorithm
 - The benefit of parallelism can easily overcome the $2\times$ work when there is sufficient hardware

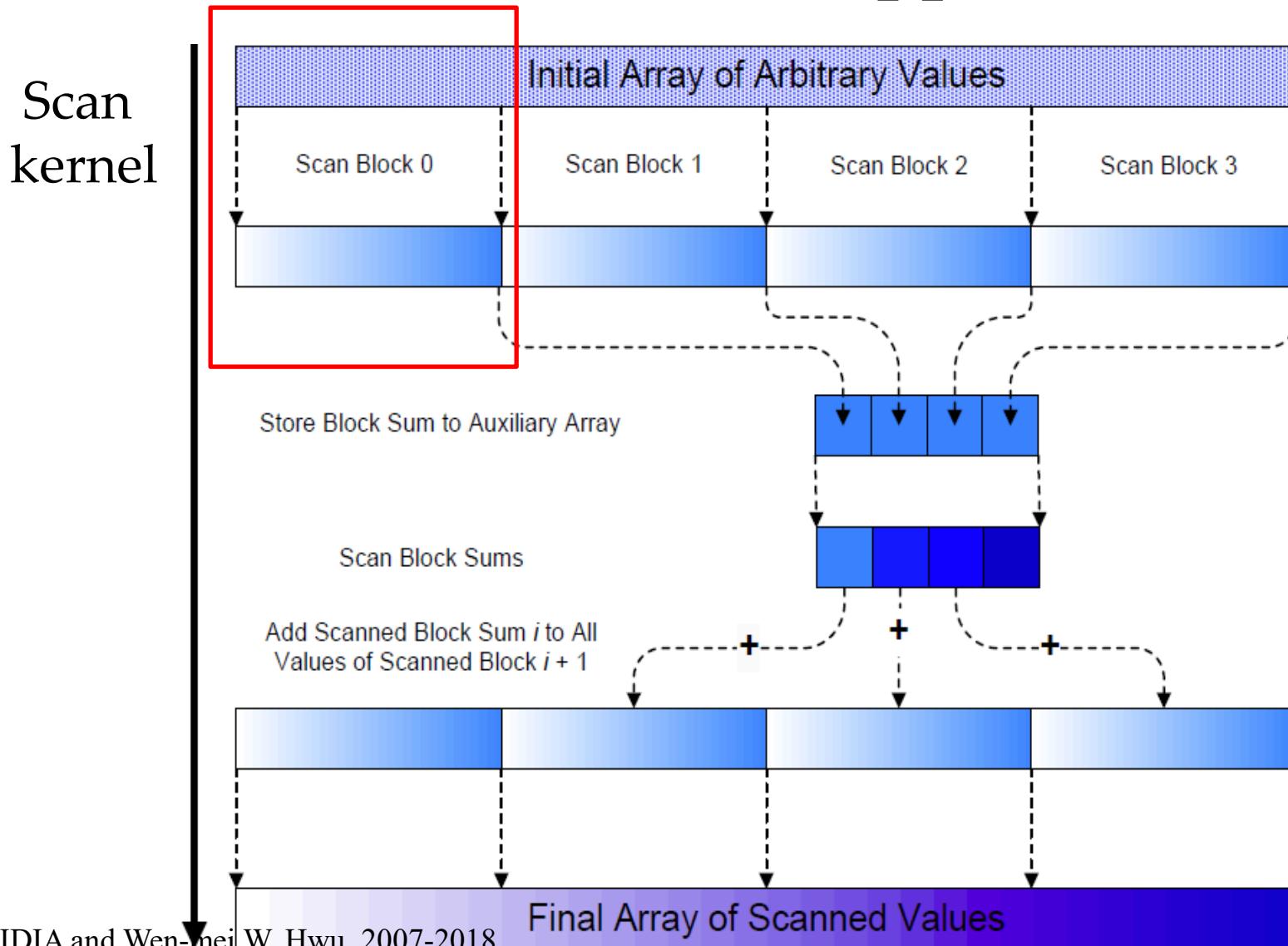
Kogge-Stone vs. Brent-Kung



- Brent-Kung uses half the number of threads compared to Kogge-Stone
 - Each thread should load two elements into the shared memory
- Brent-Kung takes twice the number of steps compared to Kogge-Stone
 - Kogge-Stone is more popular for parallel scan with blocks in GPUs

Overall Flow of Complete Scan

A Hierarchical Approach



Using Global Memory Contents in CUDA

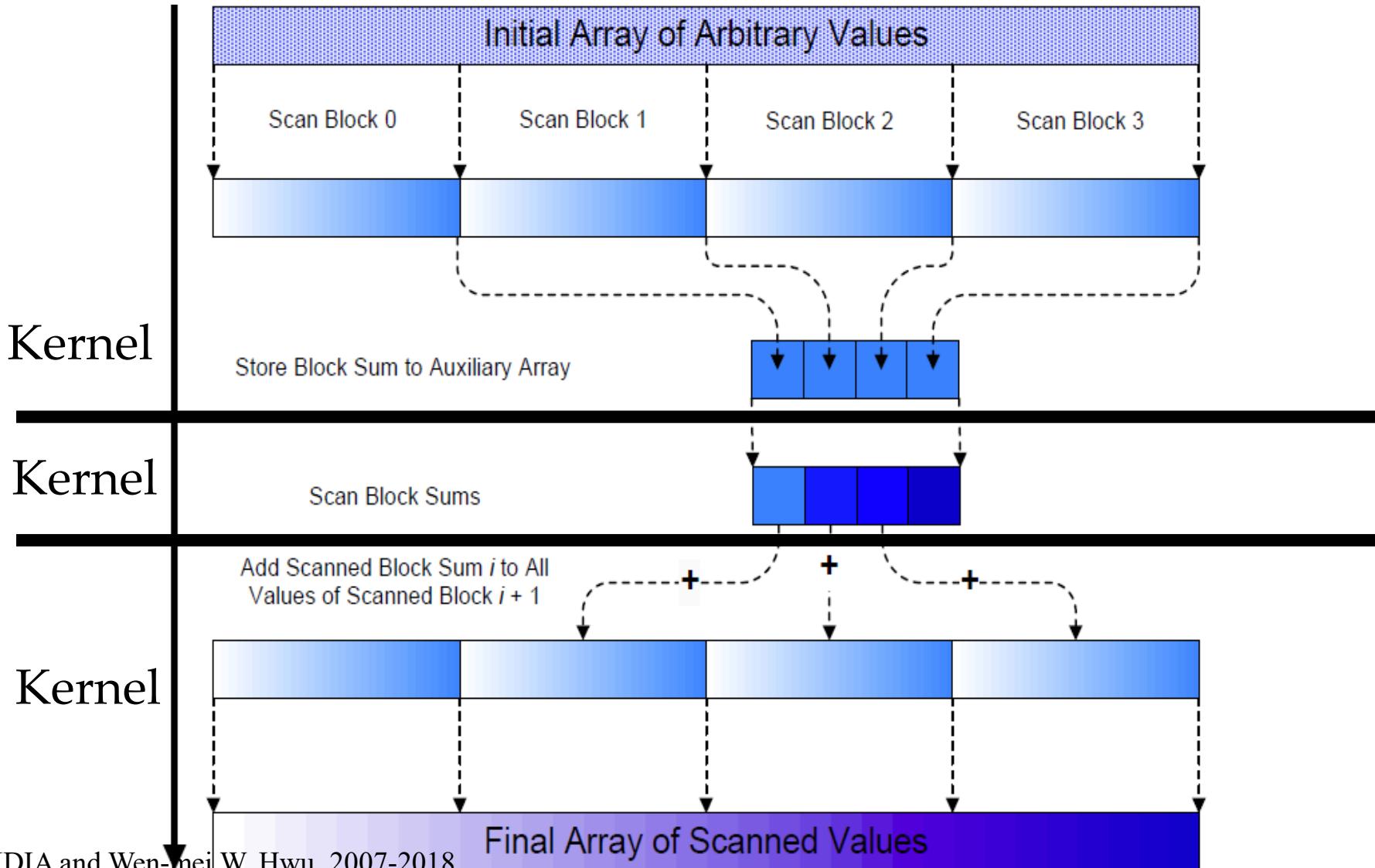
- Data in registers and shared memory of one thread block are not visible to other blocks
- To make data visible, the data has to be written into global memory
- However, any data written to the global memory are not visible until a memory fence. This is typically done by terminating the kernel execution
- Launch another kernel to continue the execution. The global memory writes done by the terminated kernels are visible to all thread blocks.

Scan of Arbitrary Length Input

- Build on the scan kernel that handles up to $2 * \text{blockDim.x}$ elements from Brent-Kung
 - For Kogge-Stone, have each section of blockDim.x elements assigned to a block
- Have each block write the sum of its section into a Sum array using its blockIdx.x as index
- Run parallel scan on the Sum array
 - May need to break down Sum into multiple sections if it is too big for a block
- Add the scanned Sum array values to the elements of corresponding sections

Overall Flow of Complete Scan

A Hierarchical Approach



(Exclusive) Scan Definition

Definition: *The exclusive scan operation takes a binary associative operator \oplus , and an array of n elements*

$$[x_0, x_1, \dots, x_{n-1}]$$

and returns the array

$$[0, x_0, (x_0 \oplus x_1), \dots, (x_0 \oplus x_1 \oplus \dots \oplus x_{n-2})].$$

Example: If \oplus is addition, then the exclusive scan operation on
[3 1 7 0 4 1 6 3],
would return [0 3 4 11 11 15 16 22].

Why Exclusive Scan

- To find the beginning address of allocated buffers
- Inclusive and Exclusive scans can be easily derived from each other; it is a matter of convenience

[3 1 7 0 4 1 6 3]

Exclusive [0 3 4 11 11 15 16 22]

Inclusive [3 4 11 11 15 16 22 25]

A simple exclusive scan kernel

- Adapt an inclusive, Kogge-Stone scan kernel
 - Block 0:
 - Thread 0 loads 0 into (shared) XY[0]
 - Other threads load (global) X[threadIdx.x-1] into XY[threadIdx.x]
 - All other blocks:
 - All threads load X[blockIdx.x*blockDim.x+threadIdx.x-1] into XY[threadIdx.x]
- Similar adaption for Brent-Kung kernel but pay attention that each thread loads two elements
 - Only one zero should be loaded
 - All elements should be shifted by only one position



**ANY MORE QUESTIONS?
READ CHAPTER 8**

Problem Solving

- Q: Suppose we have a kernel function that performs partial sum reduction. The block dimension is $(64,1,1)$. During the **second** iteration of the for loop, is there any warp that has a control divergence?

```
__shared__ int partialSum[2*BLOCK_SIZE];
for (int stride = blockDim.x; stride >= 1; stride = stride/2) {
    __syncthreads();
    if (threadIdx.x < stride) partialSum[t] += partialSum[t + stride];
}
```

- A:
 - No

Problem Solving

- Q: Consider a kernel that performs Brent-Kung scan algorithm and assume that there are 1024 elements in a section, and a warp size is 32. In which iteration will there be at least one warp that has a control divergence? The stride is 1 for the first iteration.
- A:
 - In 6th iteration

Problem Solving

- Q: Suppose we use Brent-Kung in a hierarchical approach to perform parallel scan on a 1D input array of 2^{42} elements. We use 1024 threads per block in all our Brent-Kung kernels and our GPU supports at most 2048 blocks per grid. Each block processes 2048 elements. What is the best approximation of the number of floating-point add operations performed per thread block in the reduction and post scan steps in the Brent-Kung kernel (summed together)?
- A:
 - 2048×2



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 16 Parallel Computation Patterns – Parallel Scan (Prefix Sum)

Course Reminders

- Project Milestone: we are grading it now
- Project Milestone 2: Baseline Convolution Kernel
 - Updated repo is released
 - Due 4/7/23
- Lab 5.1 – due this week
 - Implement a kernel and associated host code that performs reduction of a 1D list stored in a C array. The reduction should give the sum of the list. You should implement the improved kernel discussed in the lecture. Your kernel should be able to handle input lists of **arbitrary length**.
- Lab 5.2 – due next week
- MT1
 - Exam average is 70, stdev is 15.3
 - Regrade requests are due by the end of this Friday

Objective

- To learn parallel scan (prefix sum) algorithms based on reductions
- To learn the concept of double buffering
- To understand tradeoffs between work efficiency and latency

Scan Includes all Partial Results

Reductions are a simplified form of scans.

In scan / parallel prefix,

- we need all of the partial sums
- (or whatever the operator might be).

(Inclusive) Scan (Prefix-Sum) Definition

Definition: *The scan operation takes a binary associative operator \oplus , and an array of n elements*

$$[x_0, x_1, \dots, x_{n-1}],$$

and returns the prefix-sum array

$$[x_0, (x_0 \oplus x_1), (x_0 \oplus x_1 \oplus x_2), \dots, (x_0 \oplus x_1 \oplus \dots \oplus x_{n-1})].$$

Example: If \oplus is addition, the scan operation
on the array [3 1 7 0 4 1 6 3],
returns [3 4 11 11 15 16 22 25].

Example: Sharing a Big Sandwich

You order a 100-inch sandwich to feed 10 people, and you know how much each person wants in inches:

[3 5 2 7 28 4 3 0 8 1].

How do you cut the bread quickly?

How much of the sandwich is left over?

Method 1: sequentially!

Cut 3 inches, then cut 5 inches, then ...

Method 2: **calculate cutting offsets with prefix-sum**

[3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (39 inches left)

Typical Applications of Scan

A simple and useful parallel building block.

Convert sequential recurrences

```
for(j=1; j<n; j++)
    out[j] = out[j-1] + f(j);
```

into parallel:

```
forall(j) { temp[j] = f(j) };
scan(out, temp);
```

Typical Applications of Scan

- Useful for many parallel algorithms:
 - radix sort
 - quicksort
 - String comparison
 - Lexical analysis
 - Stream compaction
 - Polynomial evaluation
 - Solving recurrences
 - Tree operations
 - Histograms
 - Etc.

Other Applications

- Assigning camp slots
- Assigning farmer market space
- Allocating memory to parallel threads
- Allocating memory buffer to communication channels
- ...

An Inclusive Sequential Scan

Given a sequence $[x_0, x_1, x_2, \dots]$

Calculate output $[y_0, y_1, y_2, \dots]$

Such that

$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

...

Using a recursive definition

$$y_i = y_{i-1} + x_i$$

A Sequential C Implementation

```
y[0] = x[0];  
for (i = 1; i < Max_i; i++)  
    y[i] = y[i-1] + x[i];
```

Computationally efficient:

N additions needed for N elements - O(N).

A Naïve Inclusive Parallel Scan

- Assign one thread to calculate each y element
- Have every thread to add up all x elements needed for the y element

$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

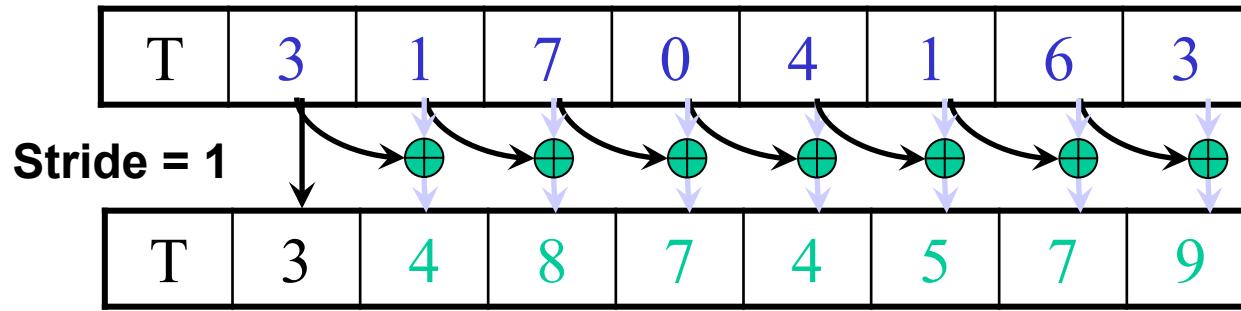
“Parallel programming is easy as long as you do not care about performance.”

Parallel Inclusive Scan using Reduction Trees

Calculate each output element as the reduction of all previous elements

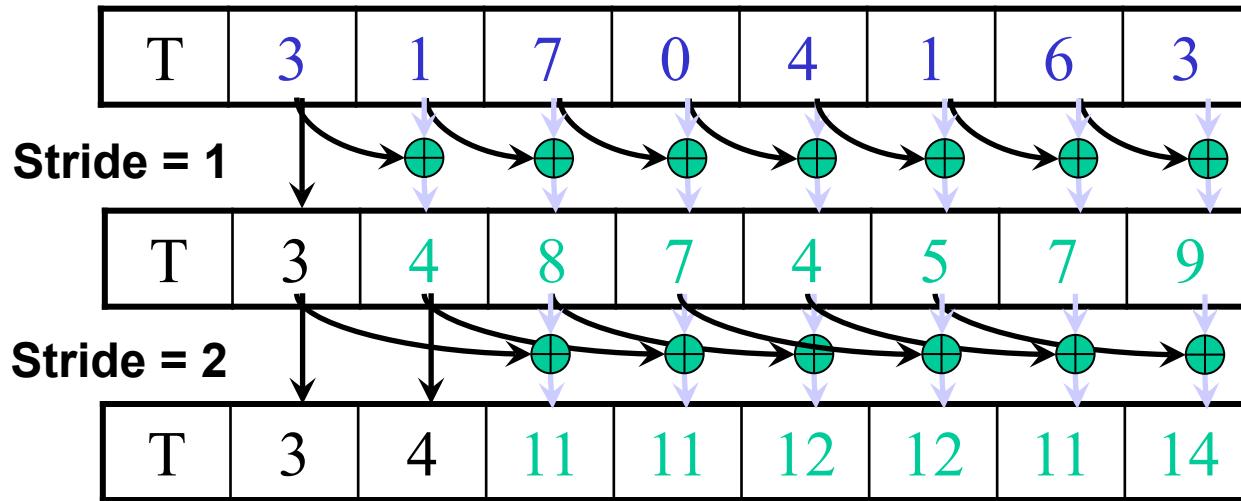
- Some reduction partial sums will be shared among the calculation of output elements
- Based on hardware added design by Peter Kogge and Harold Stone at IBM in the 1970s – Kogge-Stone Trees
- Goal: low latency

A Kogge-Stone Parallel Scan Algorithm



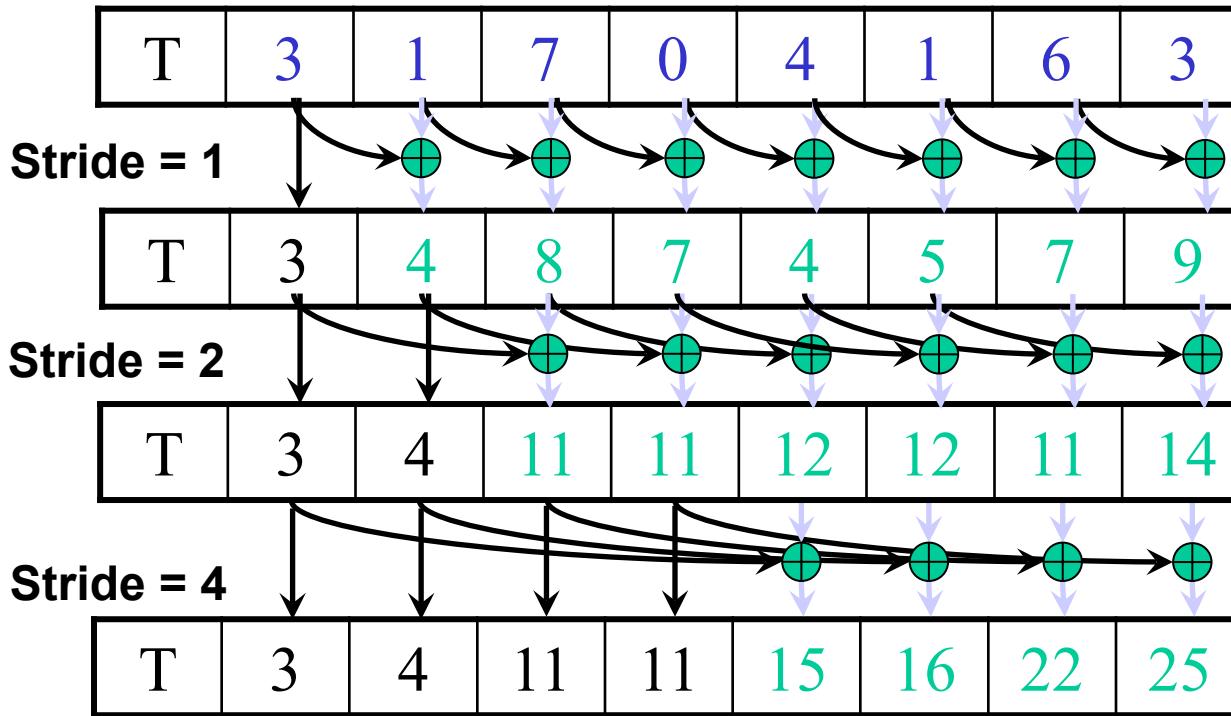
Iteration #1
Stride = 1

A Kogge-Stone Parallel Scan Algorithm



Iteration #2
Stride = 2

A Kogge-Stone Parallel Scan Algorithm



Iteration #3
Stride = 4

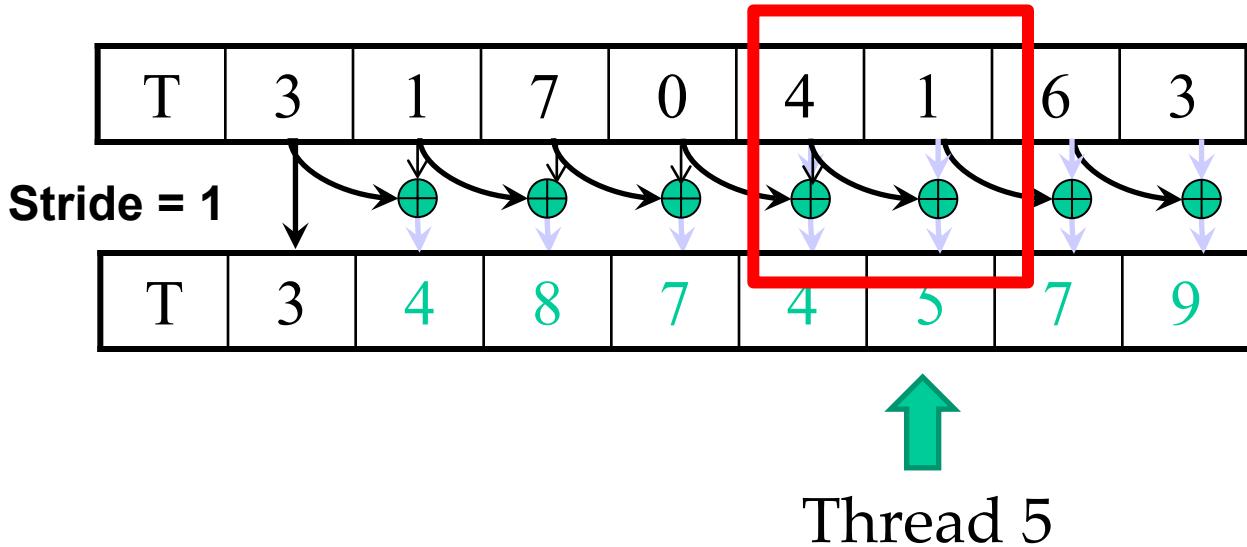
A Kogge-Stone Parallel Scan Algorithm

T	3	1	7	0	4	1	6	3
---	---	---	---	---	---	---	---	---

1. Load input from global memory into shared memory array T

Each thread loads one value from the input (global memory) array into shared memory array T.

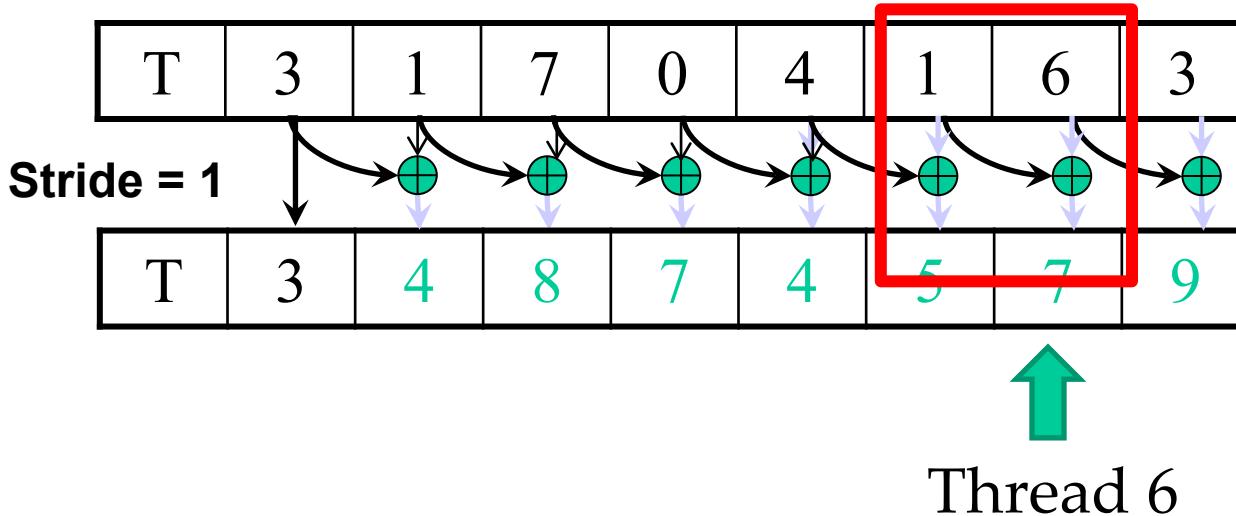
A Kogge-Stone Parallel Scan Algorithm



1. (previous slide)
2. Assuming n is a power of 2. Iterate $\log(n)$ times, stride from 1 to $n/2$. Threads *stride* to $n-1$ active: add pairs of elements that are *stride* elements apart.

- Iteration #1
Stride = 1
- Active threads: *stride* to $n-1$ ($n - \text{stride}$ active threads)
 - Thread j adds elements $T[j]$ and $T[j-\text{stride}]$ and writes result into element $T[j]$
 - Each iteration requires two syncthreads
 - make sure that input is in place
 - make sure that all input elements have been used

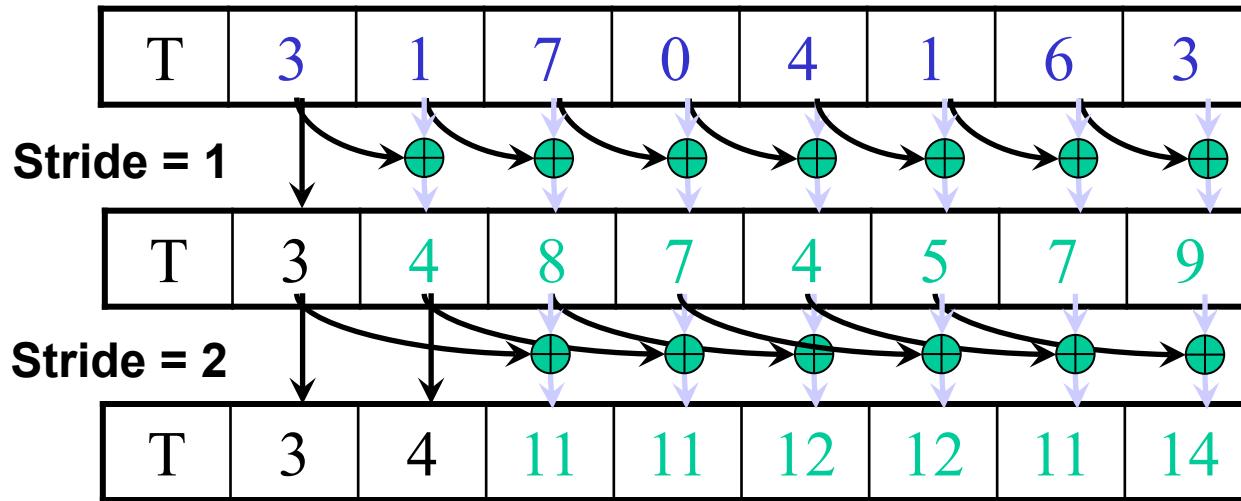
A Kogge-Stone Parallel Scan Algorithm



1. (previous slide)
2. Assuming n is a power of 2. Iterate $\log(n)$ times, stride from 1 to $n/2$. Threads *stride* to $n-1$ active: add pairs of elements that are *stride* elements apart.

- Iteration #1
Stride = 1
- Active threads: *stride* to $n-1$ ($n - \text{stride}$ active threads)
 - Thread j adds elements $T[j]$ and $T[j-\text{stride}]$ and writes result into element $T[j]$
 - Each iteration requires two syncthreads
 - `syncthreads();` // make sure that input is in place
 - `float temp = T[j] + T[j-stride];`
 - `syncthreads();` // make sure that previous output has been consumed
 - `T[j] = temp;`

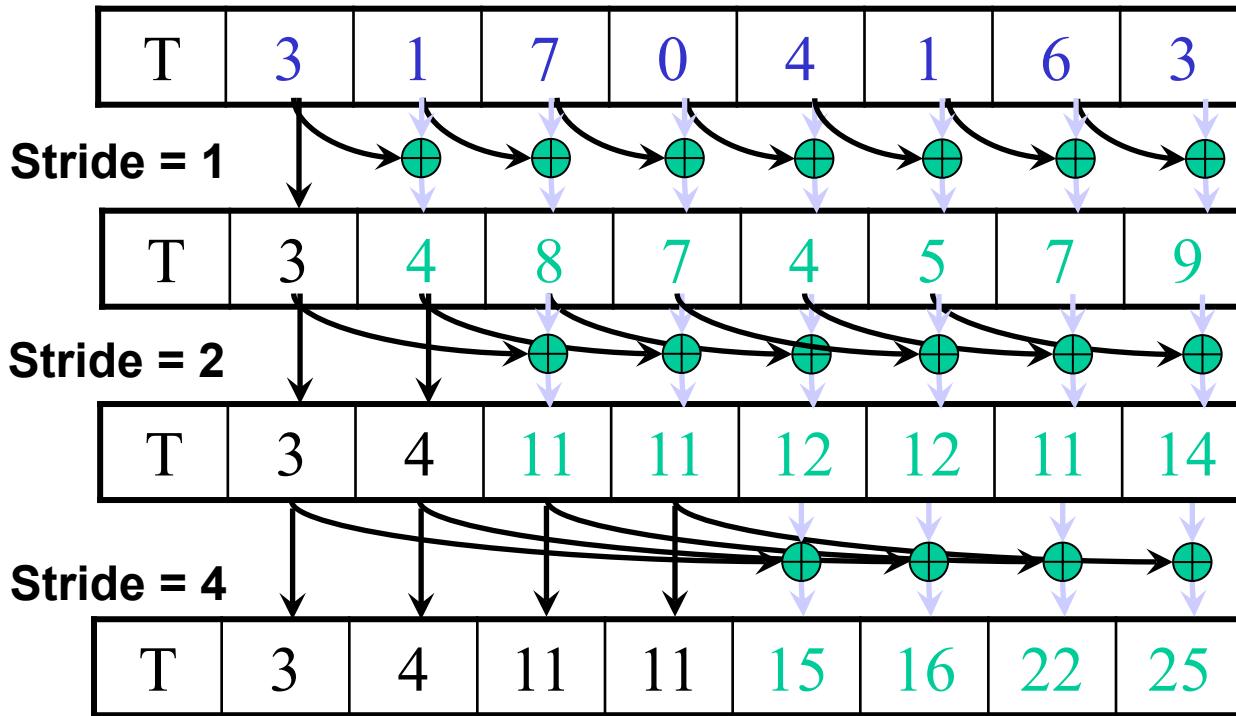
A Kogge-Stone Parallel Scan Algorithm



1. ...
2. Assuming n is a power of 2. Iterate $\log(n)$ times, stride from 1 to $n/2$. Threads *stride* to $n-1$ active: add pairs of elements that are *stride* elements apart.

Iteration #2
Stride = 2

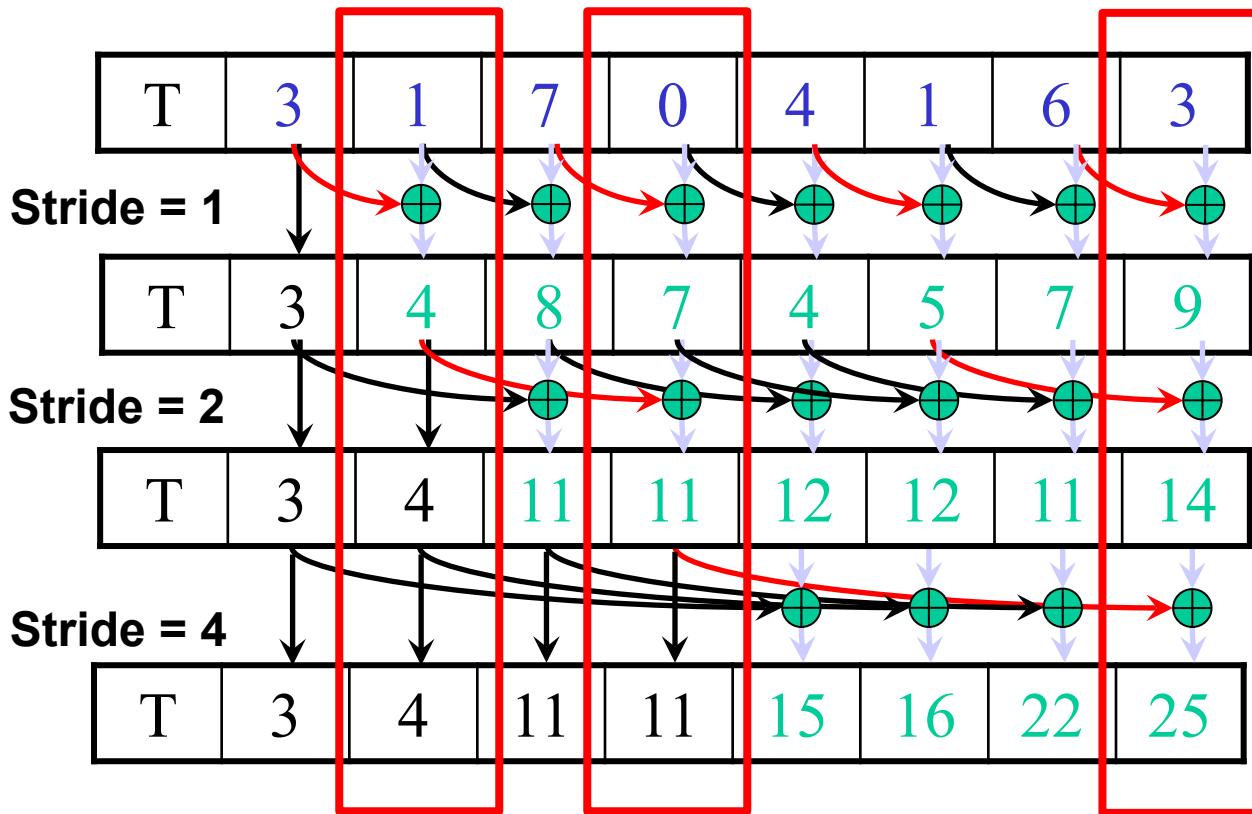
A Kogge-Stone Parallel Scan Algorithm



Iteration #3
Stride = 4

1. ...
2. ...
3. Write output from shared memory to device memory

Sharing Computation in Kogge-Stone



Iteration #3
Stride = 4

(Incomplete) Implementation

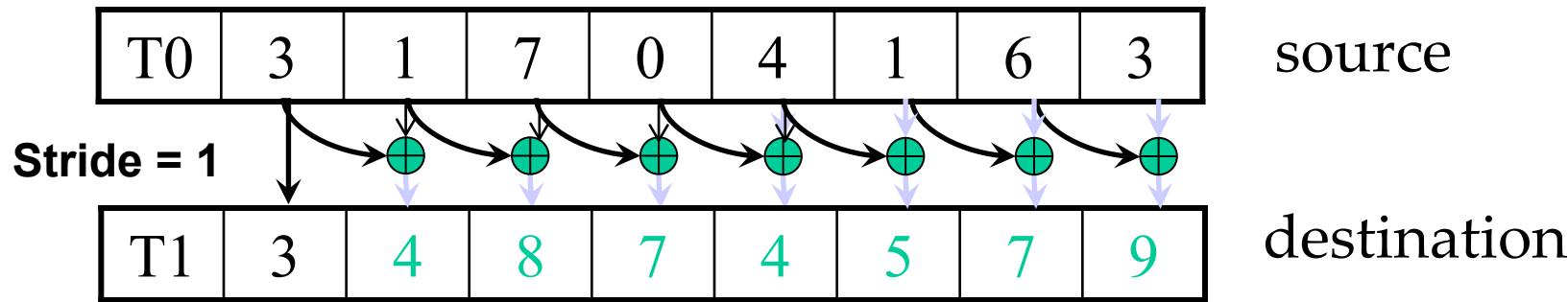
```
__global__
void Kogge_Stone_scan_kernel(float *X, float *Y, int InputSize)
{
    __shared__ float XY[SECTION_SIZE];
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < InputSize) XY[threadIdx.x] = X[i];

    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
        __syncthreads();
        if (threadIdx.x >= stride) // This code has a data race condition
            XY[threadIdx.x] += XY[threadIdx.x-stride];
    }
    Y[i] = XY[threadIdx.x];
}
```

Double Buffering

- Use two copies of data T0 and T1
- Start by using T0 as input and T1 as output
- Switch input/output roles after each iteration
 - Iteration 0: T0 as input and T1 as output
 - Iteration 1: T1 as input and T0 and output
 - Iteration 2: T0 as input and T1 as output
- This is typically implemented with two pointers, source and destination that swap their contents from one iteration to the next
- This eliminates the need for the second `__syncthreads()` call

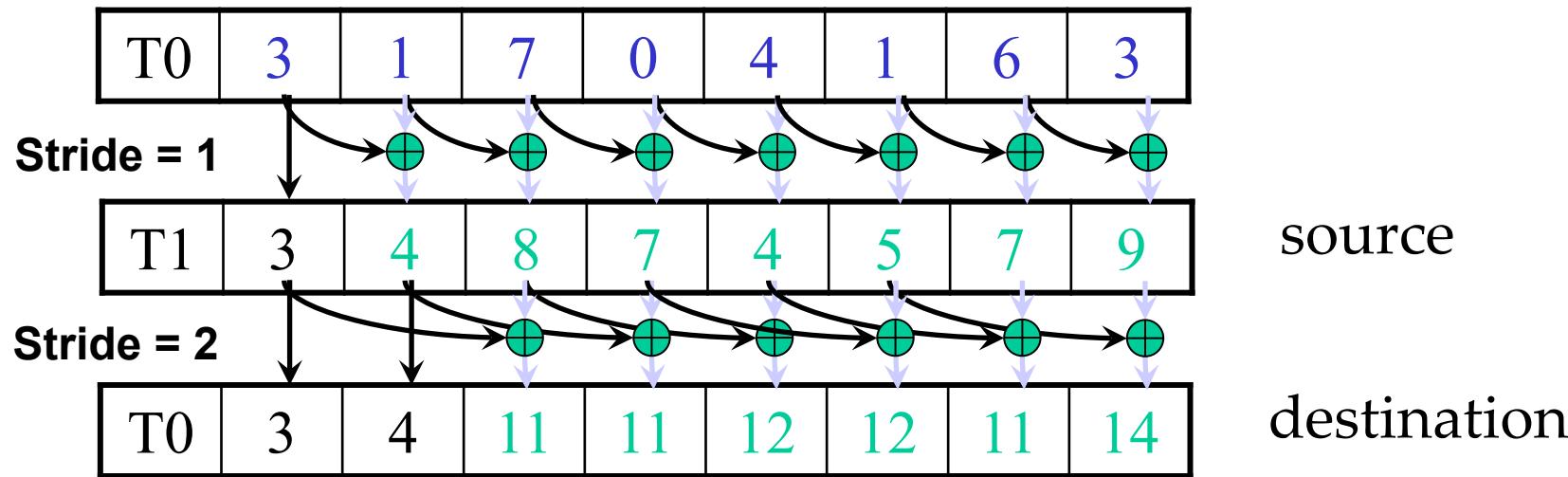
A Double-Buffered Kogge-Stone Parallel Scan Algorithm



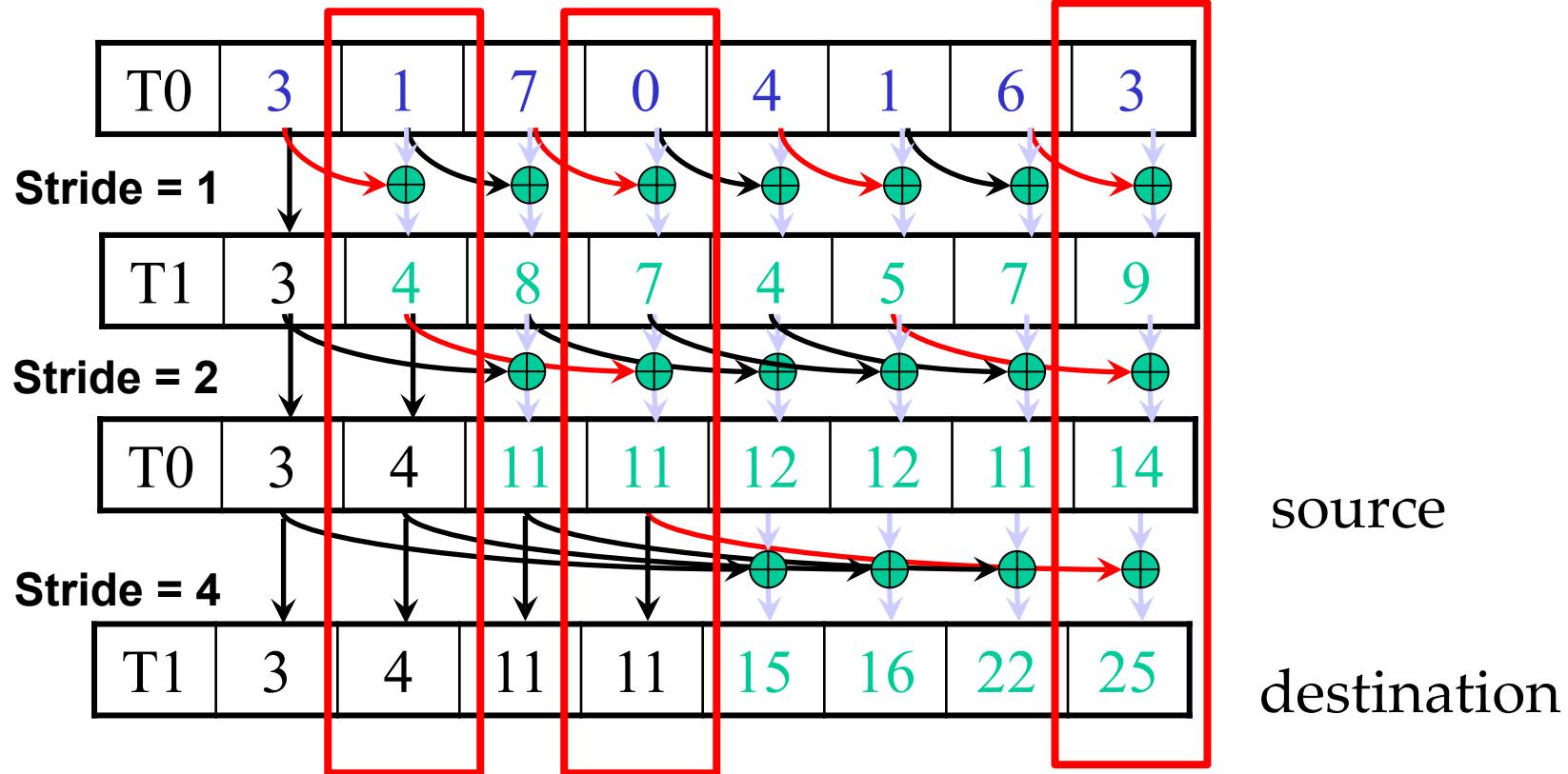
Iteration #1
Stride = 1

- `source = &T0[0]; destination = &T1[0];`
- Each iteration requires only one `syncthreads()`
 - `syncthreads(); // make sure that input is in place`
 - `float destination[j] = source[j] + source[j-stride];`
 - `temp = destination; destination = source; source = temp;`
- After the loop, write destination contents to global memory

A Double-Buffered Kogge-Stone Parallel Scan Algorithm



Sharing Computation in a Double-Buffered Kogge-Stone



Iteration #3
Stride = 4

Work Efficiency Analysis

- A Kogge-Stone scan kernel executes $\log(n)$ parallel iterations
 - The steps do $(n-1), (n-2), (n-4), \dots, (n - n/2)$ add operations each
 - Total # of add operations: $n * \log(n) - (n-1) \rightarrow O(n * \log(n))$ work
- This scan algorithm is not very work efficient
 - Sequential scan algorithm does n adds
 - A factor of $\log(n)$ hurts: 20x for 1,000,000 elements!
 - Typically used within each block, where $n \leq 1,024$
- A parallel algorithm can be slow when execution resources are saturated due to low work efficiency



**ANY MORE QUESTIONS?
READ CHAPTER 8**



ECE408/CS483/CSE408 Spring 2023

Applied Parallel Programming

Lecture 15 Parallel Computation Patterns – Reduction Trees



Course Reminders

- Midterm 1
 - We are grading the coding part of the exam
 - Average for multiple choice questions is 48/70
- Project Milestone 1
 - Baseline CPU implementation is due this Friday
- MP 5.1
 - Is due on 3/24

Objectives

- To learn the basic concept of reductions, one of the most widely used parallel computation patterns
- To learn simple strategies for parallelization of reductions
- To understand the performance issues involved with performing reductions on GPUs

Important Enough to Use in Theory

“... scan operations, also known as prefix computations, can execute in no more time than ... parallel memory references ... greatly simplify the description of many [parallel] algorithms, and are significantly easier to implement than memory references.” —Guy Blelloch, 1989*

*G. Blelloch, “Scans as Primitive Parallel Operations,” IEEE Transactions on Computers, 38(11):1526-1538, 1989.
The idea behind scans for computation goes back another 30+ years.

Trying to Bridge Theory and Practice

A generic parallel algorithm,

- in which parallel threads access memory arbitrarily,
- is likely to produce an extremely slow access pattern.

Scans

- can be implemented quickly in hardware, and
- form a useful alternative to arbitrary memory accesses.

(His hope was to enable theory
without knowledge of microarchitecture.)

Example Use: Summarizing Results

1. Start with a **large set of things** (examples: integers, social networking user information)
2. **Process** each thing **independently to produce** some **value** (examples: number of friends, timeline posts in last two weeks)
3. **Summarize!**
 - Typically, with an associative and commutative operation (+, *, min, max, ...)
 - since things in the set are unordered and independent.

Focus on Reduction Using a Tree

Pattern is so common that

- people have built frameworks around it!
- examples: Google and Hadoop MapReduce

Let's focus on the summarization, called a **reduction**:

- **no required order** for processing the values (operator is associative and commutative), so
- **partition the data set** into smaller chunks,
- have each thread to process a chunk, and
- **use a tree to compute the final answer.**

Reduction Enables Parallelization

Reduction enables common parallel transformations.

example: **privatization of output**

- **Loop iterations sum into a single output**
(examples: inner loops in matrix multiply and convolution).
- To parallelize iterations, must
make private copies of the output!
- **Use reduction** to sum private copies
into the original output.

What Exactly is a Reduction?

Reduce a set of inputs to a single value

- using a binary operator, such as
 - sum, product, minimum, maximum,
- or a user-defined reduction operation
 - must be associative and commutative
 - and have an identity value (example: 0 for sum)

Available in most parallel libraries as **collective operations** (like barriers).

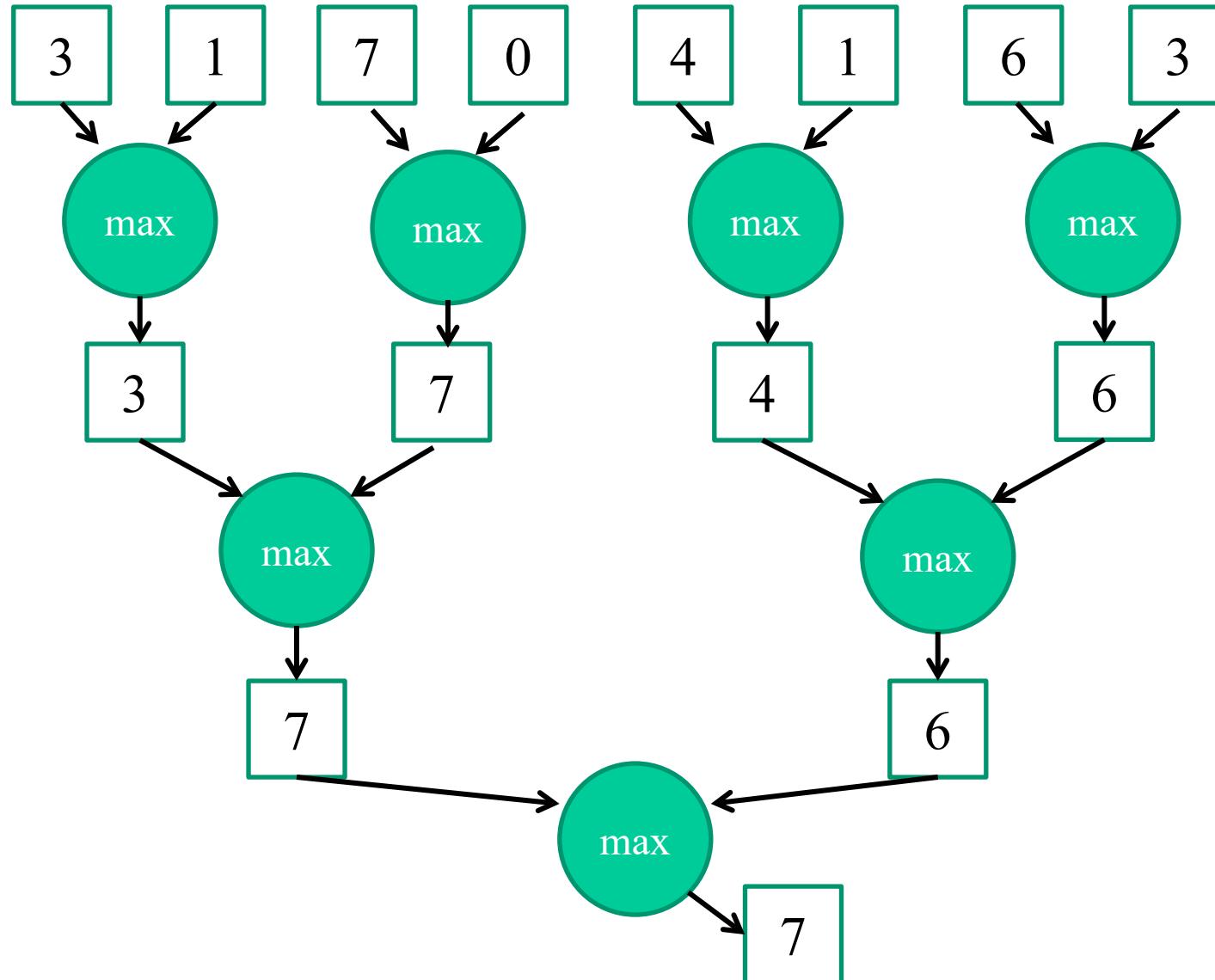
Sequential Reduction is O(N)

Given binary operator \leftrightarrow and an identity value I^\leftrightarrow

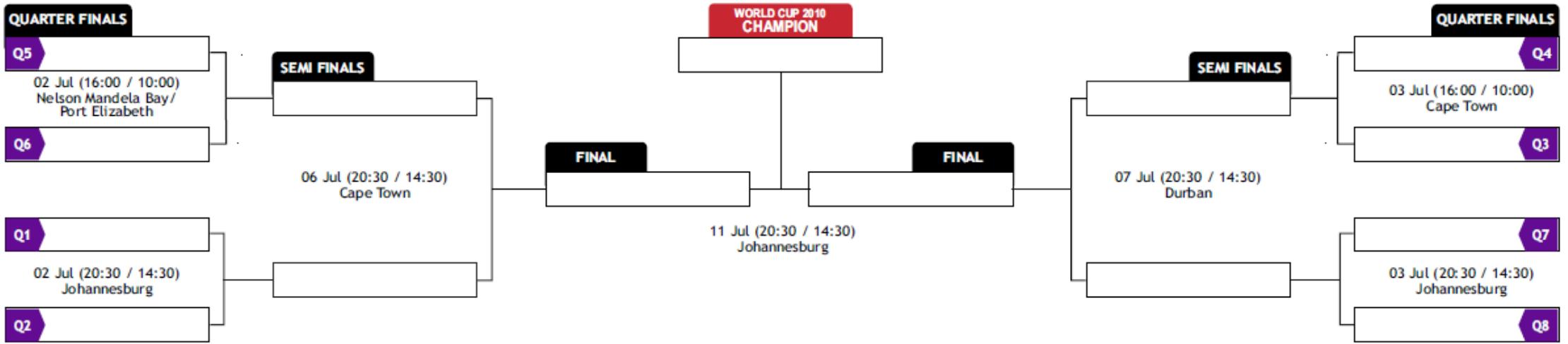
- $I^\leftrightarrow = 0$ for sum
- $I^\leftrightarrow = 1$ for product
- $I^\leftrightarrow = \text{largest possible value}$ for min
- $I^\leftrightarrow = \text{smallest possible value}$ for max

```
result <- I $\leftrightarrow$ 
for each value X in input
    result <- result  $\leftrightarrow$  X
```

Example: Parallel Max Reduction in $\log(N)$ Steps



Tournaments Use Reduction with “max”



(A more artful rendition of the reduction tree.)

Algorithm is Work Efficient

For N input values, the number of operations is

$$\frac{1}{2}N + \frac{1}{4}N + \frac{1}{8}N + \cdots + \frac{1}{N}N = \left(1 - \frac{1}{N}\right)N = N - 1.$$

The parallel algorithm shown is work-efficient:

- requires the same amount of work as a sequential algorithm
- (constant overheads, but nothing dependent on N).

Fast if Enough Resources are Available

For N input values, the number of steps is $\log(N)$.

With enough execution resources,

- $N=1,000,000$ takes 20 steps!
- Sounds great!

How much parallelism do we need?

- On average, $(N-1)/\log(N)$.
50,000 in our example.
- But peak is $N/2$!
500,000 in our example.

Diminishing Parallelism is Common

In our **parallel reduction**,

- the **number of operations**
- **halves in every step.**

This kind of **narrowing parallelism is common**

- from combinational logic circuits
- to basic blocks
- to high-performance applications.

CUDA kernels allow only a fixed number of threads.

Parallel Strategy for CUDA

Let's start simple: N values in device global memory.

Each **thread block** of M threads

- uses shared memory,
- to **reduce chunk of $2M$** values to one value
- ($2M \ll N$ to produce enough thread blocks).

Blocks operate **within shared memory**

- to reduce global memory traffic, and
- **write one value back** to global memory.

CUDA Reduction Algorithm

1. **Read** block of **2M values** into shared memory.
2. For each of **log(2M)** steps,
 - **combine two values** per thread in each step,
 - **write result** to shared memory, and
 - **halve** the number of **active threads**.
3. **Write final result** back to global memory.

A Simple Mapping of Data to Threads

Each **thread**

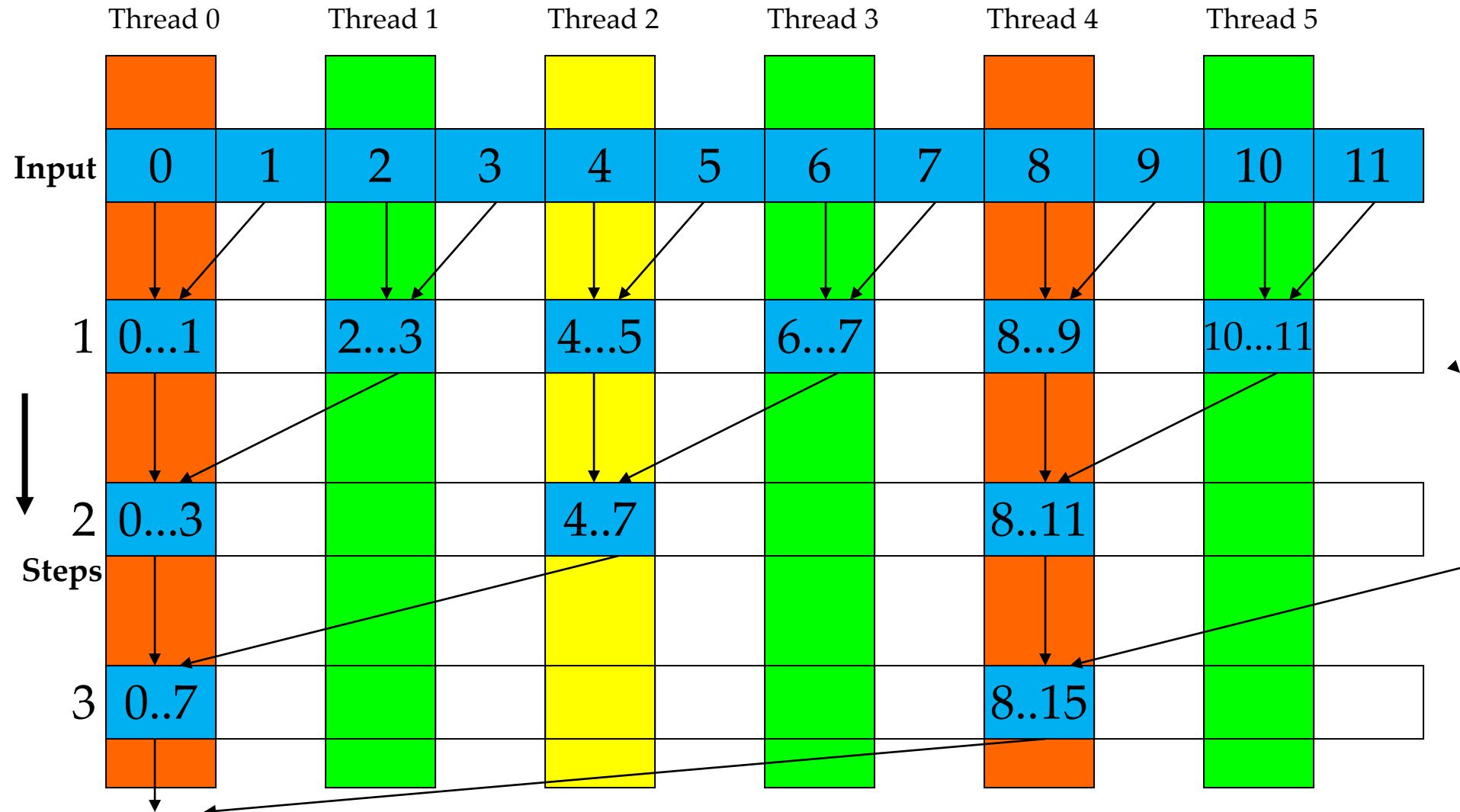
- **begins with two adjacent locations (stride of 1),**
- **even index (first)** and an odd index (second).
- Thread 0 gets 0 and 1, Thread 1 gets 2 and 3, ...
- Write **result** back **to the even index**.

After each step,

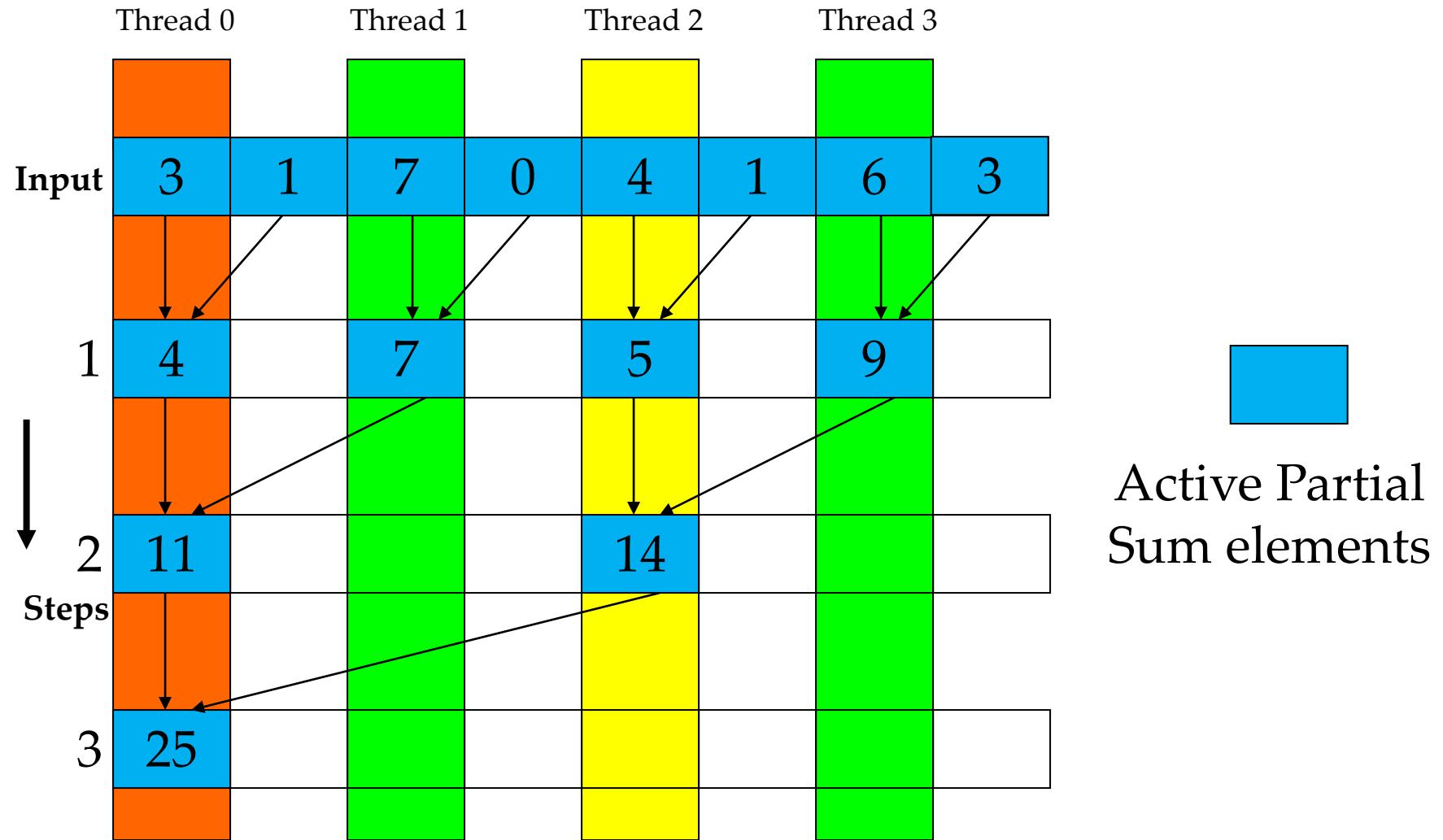
- **half of active threads** are **done**.
- **Double the stride.**

At the end, **result is at index 0**.

Naïve Data Mapping for a Reduction



A Sum Example (Values Instead of Indices)



The Reduction Steps

```
// Stride is distance to the next value being
// accumulated into the threads mapped position
// in the partialSum[] array
for (unsigned int stride = 1;
     stride <= blockDim.x;  stride *= 2)
{
    __syncthreads();
    if (t % stride == 0)
        partialSum[2*t] += partialSum[2*t+stride];
}
```

Why do we need `__syncthreads()`?

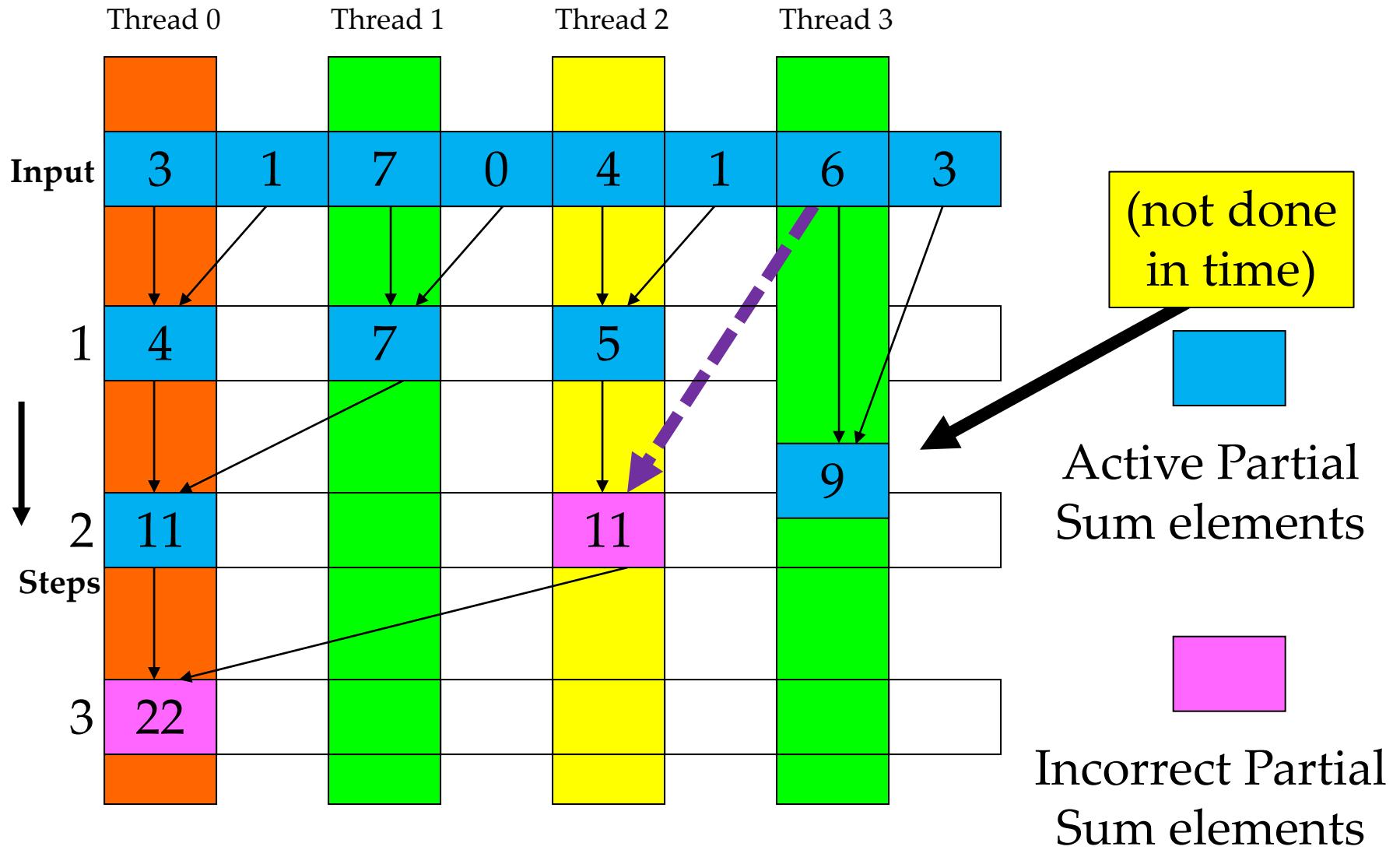
Barrier Synchronization

__syncthreads() ensures

- **all elements** of partial sum **generated**
- **before** the **next step uses them.**

Why do we not need __syncthreads() at the end of the reduction loop?

Example Without __syncthreads



Several Options after Blocks are Done

After all reduction steps, **thread 0**

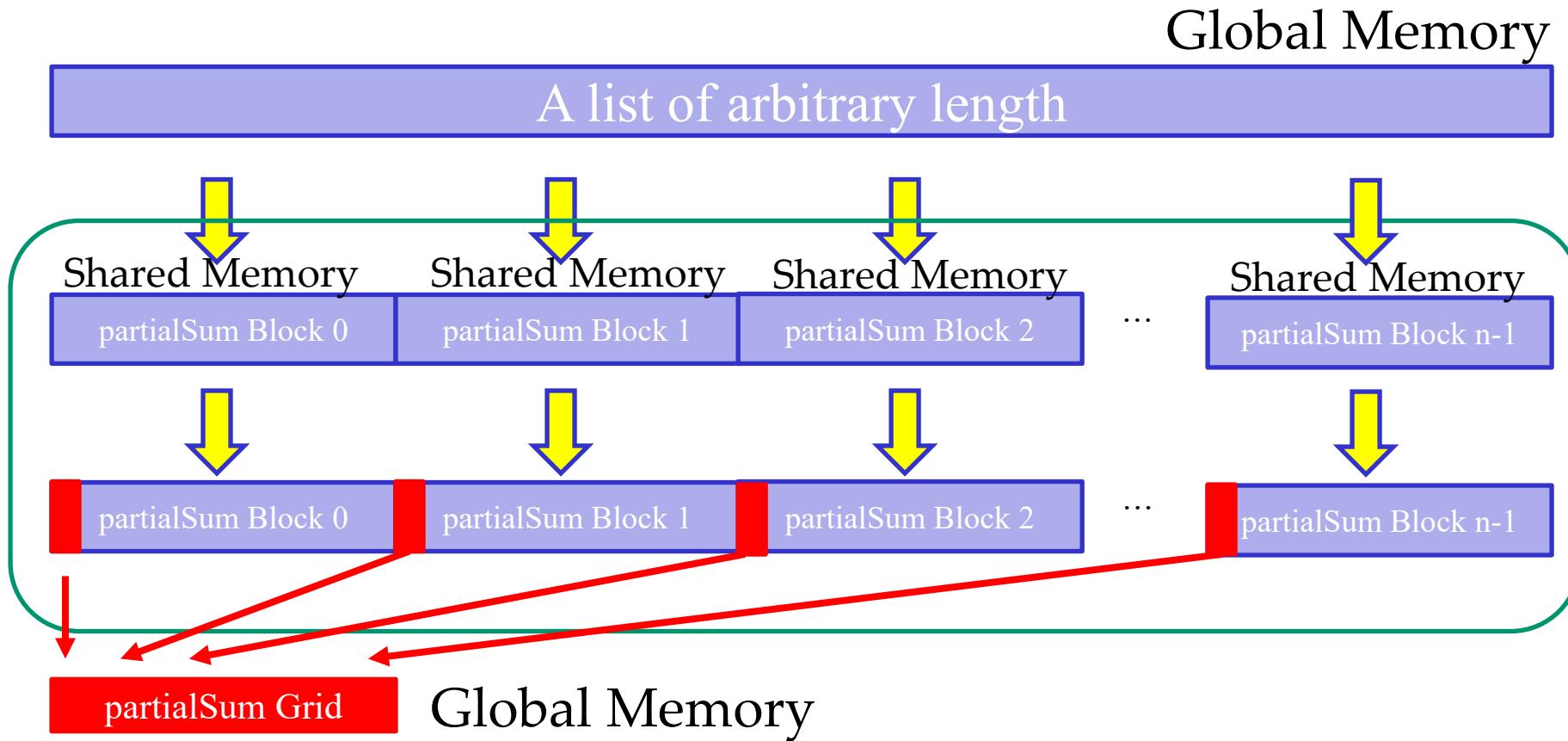
- **writes** block's **sum** from **partialSum[0]**
- **into global vector** indexed by **blockIdx.x**.

Vector has **length $N / 2M$** .

- If small, **transfer** vector **to host** and **sum** it up **on CPU**.
- If large, **launch kernel again** (and again).

(Kernel can also accumulate to a global sum using atomic operations, to be covered soon.)

“Segmented Reduction”



Copy back to host and host to finish the work.

Analysis of Execution Resources

All threads active in the first step.

In all subsequent steps, two control flow paths:

- perform addition, or do nothing.
- Doing nothing still consumes execution resources.

At most half of threads perform addition after first step

- (all threads with odd indices disabled after first step).
- After fifth step, entire warps do nothing:
poor resource utilization, but no divergence.
- Active warps have only one active thread.

Up to five more steps (if limited to 1024 threads).

Improve Performance by Reassigning Data

Can we do better?

Absolutely!

How we assign data to threads
makes a difference in some algorithms,
including reduction.

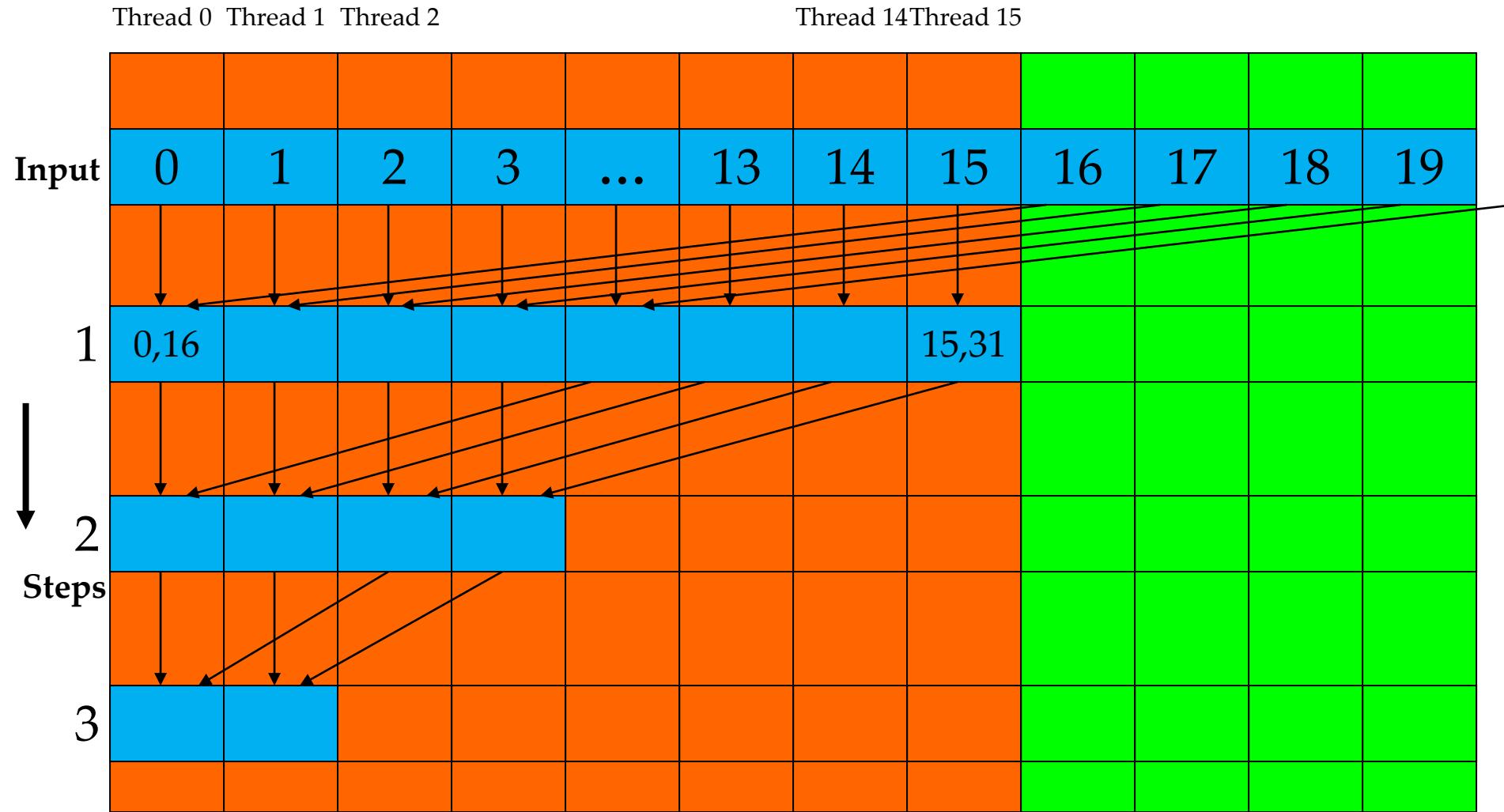
A Better Strategy

Let's try this approach:

- **in each step,**
- **compact** the partial sums
- **into the first locations**
- in the **partialSum** array

Doing so **keeps the active threads consecutive.**

Illustration with 16 Threads



A Better Reduction Kernel

```
for (unsigned int stride = blockDim.x;
     stride >= 1;  stride /= 2)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
```

Again: Analysis of Execution Resources

Given 1024 threads,

- Block loads 2048 elements to shared memory.
- **No branch divergence** in the **first six steps**:
 - 1024, 512, 256, 128, 64, and 32 consecutive threads active;
 - threads in each warp either all active or all inactive
- **Last six steps** have **one active warp** (branch divergence for last five steps).

Parallel Algorithm Overhead

```
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start+ blockDim.x+t];
for (unsigned int stride = blockDim.x;
     stride >= 1;  stride >>= 1)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
```

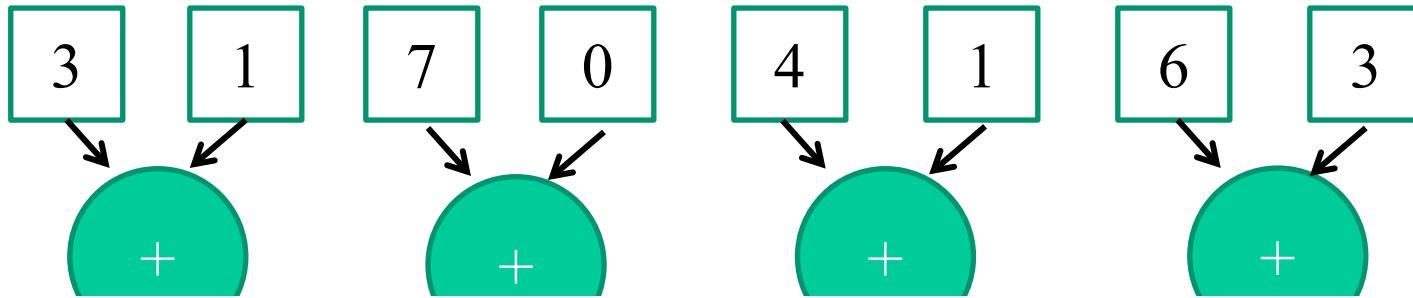
Parallel Algorithm Overhead

```
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start+ blockDim.x+t];

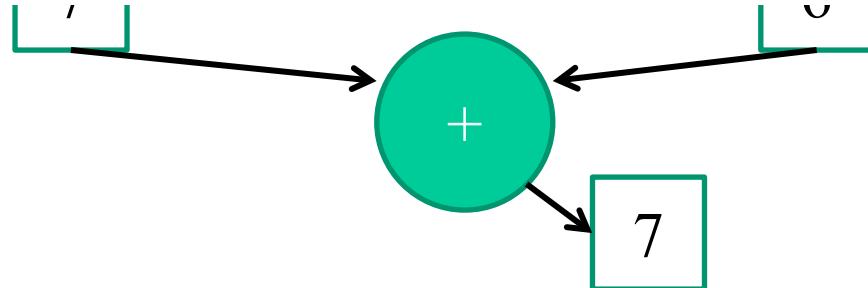
for (unsigned int stride = blockDim.x;
     stride >= 1;  stride >>= 1)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
```

Parallel Execution Overhead



Although the number of “operations” is N , each “operation” involves much more complex address calculation and intermediate result manipulation.

If the parallel code is executed on a single-thread hardware, it would be significantly slower than the code based on the original sequential algorithm.



Further Improvements

Can we further improve reduction?

The problem is **memory-bound**:

- **one operation** for every **4B value read**;
- so **focus on memory coalescing and**
avoiding poor **computational resource use**.

Make Use of Shared Memory

How much shared memory are we using?

Each block of 1,024 threads reads 2,048 values.

- Let's say two blocks per SM,
- so 16 kB ($= 2,048 \times 2 \times 4B$).

Could read 4,096 or 8,192 values

- (with 64 kB per SM)
- to slightly increase parallelism.

(For 48 kB per SM, use 6,144 values and have all threads do a 3-to-1 reduction before the current loop.)

Eliminate the Narrow Parallelism

What about parallelism?

Smaller blocks might seem attractive:

- when one warp is active,
- each SM has one warp per block.

But there are probably better ways. For example,

- **stop reducing at 32 elements** (or at 64, or 128), and
- hand off to the **next kernel**.

Get Rid of the Overhead

Launching kernels is expensive.

- Why bother tearing down and setting up the same blocks on the same SMs?
- Makes no sense.
- Remember that reduction operators are associative and commutative.

Let's **be compute-centric**:

- put **2048 threads** (as two blocks) **on each SM**, and
- just keep them there **until we're done!**

Work Until the Data is Exhausted!

Say there are 8 SMs, so 16 blocks.

1. **Divide** the whole **dataset into 16 chunks**.
2. **Read** enough **to fill shared memory**.
3. **Compute** ... only **until** some **threads not needed**.
4. Then **load more data!**
5. **Repeat** until the data are exhausted,
6. THEN let parallelism drop.

(Gather 16 values on host and reduce them.)

Caveat

I didn't try these ideas.

I'll leave them

- for those of you who feel motivated
- to **try in MP5.1**.

Do

- **save a copy of your simpler solution**, though, as
- you will need the partial sums for scan (MP5.2).



**ANY MORE QUESTIONS?
READ CHAPTER 5**

ECE408/CS483/CSE408 Exam #1, Spring 2018

Tuesday, February 27, 2018

- You are allowed one 8.0x11.5 cheat sheet with notes on both sides. The minimal font size for your text on the cheat sheet should be 8pts.
- No interactions with humans other than course staff are allowed.
- This exam is designed to take 155 minutes to complete. To eliminate time pressure and allow for any unforeseen difficulties, we will give everyone 180 minutes.
- This exam is based on lectures, textbook chapters, as well as lab MPs/projects.
- The questions are randomly selected from the topics we covered up to and including Deep Learning.
- You can write down the reasoning behind your answers for possible partial credit.
- You must write your answers with pen, not pencil, in order to request regrade.
- **Good luck!**

Name: _____

Netid: _____

UIN: _____

Question 1: _____

Question 2: _____

Question 3: _____

Question 4: _____

Question 5: _____

Question 1 (27 points, suggested time allocation 40 minutes): multiple-choice and short-answer questions. If you get more than 27 points by answering all questions (1-8), your score will saturate at 27 points.

For multiple-choice questions, give a concise explanation for your answer for possible partial credit. Answer each of the short-answer questions in as few words as you can. Your answer will be graded based on completeness, correctness, and conciseness.

1. (4 points) If we want to allocate an array of float in the GPU **global memory** and have the pointer variable device_array to point to the array, what is the correct call to cudamalloc() on the host side?

```
// pointers to host & device arrays
float *device_array;
int size = num_bytes * sizeof(float);

(A) cudaMalloc(device_array, size);
(B) cudaMalloc((void *)device_array, size);
(C) cudaMalloc((void *)&device_array, size);
(D) cudaMalloc((void **)&device_array, size);
```

(D) since we need a reference to the pointer itself and CUDA defines the first argument as void** type.

1 point partial credit for (C)

2. (4 points) If we want to copy an array of float from the host memory to the GPU **constant memory** and have the pointer variable device_array to point to the array, what is the correct call on the host side?

```
// pointers to host & device arrays
__constant__ float device_array[100];

float host[100]; // Assume that host array is already filled with data
int size = 100 * sizeof(float);

(A) cudaMemcpy(device_array, host, size, cudaMemcpyHostToDevice);
(B) cudaMemcpyToSymbol(device_array, host, size, 0,
                     cudaMemcpyHostToDevice);
(C) cudaMemcpy(host, device_array, size);
(D) cudaMemcpyToSymbol(host, device_array, size);
```

(B) correct syntax for cudaMemcpyToSymbol. (2 point partial credit if the student mentions cudaMemcpyToSymbol)

3. (4 points) We want to use each thread to calculate four (4) output elements of a vector addition. Each thread block processes four sections. All threads in each block will first process a section first, each processing one element. They will then all move to the next section, with each thread processing one more element. This repeats until all four sections are processed. Assume that variable *i* should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?

- (A) $i = blockIdx.x * blockDim.x * 4 + threadIdx.x;$
- (B) $i = blockIdx.x * threadIdx.x * 2;$
- (C) $i = (blockIdx.x * blockDim.x + threadIdx.x) * 4;$
- (D) $i = blockIdx.x * blockDim.x * 2 + threadIdx.x;$

(A) Explanation: All previous blocks cover $(blockIdx.x * blockDim.x)^4$. The index of the first element processed by each thread can be derived by adding $threadIdx.x$ to it.

4. (4 points) You would like to run a kernel on a GPU device with compute capability 6.0. There are a total of 1,000,000 threads needed to launch the kernel. The kernel uses 10 registers per thread and 10KB shared memory per block. The kernel is launched as a cubic grid of cubic blocks. The grid dimension is 10 in X, Y and Z direction. What is the maximum number of simultaneous blocks that will run on a single SM?

- (A) 1
- (B) 2
- (C) 3
- (D) 6

Technical specifications of CUDA compute capability 6.0 below. Note that you might not need all information from this table.

Maximum number of resident grids per device (Concurrent Kernel Execution)	3
Maximum number of threads per block	1024
Maximum number of resident blocks per multiprocessor	32
Maximum number of resident warps per multiprocessor	64
Maximum number of resident threads per multiprocessor	2048
Number of 32-bit registers per multiprocessor	64K

Maximum amount of shared memory per multiprocessor	64KB
Maximum amount of shared memory per thread block	48KB

(4 points for (B)) $10^6/10^3 = 10^3$ threads per block. Thus maximum of two blocks can run simultaneously on a single SM.

(2 point for (D)) Registers: $10 * 10^3 = 10K$, $64K/10K = 6$, not the limiting factor. Shared memory: $64KB/10KB=6$, not the limiting factor.

5. (4 points) For a tiled 2D convolution kernel with 30x30 output tiles and 3x3 mask (and thus 32x32 input tile), how many warps in each thread block have control divergence? (Assume strategy 2: **block size covers input tile**)

- (A) 3
- (B) 30
- (C) 32
- (D) $30*30$

Answer: (B) each row in a block is a warp. The first 30 rows will have control divergence. The last two warps are completely turned off during the calculation of inner products.

6. (4 points) How do we declare a 5x5 constant memory float array Mc?

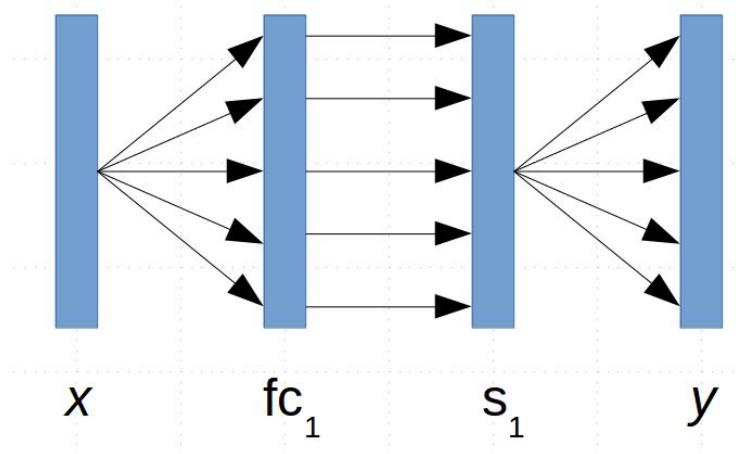
- (A) const float[5][5] Mc; in the kernel that needs to use Mc
- (B) const float Mc[5][5]; outside all kernels
- (C) __const__ float[5][5] Mc; outside all kernels.
- (D) __const__ float Mc[5][5]; outside all kernels.

Answer: (D) keyword is __const__ and must be outside all kernels

7. (4 points) True or false: A single perceptron can learn the XOR function. Explain

False - XOR is not a linearly-separable function. Perceptrons work by finding a hyperplane that splits the input data.

8. (5 points) The following figure represents a feed-forward multi-layer network operating on input x , with a fully-connected layer, a sigmoid layer, and a fully connected layer, to produce an output y . Let w_1 , b_1 and w_2 , b_2 be the weight matrices and bias vectors for the first and second fully-connected layers, respectively. In the figure below, fc_1 is the output of the first fully-connected layer, and s_1 is the output of the sigmoid layer



Recall that for a fully-connected layer, the output vector o is related to the input vector i

$$o = W * i + b$$

And for a sigmoid layer,

$$o = \sigma(i)$$

Note that for the sigmoid layer, $\frac{\partial \sigma(x)}{\partial x} = \sigma(x) * (1 - \sigma(x))$

Using the chain rule, show that if the error gradient at y is $\frac{\partial E}{\partial y}$, then the gradient of the error with respect to b_1 , i.e., $\frac{\partial E}{\partial b_1}$, is $\frac{\partial E}{\partial y} * W_2 * \sigma(fc_1) * (1 - \sigma(fc_1))$

$$\frac{\partial E}{\partial b_1} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial s_1} \frac{\partial s_1}{\partial fc_1} \frac{\partial fc_1}{\partial b_1} = \frac{\partial E}{\partial y} * W_2 * \sigma(fc_1) * (1 - \sigma(fc_1)) * 1$$

Question 2 (15 points, suggested time allocation 20 minutes): CUDA Basics.

For the color image to grey image conversion kernel and the corresponding kernel launch code, answer each of the sub-questions below.

Kernel Code:

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
01. __global__
02. void colorToGreyscaleConversion(unsigned char *grayImage, unsigned char
*rgbImage, int width, int height) {
03.     int Col = threadIdx.x + blockIdx.x * blockDim.x;
04.     int Row = threadIdx.y + blockIdx.y * blockDim.y;
05.     if (Col < width && Row < height) {
06.         // get 1D coordinate for the grayscale image
07.         int greyOffset = Row*width + Col;
08.         // one can think of the RGB image having
09.         // CHANNEL times columns of the gray scale image
10.         int rgbOffset = greyOffset*CHANNELS;
11.         unsigned char r = rgbImage[rgbOffset]; // red value for pixel
12.         unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
13.         unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pixel
14.         // perform the rescaling and store it
15.         // We multiply by floating point constants
16.         grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
17.     }
18. }
```

Host Code:

```
// the following host code is buggy
// you need to fix the bugs in question 2(a)
01. dim3 dimGrid(ceil(height/64), ceil(width/32), 1);
02. dim3 dimBlock(64, 32, 1);
03. colorToGreyscaleConversion<<<dimGrid, dimBlock>>>(d_grayImage, d_rgbImage,
width, height);
```

- 2(a). (6 points) Assume the above kernel code is correctly implemented. Based on the kernel code, there are three bugs in the host code which launches the kernel. Please clearly state where the bugs are and state how to fix these bugs.

Bug 1: **64*32 > 1024** which excesses the max available threads in a block

Fix 1: **change 64->32 or other changes make generated threads less than 1024**

Bug 2: **ceil(height/64)** is problematic when height/64 has a fractional part, as the fractional part of integer division in C is automatically truncated

Fix 2: **ceil(height/64.0) or (height-1)/64 + 1 or other correct answers**

Bug 3: the x dimension matches the column or width in the device code, while the host code assigns height into the x dimension

Fix 3: swap height and width in the host code

2(b). (3 points) Assume that the height of the grey image is 600 pixels and the width is 800 pixels. Instead of the settings in 2(a), assume that we decide to use a grid of 16X16 blocks. That is, each block is organized as a 2D 16X16 array of threads. How many warps will be generated during the execution of the kernel? **Please show your work.** (Hint: Draw a picture on how pixels are covered by thread blocks.)

$$38 \times 50 \times 8 = 15200$$

Each thread block contains $16 \times 16 / 32 = 8$ warps

On vertical axis, there are in total $\text{ceil}(600/16) = 38$ blocks

On horizontal axis, there are in total $\text{ceil}(800/16) = 50$ blocks

$$\text{Total warps} = 38 \times 50 \times 8 = 15200$$

2(c). (3 points) Based on 2(b), how many warps will have control divergence? **Please show your work.**

0

As warps are assigned in row-major order, and the number of thread blocks in x dimension is an integer value with no fractional part, so there is no divergence in the x dimension.

On the bottom, the threads in half of the warps are in use while all those in others are not. There is no divergence here.

2(d). (3 points) Based on 2(b), if we now have a grey image whose height is 800 pixels and the width is 600 pixels, how many warps will have control divergence? (Assume that we decide to use a grid of 16X16 blocks. That is, each block is organized as a 2D 16X16 array of threads.) **Please show your work.**

$$50 \times 8 = 400$$

As the warps are assigned in row-major order, all warps in last block of every row have divergence.

The number of $\text{ceil}(800/16) \times 8 = 400$ warps.

Question 3. (14 points, suggested time allocation 20 minutes): Deep Learning.

You are implementing a fully-connected neural-network layer. It takes an input vector \mathbf{x} , multiplies it by a weight matrix \mathbf{w} , adds a bias vector \mathbf{b} , and produces an output vector \mathbf{y} . $\mathbf{w}[i,j]$ corresponds to how the i-th element of the input affects the j-th element of the output.

Your layer will be used as the first fully-connected layer in a network that reads in 28x28 input image data treated as a linearized input vector and produces an output vector of 500 values.

- 3(a). (2 points) Recall our formulation for a fully-connected layer as a matrix-vector multiplication:

$$\mathbf{y} = \mathbf{w} * \mathbf{x} + \mathbf{b}$$

Fill in the following table with the vector and matrix dimensions

Data	Dimensions
x	784
y	500
w	[784, 500]
b	500

- 3(b). (4 points) You have two different implementations of the layer. In the first, the weight matrix data is stored in row-major order, and in the second, in column-major order. The figure below shows the thread access pattern for each storage type in a 4x3 example.

Row-Major Layout	Column-major Layout
<p>$y = w * x$</p>	<p>$y = w * x$</p>

Fill in the blanks in the following implementation of the fully-connected layer for row-major weight matrix.

```
1. __global__ void fc_row(float *y, const float *x,
2.                         const float *w, const float *b,
3.                         const int ySize, const int xSize) {
4.
5.     const int tx = blockDim.x * blockIdx.x + threadIdx.x;
6.     const int gx = gridDim.x * blockDim.x;
7.
8.     for (int o = tx; o < ySize; o += gx) {
9.         float acc = 0;
10.        for (int i = 0; i < xSize; ++i) {
11.            acc += x[ ] * w[ ];
12.        }
13.        y[ ] = acc + b[ ];
14.    }
15. }
```

- 11: i
 - 11: $i * ySize + o$
 - 13: o
 - 13: o

3(c). (4 points) Fill in the blanks in the following implementation of the fully-connected layer for col-major weight matrix.

```

3.                     const int ySize, const int xSize) {
4.
5.     const int tx = blockDim.x * blockIdx.x + threadIdx.x;
6.     const int gx = gridDim.x * blockDim.x;
7.
8.     for (int o = tx; o < ySize; o += gx) {
9.         float acc = 0;
10.        for (int i = 0; i < xSize; ++i) {
11.            acc += x[ ] * w[ ];
12.        }
13.        y[ ] = acc + b[ ];
14.    }
15. }
```

- 11: i
- 11: o * xSize + i
- 13: o
- 13: o

3(d). (2 point) How many threads can simultaneously contribute to y in this kernel? You may state your answer as an algebraic expression in terms of the variables in the code.

- **gx**

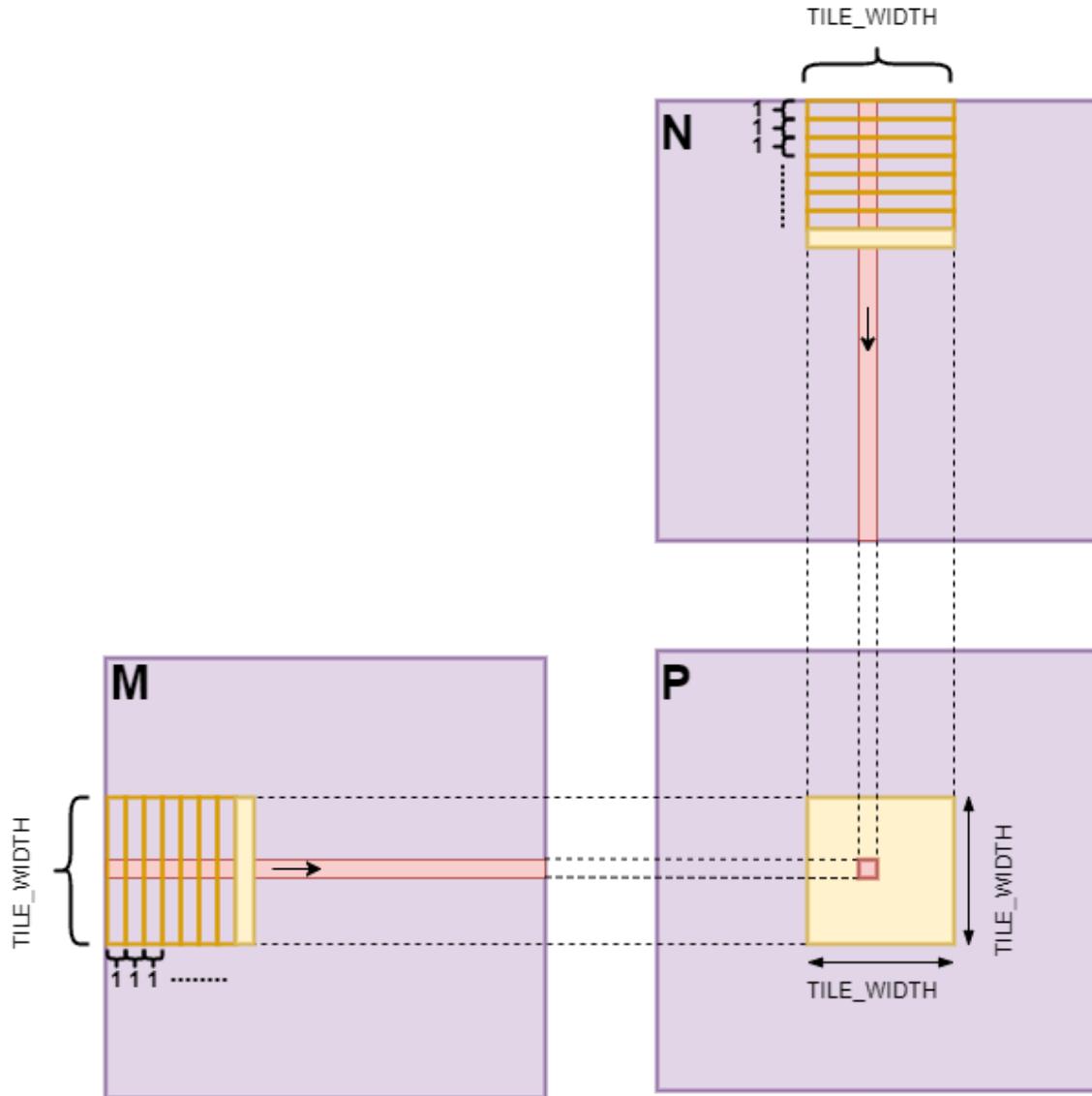
3(e). (2 points) You observe that the performance of the row-major weight kernel is substantially higher. Why?

- **Matrix accesses are coalesced**

Question 4. (24 points, suggested time allocation 35 minutes): Matrix Multiplication.

You need to do a matrix multiplication between a **632 x 632 matrix M** and a **632 x 632 matrix N**. You write a tiled matrix multiplication kernel you learned in class with a **8 x 8 square tile**. Your friend suggests you to instead use a **1 x 16 strip input tile** with **16x16 output tile** (see picture below) and gives you a partially finished code. You decided to implement the rest of code and compare the two.

The strip tiled matrix multiplication is visualized as below



```

1. #define TILE_WIDTH 16
2. __global__ void strip_sgemm(float *M, float *N, float *P, Width) {
3.     // Assume that TILE_WIDTH is set to 16
4.     __shared__ float Mds[TILE_WIDTH];
5.     __shared__ float Nds[TILE_WIDTH];
6.
7.     int bx = blockIdx.x;  int by = blockIdx.y;
8.     int tx = threadIdx.x; int ty = threadIdx.y;
9.
10.    // Identify the row and column of the P element to work on
11.    int Row = by * TILE_WIDTH + ty;
12.    int Col = bx * TILE_WIDTH + tx;
13.
14.    float Pvalue = 0;
15.    // Loop over the M and N tiles required to compute P element
16.    for (int m = 0; m < Width; ++m) {
17.        // use the first 16 threads in each block to load Nds and Mds
18.        if(threadIdx.y== 0) {
19.            if(____ < Width)
20.                Mds[____] = M[____];
21.            else
22.                Mds[____] = 0;
23.            if(____ < Width)
24.                Nds[____] = N[____];
25.            else
26.                Nds[____] = 0;
27.        }
28.
29.        __syncthreads();
30.
31.        Pvalue += Mds[____] * Nds[____];
32.        __syncthreads();
33.    }
34.
35.    if(Row < Width && Col < Width)
36.        P[Row*Width + Col] = Pvalue;
37. }

```

```

1. /* strip_sgemm is launched with the following parameters
2. dim3 gridDim(ceil(632/(float)TILE_WIDTH),ceil(632/(float)TILE_WIDTH);
3. dim3 blockDim(TILE_WIDTH, TILE_WIDTH);
4. */

```

- 4(a). (8 points) Fill in the blank to make this code run correctly. If you think it is impossible to fill in any of the blanks to make this code to run correctly, state why. (Hint: The thread index

to data index mapping for the output is still the same as we used in MP3. Use the provided picture to identify the expressions for the thread index to data index mapping for loading and using M and N tiles.)

```
19. if( by * TILE_WIDTH + tx < Width) + 1
20. Mds[tx] = M[(by * TILE_WIDTH + tx) * Width + m]; + 2
22. Mds[tx] = 0; + .5
23. if( bx * TILE_WIDTH + tx < Width) + 1
24. Nds[tx] = N [bx * TILE_WIDTH + tx + Width * m]; + 2
26. Nds[tx] = 0; + .5
31. Pvalue += Mds[ty] * Nds[tx]; + 1
```

4(b). (4 points) For the strip tile matrix multiplication kernel, which of the input matrices have/has a coalesced memory access pattern? Please explain. (Hint: use the provided picture to analyze the memory access patterns by adjacent threads when loading M and N elements.)

- (A) M
- (B) N
- (C) Both
- (D) None

Answer:

Explanation:

(B) For M, the strip elements accessed by the adjacent threads (in the x dimension) belong to the same column and are not adjacent to each other in the layout. Thus M accesses are not coalesced. For N, the strip elements accessed by the adjacent threads (in the x dimension)) belong to the same row and are adjacent to each other. Thus N accesses are coalesced.

4(c). (3 points) If the computation is limited by global memory bandwidth and the global memory bandwidth is 150GB/s bandwidth, what are the highest achievable FLOPS for the two kernel? Assume 32-bit floating point (single precision) for both kernels.(Hint: use the provided picture to analyze the number of reuses for M and N elements in the 1x16 tiling design.)

8 x 8 square tile: $(150/4) * 8 = 300$ GFlops

1x 16 strip tile: $(150/4) * 16 = 600$ GFlops

For 8x8 tiling, each input element is reused 8 times.

For 1x16 tiling, each input element is reused 16 times.

4(d). (6 points) Calculate the number of warps that will have control divergence for the two code during kernel execution for a 632x632 P matrix. Count a warp as having divergence if there is any control divergence at any of the statements during the execution of the kernel. Also, count a warp as one diverging warp even if there is control divergence at multiple statements for the warp. (Hint: use the provided picture to analyze the divergence patterns when loading M and N tiles.)

8 x 8 square tile: 0

1 x 16 strip tile: $(40 * 40) + (39 * 7 + 3) = 1876$

For 8x8 tiling, the number of pixels are multiples of the block width and height in both x and y dimensions. All threads will be active so there is no divergence.

For 1x16 tiling, there are four cases.

- (1) For every block, the first warp will always have control divergence - $40*40*1$ in this case.
- (2) In addition. The blocks that calculate the right edge (except for the lower right corner) of the output P matrix have half of their threads active. There are 39 of them and each has 7 additional diverging warps in each of these blocks. All of these warps will have control divergence.
 $39*7$ warps in this case
- (3) The blocks that calculate the bottom edge (except for the lower right corner) of the output P matrix have the last 4 warps inactive. There are no additional diverging warps.
- (4) The block that calculates the lower right corner of the output P matrix has both active and inactive threads in all its warps. There are 3 additional diverging warps in this case.
- (5) The rest of the blocks have all their threads active.

There are $40*40 + 39*7 + 3 = 1876$ warps

In the second case, the blocks that process the bottom edge of the matrices have

memory

4(e). (3 points) If the SM (streaming multiprocessor) can take up to 1,536 threads and up to 12 thread blocks and have 5,120 Bytes of shared memory, How many pending global memory reads to M and N can the two code have?

8 x 8 square tile: $10 * 64 * 2 = 1280$

1 x 16 strip tile: $6 * 16 * 2 = 192$

For 8x8 tiles, M and N tiles require $8*8*4*2 = 512$ bytes so 10 thread blocks can go into the SM
So there will be $10*64*2$ pending loads.

For 1x16 tiles, M and N tiles require $1*16*4*2 = 128$ bytes so shared memory will not be a limitation. Each block has 256 threads so we can have up to $1536/256 = 6$ blocks. So there will be $6*16*2$ pending loads

Question 5. (20 points, suggested allocation of time 40 minutes): Convolution.

Recall tiling strategy 1 that we introduced in lecture where the threads were mapped to the output tile and the entire input tile was loaded into shared memory in multiple stages. In the strategy it was presented that for the 1D case you could load the input tile in three stages as presented by the left diagram below, one for the core elements and then one for each side with halo elements. This strategy inefficiently utilizes the number of threads we have and could actually implemented in two stages like in the right diagram.



- 5(a). (2 points) Now consider the 2D case and determine the **minimum number of stages** it would require to load an **input** tile with the dimensions 16x16 and a mask with dimensions 7x7. Make sure to explain your reasoning. Feel free to draw a diagram as part of your explanation.

Answer: Output tile must be $(16-6)^2 = 10 \times 10$. (**Input / Output tile sizes**) = $256 / 100 = 2.56$, rounds up to 3. Can be done in 3 stages.

- 5(b). (12 points) Write out the indexing and declarations needed to load the input tile into shared memory below for all three sections. You may assume **x_size** and **y_size** are known variables that give the number of pixels in the horizontal and vertical dimensions when declaring block and grid dimensions.

Defines:

```
1. #define MASK_SIZE 49
2. #define MASK_WIDTH 7
3. #define OUT_TILE_WIDTH 10
4. #define IN_TILE_WIDTH 16
5. __constant__ float kernel[MASK_WIDTH][MASK_WIDTH];
```

Define block and grid dimensions here:

```
1. // Somewhere in the host code
2. dim3 DimBlock(OUT_TILE_WIDTH, OUT_TILE_WIDTH, 1);
3. dim3 DimGrid(ceil(x_size * 1.0 / OUT_TILE_WIDTH), ceil(y_size *
1.0 / OUT_TILE_WIDTH), 1);
```

Device Code:

```
1. __global__ void conv3d(float *input, float *output, const int
y_size, const int x_size) {
```

```

3. int tx = threadIdx.x, ty = threadIdx.y;
4. int bx = blockIdx.x, by = blockIdx.y;
5. // x_i and y_i are the x and y indices of the upper upper-left
6. // corner of the input tile
7. int x_i = bx * OUT_TILE_WIDTH - MASK_WIDTH/2;
8. int y_i = by * OUT_TILE_WIDTH - MASK_WIDTH/2;
9. __shared__ float inputTile[IN_TILE_WIDTH][IN_TILE_WIDTH];
11. for(int i = 0; i < 3; i++){ // +1 for matching answer from a
12.     // Below are helper variables
15.     // idx is the linearized index of an element in the input tile
16.     int idx = ((i * by + ty) * bx + tx); // Must have some
dependence on i, ty, and tx to be considered valid.
17.     // xidx and yidx are the x and y positions of the corresponding
18.     // element in the tile
19.     int yidx = idx / IN_TILE_WIDTH; // yidx and xidx must each have
20.     int xidx = idx % IN_TILE_WIDTH; // a distinct calculation
21.     // xpos and ypos are the x and y positions of the corresponding
22.     // element in the input
23.     int ypos = y_i + yidx; // need to bring y_i and x_i to correctly
24.     int xpos = x_i + xidx; // calculate the element in input
25.     if(idx < IN_TILE_WIDTH * IN_TILE_WIDTH){
26.         if (xpos >= 0 && xpos < x_size && ypos >= 0 && ypos < y_size)
// +1 for >=0 checks and +1 for size checks
27.             inputTile[yidx][xidx] = input [(ypos * x_size) + xpos];
28.         else
29.             inputTile[yidx][xidx] = 0.0;
30.     }
31. }
32. ...

```

5(c). (3 points) Does this strategy have any advantage over strategy 2 with regards to how the hardware is utilized? Recall that strategy 2 is where the threads were mapped to the input tile. Explain why or why not. Think about the kind of work each thread would do in either case and any hardware limitations. You may assume CUDA compute capability 3.0.

Short Answer: In strategy 1 has the advantage that all threads contribute to computation, while in strategy 2 many threads are turned off during computation.

This applies in general between strategy 1 and 2 since we have only changed how strategy 1 loads, not the overall algorithm.

Long Answer: When you instantiate the same number of threads in both strategies, only strategy 1 has all of the threads contributing towards calculations. In fact, you can always get more threads computing in strategy 1 than in strategy 2 due to the limits of the number of threads per block.

We also get some better global memory access to computation ratios (shared memory reuse essentially). Ideally we would want 1 global memory access for each computation. The worst case is if the ratio equals the mask size. Say we compute a 8x8x8 output in strategy 2 with a 3x3x3 mask. Then we have a 1000/512 ratio for memory access to compute = 1.95 global memory accesses per computation. If we consider strategy 1 it will have this same ratio on an output of the same size, but since strategy 1 uses less threads we can actually compute on larger output tiles and still be under the 1024 thread block limit, while strategy 2 cannot operate on larger tiles. Thus if we compare the 10x10x10 output strategy 1 can compute, we have a $1728/1000 = 1.728$ ratio for memory accesses per computation.

This helps us see the trade off. We have longer latency loading stage in strategy 1, but we can always get better shared memory reusage.

Also accepting the answers of control divergence and more blocks per SM because fewer threads to compute the same output.

5(d). (3 points) Assuming the same grid and block dimensions and that the input we perform convolution on is perfectly tiled (one of the internal tiles), how many warps will experience control divergence for an internal tile using this modified strategy? You may assume that if the number of threads don't divide evenly by the warp size, the last warp is underutilized but does **not** experience control divergence because of this. Explain your answer.

Answer: 1, the only check that will diverge with the above assumptions is `if(idx < inWidth * inWidth)`. Based on our math from part a, we have input tiles of size 256 and 100 threads. When we are on the third part of loading the input tile, we have 56 elements left among 100 threads, that means that the second warp will have $56 - 32 = 24$ threads loading elements and 8 threads not loading elements. This means warp 1 diverges (starting numbering from warp 0). All threads in the warp that precedes perform loads and the threads in the warps that follow do not load so they do not experience control divergence.

Partial credit awarded for answers that are correct for the 4 stage loading strategy up to 2 points. This method has 6 warps that diverge.

ECE408/CS483/CSE408 Exam #1, Fall 2017

Tuesday, October 17, 2017

- You are allowed one 8.0x11.5 cheat sheet with notes on both sides. The minimal font size for your text on the cheat sheet should be 8pts.
- No interactions with humans other than course staff are allowed.
- This exam is designed to take 170 minutes to complete. To eliminate time pressure and allow for any unforeseen difficulties, we will give everyone 180 minutes.
- This exam is based on lectures, textbook chapters, as well as lab MPs/projects.
- The questions are randomly selected from the topics we covered up to and including parallel scan.
- You can write down the reasoning behind your answers for possible partial credit.
- **Good luck!**

Question 1 (30 points, suggested time allocation 40 minutes): multiple-choice and short-answer questions. If you get more than 30 points by answering all questions (1-9), your score will saturate at 30 points. The bonus question is extra.

For multiple-choice questions, give a concise explanation for your answer for possible partial credit. Answer each of the short-answer questions in as few words as you can. Your answer will be graded based on completeness, correctness, and conciseness.

1. (4 points) We want to use each thread to calculate four (4) output elements of a vector addition. Each block processes $4 * \text{blockDim.x}$ consecutive elements that form 4 sections. All threads in each block will first process a section with each thread processing one element. They will then all move to the next section with each thread processing one element. For each section, consecutive threads should process consecutive elements. What would be the kernel code expression for forming the value of i , the data index of the first element to be processed by each thread?

- (A) $i = \text{blockIdx.x} * \text{blockDim.x} * 4;$
- (B) $i = \text{blockIdx.x} * \text{threadIdx.x} + \text{threadIdx.x}$
- (C) $i = \text{blockIdx.x} * \text{blockDim.x} * 4 + \text{threadIdx.x}$
- (D) $i = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x} * 4$

Answer: (C)

Explanation: All preceding blocks cover $(\text{blockIdx.x} * \text{blockDim.x}) * 4$. The beginning element is consecutive in this case so just add threadIdx.x to it.

2. (4 points) For vector addition, assume that the vector length is 8,000, each thread calculates four (4) output elements, and the thread block size is 1,024 threads. The programmer configures the kernel launch to have a minimal number of blocks to cover all output elements. How many threads will be in the grid?

- (A) 1,024
- (B) 8,196
- (C) 2,048
- (D) 8,000

Answer: (C)

Explanation: $\text{ceil}(8000/\text{float}(4 * 1024)) * 1024 = 2 * 1024 = 2048$. Another way to look at it is that each block covers 4096 elements. The minimal multiple of 4096 to cover 8000 is 2 (blocks). So $2 * 1024$ is 2048.

3. (4 points) We are to process an 565x784 ($m=565$ pixels in the y or vertical dimension, $n=784$ pixels in the x or horizontal dimension) picture with the **PictureKernel** below:

```
__global__ void PictureKernel(float* Pin, float* Pout, int m, int n) {  
  
    // Calculate the row # of the d_Pin and d_Pout element to process  
    int Row = blockIdx.y*blockDim.y + threadIdx.y;  
  
    // Calculate the column # of the d_Pin and d_Pout element to process  
    int Col = blockIdx.x*blockDim.x + threadIdx.x;  
  
    // each thread computes one element of d_Pout if in range  
    if ((Row < m) && (Col < n)) {  
        Pout[Row*n+Col] = 2*Pin[Row*n+Col];  
    }  
}
```

Assume that each block is organized as a 2D 16x32 array of threads (16 in the y dimension and 32 in the x dimension). Which of the following statements sets up the kernel configuration properly? Assume that int variable n has value 784 and int variable m has value 565. The kernel is launched with the statement `PictureKernel<<<gridDim, blockDim>>>(d_Pin, d_Pout, m, n);`

- (A) `dim3 gridDim(ceil(1, ceil(n/32), ceil(m/16); dim3 blockDim(1, 32 16);`
- (B) `dim3 gridDim(ceil(n/32.0), ceil(m/16.0), 1); dim3 blockDim(32, 16, 1);`
- (C) `dim3 gridDim(ceil(m/32.0), ceil(n/16.0), 1); dim3 blockDim(32, 16,1);`
- (D) `dim3 gridDim(ceil(m/16), ceil(n/32), 1); dim3 blockDim(16, 32, 1);`

Answer: (B)

Explanation: dim3 format is (x, y, z). Since n is the size of the picture in the x direction and m is the size of the y direction, we should use n to set up the x dimension and m to set up the y dimension.

4. (4 points) In Question 3, how many warps will have control divergence?

- (A) 0
- (B) $25*32 + 35$
- (C) 565
- (D) 576

Answer: (C)

Explanation: The size of the picture in the x dimension is not a multiple of 32. That means each warp in the 565 rows of the grid will have divergence. All threads in the last (bottom) 11 warps in the last (bottom) block at the right edge will all be inactive and thus no control divergence.

5. (4 points) Write some CUDA kernel code in which the threads in the same block transpose the elements of a matrix **mat** in shared memory in-place. That is, all threads should see that **mat[i][j]** is swapped with element **mat[j][i]** after the transposition. You can assume that that each block has dimensions dim3(16,16,1). Make your code as efficient in time and space as possible.

```
__shared__ float mat[16][16];
```

Answer:

```
float temp = mat[threadIdx.y][threadIdx.x];
__syncthreads();
mat[threadIdx.x][threadIdx.y] = temp;
__syncthreads();
// 2 points per (sync, swap)
```

6. (4 points) Your kernel runs on a device where each Streaming Multiprocessor has 32K registers and 64KB of shared memory. The kernel code uses 30 registers (local variables) and 5KB of shared memory. The kernel is launched with 1,920,000 threads in total organized into a square grid with dimension 50 on each side. What is the maximum number of simultaneous blocks that will run on a single SM?

Answer:

Each block has 768 threads, and will use $768 \times 30 = 23,040$ registers.
Each SM has 32K registers. There can only be 1 block at a time.
// calculate registers or calculate number of threads (2) correct conclusion (2)

7. (4 points) What are the possible values of *dst after this kernel execution?

```
__global__ void race_me(char *dst) {
    dst[0] = threadIdx.x;
}

// host code
cudaMalloc(&dst, 1);
cudaMemset(dst, 3); // assign value 3 to dst
race_me<<<1,2>>>(dst);
```

- (A) 0
- (B) 1
- (C) 0, 1
- (D) 0, 1, 3
- (E) 3

Answer: (C) There is one block with two threads whose threadIdx.x values are 0 and 1. Either value is possible since the code is not properly synchronized.

// (3) for D, (2) for explanation mentioning thread parallelism or lack of sync even with wrong answer

8. (4 points) Which of the following are design philosophies in GPU architecture?

- (1) GPUs use massive parallelism to hide stalls
 - (2) GPUs have many low-latency execution units
 - (3) GPUs spread the cost of managing an instruction stream across many ALUs
-
- (A) 1 and 2
 - (B) 1 and 3
 - (C) 2 and 3
 - (D) all of the above

Answer: (B) GPU execution units are not designed to have low latency.

9. (4 points) Your friend has profiled a kernel and discovered that the performance is limited by the latency of the divide operator. It turns out variable d in his kernel code (below) is 1 half the time, so your friend proposed the following optimization:

```
if (1 == d) r = d; else r = 1 / d;
```

Unfortunately, the performance did not improve. In one sentence, explain why.

Answer: The code is executed by all threads and some of the threads will still succeed so now the code has control divergence.

// (3) for correct answer, (4) for control divergence

(bonus 3 points) List the errors and typos that you reported via Piazza postings, e-mails, or in person communication with Prof. Hwu.

Question 2 (15 points, suggested time allocation 20 minutes): CUDA Basics. For the vector addition kernel and the corresponding kernel launch code, answer each of the sub-questions below.

```
01. __global__
02. void vecAddKernel(float* A, float* B, float* C, int n) {
03.     int i = threadIdx.x + 2*blockDim.x * blockIdx.x;
04.     if(i<n) C_d[i] = A_d[i] + B_d[i];
05.     i += blockDim.x;
06.     if(i<n) C_d[i] = A_d[i] + B_d[i];
07. }

08. int vectAdd(float* A, float* B, float* C, int n) {
    // assume that size has been set to the actual length of
    // arrays A, B, and C
09.     int size = n * sizeof(float);

10.     cudaMalloc((void **) &A_d, size);
11.     cudaMalloc((void **) &B_d, size);
12.     cudaMalloc((void **) &C_d, size);
13.     cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
14.     cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
15.     vecAddKernel<<<ceil(n/(2*1024.0)), 1024>>>(A_d, B_d, C_d, n);
16.     cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
17. }
```

2(a). (2 point) Assume that the size of A, B, and C is 20,000 elements each. How many thread blocks will be generated?

Answer: 10, or correct expression

2(b). (2 point) Assume that the size of A, B, and C is 20,000 elements each. How many warps are there in each block?

Answer: 32, or correct expression

2(c) (2 point) Assume that the size of A, B, and C is 20,000 elements. How many threads will be created in the grid?

Answer: 10,240, or correct expression
// (-1) for carrying an error from (a)

2(d) (4 points) Assume that the size of A, B, and C is 20,000 elements each. Is there any control divergence during the execution of the kernel? If so, identify the block index and warp index that causes the control divergence. Explain why or why not.

Answer: No, there is no control divergence. The last block covers 1568 elements. The first 1024 elements are covered by the first if/add statement. The 544 needs to be covered by the second if/add statement. Since 544 is multiples of 32, 17 warps will be all active and 15 warps will be in active.

// (2) for answer, (2) for explanation

// If the student answered yes, 1 point for block 9 and 1 point for warp 24 as partial credits.

2(e). (5 points) Assume that the size of A, B, and C is 40,000 elements each. Is there any control divergence during the execution of the kernel? If so, identify the line number of the statement that causes the control divergence. Explain why or why not.

Answer: No, there is no control divergence.. Since the total number of blocks is 20 and the total number of threads in the grid will be 20480, larger than the size of the arrays, Warp 9 of Block 19 will not have divergence.

// (2.5) for answer, (2.5) for explanation, (+1) for line 6

Question 3. (15 points, suggested time allocation 25 minutes): Your friend wants to find the sum of all elements of the **bigArr** array with GPU to boost his program performance. After he heard that you are taking ECE408, he comes to you for help, desperately. His idea is as follows: 1) load elements into shared memory of blocks; 2) use threads of each block to sum up elements in its shared memory; 3) output the sum by each block to form a new array **kernelOutput**; 4) use serial code to calculate the sum of all elements in **kernelOutput**. Assume that the kernel is launched with 1024 threads in each block.

```
#define BLOCK_SIZE = 1024
```

Global variable: **float bigArr[0..n-1]** (the length is n)
 float kernelOutput[0..ceil(n / (2* BLOCK_SIZE)) - 1]

Kernel code:

```
01.   __shared__ float partialSum[2048];
02.   unsigned int bid = blockIdx.x;
03.   unsigned int t = threadIdx.x;
04.   unsigned int start = _____ X _____;
05.   if (start + t < n)
06.     partialSum[t] = bigArr[start + t];
07.   else
08.     partialSum[t] = 0.0f;
09.   if ( _____ Y _____ < n)
10.     partialSum[_____ Z _____] = bigArr[_____ Y _____];
11.   else
12.     partialSum[_____ Z _____] = 0.0f;
13.   for (unsigned int stride = 1; stride <= BLOCK_SIZE; stride *= 2)
14.  {
15.     __syncthreads();
16.     if (t < BLOCK_SIZE / stride)
17.       partialSum[t*stride*2] += partialSum[t*stride*2+stride];
18.  }
19.  if (t == 0)
20.  kernelOutput[bid] = partialSum[0];
```

3(a). (3 pts) Please help your friend fill codes in the blanks:

X: **2 * bid * BLOCK_SIZE**

Y: **start + t + BLOCK_SIZE**

Z: **t + BLOCK_SIZE**

// No partial credit

3(b). (2 pts) Does control divergence happen in the 3rd iteration of the for-loop? Assume that the warp size is 32. (Hint: draw a picture of how the threads are mapped to the data for the next few sub-questions.)

No. All active threads are consecutive and there are multiples of 32 of them.

3(c). (3 pts) How many warps in a block are active in the 3rd iteration of the for-loop? Assume that the warp size is 32.

stride = 4. threads in active = $1024 / 4 = 256$. warps in active = $256 / 32 = 8$.
// (2) for stride, (1) for the rest

3(d). (3 pts) Do you think the method your friend uses to read the global memory is coalesced? Why?

Yes. Adjacent threads access adjacent memory.
// (2) for answer, (1) for correct defn of coalescing in explanation

3(e). (4 pts) Can we put `__syncthreads()` in the if-statement as follows? Why?

```
if (t < BLOCK_SIZE / stride) {  
    __syncthreads();  
    partialSum[t*stride*2] += partialSum[t*stride*2+stride];  
}
```

No. Because if there exists control divergence in a warp, the threads which satisfy the if-statement will be in active and wait for the threads which do not satisfy the if-statement forever because these threads are deactivated.

(2) for No, (2) for understanding/explanation of `__syncthreads()`

Question 4. (20 points, suggested allocation of time 35 minutes).

Your friend Jin suggested that doing the shared matrix-matrix multiplication ($M \times N$) with matrix M transposed (M_{trans}) might increase performance in terms of memory coalesced (M_{trans} is the transposed matrix of M). The host code would provide M in its transposed form when calling the kernel. Jin being busy could not finish the transposed shared matrix-matrix multiplication and left few blanks to fill in. His idea **may** or **may not** be correct. Note that M_{trans} and N are square matrices.

```
01      #define TILE_WIDTH 32
02
03      __global__ void sgemm(float* M_trans, float* N, float* P, int
Width) {
04
05      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
06      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
07
08      int bx = blockIdx.x;  int by = blockIdx.y;
09      int tx = threadIdx.x; int ty = threadIdx.y;
10
11      // Identify the row and column of the P element to work on
12      int Row = by * TILE_WIDTH + ty;
13      int Col = bx * TILE_WIDTH + tx;
14
15      float Pvalue = 0;
16      // Loop over the M and N tiles required to compute P element
17      for (int m = 0; m < (Width - 1)/TILE_WIDTH + 1; ++m) {
18
19          // Collaborative load of M and N tiles into shared memory
20          if(Row < Width && m * TILE_WIDTH + tx < Width) {
21              Mds[ty][tx] = M_trans[(m*TILE_SIZE+tx)*Width+Row];
22          } else {
23              Mds[ty][tx] = 0;
24          }
25          if(Col < Width && m * TILE_WIDTH + ty < Width) {
26              Nds[ty][tx] = N [(m*TILE_SIZE+ty)*Width+Col];
27          } else {
28              Nds[ty][tx] = 0;
29          }
30          __syncthreads();
31
32          if(Row < Width && Col < Width) {
33              for (int k = 0; k < TILE_WIDTH; ++k) {
34                  Pvalue += Mds[ty][k] * Nds[k][tx];
```

```

35         }
36     }
37     __syncthreads();
38 }
39 if( Row < Width && Col < Width)
40     P[Row*Width + Col] = Pvalue;
41 }
```

4(a) (6 points) Fill in the missing index calculations to make this code run correctly. If you think it is impossible to make this code to run correctly, state why. (*Hint: Draw a picture of matrix multiplication for the original layout M, study the behavior of the tiles, draw another picture of matrix multiplication based on transposed M_trans, adjust the use of threads in loading the tile elements to maximize global memory coalescing if necessary.*)

Answer above.

4(b) (3 points) For the **tiled matrix-matrix multiplication** (MxN) based on the original row-major layout, which input matrix will have coalesced access? (*Hint: Draw a picture of the tile loading access patterns.*)

- (A) M
- (B) N
- (C) Both
- (D) Neither

Answer: C

4(c) (3 points) For the **transposed (M is transposed) tiled matrix-matrix multiplication** (MxN) based on row-major layout, which input matrix will have coalesced access? (*Hint: Look at your picture and make sure that you adjust the tile loading index calculation to maximize coalescing if necessary.*)

- (A) M
- (B) N
- (C) Both
- (D) Neither

Answer: B

4(d) (3 points) For the **basic matrix-matrix multiplication** ($M \times N$) (without tiling) based on row-major layout, which input matrix will have coalesced access? (*Hint: Draw a picture of the access patterns.*)

- a) M
- b) N
- c) Both
- d) Neither

Answer: B

4(e) (3 points) For the **transposed (M is transposed) basic matrix-matrix multiplication** ($M \times N$) (without tiling) based on row-major layout, which input matrix will have coalesced access? (*Hint: Draw a picture of the access patterns.*)

- a) M
- b) N
- c) Both
- d) Neither

Answer: C

4(f) (2 points) Was Jin correct on his assumption? Suppose the transposed matrix is given. Justify your answer.

Jin is not correct. The transposition access used above leads to uncoalesced accesses.

Question 5. (20 points, suggested time allocation 40 minutes) Convolution Fun. After teaching your roommate how to do parallel convolution, they come up with the following kernel. Their kernel is similar to the Strategy 2 presented in class where enough threads are instantiated to load all the input elements of a tile in one round. There are three major differences. Each thread block calculates two adjacent tiles in the x-dimension and the tiles and mask are no longer cubes. Look at the code below and answer the following questions.

```
#define TILE_X 8           // Output tile width in the X-dimension
#define TILE_Y 4           // Output tile width in the Y-dimension
#define TILE_Z 2           // Output tile width in the Z-dimension
#define NUM_X 2
#define X_MASK_WIDTH 3
#define Y_MASK_WIDTH 3
#define Z_MASK_WIDTH 5
#define MASK_SIZE X_MASK_WIDTH * Y_MASK_WIDTH * Z*MASK_WIDTH

__ constant__ float mask[Z_MASK_WIDTH][Y_MASK_WIDTH][X_MASK_WIDTH];

__global__ void conv3d(float *input, float *output, const int z_size, const int y_size, const int
x_size) {
    __shared__ float inputTile
[TILE_Z+Z_MASK_WIDTH-1][TILE_Y+Y_MASK_WIDTH-1][TILE_X+X_MASK_WIDTH-1];
    int tx = threadIdx.x; int ty = threadIdx.y; int tz = threadIdx.z;
    int bx = blockIdx.x; int by = blockIdx.y; int bz = blockIdx.z;

    for(int x_tile = 0; x_tile < NUM_X; x_tile++) {
        (2) __syncthreads(); (solution for part B)
        int x_o = NUM_X * bx * TILE_X + tx + x_tile * TILE_X;
        int y_o = by * TILE_Y + ty;
        int z_o = bz * TILE_Z + tz;
        (2) __syncthreads(); (solution for part B)
        int x_i = x_o - X_MASK_WIDTH/2;
        int y_i = y_o - Y_MASK_WIDTH/2;
        int z_i = z_o - Z_MASK_WIDTH/2;
        (2) __syncthreads(); (solution for part B)
        if (x_i >= 0 && y_i >= 0 && z_i >= 0 && x_i < x_size && y_i < y_size && z_i < z_size)
            inputTile[tz][ty][tx] = input[(z_i * y_size + y_i) * x_size + x_i];
        else
            inputTile[tz][ty][tx] = 0.0;
        (1) __syncthreads(); (solution for part B)
        float acc = 0.0;
        (1) __syncthreads(); (solution for part B)
```

```

if(tz < TILE_Z && ty < TILE_Y && tx < TILE_X) {

    for(int z_mask = 0; z_mask < Z_MASK_WIDTH; z_mask++) {

        for(int y_mask = 0; y_mask < Y_MASK_WIDTH; y_mask++) {

            for(int x_mask = 0; x_mask < X_MASK_WIDTH; x_mask++) {

                acc += maskl[z_mask][y_mask][x_mask] * inputTile[tz+z_mask][ty+y_mask][tx+x_mask];

            }

        }

    }

}

if(z_o < z_size && y_o < y_size && x_o < x_size)
    output[(z_o * y_size + y_o) * x_size + x_o] = acc;

}

(2) __syncthreads(); (solution for part B)

}

}

```

5(a) (4 points) Write out the host code for declaring the grid and block dimensions below.

Answer:

```

dim3 DimGrid((x_size/TILE_X*2.0), y_size/TILE_Y, z_size/TILE_Z);
if(x_size%(TILE_X*2)) DimGrid.x++;
if(y_size%TILE_Y) DimGrid.y++;
if(z_size%TILE_Z) DimGrid.z++;
dim3 DimBlock(TILE_X+X_MASK_WIDTH-1, TILE_Y+Y_MASK_WIDTH-1,
TILE_Z+Z_MASK_WIDTH-1);

```

5(b) (4 points) Unfortunately your roommate wasn't paying attention when you talked about `__syncthreads()` and when you should use it. Please insert only the necessary `__syncthreads()` to the kernel above to make sure it works.

Answer: Need 2 `__syncthreads()`, 1 (1) and 1 (2). +2 For each.

5(c) (3 points) For an internal output tile, what is the average number of times that each input element will be accessed from the shared memory during the calculation an output tile? (Note:

We are referring to tiles in this case, not thread blocks. Each thread block calculates two adjacent output tiles.)

Answer:

$$\begin{aligned}(\text{Num Output elements}) * (\text{Size of mask}) / (\text{Num. Input Elements}) &= \\(8*4*2)*(3*3*5) / ((8+3-1)*(4+3-1)*(2+5-1)) &= 8\end{aligned}$$

5(d) (3 points) How does your answer to (C) compare to an output tile that is a cube that contains the same volume and why? Note: We are referring to tiles in this case, not thread blocks. Each thread block calculates two adjacent output tiles. (*Hint: Are all the input tile elements reused the same number of times? Does it matter where they are in the input tile?*)

Answer:

A cube tile would have reuse of $(4^3)*(3*3*5) / ((4+3-1)*(4+3-1)*(4+5-1)) = 10$

A rectangular prism has a smaller average than a cube with the same volume because there are more elements in the center that use more input elements in a cube than a rectangular prism. That is, a rectangular prism has more surface elements than a cube of the same volume. As a result, the volume of the input tile for a cube is larger than the volume of the input tile for a rectangular prism. More input elements need to be loaded to calculate the same number of output elements.

+3 For correct explanation (calculation does not need to be included)

+3 For correct reuse calculation and conclusion that cube has more reuse

+2 For reuse calculation with mistakes and conclusion that cube is better

+1 For correct reuse calculation, but incorrect conclusion

+0 For incorrect conclusion

5(e) (3 points) Consider another output tile with dimensions 2x4x8. Is there a difference in terms of execution efficiency of the kernel between this 2x4x8 output tile and the output tile in the kernel above with dimensions 8x4x2? If so explain why, if not, explain why not.

Answer: Because the elements are stored along the x then y dimensions, the 8x4x2 tile will have better memory coalescing than the 2x4x8. (As long as the assumption for which dimension is x is clear than either 2x4x8 or 8x4x2 would be accepted)

+3 Talks about what dimension needs to be larger and memory coalescing

+2 For x dimension being larger but no memory coalescing

+1 For memory reuse, as it does have an impact, but when comparing DRAM accesses, the one with lower reuse, i.e. 8 in the x, 4 in the y, 2 in the z will only access DRAM $6 * 6$ times for each tile while the one with higher reuse, 2 in the x, 4 in the y, and 8 in the z will access DRAM $6 * 12$ times for each tile which is twice as many accesses.

+1 For talking about control divergence as it again ignores memory coalescing. The one with 8 in the x, 4 in the y, 2 in the z will have less control divergence, but more warps.

+0 For all other answers.

5(f) (3 points) Is there any inefficiency in loading the input tiles for the two adjacent output tiles processed by each block? If so, identify the cause of the inefficiency and briefly suggest a strategy to eliminate the inefficiency. (*Hint: Draw a picture of the input tiles for these two output tiles.*)

Answer:

The left halo surface elements are loaded redundantly during the second iteration of the outermost loop. One should use a larger output tile rather than iterating over two output tiles. Or one could try to use if-statements to avoid the redundant loads during the second iteration. .

+3 A student will receive full credit if they spot this inefficiency and choose one of these strategies.

+2 For an inefficiency that is particular to the loading of input tiles for the two adjacent output tiles and no correct solution.

+1 For an inefficiency that is not particular to loading two adjacent output tiles, i.e. it is also present in the 1 tile per block case and correct solution.

+1 For a reasonable inefficiency, but no correct or incorrect solution.

+0 for an incorrect inefficiency.

ECE 408 Exam #1 Study Guide, Fall 2019

1. Exam format

- You are allowed one 8.0x11.5 cheat sheet with notes on both sides. The minimal font size for your text on the cheat sheet should be 8pts.
- No interactions with humans other than course staff are allowed.
- This exam is designed to take 150 minutes to complete. To allow for any unforeseen difficulties, we will give everyone 180 minutes.
- This exam is based on lectures, textbook chapters, as well as lab MPs/projects.
- The questions are randomly selected from the topics we covered.
- You can write down the reasoning behind your answers for possible partial credit.

2. Topics to Review from Lectures

2.1. CUDA Basic concepts

- Mapping thread index into data index
 - Memory hierarchies – characteristics and usage of each type of memory
1. We need to use each thread to calculate one output element of a vector addition. Assume that variable i should be the index for the element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index?
 - (A) $i = \text{threadIdx.x} + \text{threadIdx.y};$
 - (B) $i = \text{blockIdx.x} + \text{threadIdx.x};$
 - (C) $i = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x};$
 - (D) $i = \text{blockIdx.x} * \text{threadIdx.x};$
 2. We want to use each thread to calculate two (adjacent) output elements of a vector addition. Assume that variable i should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?
 - (A) $i = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x} + 2;$
 - (B) $i = \text{blockIdx.x} * \text{threadIdx.x} * 2$
 - (C) $i = (\text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}) * 2$

- (D) $i = blockIdx.x * blockDim.x * 2 + threadIdx.x$
3. We want to use each thread to calculate two output elements of a vector addition. Each thread block processes $2 * blockDim.x$ consecutive elements that form two sections. All threads in each block will first process a section first, each processing one element. They will then all move to the next section, each processing one more element. Assume that variable i should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?
- (A) $i = blockIdx.x * blockDim.x + threadIdx.x + 2;$
(B) $i = blockIdx.x * threadIdx.x * 2$
(C) $i = (blockIdx.x * blockDim.x + threadIdx.x) * 2$
(D) $i = blockIdx.x * blockDim.x * 2 + threadIdx.x$
4. For a vector addition, assume that the vector length is 8000, each thread calculates one output element, and the thread block size is 1024 threads. The programmer configures the kernel launch to have a minimal number of thread blocks to cover all output elements. How many threads will be in the grid?
- (A) 8000
(B) 8196
(C) 8192
(D) 8200
5. If we want to allocate an array of v integer elements in CUDA device global memory, what would be an appropriate expression for the second argument of the `cudaMalloc()` call?
- (A) n
(B) v
(C) $n * \text{sizeof(int)}$
(D) $v * \text{sizeof(int)}$
6. If we want to allocate an array of n floating-point elements and have a floating-point pointer variable d_A to point to the allocated memory, what would be an appropriate expression for the first argument of the `cudaMalloc()` call?
- (A) n
(B) $(\text{void} *) d_A$
(C) $*d_A$
(D) $(\text{void} **) \&d_A$

7. If we want to copy 3000 bytes of data from host array `h_A` (`h_A` is a pointer to element 0 of the source array) to device array `d_A` (`d_A` is a pointer to element 0 of the destination array), what would be an appropriate API call for this in CUDA?
 - (A) `cudaMemcpy(3000, h_A, d_A, cudaMemcpyHostToDevice);`
 - (B) `cudaMemcpy(h_A, d_A, 3000, cudaMemcpyDeviceToHost);`
 - (C) `cudaMemcpy(d_A, h_A, 3000, cudaMemcpyHostToDevice);`
 - (D) `cudaMemcpy(3000, d_A, h_A, cudaMemcpyHostToDevice);`
8. How would one declare a variable `err` that can appropriately receive returned value of a CUDA API call?
 - (A) `int err;`
 - (B) `cudaError err;`
 - (C) `cudaError_t err;`
 - (D) `cudaSuccess_t err;`

1. Answer: (C)

Explanation: This is the linearized global thread index for the whole grid.

2. Answer: (C)

Explanation: Every thread covers 2 consecutive elements. The starting data index is simply twice the global thread index. Another way to look at it is that all previous blocks cover $(blockIdx.x * blockDim.x) * 2$. Within the block, each thread covers 2 elements so the beginning position for a thread is $2 * threadIdx.x$ beyond what is covered by all the previous blocks.

3. Answer: (D)

Explanation: All previous blocks cover $(blockIdx.x * blockDim.x) * 2$. The beginning element is consecutive in this case so just add `threadIdx.x` to it.

4. Answer: (C)

Explanation: $\lceil 8000 / 1024.0 \rceil * 1024 = 8 * 1024 = 8192$. Another way to look at it is the minimal multiple of 1028 to cover 8000 is $1024 * 8 = 8192$.

5. Answer: (D)

Explanation: This one should be self-evident.

6. Answer: (D)

Explanation: `&d_A` is pointer to a pointer of float. To convert it to a generic pointer required by `cudaMalloc()` should use `(void **)` to cast it to a generic double-level pointer.

7. Answer: (C)

Explanation: See Lecture slides.

8. Answer: (C)

Explanation: See Lecture slides.

2.2. Parallel Kernel Execution Concepts

- Using multi-dimensional thread indices to easily access multi-dimensional data structures
 - Boundary condition checking and control divergence
 - Thread capacity and thread block capacity of Streaming Multiprocessor
1. We are to process an 800X600 (800 pixels in the x or horizontal direction, 600 pixels in they or vertical direction) picture with the PictureKernel in Lecture slides:

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m) {  
  
    // Calculate the row # of the d_Pin and d_Pout element to process  
    int Row = blockIdx.y*blockDim.y + threadIdx.y;  
  
    // Calculate the column # of the d_Pin and d_Pout element to process  
    int Col = blockIdx.x*blockDim.x + threadIdx.x;  
  
    // each thread computes one element of d_Pout if in range  
    if ((Row < m) && (Col < n)) {  
        d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];  
    }  
}
```

Assume that we decided to use a grid of 16X16 blocks. That is, each block is organized as a 2D 16X16 array of threads. Which of the following statements sets up the kernel configuration properly? Assume that int variable n has value 800 and int variable m has value 600. The kernel is launched with the statement
PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, n, m);

- (A) dim3 DimGrid(ceil(n/16.0), ceil(m/16.0), 1); dim3 DimBlock(16, 16, 1);
- (B) dim3 DimGrid(1, ceil(n/16.0), ceil(m/16.0)); dim3 DimBlock(1, 16, 16);
- (C) dim3 dimGrid(ceil(m/16.0), ceil(n/16.0), 1); dim3 DimBlock(16,16,1);
- (D) dim3 DimGrid(1, ceil(m/16.0), ceil(n/16.0)); dim3 DimBlock(1, 16, 16);

2. In Question 1, how many warps will have control divergence?

- (A) 37*16 + 50
- (B) 38*16
- (C) 50
- (D) 0

3. If a CUDA device's SM (streaming multiprocessor) can take up to 1,536 threads and up to 8 thread blocks. Which of the following block configuration would result in the most number of threads in each SM?

(A) 64 threads per block
(B) 128 threads per block
(C) 512 threads per block
(D) 1024 threads per block

4. Assume the following simple matrix multiplication kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    int Col = blockIdx.x*blockDim.x+threadIdx.x;
    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        for (int k = 0; k < Width; ++k) {Pvalue += M[Row*Width+k] * N[k*Width+Col];}
        P[Row*Width+Col] = Pvalue;
    }
}
```

If we launch the kernel with a block size of 16X16 on a 1000X1000 matrix, how many warps will have control divergence?

(A) 1000
(B) 500
(C) 1008
(D) 508

1. Answer: (A)

Explanation: dim3 format is (x, y, z). Since n is the size of the picture in the x direction and m is the size of the y direction, we should use n to set up the x dimension and m to set up the y dimension.

2. Answer: (D)

Explanation: The size of the picture in the x dimension is a multiple of 16 so there is no block in the x direction that has any threads in the invalid range. The size of the picture in the y dimension is 37.5 times of 16. This means that the threads in the last block are divided into halves: 128 in the valid range and 128 in the invalid range. Since 128 is a multiple of 32, all warps will fall into either one or the other range. There is no control divergence.

3. Answer: (C)

Explanation: (A) and (B) are limited by the number of thread blocks that can be accommodated by each SM. (D) is not a divider of 1,536, leaving 1/3 of the thread space open. (C) results in 3 blocks and fully occupies the capacity of 1,536 threads in each SM.

4. Answer: (B)

Explanation: There will be 63 blocks in the horizontal direction. 8 threads in the x dimension in each row will be in the invalid range. Every two rows form a warp. Therefore, there are $1000/2 = 500$ warps that will straddle the valid and invalid ranges in the horizontal direction. As for the warps in the bottom blocks, there are 8 warps in the valid range and 8 warps in the invalid range. Threads in these warps are either totally in the valid range or invalid range.

2.3. Tiling

- Proper use of CUDA shared memory
 - Explain the principles and scope of memory coalescing
 - Derive the necessary array indexing for a tiled matrix multiplication
 - Understand the use of barrier synchronization in a tiled algorithm
 - Estimate the reduction of memory bandwidth usage
 - Overhead due to halo cells in algorithms such as convolution
 - Understanding how to extend square tiles to rectangular tiles
1. Assume that a kernel is launched with 1000 thread blocks each of which has 512 threads. If a variable is declared as a shared memory variable, how many versions of the variable will be created through the lifetime of the execution of the kernel?
 - (A) 1
 - (B) 1000
 - (C) 512
 - (D) 51200
 2. For our tiled matrix-matrix multiplication kernel, if we use a 32X32 tile, what is the reduction of memory bandwidth usage for input matrices M and N?
 - (A) 1/8 of the original usage
 - (B) 1/16 of the original usage
 - (C) 1/32 of the original usage
 - (D) 1/64 of the original usage
 3. For the tiled single-precision matrix multiplication kernel as shown in Lecture, assume that the tile size is 32X32 and the system has a DRAM burst size of 128 bytes. How many DRAM bursts will be delivered to the processor as a result of loading one M-matrix tile by a thread block?

- (A) 16
(B) 32
(C) 64
(D) 128
4. Assume that A is a global memory float array that is properly aligned to a DRAM burst section boundary. Further assume that the number of threads in the x-dimension of each thread block is greater than or equal to the warp size. Which of the following accesses in a kernel will make the most effective use of the DRAM bandwidth? Assume that k and Width are integer variables that do not depend on threadIdx.x or threadIdx.y. The Width value can be assumed to be a multiple of the DRAM burst size.
-
- (A) $A[2 * \text{threadIdx.x}]$
(B) $A[\text{threadIdx.x} * \text{Width} + k]$
(C) $A[\text{threadIdx.x} + \text{Width} * k]$
(D) $A[k * \text{threadIdx.x}]$
5. Assume a DRAM system with a burst size of 512 bytes and a peak bandwidth of 240 GB/s. Assume a thread block size of 1024 and warp size of 32 and that A is a float array in the global memory. What is the maximal memory data access throughput we can hope to achieve in the following access to A?
- ```
int i = blockIdx.x * blockDim.x + threadIdx.x;
float temp = A[8*i];
```
- (A) 240 GB/s  
(B) 120 GB/s  
(C) 60 GB/s  
(D) 30 GB/s
6. Assume a tiled matrix multiplication that handles boundary conditions as explained in Lecture. If we use 32X32 tiles to process square matrices of 1000X1000, within EACH thread block, what is the maximal number of warps that will have control divergence due to handling boundary conditions for loading M matrix tiles throughout the kernel execution?
- (A) 32  
(B) 24  
(C) 16  
(D) 8

7. For a tiled 2D convolution, if each output tile is a square with 256 elements on each side and the mask is a square with 5 elements on each side, how many elements are in each input tile?
- (A)  $256 \times 256 = 65,536$   
(B)  $5 \times 5 = 25$   
(C)  $(256+2) \times (256+2) = 66,564$   
(D)  $(256+4) \times (256+4) = 67,600$
8. For a tiled 2D convolution, assume that we load an entire input tile, including the halo elements into the shared memory when calculating an output tile. Further assume that the tiles are internal and thus do not involve any ghost elements. If each output tile is a square with 256 elements on each side and the mask is a square with 5 elements on each side, which of the following best approximate the average number of times each input element will be accessed from the shared memory during the calculation of an output tile?
- (A) 256  
(B) 25  
(C) 24  
(D) 4.9
9. For a tiled 3D convolution, assume that we load an entire input tile, including the halo elements into the shared memory when calculating an output tile. Further assume that the tiles are internal and thus do not involve any ghost elements. If the mask is a cube with 5 elements on each side and due to the limited size of the shared memory, each output tile is a cube with 16 elements on each side. What is the average number of times each input element will be accessed from the shared memory during the calculation an output tile?
- (A) 256  
(B) 64  
(C) 24  
(D) 4
10. For a tiled 3D convolution, assume that we load an entire input tile, including the halo elements into the shared memory when calculating an output tile. Further assume that the tiles are internal and thus do not involve any ghost elements. If the mask is a cube with 5 elements on each side what is the trend of the average number of times each input element will be accessed from the shared memory during the calculation an output tile as a function of the input tile width?
- (A) Increases with the width of the tile size with a limit of 25  
(B) Decreases with the width of tile size with a limit of 25  
(C) Increases with the width of the tile size with a limit of 125  
(D) Decreases with the width of the tile size with a limit of 125

1. Answer: (B)

Explanation: Shared memory variables are allocated to thread blocks. So, the number of versions is the number of thread blocks, 1000.

2. Answer: (C)

Explanation: Each element in the tile is used 32 times, as explained in Lecture

3. Answer: (B)

Explanation: For an 32X32 M-tile, each row in the tile consists of 32 consecutive words and is accessed by a warp. The total amount of data in the row is just a single burst. We have 32 rows in a tile so there will be 32 bursts delivered to the processor.

4. Answer: (C)

Explanation: All consecutive threads in  $A[\text{threadIdx.x} + \text{Width} * k]$  access consecutive memory locations. Since A is properly aligned to the DRAM burst sections and  $\text{Width} * k$  will be always a multiple of DRAM burst sizes, all DRAM burst bytes will be fully utilized. All other accesses are strided accesses that will waste at least some of the burst bytes.

5. Answer: (D)

Explanation: Each warp is going to access 64 bytes of data from a 512-byte section. It will waste at least  $7/8$  of the DRAM bandwidth. The access cannot achieve more 30 GB/s.

6. Answer: (A)

Explanation: Control divergence happens due to the handling of right-side boundary. For thread blocks that process tiles that are totally within the valid range in the y-dimension, all 32 warps in a block will experience divergence at the right boundary. For the thread block that process the bottom M tiles, only 8 warps will experience control divergence because 24 warps will always fail the boundary test.

7. Answer: (D)

Explanation: As shown in Lecture 7.3, Slide 12, the number of elements in an input tile is  $(\text{tile\_width} + \text{mask\_width} - 1) * (\text{tile\_width} + \text{mask\_width} - 1)$ , where tile-width is the width the output tiles.

8. Answer: (C)

Explanation: As shown in Lecture 7.3, Slide, the answer is  $\text{tile\_width}^2 * \text{mask\_width}^2 / (\text{tile\_width} + \text{mask\_width} - 1)^2 = 256^2 * 5^2 / (256 + 5 - 1)^2 = 24.2$

9. Answer: (B)

Explanation: As generalized from Lecture 7.3, the answer is  $(\text{tile\_width}^3 * \text{mask\_width}^3) / (\text{tile\_width} + \text{mask\_width} - 1)^3 = 16^3 * 5^3 / (16 + 5 - 1)^3 = 64$

10. Answer: (C)

The average number of times an input tile element is accessed from the shared memory is  $(\text{tile\_width}^3 * \text{mask\_width}^3) / (\text{tile\_width} + \text{mask\_width} - 1)^3$ . For a given mask\_width, the value increases as the tile\_width increases. When tile-width becomes much larger than tile\_width, the mask\_width term can be dropped from the denominator. This makes the expression  $\text{mask\_width}^3$ .

### 3. Topics to Review from Lab

The answers have not been fully verified. Please check with TAs and Prof. Hwu if you would like to challenge any of the answers.

Common sources of bugs

- Function prototype problems
- Barrier synchronization problems
- Indexing problems

Performance Issues

- Access patterns that result non-coalesced global memory accesses
- Sources of control divergence
- Thread utilization

Tiling

- Square vs. Rectangular tiling
- Handling boundary conditions in tiling
- Barrier synchronization usage

Convolution

- Different strategies for convolution implementations, their pros and cons, and how they reflect on kernel launch configurations

Reduction Tree

- Reduction trees, memory access patterns, and branch divergence

**Lab Question 1.** CUDA Basics. For the vector addition kernel and the corresponding kernel launch code, answer each of the sub-questions below.

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
 1. int i = threadIdx.x + blockDim.x * blockIdx.x;
 2. int Stride = blockDim.x * gridDim.x;
```

```

3. while (i < n) {
4. C_d[i] = A_d[i] + B_d[i];
5. i+= Stride;
6. }
7. }

8. int vectAdd(float* A, float* B, float* C, int n)
9. {
//assume that size has been set to the actual length of
//arrays A, B, and C
10. int size = n * sizeof(float);
11.
12. cudaMalloc((void **) &A_d, size);
13. cudaMalloc((void **) &B_d, size);
14. cudaMalloc((void **) &C_d, size);
15. cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
16. cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
17. vecAddKernel<<<8,1024>>>(A_d, B_d, C_d, n);
18. cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
}

```

- (A) Assume that the size of A, B, and C is 20,000 elements each. How many thread blocks will be generated?

Answer:

- (B) Assume that the size of A, B, and C is 20,000 elements each. How many warps are there in each block?

Answer:

- (C) Assume that the size of A, B, and C is 20,000 elements. How many threads will be created in the grid?

Answer:

- (D) Assume that the size of A, B, and C is 20,000 elements each. Is there any control divergence during the execution of the kernel? If so, identify the block number and warp number that causes the control divergence. Explain why or why not.

Answer:

- (E) Assume that the size of A, B, and C is 10,000 elements each. Is there any control divergence during the execution of the kernel? If so, identify the line number of the statement that causes the control divergence. Explain why or why not.

Answer:

Answers:

- (A) 8
- (B) 32
- (C) 8,192

- (D) No. We have 8,192 threads. All threads will iterate at least two iterations to process 16,384 elements. During the last iteration, only  $20,000 - 16,384 = 3,616$  threads will be active. These threads form 113 warps. So, all threads in all the first 113 warps will be active. The rest 143 warps will see all their threads inactive.
- (E) We have 8,192 threads. All threads will iterate at least one iteration to process 8,192 elements. During the last iteration, only  $10,000 - 8,192 = 1,808$  threads will be active. These threads form 56.5 warps. So, all threads in all the first 56 warps will be active. Warp 24 on Block 1 will have control divergence. All remaining 199 warps will see all their threads inactive and thus see no control divergence.

**Lab Question 2.** Many numerical libraries offer matrix multiplication functions that accept one of the matrices in transposed form. Redo MP-2 simple matrix multiplication assuming that the first input matrix M is in transposed form ( $M_T$ ).

(A) Write a new sgemm kernel for this transposed M input:

- (B) Assume that the height and width values of the input matrices are in host variables Height\_M\_T, Width\_M\_T, Height\_N, Width\_N, write the host code that sets up the grid dimensions for launching a kernel with 32x32 threads per block.
- (C) Explain the benefit of having a transposed M as input for a CUDA GPU.

Answer:

(D) What is the minimal block width in the x dimension for full memory bandwidth utilization?

Answer:

- (E) Does it make sense for us to have a version of the tiled matrix multiplication kernel that accepts transposed M input matrix?

Answer:

Answers:

```
(A)
__global__ void sgemm(float* M_T, float* N, float* P, int HeiM_T, int WidM_T, int WidN)
{
 // Height N is the same as Height M_T

 // Calculate the row index of the P element, this will be used as the Col index for M_T
 int Row = blockIdx.y*blockDim.y+threadIdx.y;
```

```

// Calculate the column index of P element and N
int Col = blockIdx.x*blockDim.x+threadIdx.x;

if ((Row < WidM_T) && (Col < WidN)) {
 float Pvalue = 0;
 // each thread computes one element of the block sub-matrix
 for (int k = 0; k < HeiM; ++k){
 Pvalue += M[k*WidM+Row] * N[k*WidN+Col];
 }
 P[Row][Col] = Pvalue;
}
}

```

(B)

```

dim3 DimGrid((WidM/32, WidN/32, 1);
if(WidM%32) DimGrid.x++;
if(WidN%32) DimGrid.y++;
dim3 DimBlock(32,32, 1);

```

(C) With the transposed input, the accesses to M\_T are now coalesced. This should improve the speed significantly.

- (D) The number of threads in the X dimension should be at least the DRAM burst size. So if the DRAM burst size is 128 bytes, the 32X32 block configuration would fully utilize the memory bandwidth.
- (E) Read the textbook section that explains “corner turning”.

### Lab Question 3

To improve the performance of her tiled matrix multiplication kernel, Jill decided to try to use rectangular tiling. Instead of using 32x32 input and output tiles, she would like to try to use 16x64 (16 in the Y dimension and 64 in the X dimension) input M tiles and 64x16 input N tiles.

- (A) Based on the input tile dimensions, what is the maximal output tile dimensions that can be supported by these input tiles? Explain your answer.

Answer:

- (B) The code below shows her code. Fill in the missing index calculations.

```

01 #define M_TILE_H 16 // Height of input M tiles (Y dimension)
02 #define M_TILE_W 64 // Width of input M tiles (X dimension)
03 #define N_TILE_H 64 // Height of input N tiles (Y dimension)
04 #define X_N_TILE_WIDTH 16 // Weight of input N tiles (X dimension)
02
03 __global__ void sgemm(float* M, float* N, float* P, int HeiM, int WidM, int WidN) {
04

```

```

05 __shared__ float Mds[M_TILE_H][M_TILE_W];
06 __shared__ float Nds[N_TILE_H][N_TILE_W];
07
08 int bx = blockIdx.x; int by = blockIdx.y;
09 int tx = threadIdx.x; int ty = threadIdx.y;
10
11 // Identify the row and column of the P element to work on
12 int Row = by * [REDACTED] + ty;
13 int Col = bx * [REDACTED] + tx;
14
15 float Pvalue = 0;
16 // Loop over the M and N tiles required to compute P element
17 for (int m = 0; m < (WidM - 1)/M_TILE_W + 1; ++m) {
18
19 // Collaborative load of M and N tiles into shared memory
20 if(Row < HeiM) {
21 Mds[ty][tx] = M[[REDACTED]];
22 } else {
23 Mds[ty][tx] = 0.0;
24 }
25 if(Col < WidN) {
26 Nds[ty][tx] = N[[REDACTED]];
27 } else {
28 Nds[ty][tx] = 0.0;
29 }
30 __syncthreads();
31
32 if (Row < HeiM && Col < WidN) {
33 for (int k = 0; k < M_TILE_W; ++k) {
34 Pvalue += [REDACTED];
35 }
36 }
37 __syncthreads();
38 }
39 if (Row < HeiM && Col < WidN) P[Row*Width + Col] = Pvalue;
40 }
```

Answer:

(C) What is the expected number of times each input element is reused in the Shared Memory?

Answer:

(D) Compare this tiling configuration with what you had in MP3. What are the pros and cons of this rectangular configuration?

Answer:

Answers:

- (A) If we draw a picture of the input and output tiles like the one we showed for square tiles in Lecture slides, the output tile has the same height as the input M tile and the same width as the input N tile. So the maximal output tile dimensions that can be supported by the input tiles are 16x16.

(B)

Line 12: M\_TILE\_H  
Line 13: N\_TILE\_W  
Line 21: Row \* WidM + (m \* M\_TILE\_W)+tx  
Line 22: (m\*N\_TILE\_H+ty) \* WidN + Col  
Line 34: Mds[ty][k] \* Nds[k][tx]

Please try to run the code on WebGPU as another attempt for MP3.

- (C) 16, each M tile row is used to calculate 16 P elements in the same row. Each N tile column is used to calculate 16 P elements in the same column.

(D)

Cons:

1. It results in less memory reuse but requires the same amount of shared memory for each block.
2. If the DRAM burst size is 32 words, the new method will use only half of the memory bandwidth when loading input tiles and writing output elements.

Pros:

1. It allows each thread to execute twice the number of inner-product iterations (64 vs. 32) between syncthreads
2. The thread block size is 256. We may be able to fit more thread blocks into a streaming multiprocessor. Since these thread blocks do syncthreads independently, we may have a better utilization of the execution resources since there may be more thread blocks that are not executing syncthreads at any moment in time.

**Lab Question 4.** To further improve the memory access efficiency and reduce the memory bandwidth consumption of his 3D convolution kernel, John decided to use rectangular prism tiles rather than cube tiles. Assume that the convolution mask is 3x3x3. His current cube tile design is 6x6x6 for output tiles and 8x8x8 for input tiles (Strategy 2). He decided to try to change the output tiles 14x6x6 and input tiles to input tiles to 16x8x8.

- (A) For an internal 16x8x8 input tile, what is the average number of times an input element is accessed from the shared memory? Show your work.

Answer:

- (B) Does the number of threads in each block fit within the limits of CUDA block size? Why or why not?

Answer:

- (C) How many thread blocks will be generated if we process a 512x768x256 volume of data with the new kernel? Show your work.

Answer:

- (D) For an internal 8x8x8 input tile in the original cube tile design, what is the average number of times each input element is reused once it is loaded into the shared memory? Show your work.

Answer:

- (E) How many thread blocks will be generated if we process a 512x768x256 volume of data with the original cube tiling kernel? Show your work.

Answer:

- (F) Compare the pros and cons of the new rectangular kernel vs. the cube kernel. Assume that the DRAM burst size is 128 bytes.

Answer:

- (G) Fill in the index calculations and missing code of the following rectangular prism kernel.

```
#define TILE_X 14 // Output tile width in the X-dimension
#define TILE_Y 6 // Output tile width in the Y-dimension
#define TILE_Z 6 // Output tile width in the Z-dimension
#define MASK_WIDTH 3
#define MASK_SIZE MASK_WIDTH * MASK_WIDTH * MASK_WIDTH

__constant__ float mask[MASK_WIDTH][MASK_WIDTH][MASK_WIDTH];

__global__ void conv3d(float *input, float *output, const int z_size, const int y_size, const int x_size) {
 __shared__ float inputTile [TILE_Z+MASK_WIDTH-1][TILE_Y+MASK_WIDTH-1][TILE_X+MASK_WIDTH-1];
 int tx = threadIdx.x; int ty = threadIdx.y; int tz = threadIdx.z;
 int bx = blockIdx.x; int by = blockIdx.y; int bz = blockIdx.z;
```

```

int x_o = bx * _____ + tx
int y_o = by * _____ + ty;
int z_o = bz * _____ + tz;

int x_i = _____;
int y_i = _____;
int z_i = _____;

if (x_i >= 0 && y_i >= 0 && z_i >= 0 && x_i < x_size && y_i < y_size && z_i < z_size)
 inputTile[tz][ty][tx] = input[_____];
else
 inputTile[tz][ty][tx] = 0.0;

_____;

float acc = 0.0;

if(tz < TILE_Z && ty < TILE_Y && tx < TILE_X) {

 for(int z_mask = 0; z_mask < Z_MASK_WIDTH; z_mask++) {

 for(int y_mask = 0; y_mask < Y_MASK_WIDTH; y_mask++) {

 for(int x_mask = 0; x_mask < X_MASK_WIDTH; x_mask++) {

 acc += mask[____][____][____] * inputTile[____][____][____];
 }
 }
 }

 if(z_o < z_size && y_o < y_size && x_o < x_size)
 output[(z_o * y_size + y_o) * x_size + x_o] = acc;
 }
}

```

Answers:

- (A)  $(14*6*6)((3*3*3) / (16*8*8)) = 13.3$
- (B) The number of threads is the same as the block size so the block size is  $16*8*8 = 1024$ . It fits into the limit of the CUDA block size, which is 1024.
- (C)  $\text{ceil}(512/14.0) * \text{ceil}(768/6.0) * \text{ceil}(256/6.0) = 203,648$
- (D)  $(6*6*6) * (3*3*3) / (8*8*8) = 11.39$
- (E)  $\text{ceil}(512/6.0) * \text{ceil}(768/6.0) * \text{ceil}(256/6.0) = 86 * 128 * 43 = 473,344$
- (F)

Pros:

1. The input elements are reused more in the shared memory than the original tile configuration due to larger tile size.
2. The memory bandwidth is better utilized since the input tile width in the X dimension is the same as the DRAM burst size. The original tile configuration uses more than double the memory bandwidth since it wastes half of the memory bandwidth.

Cons:

1. The new kernel generates fewer, larger thread blocks. This may increase the negative impact of the barrier synchronizations.

(G)

```
#define TILE_X 14 // Output tile width in the X-dimension
#define TILE_Y 6 // Output tile width in the Y-dimension
#define TILE_Z 6 // Output tile width in the Z-dimension
#define MASK_WIDTH 3
#define MASK_SIZE MASK_WIDTH * MASK_WIDTH * MASK_WIDTH

__constant__ float mask[MASK_WIDTH][MASK_WIDTH][MASK_WIDTH];

__global__ void conv3d(float *input, float *output, const int z_size, const int y_size, const int x_size) {
 __shared__ float inputTile [TILE_Z+MASK_WIDTH-1][TILE_Y+MASK_WIDTH-1][TILE_X+MASK_WIDTH-1];
 int tx = threadIdx.x; int ty = threadIdx.y; int tz = threadIdx.z;
 int bx = blockIdx.x; int by = blockIdx.y; int bz = blockIdx.z;

 int x_o = bx * TILE_X + tx
 int y_o = by * TILE_Y + ty;
 int z_o = bz * TILE_Z + tz;

 int x_i = x_o - MASK_WIDTH/2;
 int y_i = y_o - MASK_WIDTH/2;
 int z_i = z_o - MASK_WIDTH/2;

 if (x_i >= 0 && y_i >= 0 && z_i >= 0 && x_i < x_size && y_i < y_size && z_i < z_size)
 inputTile[tz][ty][tx] = input[(z_i * y_size + y_i) * x_size + x_i];
 else
 inputTile[tz][ty][tx] = 0.0;

 __syncthreads();

 float acc = 0.0;

 if(tz < TILE_Z && ty < TILE_Y && tx < TILE_X) {

 for(int z_mask = 0; z_mask < Z_MASK_WIDTH; z_mask++) {

 for(int y_mask = 0; y_mask < Y_MASK_WIDTH; y_mask++) {

 for(int x_mask = 0; x_mask < X_MASK_WIDTH; x_mask++) {

 acc += mask[z_mask][y_mask][x_mask] * inputTile[tz+z_mask][ty+y_mask][tx+x_mask];
 }
 }
 }
 }
}
```

```

 }

 }

}

if(z_o < z_size && y_o < y_size && x_o < x_size)
 output[(z_o * y_size + y_o) * x_size + x_o] = acc;

}

```

**Lab Question 5.** Out of curiosity, Emily decided to try Strategy 3 in her 2D kernel. She decided to use square 32x32 input and output kernel. Assume that the mask is 3x3.

- (A) For an input tile, what is the average number of times each input element is reused once it is loaded into the shared memory. (Hint: draw a picture any check the number of cases that you need to analyze.)

Answer:

- (B) How many thread blocks will be generated if we process 512x768 input data?

Answer:

- (C) Fill in the missing index calculations and boundary condition checks in the following Strategy 3 kernel.

Answer:

```

#define MASK_WIDTH 3
#define TILE_WIDTH 32
__constant__ float mask[MASK_WIDTH][MASK_WIDTH][MASK_WIDTH];

__global__ void conv2d(float *input, float *output, const int y_size, const int x_size) {

 __shared__ float inputTile [TILE_WIDTH][TILE_WIDTH-1][TILE_WIDTH-1];
 int tx = threadIdx.x;
 int ty = threadIdx.y;
 int row = blockIdx.y * TILE_WIDTH + ty;
 int col = blockIdx.x * TILE_WIDTH + tx;
 int radius = MASK_WIDTH/2;

```

```

float output = 0.0f;

if ((row < y_size) && (col < x_size)) {
 inputTile[ty][tx] = input[_____];
}
else{
 inputTile[ty][tx] = 0.0f;
}

__syncthreads();

if (row < size_y && col < size_x) {

 int This_tile_start_point_x = blockIdx.x * blockDim.x;
 int Next_tile_start_point_x = (blockIdx.x + 1) * blockDim.x;

 int This_tile_start_point_y = blockIdx.y * blockDim.y;
 int Next_tile_start_point_y = (blockIdx.y + 1) * blockDim.y;

 int input_start_point_x = _____;
 int input_start_point_y = _____;

 float Pvalue = 0;
 for (int i = 0; i < MASK_WIDTH; i++) {
 int input_index_y = N_start_point_y + i;
 for (int j = 0; j < MASK_WIDTH; j++) {
 int input_index_x = N_start_point_x + j;
 if (input_index_y >= This_tile_start_point_x
 && input_index_y < Next_tile_starting_point_y
 && input_index_x >= This_tile_start_point_x
 && input_index_y < Next_tile_start_point_y) {
 Pvalue += inputTile[____][____]*mask[____][____];
 } else {
 Pvalue += input[____] * mask[____][____];
 }
 }
 }
 P[i] = Pvalue;
}

```

- (D) Compare the pros and cons of the new Strategy 3 kernel vs. a Strategy 2 kernel.

Answer:

**Answers:**

- (A) All edge elements of the output square except for the corner ones will have to access three of their input elements from the global memory. The corner output elements will have to access five of their input elements from the global memory. Thus the total number of global memory accesses during the convolution calculation will be

$$3 * (\text{TILE\_WIDTH}-2) * 4 + 5 * 4 = 3 * 30 * 4 - 20 = 340$$

The total number of memory accesses served by the shared memory is thus

$$32 * 32 * 3 * 3 - 340 = 9216 - 340 = 8,876$$

The average number of reuses for each input element loaded into the shared memory is thus

$$8,876 / (32 * 32) = 8.67$$

Another way to look at it is that total number of global memory accesses made for calculating the 32x32 output elements is

$$1024 + 340 = 1,364$$

So the average number of global accesses for input elements per output element calculation is

$$1,364 / 1,024 = 1.33$$

It is interesting to compare this with Strategy 2, where the average number of global memory accesses for input elements per output element calculation is

$$1,024 / ((32-2) * (32-2)) = 1,024 / 900 = 1.14$$

So Strategy 2 is slightly better than strategy 3. (However, if some of the accesses to halo elements are serviced by the L2 cache, this benefit diminishes quite quickly.)

- (B)  $\text{ceil}(512/32.0) * \text{ceil}(768/32.0) = 16 * 24 = 384$

- (c)

#define MASK\_WIDTH 3

```

#define TILE_WIDTH 32
__constant__ float mask[MASK_WIDTH][MASK_WIDTH][MASK_WIDTH];

__global__ void conv2d(float *input, float *output, const int y_size, const int x_size) {

 __shared__ float inputTile [TILE_WIDTH][TILE_WIDTH-1][TILE_WIDTH-1];
 int tx = threadIdx.x;
 int ty = threadIdx.y;
 int row = blockIdx.y * TILE_WIDTH + ty;
 int col = blockIdx.x * TILE_WIDTH + tx;
 int radius = MASK_WIDTH/2;

 float output = 0.0f;

 if ((row < y_size) && (col < x_size)) {
 inputTile[ty][tx] = input[row*x_size + col];
 }
 else{
 inputTile[ty][tx] = 0.0f;
 }

 __syncthreads();

 If (row < size_y && col < size_x) {

 int This_tile_start_point_x = blockIdx.x * blockDim.x;
 int Next_tile_start_point_x = (blockIdx.x + 1) * blockDim.x;

 int This_tile_start_point_y = blockIdx.y * blockDim.y;
 int Next_tile_start_point_y = (blockIdx.y + 1) * blockDim.y;

 int input_start_point_x = col - radius;
 int input_start_point_y = row - radius;

 float Pvalue = 0;
 for (int i = 0; i < MASK_WIDTH; i++) {
 int input_index_y = N_start_point_y + i;
 for (int j = 0; j < MASK_WIDTH; j++) {
 int input_index_x = N_start_point_x + j;
 if (input_index_y >= This_tile_start_point_x
 && input_index_y < Next_tile_start_point_y
 && input_index_x >= This_tile_start_point_x
 && input_index_x < Next_tile_start_point_x) {

```

```

 Pvalue += inputTile[ty-radius+i][tx-radius+j]*mask[i][j];
 } else {
 Pvalue += input[input_index_y*x_size + input_index_x] * mask[i][j];
 }
}
P[i] = Pvalue;
}
}

```

(D)

Pros:

1. Simple input tile loading code
2. Better thread utilization
3. Simple output element writing code

Cons:

1. More complex calculation code
2. More global memory accesses

# ECE408/CS483 Exam #1, Fall 2016

Tuesday, October 25, 2016

- You are allowed to use any notes, books, papers, and other reference material as you desire.
- No interactions with humans other than course staff are allowed.
- This exam is designed to take 120 minutes to complete. To eliminate time pressure and allow for any unforeseen difficulties, we will give everyone 180 minutes.
- This exam is based on lectures as well as lab MPs/projects.
- The questions are randomly selected from the topics we covered up to and including parallel scan.
- You can write down the reasoning behind your answers for possible partial credit.
- **Good luck!**

### **Question 1 (30 points, suggested time allocation 30 minutes): Short Answer**

Answer each of the following questions in as few words as you can. Your answer will be graded based on completeness, correctness, and conciseness.

(Question about very basic CUDA programming skills)

1. (3 points) If we want to copy host array `h_A` (`h_A` is a pointer to element 0 of the source array) to device array `d_A` (`d_A` is a pointer to element 0 of the destination array), what would be an appropriate API call for this in CUDA? Assume that array `h_A` and array `d_A` each consists of 2,048 single-precision floating elements.

- (A) `cudaMemcpy(2048*sizeof(float), h_A, d_A, cudaMemcpyHostToDevice);`
- (B) `cudaMemcpy(h_A, d_A, 2048, cudaMemcpyDeviceTHost);`
- (C) `cudaMemcpy(d_A, h_A, 2048, cudaMemcpyHostToDevice);`
- (D) `cudaMemcpy(d_A, h_A, 2048*sizeof(float), cudaMemcpyHostToDevice);`

(Question about 1D kernel launch)

2. (3 points) We want to use each thread to calculate eight (8) output elements of a vector addition. Each thread block process  $8 * \text{blockDim.x}$  consecutive elements that form 8 sections. All threads in each block will first process a section first with each processing one element. They will then all move to the next section with each processing one element. What would be the kernel code expression for forming the value of `i`, the data index of the first element to be processed by each thread?

- (A) `i=blockIdx.x*blockDim.x + threadIdx.x +2;`
- (B) `i=blockIdx.x*threadIdx.x*2`
- (C) `i=blockIdx.x*blockDim.x*8 + threadIdx.x`
- (D) `i=blockIdx.x*blockDim.x*2 + threadIdx.x*8`

3. (3 points) For a vector addition, assume that the vector length is 8000, each thread calculates eight (8) output element, and the thread block size is 512 threads. The programmer configures the kernel launch to have a minimal number of thread blocks to cover all output elements. How many threads will be in the grid?

- (A) 1024
- (B) 8196
- (C) 8192
- (D) 8200

4. (3 points) We are to process an 800X565 (784 pixels in the x or horizontal direction, 565 pixels in the y or vertical direction) picture with the PictureKernel below:

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m) {

 // Calculate the row # of the d_Pin and d_Pout element to process
 int Row = blockIdx.y*blockDim.y + threadIdx.y;

 // Calculate the column # of the d_Pin and d_Pout element to process
 int Col = blockIdx.x*blockDim.x + threadIdx.x;

 // each thread computes one element of d_Pout if in range
 if ((Row < m) && (Col < n)) {
 d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
 }
}
```

Assume that each block is organized as a 2D 16X16 array of threads. Which of the following statements sets up the kernel configuration properly? Assume that int variable n has value 784 and int variable m has value 565. The kernel is launched with the statement PictureKernel<<<DimGrid,DimBlock>>>(d\_Pin, d\_Pout, n, m);

- (A) dim3 DimGrid(ceil(1, ceil(n/16), ceil(m/16)); dim3 DimBlock(1, 16, 16);
- (B) dim3 DimGrid(ceil(n/16.0), ceil(m/16.0), 1); dim3 DimBlock(16, 16, 1);
- (C) dim3 dimGrid(ceil(m/16.0), ceil(n/16.0), 1); dim3 DimBlock(16,16,1);
- (D) dim3 DimGrid(ceil(m/16), ceil(n/16), 1); dim3 DimBlock(16, 16, 1);

5. In Question 4, how many warps will have control divergence?

- (A)  $37*16 + 50$
- (B)  $38*16$
- (C) 49
- (D) 0

(3 points) Write some CUDA kernel code in which the threads in the same block rotate the elements of an array **list** in shared memory in-place. That is, element `list[i]` after rotation should be element `list[i-1]` before rotation and element `list[0]` should be element `list[255]` before. You can assume that each block has exactly 256 threads. Make your code as efficient in time and space as possible.

```
__shared__ float list[256]
```

3. (5 points) For a 1D tiled convolution kernel (Lecture 9), assume that we use a block size of 1024 and a mask\_width of 9. What is the average number of times each data element of an internal tile is reused from the Shared Memory?
4. (5 points) Your kernel runs on a device with compute capability 2.0. The kernel resource usage is 30 registers/thread and 5KB of shared memory per block. The kernel has 1,920,000 threads in total and a square grid with dimension 50 on each side. What is the maximum number of simultaneous blocks that will run on a single SM?
5. (5 points) Assume that a kernel has an atomic operation on a variable in the global memory. If you know that each load or store access to the global memory takes 1,000 clock cycles and the clock runs at the 1 GHz. What is the maximal throughput one can hope for the atomic operations on this variable?
6. (5 points) Assume a DRAM system with a burst size of 512 bytes and a peak bandwidth of 240 GB/s. Assume a thread block size of 1024 and warp size of 32 and A as a double-precision array in the global memory. What is the maximal memory bandwidth we can hope to achieve? Explain.

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

```
double temp = A[8*i];
```

7. (bonus 3 points) List the errors and typos that you reported via Piazza postings, e-mails, or in person communication with Prof. Hwu.

**Question 2 (15 points, suggested time allocation 20 minutes):** CUDA Basics. For the vector addition kernel and the corresponding kernel launch code, answer each of the sub-questions below.

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
 1. int i = threadIdx.x + blockDim.x * blockIdx.x;
 2. if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vectAdd(float* A, float* B, float* C, int n)
{
 //assume that size has been set to the actual length of
 //arrays A, B, and C
 3. int size = n * sizeof(float);
 4.
 5. cudaMalloc((void **) &A_d, size);
 6. cudaMalloc((void **) &B_d, size);
 7. cudaMalloc((void **) &C_d, size);
 8. cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
 9. cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

10. vecAddKernel<<<ceil(n/1024.0),1024>>>(A_d, B_d, C_d, n);

11. cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
}
```

- 2(a). (2 point) Assume that the size of A, B, and C is 10,000 elements each. How many thread blocks will be generated?

Answer:

- 2(b). (2 point) Assume that the size of A, B, and C is 10,000 elements each. How many warps are there in each block?

Answer:

- 2(c) (2 point) Assume that the size of A, B, and C is 10,000 elements. How many threads will be created in the grid?

Answer:

- 2(d) (4 points) Assume that the size of A, B, and C is 10,000 elements each. Is there any control divergence during the execution of the kernel? If so, identify the block number and warp number that causes the control divergence. Explain why or why not.
- 2(e). (5 points) Assume that the size of A, B, and C is 20,000 elements each. Is there any control divergence during the execution of the kernel? If so, identify the line number of the statement that causes the control divergence. Explain why or why not.

**Question 3. (15 points, suggested time allocation 20 minutes):** The following CUDA code is intended to perform a sum of all elements of the **partialSum** array. Assume that the kernel is launched with 1024 threads in each block. First, provide the missing operation, and second estimate the fraction of the iterations of the **for** loop that will have branch divergence.

```
__shared__ float partialSum[2048];
unsigned int tid = threadIdx.x;

for (unsigned int stride = blockDim.x;
 stride > 0;
 stride = stride / 2)
{
 __syncthreads();
 if (tid < stride)
 partialSum[tid] += partialSum[tid+stride];
}
```

3(a). (2 pts) How many versions of `partialSum[2048]` will be created for each block?

Answer:

3(b). (2 pts) How many iterations will the for-loop take?

Answer:

3(b). (3 pts) Under what condition will there be control divergence in the if-statement?  
Assume that the warp size is 32.

Answer:

3(c). (2 pts) How many iterations of the for-loop will have control divergence in the if-statement?

Answer:

3(d). (3 pts) How many warps have control divergence in the for-loop where `stride` value is 16?

Answer:

3(e). (3 pts) In each block, how many warps have control divergence in the for-iteration where `stride` value is 4?

Answer:

**Question 4. (20 points, suggested time allocation 40 minutes):** Consider the following fragment of C code:

```
for(unsigned int x = 0; x < 512; ++x) {
 for(unsigned int y = 0; y < 512; ++y) {
 for(unsigned int z = 0; z < 512; ++z) {
 out[x][y] = out[x][y] <OP> in[x][y][z];
 }
 }
}
```

Explain how you would optimally parallelize this code and why if:

- (a)  $\langle OP \rangle$  was not associative
- (b)  $\langle OP \rangle$  was associative

You need to state to what you will assign your blocks and your threads to and why, then write out the kernel function code.

Assume  $out[x][y]$  is initialized correctly.

Part (a):

Kernel code:

Part (b):

Kernel code:

**Question 5. (20 points, suggested allocation of time 30 minutes).** This question tests your ability to handle boundary conditions in a tiled matrix multiplication algorithm. For simplicity, we will only handle square matrices.

Consider the following tiled matrix multiplication code:

```
01 #define TILE_WIDTH
02
03 __global__ void sgemm(float* M, float* N, float* P, int Width) {
04 __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
```

```

06 __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
07
08 int bx = blockIdx.x; int by = blockIdx.y;
09 int tx = threadIdx.x; int ty = threadIdx.y;
10
11 // Identify the row and column of the P element to work on
12 int Row = by * TILE_WIDTH + ty;
13 int Col = bx * TILE_WIDTH + tx;
14
15 float Pvalue = 0;
16 // Loop over the M and N tiles required to compute P element
17 for (int m = 0; m < _____; ++m) {
18
19 // Collaborative load of M and N tiles into shared memory
20 if(_____) {
21 Mds[ty][tx] = M[Row*Width + m*TILE_WIDTH + tx];
22 Nds[ty][tx] = N[(m*TILE_WIDTH + ty)*Width + Col];
23 }
24 __syncthreads();
25
26 if(_____) {
27 for (int k = 0; k < _____; ++k) {
28 Pvalue += Mds[ty][k] * Nds[k][tx];
29 }
30 }
31 __syncthreads();
32 }
33 P[Row*Width + Col] = Pvalue;
34 }
```

Fill in the missing boundary checks on lines 17, 20, 26, and 27 to make this code run correctly. If you think it is impossible to make this code to run correctly as is, state why, and then make the necessary changes to fix it by inserting a maximum of 6 lines.

# **408: Applied Parallel Programming**

## **Fall 2018 – Midterm Exam 1**

October 9<sup>th</sup>, 2018

1. This is a closed book exam except for 1 sheet of hand-written notes
2. You may not use any personal electronic devices except for calculator
3. Absolutely no interaction between students is allowed
4. Illegible answers will likely be graded as incorrect

**Good Luck!**

**Name:** \_\_\_\_\_

**NetID:** \_\_\_\_\_

**Exam Room:** \_\_\_\_\_

Question 1 (25 points): \_\_\_\_\_

Question 2 (25 points): \_\_\_\_\_

Question 3 (25 points): \_\_\_\_\_

Question 4 (25 points): \_\_\_\_\_

**Total Score:** \_\_\_\_\_

## Problem 1 (25 points): Multiple Choice

Choose the proper response, and if multiple responses are correct, choose all. No partial credit will be provided if the answer is partially correct, or wrong.

**Part 1(a) (2 points)** We want to use each thread to calculate eight elements of a vector addition. Each thread block process  $8 * \text{blockDim.x}$  consecutive elements that form eight sections. All threads in each block will first process a section first, each processing one element. They will then all move to the next section, each processing one element. Assume that variable  $i$  should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?

- $i = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x} + 8$
- $i = (\text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}) * 8$
- $i = \text{blockIdx.x} * \text{blockDim.x} * 8$
- $i = \text{blockIdx.x} * \text{blockDim.x} * 8 + \text{threadIdx.x}$
- None of the above

**Part 1(b) (2 points)** What are the scopes of shared memory and barrier synchronization?

- per block; per warp
- per warp; per warp
- per block; per block
- per SM; per block
- None of the above

**Part 1(c) (2 points)** For a vector addition, assume that the vector length is 9000, each thread calculates 9 output elements, and the thread block size is 256 threads. The programmer configures that kernel launch to have a minimal number of thread blocks to cover all output elements. How many thread will be created in the grid?

- 1000
- 900
- 1024
- 2000
- 2048
- 8292

**Part 1(d) (2 points)** If a CUDA device's SM (streaming multiprocessor) can take up to 1536 threads and up to 8 thread blocks. Which of the following block configuration would result in the most number of threads in the SM?

- 64 threads per block
- 128 threads per block
- 256 threads per block
- 1024 threads per block

**Part 1(e) (2 points)** We would like to launch a matrix multiplication kernel to multiply an 80 X 96 matrix M and a 96 X 40 matrix N, using 16 X 16 thread blocks. How many blocks will be launched if each thread is responsible for four elements?

- 4
- 6
- 8
- 10

**Part 1(f) (2 points)** For a tiled-matrix multiplication kernel, if we use a 16 X 16 tile, what is the reduction of memory bandwidth usage for input matrices M and N?

- 1/8 of original usage
- 1/16 of original usage
- 1/32 of original usage
- 1/64 of original usage

**Part 1(g) (3 points)** For a 1D tiled convolution kernel using shared memory (using strategy 2), assume that we use a block size of 1024 and a mask\_width of 7. What is the average number of times each data element of an internal tile (no ghost cells) is reused from the shared memory? If you don't have a calculator, provide an expression that indicates how to derive the answer

- 6.86
- 6.96
- 8.93
- 8.89

**Part 1(h) (3 points)** For a tiled 3D convolution, assume that we load an entire input tile, including the halo elements into the shared memory when calculating an output tile.

Further assume that the tiles are internal and thus do not involve any ghost elements. The mask is a 3x3x3 cube and each output tile is a 4x4x4 cube. Which of the choices is the closest to the average number of times each input element will be accessed from the shared memory during the calculation of an output tile?

- 32
- 16
- 8
- 4

**Part 1 (i) (3 points)** In matrix multiplication, suppose we use 16 X 32 rectangular tiles to process output matrices of 1000 X 1200. Within EACH thread block, what are the possible number of warps that will have control divergence due to handling boundary conditions? For example, the response 32, 16, 0 means that each thread block will have one of 32, 16 or 0 divergent warps. Chose the answer that is most precise.

- 32, 16, 0
- 32, 8, 0
- 32, 0
- 16, 8, 0
- None of the above

**Part 1(j) (4 points)** For a tiled 2D convolution, assume that we load an entire input tile, including the halo elements into the shared memory when calculating an output tile. Assume we use Strategy 2. The mask is a 3x3 square and each output tile is a 4x4 square. Which of the choices is the closest to the average number of different blocks any particular input element will be accessed by?

- 40/16
- 36/8
- 9/4
- 3/2

## Solution:

1.a D  
1.b C  
1.c C  
1.d C  
1.e B  
1.f B  
1.g B  
1.h C  
1.i D  
1.j C

## Problem 2 (22 points): Column Permutation

In real-world applications, programmers may need to reorder columns in a matrix based on a specified permutation vector. For example, if the original matrix **idata** is [0 1 2 3 6 4 7 5 8 ] and the permutation vector (**perms** in the code; the length equal to number of columns in the input/output matrix) is [2 0 1 ], the result matrix **odata** should be [2 0 1 4 3 6 8 7 5 ] after the column permutation. The “2” in **perms**[0] vector means the column **odata**[0] comes from the column 2 in **idata** and so forth. Please answer the following questions.

**Part 2(a) (5 points):** To do this on GPU, please fill in the missing index calculations in the CUDA code below. The matrices are stored in one-dimensional arrays in the row-major layout. In the following code, we only handle square input/output matrix (**rows = cols**) with **rows/cols** as multiples of 32.

```
1. #define BLOCK_WIDTH 32
2. // Kernel Code
3. __global__
4. void column_reorder(float *odata, float *idata, int *perms,
5. int cols, int rows)
6. {
7. int x = blockIdx.x * BLOCK_WIDTH + threadIdx.x;
8. int y = blockIdx.y * BLOCK_WIDTH + threadIdx.y;
9.
10. odata[_____] = idata[_____];
11. }
12.
13. // Host Code
14. int main ()
15. {
16. ...
}
```

```
17. // Invoke Kernel Here
18. dim3 dimGrid(cols/BLOCK_WIDTH, rows/BLOCK_WIDTH, 1);
19. dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH, 1);
20. column_reorder<<<dimGrid, dimBlock>>>(d_odata, d_idata, d_perms,
21. cols, rows);
22. ...
23. }
```

Odata blank:  $y * \text{cols} + x$  (2 points)  
Idata blank:  $y * \text{cols} + \text{perms}[x]$  (3 points)

**Part 2(b) (5 points):** For the access pattern of **idata** and **odata**, which of them will have coalesced access, or both, or neither? Please explain.

Access to odata (2.5 points) is coalesced while access to idata (2.5 points) is not. + reasonable explanation

**Part 2(c) (5 points):** Consider the case where we decide to put the **perm** vector in shared memory and call it **perm\_shared**. Assuming the numbers in the vector are 32-bits integers, what is the minimum amount of shared memory in bytes we need to allocate for each block with the block and grid dimensions shown in the code line 18 and line 19. Show your computation to help explain your answer.

BLOCK\_WIDTH\*4 bytes (5 points)  
Anything greater than that (1 point)

**Part 2(d) (2 points):** For question 2(c), how many total blocks will be allocated? Show your computation to help explain your answer.

(`cols/BLOCK_WIDTH`) \* (`row/BLOCK_WIDTH`) (2 points)

**Part 2(e) (5 points):** We now need to make sure this code can be applied on rectangular matrices and arbitrary `BLOCK_WIDTH`. What changes do you need to make on the original code? You can modify, add, or delete lines in the original code, both kernel side and host side. Please note that you may not need all the empty lines below and overly complex answers will result in lost points.

Example:

[Add between line 1 and line 2]: `_syncthreads()`

[Add between line 7 & 8]: `if (x < cols && y < rows)` (2.5 points)

[Change line 15 to]:

`dimGrid(ceil(1.0*cols/BLOCK WIDTH).ceil(1.0*rows/BLOCK WIDTH), 1);` (2.5 points)

[\_\_\_\_\_]: \_\_\_\_\_

[\_\_\_\_\_]: \_\_\_\_\_

[\_\_\_\_\_]: \_\_\_\_\_

[\_\_\_\_\_]: \_\_\_\_\_

## Problem 3 (28 points): Convolution

For this question, we will developed a tiled 2D convolution where each thread is responsible for computing 2 consecutive output elements in the x dimension. This means each thread block will need to compute 2 output tiles in the resulting output. Each thread block should load enough data from the input into the shared memory for computing its output without loading any row or column of the input tile more than once. For this problem we will use Strategy 2, where all threads participate in loading shared memory, and some of the threads participate in generating output. For this question, the mask is 3x3 and each thread block consists of 5x5 threads.

**Part 3(a) (15 points):** Fill in the blanks in the code below to complete the kernel.

```
1. #define MASK_WIDTH 3
2. #define MASK_RADIUS 1
3. #define BLOCK_WIDTH 5
4. #define INPUT_TILE_WIDTH_X 8 (10)
5. #define OUTPUT_TILE_WIDTH_X 6 (8)
6.
7. #define INPUT_TILE_WIDTH_Y BLOCK_WIDTH
8. #define OUTPUT_TILE_WIDTH_Y MASK_WIDTH
9.
10. __constant__ float mask[MASK_WIDTH][MASK_WIDTH];
11.
12. __global__
13. void conv2d(float *input, float *output, int y_size, int x_size) {
14.
15. __shared__ float inputTile [INPUT_TILE_WIDTH_Y][INPUT_TILE_WIDTH_X];
16. int tx = threadIdx.x; int ty = threadIdx.y;
17. int bx = blockIdx.x; int by = blockIdx.y;
18.
19. //Calculate index of first output element
20. int first_x_o = bx * OUTPUT_TILE_WIDTH_X + (2*tx);
21. int first_y_o = by * OUTPUT_TILE_WIDTH_Y + ty ;
22.
23. //Calculate index of input element to put in shared memory
24.
25. int x_i = first_x_o - tx - MASK_RADIUS; (or MASK_RADIUS)
26. int y_i = first_y_o - MASK_RADIUS;
27. //Iterate to load whole input tile
28. for (int i = 0; i < 2; i++) {
29. //Calculate where to put element from input into shared memory
30.
31. int tile_x_idx = tx + + i * BLOCK_WIDTH; (or i + tx)
32. int tile_y_idx = ty;
33. if ((x_i >= 0) && (x_i < x_size) &&
34. (y_i >= 0) && (y_i < y_size) &&
35. (tile_x_idx < INPUT_TILE_WIDTH_X) &&
```

```

36. (tile_y_idx < INPUT_TILE_WIDTH_Y)) {
37.
38. inputTile[tile_y_idx][tile_x_idx] =
39. input[(y_i * x_size) + (x_i)];
40. }
41. else if ((tile_x_idx < INPUT_TILE_WIDTH_X) &&
42. (tile_y_idx < INPUT_TILE_WIDTH_Y)) {
43.
44. inputTile[tile_y_idx][tile_x_idx] = 0.0f;
45. }
46. //Determine which column to load the next iteration
47. x_i += BLOCK_WIDTH; (or 1)
48. }
49.
50.
51. //Iterate to compute 2 output elements per thread
52. for (int i = 0; i < 2; i++) {
53. float val = 0.0f;
54. int new_tx = 2*tx + i;
55. if (ty < OUTPUT_TILE_WIDTH_Y &&
56. new_tx < OUTPUT_TILE_WIDTH_X) {
57. for (int j = 0; j < MASK_WIDTH; j++)
58. for (int k = 0; k < MASK_WIDTH; k++)
59. val += mask[j][k] *
60. inputTile[j + ty][k + new_tx];
61. if (first_y_o < y_size && first_x_o < x_size)
62. output[first_y_o * x_size + first_x_o] = val;
63. }
64. //Determine next element in x dimension to compute
65.
66. first_x_o += 1;
67. }
68. }

```

Every line is worth 2 point each, except lines 65 is 1 points.

The bracket describes the alternate solution. Every student is given points based on the best of the two complete solution. i.e if the student uses all answer in brackets, he gets full points. A mixed solution will get you best of the two answers.

Example:

**MASK\_RADIUS, i\*BLOCK\_WIDTH, BLOCK\_WIDTH** will give you 4 points and not 6.

If you have mentioned 10,8 for lines 4 and 5→ request regrade for lines 4,5

The two solutions for lines 4 and 5 are independent of the solutions for the other lines

**Part 3(b) (5 points):** The code above is incorrect in that it contains no synchronization (i.e., `__syncthreads()`). Please provide pair(s) of line numbers between which

`__syncthreads()` is required for correct execution. For the sake of efficiency, we want to **execute** as few `__syncthreads()` as possible.

**Between 45 and 47 (only 2 points) Its correct but its not minimal number in execution**

**Anywhere Between 47 and 50 (full 5 points)**

**Part 3(c) (5 points):** For an internal output tile (no ghost elements), what is the average number of times each input element will be accessed from shared memory during the calculation of an output tile for the correctly working version of this code?

```
(OUTPUT_TILE_WIDTH_X * OUTPUT_TILE_WIDTH_Y) *
(MASK_WIDTH * MASK_WIDTH) /
(INPUT_TILE_WIDTH_X * INPUT_TILE_WIDTH_Y)
```

$(6*3) * (3 * 3) / (8 * 5) = 4.05$  if they used `INPUT_TILE_WIDTH_X` as 8 and `OUTPUT_TILE_WIDTH_X` as 6

$(8*3) * (3 * 3) / (10 * 5) = 4.32$  if they used `INPUT_TILE_WIDTH_X` as 10 and `OUTPUT_TILE_WIDTH_X` as 8

**Part 3(d) (3 points):** Which of the lines in the code in 3(a) might suffer from lack of memory coalescing?

**Lines 37-38 (reading from global memory) (1.5 points)**

**Line 61 (writing to global memory) (1.5 points)**

## Problem 4 (25 points): Machine Learning

**Part 4(a) (10 points):** Pied Piper is hiring interns with CUDA + ML experience for Silicon Valley Season 6. Richard Hendricks has a bunch of questions to check how knowledgeable you are.

Choose the proper response, and if multiple responses are correct, choose all. No partial credit will be provided if the answer is partially correct, or wrong. (Each carries 2 point)

1. A multi-layer perceptron can perfectly learn a linear function given enough training steps.

True

False

2. A single perceptron can compute the XOR function.

True

False

3. A 3 layer perceptron with 10 neurons in each layer has a total of X connections, requires Y weight parameters, and Z biases.

X = 200, Y = 200, Z = 200

X = 100, Y = 200, Z = 20

X = 200, Y = 200, Z = 10

X = 200, Y = 200, Z = 20

4. With back propagation, we are evaluating the gradient of the \_\_\_\_\_ relative to the \_\_\_\_\_.

loss function, weights

activation function, cost function

cost function, input

cost function, biases

5. With stochastic gradient decent, a mini-batch requires processing all inputs in the training set.

True

False

Neither

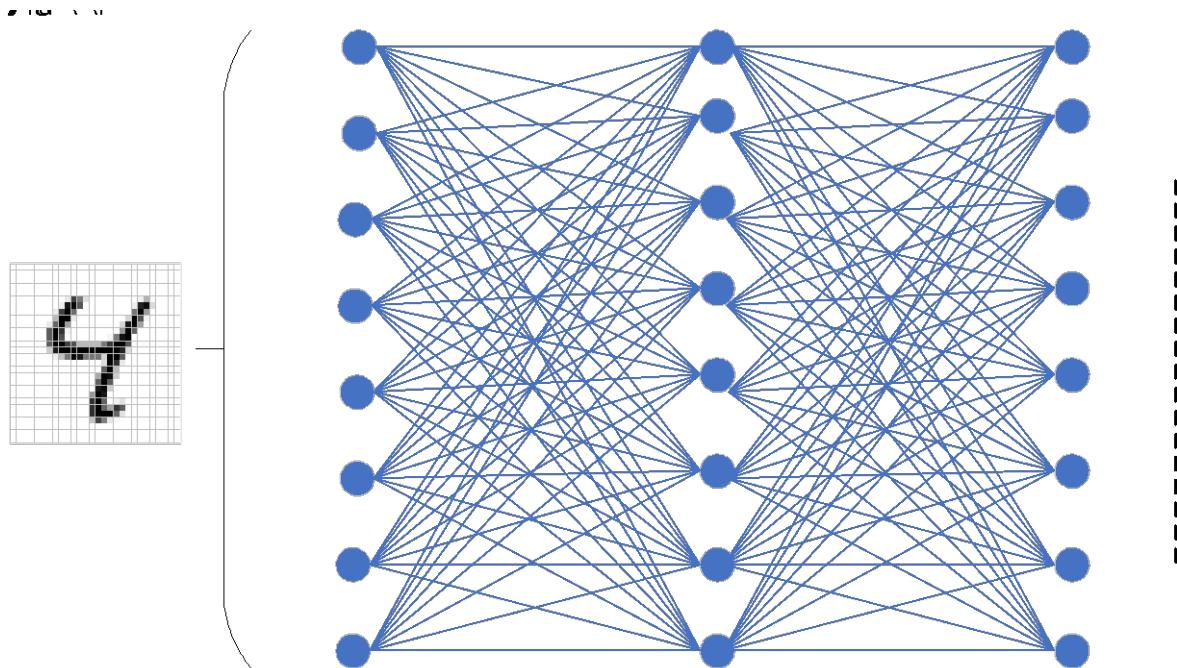
Q1.

1. True
2. False
3.  $X = 200, Y = 200, Z = 10$  or  $X = 200, Y = 200, Z = 20$  (request regrade if you wrote  $Z = 20$  - it based on assumption and the question is not clear)
4. A and D – Give 2 point only if both are mentioned. No point for 1 correct.
5. False

**Part 4(b) (10 points):** Congratulation, You are hired! Now, you are an intern working for Pied Piper! You work very closely with Richard Hendricks to pivot a digit recognition application for Season 6! Richard asks you to implement optimized multi-layer perceptron with 3 layers as shown in the figure. (NOTE: Image is not scaled to dimensions). It takes input  $x$  gray scale image of size  $32 \times 32$  and has 10 classes for output  $y$ , each representing a digit. The inputs and outputs are represented as linearized vectors,  $x$  and  $y$ . The hidden layer  $h$  has 100 neurons in it. The overall equation of the model can be given by

$$h = \sigma(1x + b_1)$$
$$y = \sigma(2h + b_2)$$

Where  $b_1$  and  $b_2$  are vectors holding the bias values, and  $W_1$  and  $W_2$  are weight matrices, and the function  $\sigma$  is the sigmoid function.



Fill in the dimensions in the following table, based on the architecture of the deep network:  
 (Each carries 2 points)

|    |                                          |  |
|----|------------------------------------------|--|
| Q1 | Dimension of $b_1$                       |  |
| Q2 | Dimension of $b_2$                       |  |
| Q3 | Dimension of $W_1$                       |  |
| Q4 | Dimension of $W_2$                       |  |
| Q5 | Total number of parameters to be learned |  |

Q2.

Initial math:

$$\text{First layer output} - 100 = [100,1024] * [1024] + 100$$

$$\text{Second layer output} - 10 = [10,100] * [100] + 10$$

Total parameters:

$$100*1024 = 102400 + 100 = 102500$$

$$10*100 = 1,000 + 10 = 1,010$$

$$\text{Total} = 1010 + 102500 = 103510$$

|    |                                          |                      |
|----|------------------------------------------|----------------------|
| Q1 | Dimension of $b_1$                       | 100                  |
| Q2 | Dimension of $b_2$                       | 10                   |
| Q3 | Dimension of $W_1$                       | 100,1024 or 1024,100 |
| Q4 | Dimension of $W_2$                       | 10,100 or 100,10     |
| Q5 | Total number of parameters to be learned | 103510               |

**Part 4(c) (5 points):** You realize the both forward-pass equations (for  $h$  and  $y$ ) are the same computation, but with different input dimensions. You want to use a single GPU kernel general enough to perform both. You can disregard the sigmoid function  $\sigma$  for this

question. Please complete below code base to complete the implementation. More credit will be given to code that is better optimized. Assume **x** is the input vector, **w** is the weight matrix in row-major order, **b** is the bias vector, and **y** is the output vector.

```
1. __global__
2. void general_layer(float *y, const float *x, const float *w, const
 float *b, const int ySize, const int xSize) {
3.
4. int tx = blockDim.x * blockIdx.x + threadIdx.x;
5. int bx = gridDim.x * blockDim.x;
6.
7. for(int j = tx; j < ySize; j+= bx){
8. float sum =0;
9. for(int i =0; i< xSize; i++){
10.
11. sum += x[_____] * w[_____];
12. }
13.
14. y[_____] = sum + b[_____];
15. }
16. }
```

Q3.

Answer:

Line 10:      i                i \* ySize + j

Line 12:      j                j

Reason: We use row-major approach which has coalesced access.

why ySize ---> it is W\*x calculation and not x\*W.

4 points for fill up the blank. (each blank carries 1 points)

1 point if they have used row major approach to fill it up.

# **408: Applied Parallel Programming**

## **Spring 2019 – Midterm Exam 1**

February 26<sup>th</sup>, 2019

1. This is a closed book exam except for 1 sheet of hand-written notes
2. You may not use any personal electronic devices except for a calculator
3. Absolutely no interaction between students is allowed
4. Illegible answers will likely be graded as incorrect

**Good Luck!**

**Name:** \_\_\_\_\_

**NetID:** \_\_\_\_\_

**Exam Room:** \_\_\_\_\_

Question 1 (24 points): \_\_\_\_\_

Question 2 (18 points): \_\_\_\_\_

Question 3 (30 points): \_\_\_\_\_

Question 4 (28 points): \_\_\_\_\_

**Total Score:** \_\_\_\_\_

Name: \_\_\_\_\_

NetID: \_\_\_\_\_

## Problem 1 (26 points): Multiple Choice

Choose the proper response, and **if multiple responses are correct, choose all**. No partial credit will be provided if the answer is partially correct, or wrong.

**Part 1a (2 points)** A particular CUDA device's streaming multiprocessor (SM) can take up to 1536 threads and up to 4 thread blocks. Which of the following block configurations would result in the most number of total threads in the SM?

- a. 256 threads per block
- b. 384 threads per block
- c. 512 threads per block
- d. 1024 threads per block
- e. Either (a) or (b) depending on whether block dimension is a power of 2.

**Part 1b (2 points)** For a vector addition, assume that the vector length is 4000, each thread calculates 10 output elements, and each block contains 64 threads. The programmer configures that kernel launch to have a minimal number of thread blocks to cover all output elements. How many threads will be created in the grid?

- 384
- 448
- 640
- 3840
- 4000
- 4480
- None of the above

**Part 1c (2 points)** A CUDA kernel is launched with 1000 thread blocks each of which has 512 threads. If a variable is declared as a local variable in the kernel, how many versions of the variable will be created through the lifetime of the execution of the kernel?

- 1
- 512
- 1000
- 512000
- None of the above

Name: \_\_\_\_\_

NetID: \_\_\_\_\_

**Part 1d (2 points)** Consider the following code in a CUDA kernel.

```
__global__ void do_work(int i, int *A)
{
 int result = 0;
 if (i < 5)
 result = threadIdx.x;

 A[threadIdx.x] = result;
}
```

- There is control divergence in this code
- There is no control divergence in this code
- The control divergence depends on the value of **i**

**Part 1e (2 points)** Consider the following code in a CUDA kernel.

```
__global__ void do_work(int i, int *A)
{
 int result = 0;

 for (j = 0; j<blockIdx.x; j++)
 result += j;

 A[threadIdx.x] = result;
}
```

- There is control divergence in this code
- There is no control divergence in this code
- Control divergence depends on the number of blocks in the x dimension

**Part 1f (2 points)** Consider the following code in a CUDA kernel.

```
__global__ void do_work(int i, int *A)
{
 int result = 0;

 for (j = 0; j<threadIdx.x; j++)
 result += j;

 A[threadIdx.x] = result;
}
```

- There is control divergence in this code
- There is no control divergence in this code
- Control divergence depends on the number of threads in the x dimension

Name: \_\_\_\_\_

NetID: \_\_\_\_\_

**Part 1g (2 points)** Consider the following statements, then select those are correct:

- i. All the threads in a CUDA warp execute the same instruction at the same time
- ii. Only one block on an SM can use the shared memory in that SM
- iii. CUDA constant memory is cached
- iv. Memory coalescing is an optimization to maximize memory utilization
- v. A `__syncthreads()` call synchronizes across all threads in all blocks

- ii, iii, iv, v
- i, iii, iv
- i. iii, iv, v
- ii, iii, iv
- i, ii, iii, iv, v

**Part 1h (2 points)** Consider a 3D video filtering (convolution) code in CUDA with a 3x3x5 mask, which is stored in constant memory. Shared memory is used to fully store the input tile required for a 16x16x16 output tile. What is the ratio of global memory loads to shared memory accesses for one output tile? For this question, only consider interior tiles with no ghost elements.

- 16\*16\*16 to 3\*3\*5\*16\*16\*16
- 18\*18\*20 to 16\*16\*16
- 15\*15\*12 to 16\*16\*16
- 15\*15\*12 to 3\*3\*5\*16\*16\*16
- 18\*18\*20 to 3\*3\*5\*16\*16\*16
- None of the above

**Part 1i (2 points)** Consider a DRAM system with a burst size of 512 bytes and a peak bandwidth of 240 GB/s. Assume a thread block size of 1024 and warp size of 32 and that **A** is a float array in the global memory. What is the maximal memory data access throughput we can hope to achieve in the following access to **A**?

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
float temp = A[4*i]+A[4*i+1];
```

- 240 GB/s
- 120 GB/s
- 60 GB/s
- 30GB/s

Name: \_\_\_\_\_

NetID: \_\_\_\_\_

**Part 1j (2 points)** The following vector addition kernel and launch code applies to parts (j) through (l)

```
1 __global__ void vecAddKernel(float* A, float* B, float* C, int n)
2 {
3 int i = threadIdx.x + blockDim.x * blockIdx.x * 2;
4
5 if (i < n) C_d[i] = A_d[i] + B_d[i];
6 i += blockDim.x;
7 if (i < n) C_d[i] = A_d[i] + B_d[i];
8 }
9
10 int vectAdd (float* A, float* B, float* C, int n)
11 {
12 int size = n * sizeof (float);
13 cudaMalloc ((void **) &A_d, size);
14 cudaMalloc ((void **) &B_d, size);
15 cudaMalloc ((void **) &C_d, size);
16 cudaMemcpy (A_d, A, size, cudaMemcpyHostToDevice);
17 cudaMemcpy (B_d, B, size, cudaMemcpyHostToDevice);
18
19 vecAddKernel<<<ceil(n/1024.0), 512>>> (A_d, B_d, C_d, n);
20 cudaMemcpy (C, C_d, size, cudaMemcpyDeviceToHost);
21 }
```

If the size of the vectors is 50,000 elements, identify the block number(s) that will have control divergence.  
(Assume the index is starting from 0)

- Block 47
- Block 48
- Both A and B
- Neither A or B

**Part 1k (2 points)** If the size of the vectors is 50,000 elements, which lines in the kernel code will experience control divergence?

- line 5
- line 7
- both line 5 and line 7
- None of the above

**Part 1l (2 points)** Again, consider the vector add kernel from part j. If the size of the vectors is *num* elements, identify the number of warps that will have control divergence.

- 0 or 1
- 1 or 2
- 0 or 2
- ceil(*num*/1024)
- None of the above

## Problem 2 (18 points): Matrix Multiply

Following is part of a tiled 2D matrix multiplication CUDA kernel, similar to the one in MP3. However, instead of calculating a single element, each thread calculates a 2x2 section of the output matrix. Adjacent threads calculate adjacent sections. For example, thread (0, 0) in the block (0, 0) would calculate the (0, 0), (0, 1), (1, 0), (1, 1) elements in the output matrix. Likewise thread (1, 0) in the block (0, 0) would calculate the (2, 0), (2, 1), (3, 0), (3, 1) elements in the output matrix. When loading the tiles into the shared memory, each thread in the block will also load a 2x2 section of the corresponding tile.

```

1. #define BLOCK_WIDTH 8
2. #define TILE_WIDTH 16
3. __global__
4. void matrixMultiplyShared(float *M, float *N, float *P,
5. int numMRows, int numMColumns, int numNRows,
6. int numNColumns, int numPRows, int numPColumns)
7. {
8. __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
9. __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
10.
11. int bx = blockIdx.x; int by = blockIdx.y;
12. int tx = threadIdx.x; int ty = threadIdx.y;
13.
14. // Identify the row and column of the first P element to work on
15. int Row = by * TILE_WIDTH + ty * 2;
16. int Col = bx * TILE_WIDTH + tx * 2;
17.
18. // Thread-local array to store 2x2 result of P
19. float Pvalue[2][2];
20. for (int j = 0; j < 2; ++j)
21. for (int i = 0; i < 2; ++i)
22. Pvalue[j][i] = 0;
23.
24. // number of iterations needed to loop over all tiles required
25. int ph_count = ceil(numMColumns/(float)TILE_WIDTH);
26.
27. // loop over the M and N tiles required to compute the P elements
28. for (int ph = 0; ph < ph_count; ++ph) {
29. // Collaborative loading of M and N tiles into shared memory
30. for (int j = 0; j < 2; ++j)
31. for (int i = 0; i < 2; ++i){
32. if ((Row+j) < numMRows && (ph*TILE_WIDTH+tx*2+i) < numMColumns)
33. Mds[ty*2+j][tx*2+i] = M[(Row+j)*numMColumns+ph*TILE_WIDTH+tx*2+i];
34. else
35. Mds[ty*2+j][tx*2+i] = 0;
36. if ((ph*TILE_WIDTH+ty*2+j) < numNRows && (Col+i) < numNColumns)
37. Nds[ty*2+j][tx*2+i] = N[(ph*TILE_WIDTH+ty*2+j)*numNColumns+Col+i];
38. else
39. Nds[ty*2+j][tx*2+i] = 0;
40. }
41. __syncthreads();
42. }

```

Name: \_\_\_\_\_

NetID: \_\_\_\_\_

```
43. // Calculate partial dot product for 2x2 array per thread
44. for (int k = 0; k < TILE_WIDTH; ++k) {
45. for (int j = 0; j < 2; ++j)
46. for (int i = 0; i < 2; ++i)

47. Pvalue[j][i] += Mds[_____][_____] * Nds[_____][_____];
48. }
49. __syncthreads();
50. } // ph loop
51.
52. for (int j = 0; j < 2; ++j)
53. for (int i = 0; i < 2; ++i)
54. if (Row + j < numPRows && Col + i < numPColumns)
55. P[(Row + j) * numNColumns + Col + i] = Pvalue[j][i];
56. }
57.
58. // Below are some host code to calculate grid_dim and block_dim

59. dim3 grid_dim(_____, _____, 1);
60. dim3 block_dim(BLOCK_WIDTH, BLOCK_WIDTH, 1);
```

**Part 2a (12 points)** Fill in the missing code in the 6 blanks above

**Line 47:** `Pvalue[j][i] += Mds[ty * 2 + j][k] * Nds[k][tx * 2 + i]`

**Line 59:** `dim3 grid_dim(ceil(numPColumns/(float)TILE_WIDTH),  
ceil(numPRows / (float)TILE_WIDTH), 1);`

2 point/blank

Sorry, no partial credit for line 47.

For line 59, to get full credit (4 points), you need to get three things correct.

- First is to correctly use floating point arithmetic so the result won't be off by 1 or stay as floating-point numbers.
- Second is to divide the Columns/Rows by the correct number, which could be TILE\_WIDTH, or BLOCK\_WIDTH \* 2, or 16, or any equivalent equations.
- Third is to get the order of Columns/Rows correctly. It should be numPColumns or numNColumns for the first blank and numPRows or numMRows for the second column.

If you get two of the above three points correct, you get 2-point partial credit. If you only get one correct, sorry you get nothing.

Name: \_\_\_\_\_

NetID: \_\_\_\_\_

**Part 2b (2 points)** Suppose we are using the above code to multiply two matrices M and N, where M is a  $80 * 104$  matrix and N is a  $104 * 80$  matrix (here, an  $m * n$  matrix means a matrix with  $m$  elements in the y or vertical direction and  $n$  elements in the x or horizontal direction), how many warps in the whole grid will have control divergence during the loading of **Mds**? (Warp size is 32 threads)

- 0
- 25
- 50
- 100
- None of the above

Explanation: Since  $104/16 = 6.5$ , each block needs to go through 7 tiles in each of matrix M and N. Consider a tile in the last column in matrix M. Only the first 8 columns in the tile corresponds to a valid element in the matrix. Since block size is  $8 * 8$  here, each block consists of 2 warps each with 4 rows and 8 columns. Given the fact that each thread loads a  $2 * 2$  section from the tile, only the threads in the first four columns of each warp will load from the matrix into the shared memory, causing control divergence in every warp when loading Mds when  $ph\_count = 6$ . Since the grid dimension is  $(80/16, 80/16) = (5, 5)$ , there are 25 blocks total each with 2 warps with control divergence. Thus, there are 50 warps with control divergence in total.

**Part 2c (2 points)** How many floating point operations (ADDS and MULTs) will each thread perform?

- $2 * \text{numMColumns}$  ADDs and  $2 * \text{numMColumns}$  MULTs
- $2 * \text{numNColumns}$  ADDs and  $2 * \text{numNColumns}$  MULTs
- $4 * \text{numMColumns}$  ADDs and  $4 * \text{numMColumns}$  MULTs
- $4 * \text{numNColumns}$  ADDs and  $4 * \text{numNColumns}$  MULTs
- None of the above

Note: Points are given for “None of the above” as `numMColumns` might not divide `TILE_WIDTH`, causing extra calculations.

**Part 2d (2 points)** How many loads from global memory will each thread perform?

- $ph\_count * 2$
- $ph\_count * 4$
- $ph\_count * 8$
- $ph\_count * 16$
- None of the above

Note: Points are given for “None of the above” as there might be edge cases.

Name: \_\_\_\_\_

NetID: \_\_\_\_\_

**Problem 3 (30 points): Separable 2D Convolution**

Under certain conditions, the 2D mask of a 2D convolution can be decomposed to two 1-D masks as shown in the below example. This is called a separable convolution.

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * A = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * [-1 \ 0 \ 1] * A$$

For convolutions that are separable, we can apply the two 1-D masks sequentially to the input matrix: first apply the horizontal 1-D mask to each element, generating an intermediate result. Then apply the vertical 1-D mask on the intermediate result to generate the output matrix. In the following questions, you will work with kernel code that uses shared memory tiles to compute a separable 2D convolution. It uses Strategy 1 to load a tile's worth of the input to shared memory, then it computes the horizontal 1-D convolution of size MASK\_WIDTHx1, and then the vertical convolution of 1xMASK\_WIDTH. In the code, **mask1** is the horizontal mask, **mask2** is the vertical mask.

**Part 3a (2 points):** Consider the following input array A, and a 3x3 mask, decomposed into two 1D masks

$$mask = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \text{ decomposed masks} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}, [-1 \ 0 \ 1], A = \begin{bmatrix} 3 & 2 & 5 & 8 & 1 & 4 \\ 2 & 4 & 7 & 1 & 2 & 6 \\ 6 & 9 & 6 & 5 & 9 & 1 \\ 0 & 4 & 7 & 2 & 8 & 5 \\ 9 & 3 & 1 & 3 & 4 & 8 \\ 7 & 4 & 9 & 2 & 1 & 4 \end{bmatrix}$$

Assume we are using a CUDA kernel has an output tile width of 2. Calculate values for tile (1,1), i.e., tile=  $\begin{bmatrix} 6 & 5 \\ 7 & 2 \end{bmatrix}$  after the horizontal mask =  $[-1 \ 0 \ 1]$  is applied.

**Answer:**  $\begin{bmatrix} -4 & 3 \\ -2 & 1 \end{bmatrix}$

**Part 3b (2 points):** Calculate the output values for tile (1, 1). That is, apply the vertical mask to the results from **part 3a**. Hint: ensure that all values used for this convolution have been first convolved by the horizontal mask.

**Answer:**  $\begin{bmatrix} -13 & 2 \\ -8 & 8 \end{bmatrix}$

Name: \_\_\_\_\_

NetID: \_\_\_\_\_

**Part 3c (20 points)** In the following questions, you will work with kernel code that uses shared memory tiles to compute a separable 2D convolution by applying two 1-D convolutions. It uses Strategy 1 to load a tile's worth of the input to shared memory, then it computes the horizontal 1-D convolution of size MASK\_WIDTHx1, and then the vertical convolution of 1xMASK\_WIDTH. In the code, **mask1** is the horizontal mask, **mask2** is the vertical mask. Fill in the blanks in the code to complete the kernel. There are 10 blanks in total.

```
1. #define MASK_WIDTH 5
2. #define MASK_RADIUS 2
3. #define INPUT_TILE_WIDTH 12
4. #define OUTPUT_TILE_WIDTH 8
5.
6.
7. __constant__ float mask1[MASK_WIDTH]; //horizontal mask
8. __constant__ float mask2[MASK_WIDTH]; //vertical mask
9. __global__
10. void Separable2DConv(float *input, float *output, int x_size, int y_size)
11. {
12. __shared__ float input_tile[INPUT_TILE_WIDTH][INPUT_TILE_WIDTH];
13. int tx = threadIdx.x; int ty = threadIdx.y;
14. int bx = blockIdx.x; int by = blockIdx.y;
15.
16. //output index
17. int row_o = by * OUTPUT_TILE_WIDTH + ty;
18. int col_o = bx * OUTPUT_TILE_WIDTH + tx;
19.
20. // load input tile into shared memory
21. int num_iters = 2 or ceil(INPUT_TILE_WIDTH/OUTPUT_TILE_WIDTH);
22. for(int i = 0; i< num_iters; i++){
23. for(int j = 0; j<num_iters; j++){
24. int row_i = row_o - MASK_RADIUS + i * OUTPUT_TILE_WIDTH;
25.
26. int col_i = col_o - MASK_RADIUS + j * OUTPUT_TILE_WIDTH;
27. int tile_y_idx = ty + i* OUTPUT_TILE_WIDTH;
28. int tile_x_idx = tx + j* OUTPUT_TILE_WIDTH;
29. if(tile_y_idx < INPUT_TILE_WIDTH && tile_x_idx < INPUT_TILE_WIDTH){
30. if(row_i>= 0 && row_i < y_size && col_i >= 0 && col_i < x_size)
31. input_tile[tile_y_idx][tile_x_idx] = input[row_i*x_size + col_i];
32. else
33. input_tile[tile_y_idx][tile_x_idx] = 0;
34. }
35. }
36. }
37. __syncthreads();
38.
39. float val;
40. for(int iter = 0; iter < num_iters; iter++){
41. val = 0.0;
42. int y_index = ty + iter*OUTPUT_TILE_WIDTH;
43. for(int k =0; k< MASK_WIDTH; k++)
44. if(y_index < INPUT_TILE_WIDTH)
45. val += mask1[k] * input_tile[y_index][tx+k];
46. if(y_index < INPUT_TILE_WIDTH)
```

Name: \_\_\_\_\_

NetID: \_\_\_\_\_

```
47.
48. input_tile[y_index][tx] = val;
49. }
50. val = 0.0;
51. for(int k =0; k< MASK_WIDTH; k++){
52. val += mask2[k] * input_tile[ty+k][tx];
53. }
54. if(row_o < y_size && col_o < x_size)
55. output[row_o * x_size + col_o] = val;
56. }
57.
58.
```

**Answer:** see the corresponding blanks.

**Part 3d (4 points)** The code above is incorrect in that it lacks synchronization. Please specify where `__syncthreads()` is required for correct execution, by stating which line numbers in the code the `__syncthreads()` should appear after. For the sake of efficiency, we want to execute as few `__syncthreads()` as possible. Hint: more than one `__syncthreads()` is required.

**Answer:** After line 45

After line 49 (after line 48 is also correct but less efficient).

**Part 3e (2 points)** Provide 1 possible advantage and 1 possible disadvantages of using separable masks over a standard 2D convolution?

**Answer:**

Pros: less computation, less constant memory required, etc

Cons: More divergence, more `__syncthreads()` required, not all masks are separable, etc

Any reasonable answer (not overly vague or incorrect) will be accepted.

Name: \_\_\_\_\_

NetID: \_\_\_\_\_

## Problem 4 (28 points): Machine Learning

**Part 4a (6 points):** Choose the proper responses for the questions below. No partial credit will be provided if the answer is partially correct, or wrong.

1. Mark **all** statements below that are true.

- A multi-layer perceptron can learn the XOR function
- A single-layer perceptron can learn the XOR function
- A convolutional layer has a smaller receptive field than a fully connected layer
- The learning process involves finding weights and biases that minimize the loss function

2. Mini-batch stochastic gradient descent generally converges to the optimized point faster than batched stochastic gradient descent.

- True
- False

3. Unlike classical machine learning algorithms, deep learning does not require hand-designed features.

- True
- False

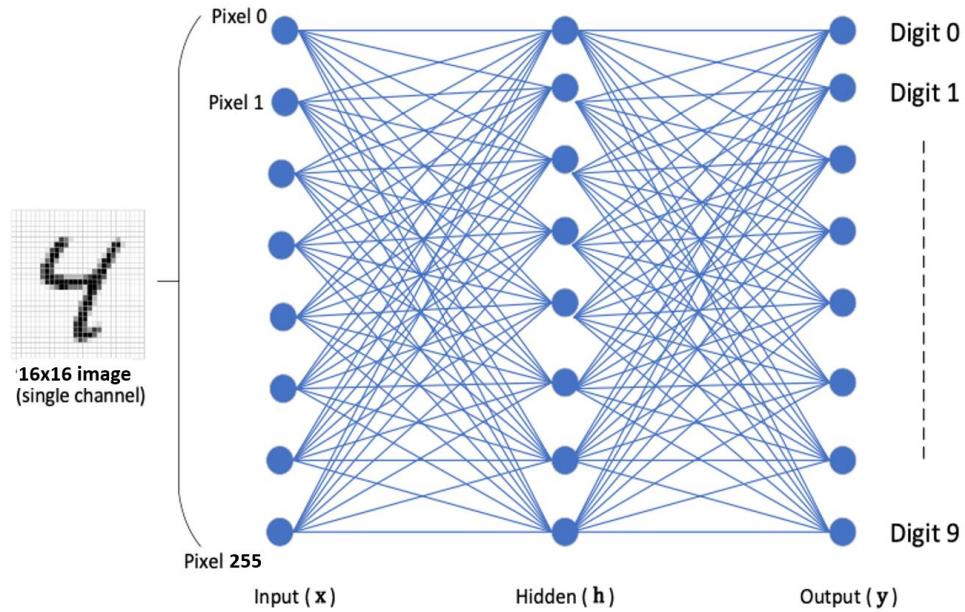
Name: \_\_\_\_\_

NetID: \_\_\_\_\_

**Part 4b (8 points):** You are required to implement optimized multi-layer perceptron with 3 layers as shown in the figure. (NOTE: Image is not scaled to dimension). It takes input  $\mathbf{x}$  gray scale image of size  $16 \times 16$  and has 10 classes for output  $\mathbf{y}$ , each representing a digit. The inputs and outputs are represented as linearized vectors,  $\mathbf{x}$  and  $\mathbf{y}$ . The hidden layer  $\mathbf{h}$  has 100 neurons in it. The overall equation of the model can be given by

$$\begin{aligned}\mathbf{h} &= \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) \\ \mathbf{y} &= \sigma(\mathbf{W}_2 \mathbf{h} + \mathbf{b}_2)\end{aligned}$$

Where  $\mathbf{b}_1$  and  $\mathbf{b}_2$  are vectors holding the bias values, and  $\mathbf{W}_1$  and  $\mathbf{W}_2$  are weight matrices, and the function  $\sigma$  is the sigmoid function.



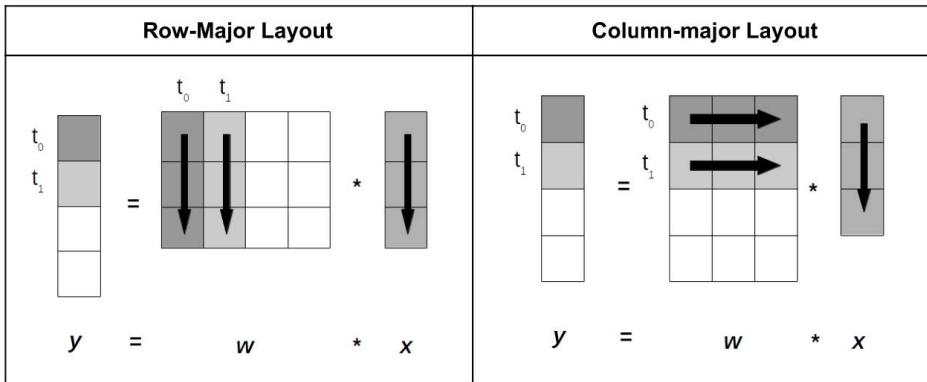
Fill in the dimensions in the following table, based on the architecture of the network:

|    |                             |                             |
|----|-----------------------------|-----------------------------|
| Q1 | Dimension of $\mathbf{b}_1$ | [ <u>100</u> , 1]           |
| Q2 | Dimension of $\mathbf{b}_2$ | [ <u>10</u> , 1]            |
| Q3 | Dimension of $\mathbf{W}_1$ | [ <u>100</u> , <u>256</u> ] |
| Q4 | Dimension of $\mathbf{W}_2$ | [ <u>10</u> , <u>100</u> ]  |

Name: \_\_\_\_\_

NetID: \_\_\_\_\_

**Part 4c (8 points):** You realize the both forward-pass equations (for  $\mathbf{h}$  and  $\mathbf{y}$ ) are the same computation, but with different input dimensions. You want to use a single GPU kernel general enough to perform both. You can disregard the sigmoid function  $\sigma$  for this question. Please complete below code base to complete the implementation in column-major layout weight matrix. The figure below shows the thread access pattern for the column major layout in a 4x3 example.



```

1. __global__
2. void fc_col(float *y, const float *x, const float *w, const float *b, const
 int ySize, const int xSize) {
3.
4. int tx = blockDim.x * blockIdx.x + threadIdx.x;
5. int gx = gridDim.x * blockDim.x;
6.
7. for(int o = tx; o < ySize; o += gx){
8. float sum =0;
9. for(int i =0; i< xSize; i++){
10.
11. sum += x[i] * w[o * xSize + i];
12. }
13.
14. y[o] = sum + b[o];
15. }
16. }
```

**Part 4d (2 points):** Based on the code in 4c, are the weight matrix accesses coalesced?

**Answer: No, if assuming row-major layout.**

**Yes, if assuming column-major layout.**

Name: \_\_\_\_\_

NetID: \_\_\_\_\_

**Part 4e (4 points):** Your partner for ECE 408 optimized the code in 4c by using shared memory for the input matrix (x). However, his implementation has bugs. What changes do you need to make on the code below to make it correct? You can modify, add, or delete lines in the code. Please note that you may not need all the empty lines below and overly complex answers will result in lost points. Assume all the indices with \$ means the same indices you answered in 4c.

```
1. __global__
2. void fc_col_shared(float *y, const float *x, const float *w, const float *b,
 const int ySize, const int xSize) {
3.
4. __shared__ x_shared[xSize];
5.
6. int tx = blockDim.x * blockIdx.x + threadIdx.x;
7. int gx = gridDim.x * blockDim.x;
8.
9. for(int s = tx; s < xSize; s += gx){
10. if (s < xSize)
11. x_shared[s] = x[s];
12. else
13. x_shared[s] = 0;
14. }
15.
16. for(int o = tx; o < ySize; o += gx){
17. float sum =0;
18. for(int i =0; i< xSize; i++){
19.
20. sum += x_shared[$] * w[$];
21. }
22.
23. y[$] = sum + b[$];
24. }
25. }
```

Example:

[Add between line 6 and line 7]: int bx = blockIdx.x;

[Add at line 15]: \_\_syncthreads();

[Modify at line 9]: s += gx to s+= blockDim.x

[Delete lines 10, 12, 13]: \_\_\_\_\_.

[\_\_\_\_\_]: \_\_\_\_\_.

# ECE 408 Exam #2 Study Guide, Fall 2019

## 1. Exam format

- You are allowed one 8.0x11.5 cheat sheet with notes on both sides. The minimal font size for your text on the cheat sheet should be 8pts.
- No interactions with humans other than course staff are allowed.
- This exam is designed to take 150 minutes to complete. To allow for any unforeseen difficulties, we will give everyone 180 minutes.
- This exam is based on lectures, textbook chapters, as well as lab MPs/projects.
- The questions are randomly selected from the topics we covered.
- You can write down the reasoning behind your answers for possible partial credit.

## 2. Topics to Review from Lectures

### Reduction

- Thread index to data index mapping and effect on control divergence
1. For the following basic reduction kernel code fragment, if the block size is 1024 and warp size is 32, how many warps in a **block** will have divergence during the iteration where stride is equal to 1?

```
unsigned int t = threadIdx.x;
Unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start+ blockDim.x+t];
for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2)
{
 __syncthreads();
 if (t % stride == 0) {partialSum[2*t]+= partialSum[2*t+stride];}
}
```

- (A) 0
- (B) 1
- (C) 16
- (D) 32

2. In the Question 1, how many warps in a block will have divergence during the iteration

where stride is equal to 16?

- (A) 0
- (B) 1
- (C) 16
- (D) 32

3. In the Question 1, how many warps in a block will have divergence during the iteration where stride is equal to 64?
  - (A) 0
  - (B) 1
  - (C) 16
  - (D) 32
4. For the following improved reduction kernel, if the block size is 1024 and warp size is 32, how many warps will have divergence during the iteration where stride is equal to 16?

```
unsigned int t = threadIdx.x;
Unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start+ blockDim.x+t];
for (unsigned int stride = blockDim.x; stride > 0; stride /= 2)
{
 __syncthreads();
 if (t < stride) {partialSum[t] += partialSum[t+stride];}
}
```

- (A) 0
- (B) 1
- (C) 16
- (D) 32

5. In the previous question, how many warps in a block will have divergence during the iteration where stride is 64?
  - (A) 0
  - (B) 1
  - (C) 16
  - (D) 32

1. Answer: (A)

Explanation: During the first iteration, all threads in each warp are active. There is no control divergence.

2. Answer: (D)

Explanation: During each iteration, 1/stride of the threads in each warp are active. When stride is 16, every warp will have  $32/16 = 2$  active threads that execute the addition statement. All 32 warps will have control divergence.

2. Answer: (C)

Explanation: There will be one active thread in every 64 threads or 2 warps. So, 1/2 of the warps or 16 warps have divergence. The other  $\frac{3}{4}$  of the warps have only inactive threads and thus no divergence.

3. Answer: (B)

Explanation: In each iteration, there are stride consecutive active threads. During the iteration where stride is 16, there are 16 consecutive active threads, all in the same warp. All other threads have only inactive threads. So one warp has control divergence and 31 will not.

5. Answer: (A)

Explanation: There are 64 consecutive active threads, which is a multiple of warp size. So two warps will have all their threads active and 30 warps will have all their threads inactive. None of them will have control divergence.

### Prefix Sum Patterns

- Thread index to data index mapping and effect on control divergence
- Work efficiency of parallel prefix-sum algorithms
- Thread block prefix-sum algorithm using shared memory
- Global combination of partial prefix sums

1. For the Kogge-Stone scan kernel based on reduction trees, assume that we have 1024 elements, which of the following gives the closest approximation for the number of useful floating-point add operations performed?  
(A)  $(1024-1)*2$   
(B)  $(512-1)*2$   
(C)  $1024*1024$   
(D)  $1024*10$

2. For the Brent-Kung scan kernel based on reduction trees and inverse reduction trees, assume that we have 1024 elements, which of the following gives the closest approximation on the total number of useful floating-point add operations performed in both the reduction tree phase and the inverse reduction tree phase?  
(A)  $(1024-1)*2$   
(B)  $(512-1)*2$   
(C)  $1024*1024$   
(D)  $1024*10$

3. For the Brent-Kung scan kernel based on reduction trees and inverse reduction trees, assume that we have 2048 elements in each section and warp size is 32, how many warps in each

- block will have control divergence during the reduction tree phase iteration where stride is 16?
- (A) 0
  - (B) 1
  - (C) 16
  - (D) 32
4. For the Kogge-Stone scan kernel based on reduction trees, assume that we have 1024 elements in each section and warp size is 32, how many warps in each block will have control divergence during the iteration where stride is 16?
- (A) 0
  - (B) 1
  - (C) 16
  - (D) 32
5. In the previous question, how many warps in each block will have control divergence during the iteration where stride is 64?
- (A) 0
  - (B) 1
  - (C) 16
  - (D) 32

Answers and Explanations:

1. Answer: (D)

Explanation: The number of useful add operations performed by the Kogge-Stone scan kernel is approximately  $N * \log(N)$ , where N is the number of elements.

2. Answer: (A)

Explanation: The reduction tree performs  $N - 1$  floating-point add operations performed in the reduction tree phase and the inverse reduction performs  $N - \log(N)$  floating-point add operations. So the total is  $2 * (N - 1) - \log(N)$ . When N is large, this is approximately  $2 * (N - 1)$ .

3. Answer: (A)

Explanation: All active threads are consecutive starting with index 0. There are a total of 1024 threads and 64 active threads in the iteration in each block. The active threads form two whole warps. The warps are either all active or all inactive. None will have control divergence.

4. Answer: (B)

Explanation: All inactive threads are consecutive at the end of the block. There are a total of 1024 threads. When stride is 16, there are 16 inactive threads at the beginning of the block. So, Warp 0 has control divergence. No other warp in the block will have control divergence.

5. Answer: (A)

Explanation: At stride value 64, there are 64 inactive threads that are at the front of the block. Therefore, all threads in the first two warps are inactive. All threads in the remaining warps are active. There is no control divergence.

### Histogram and Atomic Operations

- Thread index to data index mapping and its effect on memory coalescing
  - The reason why parallel histogram algorithms need atomic operations
  - The use of privatization to increase efficiency
1. Assume that each atomic operation in a DRAM system has a total read-modify-write latency of 100ns. What is the maximal throughput we can get for atomic operations on the same global memory variable?
    - (A) 100G atomic operations per second
    - (B) 1G atomic operations per second
    - (C) 0.01G atomic operations per second
    - (D) 0.0001G atomic operations per second
  2. For a processor that supports atomic operations in L2 cache, assume that each atomic operation takes 4ns to complete in L2 cache and 100ns to complete in DRAM. Assume that 90% of the atomic operations hit in L2 cache. What is the approximate throughput for atomic operations on the same global memory variable?
    - (A) 0.25G atomic operations per second
    - (B) 2.5G atomic operations per second
    - (C) 0.0735G atomic operations per second
    - (D) 100G atomic operations per second
  3. In question 1, if a kernel performs 5 floating-point operations per atomic operation, what is the maximal floating-point throughput of the kernel execution as limited by the throughput of the atomic operations?
    - (A) 500 GFLOPS
    - (B) 5 GFLOPS
    - (C) 0.05 GFLOPS
    - (D) 0.0005 GFLOPS
  4. In Question 2, if a kernel performs 5 floating-point operations per atomic operation, what is the maximal floating-point throughput of the kernel execution as limited by the throughput of the atomic operations?
    - (A) 1.25 GFLOPS
    - (B) 12.5 GFLOPS
    - (C) 0.368 GFLOPS
    - (D) 500 GFLOPS

5. In Question 3, after Gather-to-Scatter transformation, there is no longer atomic operation in the kernel. Assume that the kernel performs 2 floating-point operations for every global memory access. Also, assume that kernel performs single-precision floating-point arithmetic and the global memory bandwidth is 160GB/second. What is the approximate floating-point throughput of the kernel execution as limited by the memory bandwidth?
- 1000 GFLOPS
  - 100 GFLOPS
  - 80 GFLOPS
  - 2 GFLOPS

#### Answers and Explanations

1. Answer: (C)

Explanation: No other atomic operation can touch the same variable for the entire duration of 100ns. The maximal rate is  $1/100n = 0.01G$

2. Answer: (C)

Explanation: The average latency is  $4ns * 90\% + 100ns * 10\% = 13.6ns$ . The average throughput is approximately  $1/13.6 = 0.0735G$  atomic operations per second

3. Answer: (C)

Explanation: The maximal is 5 operations per every 100ns, or  $5*0.01G$

4. Answer: (C)

Explanation: 5 floating-point operations every 13.6 ns, approximately  $5/(13.6ns) = 0.368$  GFLOPS

5. Answer: (C)

Explanation: 40G operand fetches per second which supports 80 GFLOPS since each fetched operand is used 2 times.

#### Sparse Matrix-Vector Multiplication

- Cost and benefit of each format – CSR, ELL, COO, JDS, JDS-Transpose
- No data reuse for matrix elements

1. Given a sparse matrix of integers with m rows, n columns, and k non-zeros. How many integers are needed to represent the matrix in COO?

- $m+n+k$
- $3m$
- $3n$
- $3k$

2. Given a sparse matrix of integers with m rows, n columns, and k non-zeros. How many integers are needed to represent the matrix in CSR?

- (A)  $m+n+k$   
 (B)  $2k+m+1$   
 (C)  $2k+n$   
 (D)  $3k$
3. Given a sparse matrix of integers with  $m$  rows,  $n$  columns, and  $k$  non-zeros. How many integers are needed to represent the matrix in ELL?  
 (A)  $m+n+k$   
 (B)  $2k+m+1$   
 (C)  $2k+n$   
 (D) None of the above
4. Given a sparse matrix of integers with  $m$  rows,  **$n$  non-zero elements in the row with the largest number of non-zeros**, and  $k$  non-zeros. How many integers are needed to represent the matrix in JDS-T? Recall that JDS-T has a transposed representation. Also, assume that we keep track of the number of non-zero's in each row, as we specified in the MP assignment.  
 (A)  $m+n+k$   
 (B)  $2k+m+1$   
 (C)  $2k+2m+n$   
 (D)  $2m+1$
5. Assume that a GPU has a global memory bandwidth of 160 GB/s. For a single-precision JDS-T kernel, what is the approximate floating-point throughput of the kernel execution as limited by memory bandwidth?  
 (A) 4000 GFLOPS  
 (B) 400 GFLOPS  
 (C) 40 GFLOPS  
 (D) 4 GFLOPS

#### Answers and Explanations:

1. Answer: (D)

Explanation: Each non-zero element needs three integers: value, row index, column index

2. Answer: (B)

Explanation: Each non-zero element needs two integers: value and column. Each row needs an integer: start but we also need the one more at the end

3. Answer: (D)

Explanation: We need to know the row with the largest number of non-zero elements!

4. Answer: (C)

Explanation: We need  $k$  integers for the non-zero elements,  $k$  integers for their column indices,  $n$  pointers to the beginning of each column after transposition,  $m$  integers to track the length of each row, and  $m$  integers to track the original row index before permutation

5. Answer: (C)

Explanation: 160 GB/s supports 40 GB/s single-precision operands per second. Each pair of floating-point multiplication and addition in the inner product requires one matrix element and one vector element. There is no reuse for the matrix elements. In our MP, we do not have anyway to exploit the reuse of the vector elements. So the memory bandwidth limits the floating point execution throughput to approximately 40 GFLOPS.

### PC System Architecture

- Calculation of PCIe Gen 2 and Gen 3 bandwidth given an X configuration
  - Understand the nature, use, and benefit of pinned (page-locked) memory
  - Understand the DMA used in CPU-GPU data transfers
1. If Carl's PC has a PCIe Gen3 x16 interconnect for his GPU, what is the closest approximation of the maximal cudaMemcpyAsync() copy throughput from host to GPU that he can expect?
    - (A) 100 GB/sec
    - (B) 500 GB/sec
    - (C) 16 GB/sec
    - (D) 1 GB/sec
  2. After ran a few test of cudaMemcpy(), Carl realized that the achieved copy throughput was about half of the PCIe bandwidth, what do you think was most likely the main cause of this degradation?
    - (A) cudaMemcpy() has a lot of software overhead
    - (B) cudaMemcpy() requires an extra copy from the user space to pinged buffer and the extra copy nearly doubles the total copying time.
    - (C) The manufacturer lied. The PCIe in the system is actually Gen2.
    - (D) The execution time of cudaMemcpy() was measured incorrectly.
  3. Peter has a 200MB array that he would like to process with GPU. He measured that the execution time of the code on CPU was 0.02 seconds. He also implemented a kernel and measured that the kernel execution on the GPU was 0.0005, a 40x speedup! However, he needs to transfer the data into the GPU memory and transfer 200MB data output data back to the host memory. His system has a PCIe Gen3 x16 interconnect. What would the real speedup be?
    - (A) 40x speedup
    - (B) 20x speedup
    - (C) 8x slow down
    - (D) 1.275x slow down

### Answers and Explanations:

1. Answer: (C)

Explanation: cudaMemcpyAsync() operates on pinged memory so there is no need to made an extra copy. The throughput should be close to 16GB/sec.

2. Answer: (B)

Explanation: The most likely reason is that the data needs to be first copied to a pinged memory buffer.

3. Answer: (C)

Explanation: The data copy will take  $(200M)/(16*1000M) = 0.0125$  sec to copy the input data in and another 0.0125 sec to copy the output data back. Thus the total speedup is  $0.02/(0.0005+0.0125+0.0125) = 0.78x$  speedup, or a 1.275x slow down

## CUDA Streams and Task Parallelism

- Use of CUDA streams – creation, and insertion into queues
- Correspondence between stream queues and engine queues
- Loop unrolling and call ordering to overlap computation with data transfer

1. For the following code:

- 1) `cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),.., stream0);`
- 2) `cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float),.., stream0);`
- 3) `cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),.., stream0);`
- 4) `cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),.., stream1);`
- 5) `cudaMemcpyAsync(d_B1, h_B+i+SegSize, SegSize*sizeof(float),.., stream1);`
- 6) `cudaMemcpyAsync(h_C+i+SegSize, d_C1, SegSize*sizeof(float),.., stream1);`

Which of the statements could be executed in parallel on the GPU

- (A) 1), 2) and 3)
- (B) 1), 2) and 4)
- (C) 1) and 4)
- (D) 3), and 4)

## Answers and Explanations:

1. Answer: (D)

Explanation: API operations in different streams can be executed in parallel. However, there is only one PCIe copy engine in each direction. So, only 3) and 4) can go in parallel

## OpenACC

- The semantics of Parallel regions vs. Parallel Loops. The statements in a parallel region will be redundantly executed by all the gang leaders
- Loops in the parallel regions that are not marked as OpenACC Loops are executed redundantly by all gang leaders
- Iterations of loops in the parallel regions that are marked as OpenACC Loops are distributed to gangs and workers to be executed in parallel.

1. In the following OpenACC code, which of the following is false?

- (A) Statement 1 will be executed redundantly executed by the the 32 gangs.
- (B) The n iterations of loop i will be divided and distributed to the 32 gangs for execution.

- (C) Statement 2 will be executed a total of n times.
- (D) Statement 3 will be executed a total of m times.

```
#pragma acc parallel num_gangs(32)
{
 Statement 1;
 #pragma acc loop gang
 for (int i=0; i<n; i++) {
 Statement 2;
 }
 for (int j=0; j<m; j++) {
 Statement 3;
 }
}
```

Answer: (D)

Explanation: The j loop will be redundantly executed by all 32 Gang leaders. So statement 3 will be executed  $32*m$  times.

#### MPI+CUDA

- The basic API functions of MPI, especially MPI\_Send and MPI\_Receive
- What are the meanings of MPI Ranks
- How do the MPI processes specialize themselves after they enter main() function.

### 3. Topics to Review from Lab

#### Common sources of bugs

- Function prototype problems
- Barrier synchronization problems
- Indexing problems

#### Performance Issues

- Access patterns that result non-coalesced global memory accesses / shared memory bank conflicts

#### Convolution

- Different convolution implementations, their pros and cons, and how they reflect on kernel launch configurations

#### Reduction and Prefix Sum

- Reduction trees, memory access patterns, thread utilization, and branch divergence
- Kogge-Stone vs. Brent-Kung thread organization and element indexing
- Parallel execution overhead and tradeoff between parallel execution and sequential execution

### Histogram and Privatization

- Levels of privatization and their applicability
- Allocation and indexing of Shared Memory
- Barrier synchronization and final contribution to the global histogram

### Question: Privatization

This question tests your understanding of parallel histogram computation and privatization. Assume that we would like to privatize a histogram that has 2048 bins. Each input data value (buffer array elements) will range from 0 to 2047. However, the shared memory can only accommodate 1024 bins for each block. As a compromise, we decide to privatize the first half of the bins into the shared memory. Whenever the data value falls into a bin in the second half, we will have to increment the global bin.

- (A) Complete the following kernel to implement the partial privatization of the histogram.

```

__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{
 __shared__ unsigned int histo_private[1024];
 int i;
 for (i = _____; i <_____ ; i+=_____) histo_privat[i] = 0;
 __syncthreads();
 int i = threadIdx.x + blockIdx.x * blockDim.x;
 // stride is total number of threads
 int stride = blockDim.x * gridDim.x;
 while (i < size) {
 if (_____) atomicAdd(&(private_histo[buffer[i]]), 1);
 else atomicAdd(_____,1);
 i += stride;
 }
 __syncthreads();
 for (i= _____; i < _____ ; i+= _____)
 atomicAdd(_____);
}

```

Answer:

```
__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)

{
 __shared__ unsigned int histo_private[1024];

 int i;

 for (i = threadIdx.x; i < 1024; i+=blockDim.x) histo_private[i] = 0;

 __syncthreads();

 i = threadIdx.x + blockIdx.x * blockDim.x;

 // stride is total number of threads

 int stride = blockDim.x * gridDim.x;

 while (i < size) {

 if (buffer[i] < 1024) atomicAdd(&(private_histo[buffer[i]]), 1);

 else atomicAdd(&histo[buffer[i]] ,1);

 i += stride;

 }

 __syncthreads();
}
```

(B) Can you think of a better partial privatization strategy that will likely result in less contention in the while loop? Outline your strategy.

Each thread can start its privatized section at a position

### Sparse Matrix

- Index calculation for the various formats.
- Control divergence and memory coalescing.

## Sparse Matrix Multiplication in JDS\_T

This question tests your knowledge of Sparse Matrix representation and operation.

- (A) In the following JDS\_T kernel, fill in the missing indexing expressions for accessing data (input matrix), x (input vector) and y (output vector).

```
1.__global__ void SpMV_JDS_T(int num_rows, float *data, int *col_index, int *jds_t_col_ptr,
 int jds_row_index, float *x, float *y) {
2. int row = blockIdx.x * blockDim.x + threadIdx.x;
3. if (row < num_rows) {
4. float dot = 0;
5. unsigned int sec = 0;
6. while (jds_t_col_ptr[sec+1]-jds_t_col_ptr[sec] > row){
7. dot += data[_____] * x[_____];
8. sec++;
9. }
10. Y[_____] = dot;
11. }
12. }
```

Answer:

```
1.__global__ void SpMV_JDS_T(int num_rows, float *data, int *col_index, int *jds_t_col_ptr,
 int jds_row_index, float *x, float *y) {
2. int row = blockIdx.x * blockDim.x + threadIdx.x;
3. if (row < num_rows) {
4. float dot = 0;
5. unsigned int sec = 0;
6. while (jds_t_col_ptr[sec+1]-jds_t_col_ptr[sec] > row){
7. dot += data[jds_t_col_ptr[sec]+row] * x[col_index[jds_t_col_ptr[sec]+row]];
8. sec++;
9. }
10. y[jds_row_index[row]] = dot;
11. }
12. }
```

(B) Assume a matrix that has 32 original rows, 64 columns, and 10 non-zeros in every row. After we transform the matrix into JDS-transposed layout, and launch the SpMV\_JDS\_T kernel. Is there any control divergence? Why or why not?

There is no control divergence. All 32 threads will take 10 iterations.

(C) In (B), are the memory accesses to the matrix in the for-loop (line 6) coalesced? Why or why not?

The memory accesses to the matrix are coalesced. All elements in the same column of the original matrix are laid out consecutively.

### Final Project

- Basic convolution layer kernel
- Tiled convolution layer kernel
- Kernel for unrolling X (input feature maps) used in the matrix-matrix multiplication implementation of convolution layer
- Properties of unrolled matrix and the parallelism, data reuse in the corresponding grid executing the matrix multiplication

### *Convolution Neural Network*

This question tests your understanding of the convolution layer of a CNN. We will start with a basic kernel implementation.

W is the convolution filter weight tensor, organized a tensor  $W[M, C, K, K]$ , M is the number of output feature maps, C is the number of input feature maps, K is the height and width of each filter.

X is the input feature map, organized as a tensor  $X[C, H_{out}+K-1, W_{out}+K-1]$ , where  $H_{out}$  is the height of each output feature map,  $W_{out}$  is the width of each output feature map.

Y is the output feature map, organized as a tensor  $Y[M, H_{out}, W_{out}]$ .

- (A) Fill in the missing parts of the basic kernel implementation. Use multidimensional indexing notation for simplicity. The kernel has each thread block to calculate a tile of output feature map elements. Each thread generates one output map element.

Assume that the blockDim is set to (TILE\_WIDTH, TILE\_WIDTH, 1) and that gridDim is set to (M,  $H_{grid} \times W_{grid}$ , 1).

```

__global__ void ConvLayerForward_Basic_Kernel(int C, int W_grid, int K,
 float* X, float* W, float* Y)
{
 int m = blockIdx.x;
 int h = blockIdx.y / W_grid + threadIdx.y;
 int w = blockIdx.y % W_grid + threadIdx.x;
 float acc = 0.;
 for (int c = 0; c < C; c++) { // sum over all input channels
 for (int p = 0; p < K; p++) // loop over KxK filter
 for (int q = 0; q < K; q++)
 acc += X[_____,_____,_____] * W[_____,_____,_____,_____];
 }
 Y[_____,_____,_____] = acc;
}

```

Answer: **X[c, h+p, w+q], W[m, c, p, q], Y[m, h, q]**

(B) Define the meaning of variables h, and w in ConLayerForward\_Basic\_Kernel.

h: \_\_\_\_\_

w: \_\_\_\_\_

Answer:

**h is the row index of the output feature map generated by each thread**

**w is the column index of the output generated by each thread**

(C) Fill in the missing parts of the tiled kernel implementation. Use multidimensional indexing notation for simplicity. The kernel has each thread block to calculate a tile of output feature map elements. Each thread generates one output map element.

```

__global__ void
ConvLayerForward_Kernel(int C, int W_grid, int K, float* X, float* W, float* Y)
{
 int 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];

```

```

m = blockIdx.x;
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

tx = threadIdx.x;
ty = threadIdx.y;
h = h_base + tx;
w = w_base + ty;

float acc = 0.;
int c, j, k, p, q;
for (c = 0; c < C; c++) { // sum over all input channels
 // load weights for W [m, c,..],
 // tx and ty used as shorthand for threadIdx.x and threadIdx.y
 if ((ty < K) && (tx < K))
 W_shared[____,____]= W [____,____,____];
 // load tile from X[n, c,...] into shared memory

 for (int i = h; i < h_base + X_tile_width; i += TILE_WIDTH) {
 for (int j = w; j < w_base + X_tile_width; j += TILE_WIDTH)
 X_shared[_____,_____] = X[_____,_____,_____]
 }

 for (p = 0; p < K; p++) {
 for (q = 0; q < K; q++)
 acc = acc + X_shared[_____,_____] * W_shared[_____,_____];
 }

 Y[_____,_____,_____,_____] = acc;
}

```

Answer:

```

// tx and ty used as shorthand for threadIdx.x and threadIdx.y
if ((ty < K) && (tx < K))
 W_shared[ty, tx]= W [m, c, ty, tx];
__syncthreads();
// load tile from X[n, c,...] into shared memory

for (int i = h; i < h_base + X_tile_width; i += TILE_WIDTH) {
 for (int j = w; j < w_base + X_tile_width; j += TILE_WIDTH)
 X_shared[i - h_base, j - w_base] = X[n, c, i, j]

```

```

 }
 __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;

```

- (D) For a 72x64 input feature map, 8x8 tiles and 3x3 convolution filters, if we use the tiled 2D convolution, what is the average number of times that each input feature map element is reused once it is loaded into the shared memory?

Answer:

$$\text{TILE\_WIDTH}^2 * \text{K}^2 / \text{X\_TILE\_WIDTH}^2 = 8^2 * 3^2 / (8+3-1)^2 = 5.76$$

- (E) If we use each thread block to generate one tile of output feature map elements, how many thread blocks will be generated when we launch the kernel?

Answer:

$$(72/8) * (64/8) = 72 \text{ thread blocks}$$

- (F) Fill in the missing parts of the unroll kernel implementation. Use multidimensional indexing notation for simplicity. The kernel has each thread block to calculate a tile of output feature map elements. Each thread generates one output map element.

```

void unroll_host_code(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>>>();
}

__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 = _____;
 w_base = c * K * K;
 for(p = 0; p < K; p++)
 for(q = 0; q < K; q++) {
 w_unroll = _____;
 X_unroll(_____, _____) = X(_____, _____, _____);
 }
 }
}

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);
 }
}

```

(G) How many times on average will be each X element be replicated by the unrolling kernel?

Answer:

The size of the unrolled matrix will be  $C \times K \times H_{out} \times W_{out}$ .

The size of the input feature maps is  $C \times (H_{out}+K-1) \times (W_{out}+K-1)$

The ratio of the two gives the answer:  $K \times (H_{out} \times W_{out}) / ((H_{out}+K-1) \times (W_{out}+K-1))$ .

Note that C and M does not play a role. Why?

You should explore the effect of different H\_out and W\_out on the answer. What happens when H\_out and W\_out are much larger than K? What is the intuition?

- (H) Think about the level of parallelism (number of thread blocks) that will be present when launching matrix multiplication kernel for a convolution layer, (1) for the C1 layer in the LeNet, (2) towards the end of the network, such as C5.

#### Parallelization

- Understanding dependence constraints
- Understanding communitivity and associativity

Question: Consider the following fragment of C code:

```
for(unsigned int x = 0; x < 512; ++x) {
 for(unsigned int y = 0; y < 512; ++y) {
 for(unsigned int z = 0; z < 512; ++z) {
 out[x][y] = out[x][y] <OP> in[x][y][z];
 }
 }
}
```

Explain how you would optimally parallelize this code and why if:

- (a) <OP> was not associative nor communitive
- (b) <OP> was associative and communitive

You need to state to what you will assign your blocks and your threads to and why, then write out the kernel function code.

Assume out[x][y] is initialized correctly.

Part (a):

<OP> is not associative/communitive => the z loop must be done sequentially  
The values of x and y are both small (<=512) so there is no need to go to 2D grids.  
y is the contiguous dimension => threads assigned to y dimension  
remaining dimension is x => blocks assigned to x dimension

Kernel code:

```
unsigned int x = blockIdx.x;
unsigned int y = threadIdx.x;
tmp = out[x][y]; // Use temporary to avoid global memory access
for(unsigned int z = 0; z < 512; ++z) {
```

```

 tmp = tmp <OP> in[x][y][z];
 }
 out[x][y] = tmp;
}

```

Part (b):

<OP> is associative and communitive => the z loop could be done using a reduction by a block  
=> assign a thread for each z value  
assign a block (256 threads) to each (x,y) pair

Kernel code:

```

unsigned int y= blockIdx.y;
unsigned int x = blockIdx.x;
unsigned int z = threadIdx.x;
__shared__ in_out_s[512]; // Use shared memory to avoid global access

in_out_s[z] = in[x][y][z];

for(unsigned int stride = 256; stride >= 1; stride >>= 1) {
 __syncthreads(); // Don't forget to sync
 if(z < stride) { // Stride in a manner that favors coalescing
 in_out_s[z] = in_out_s[z] <OP> in_out_s[z + stride];
 }
}

__syncthreads(); // The synchronization here is not necessary

if(z == 0) {
 out[x][y] = in_out_s[z];
}

```

# 408: Applied Parallel Programming

## Fall 2018 - Midterm Exam 2

December 4<sup>th</sup>, 2018

1. This is a closed book exam except for 1 sheet of hand-written notes
2. You may not use any personal electronic devices except for a calculator
3. **Please write legibly!! We are using OCR to help grade your exam**
4. Absolutely no interaction between students is allowed
5. Illegible answers will likely be graded as incorrect

**Good Luck!**

**Name:** \_\_\_\_\_ **SOLUTION** \_\_\_\_\_

**NetID:** \_\_\_\_\_

**Exam Room:** \_\_\_\_\_

Question 1 (20 points): \_\_\_\_\_

Question 2 (20 points): \_\_\_\_\_

Question 3 (15 points): \_\_\_\_\_

Question 4 (20 points): \_\_\_\_\_

Question 5 (20 points): \_\_\_\_\_

Question 6 ( 5 points): \_\_\_\_\_

**Total Score (100 points):** \_\_\_\_\_

Name: \_\_\_\_\_

## Problem 1 (20 points): Multiple Choice

Choose the proper response, and if multiple responses are correct, choose all. No partial credit will be provided if the answer is partially correct, or wrong.

**Part 1a (3 points)** For the following reduction kernel fragment, if the block size is 512 and warp size is 32, how many warps in a block will have control divergence during the iteration where stride is equal to 64?

```
1. __shared__ float partialSum[2 * blockDim.x];
2. unsigned int t = threadIdx.x;
3. unsigned int start = 2 * blockIdx.x * blockDim.x;
4. partialSum[t] = input[start + t];
5. partialSum[blockDim.x + t] = input[start + blockDim.x + t];
6. for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2)
7. {
8. __syncthreads();
9. if (t % stride == 0)
10. partialSum[2 * t] += partialSum[2 * t + stride];
11. }
```

- 0
- 1
- 2
- 8
- 16

**Part 1b (3 points)** For the Brent-Kung scan kernel based on reduction trees and inverse reduction trees, assume that we have 2048 elements in each section and warp size is 32, how many warps in each block will have control divergence during the reduction tree phase iteration where stride is 64?

- 0
- 1
- 16
- 32

Name: \_\_\_\_\_

**Part 1c (3 points)** Suppose we need to run Brent-Kung scan algorithm on a very large input consisting of  $2^{30}$  elements. For our CUDA device, the maximum number of threads in a block is  $2^{10}$  and the maximum number of blocks in the x-dimension of the grid is  $2^{11}$ . Further, suppose that we are using a one-dimensional grid along the x dimension. If we choose to use hierarchical parallel scan to process the input, what is the minimum number of times we need to launch the scan kernel?

- 2
- $2^8 + 2$
- $2^9 + 1$
- $2^9 + 2$
- $2^{19} + 2$

**Part 1d (3 points)** Suppose a processor supports atomic operations in L2 cache, assume that each atomic operation takes 5ns to complete in L2 cache and 180ns to complete in DRAM. Assume that 80% of the atomic operations hit the L2 cache. Further, assume that the kernel performs 10 floating-point operations per atomic operation. What is the floating-point throughput of the kernel execution as limited by the throughput of the atomic operations?

- 0.0025 GFLOPS
- 0.054 GFLOPS
- 0.25 GFLOPS
- 18.52 GFLOPS

**Part 1e (3 points)** To transfer data from the device to host (or vice versa), we use cudaMemcpy which essentially requires a pointer to a data block and the size of that block. To transfer data from nodes in an MPI system, we use MPI\_Send (and MPI\_Recv). One key difference to CUDA is that MPI\_Send requires a pointer to the data block, the number of elements in the block, and the datatype of these elements. Which of the following accounts for this difference?

- MPI\_Send is non-blocking
- cudaMemcpy only works on floating point data
- MPI needs to work on heterogeneous systems with different endian-ness (byte ordering).
- MPI\_Send can work on non-contiguous data blocks, whereas cudaMemcpy moves a single contiguous block

Name: \_\_\_\_\_

**Part 1f (3 points)** We need to calculate the histogram of an array with  $10^9$  elements. The histogram has four bins. Assume that each atomic operation in the global memory has a constant total latency of 100ns and each atomic operation in a shared memory has a constant total latency of 1ns. Further, assume that when we launch the kernel, blockDim =  $10^3$  and gridDim =  $10^5$ , thus thread is responsible for 10 elements. If we only consider the latencies caused by atomic operations, what is the theoretical minimum runtime if privatization is implemented and the elements of the array has a distribution of (50%, 30%, 10%, 10%)? Suppose that there is only one global histogram and one shared memory histogram in each block.

- $(10^7 + 1000)$  ns
- $(10^7 + 5000)$  ns
- $(10^7 + 10^4)$  ns
- $(4 * 10^7 + 1000)$  ns
- $(4 * 10^7 + 5000)$  ns
- $(4 * 10^7 + 10^4)$  ns
- None of the above

**Part 1g (2 points)** Given a sparse matrix of integers with m rows, n non-zero elements in the row with the largest number of non-zeros, and k non-zeros. How many integers are needed to represent the matrix in JDS-T? Recall that JDS-T has a transposed representation. Remember that we need to keep track of the number of non-zeros in each row, to assist the SpMV calculation.

- $2m+1$
- $m+n+k$
- $2k+m+n+1$
- $2k+m+1$

**Solution:**

- 1a. D, 8  
1b. B, 1  
1c. B,  $2^8 + 2$   
1d. C, 0.25 GFLOPS  
1e. C, need to work on systems with different endian-ness  
1f. B,  $(10^7 + 5000)$  ns  
1g. C,  $2k+m+n+1$

Name: \_\_\_\_\_

## Problem 2 (20 points): Histogramming

For this question, we'll consider a histogram with  $2^{18}$  bins (much larger than shared memory), and an input data stream (with values in the range  $[0, 2^{18} - 1]$ ) that needs to be binned. We want to take advantage of privatization, and observe that the input stream only has 256 bins that contain a significant fraction of the data (hot bins). Say we are provided a vector that's also  $2^{18}$  in length like the histogram, but contains a -1 for bins that are infrequently used, and a unique number between 0 and 255 for each of the hot bins. Our algorithm uses this unique value to create a private histogram in shared memory.

The following kernel will use shared memory for the hot bins and global memory for the not-so-hot bins. Once the work for a thread block is completed, the histogram will be updated in global memory.

**Part 2a (12 Points)** Complete the following kernel to implement the hot bin histogram. Note that each thread block will have its own privatized histogram of the hot bins in the shared memory with name **histo\_private**. The vector **data** contains the input data stream of length **size**, and the vector **freq** contains the hot bin vector.

For this problem, pay close attention to the array **global\_index**. It serves as a mapping back from the private histogram bins in **histo\_private** to the global histogram bins in **histo**.

Name: \_\_\_\_\_

```
1 // histogram is launched with the following parameters.
2 dim3 gridDim(1000);
3 dim3 blockDim(256);
4 histo_kernel<<<gridDim,blockDim>>>(data, freq, length, histo)
5
6 // Kernel Code
7 __global__
8 void histo_kernel(unsigned int *data, int *freq, long size, unsigned int *histo)
9 {
10 __shared__ unsigned int histo_private[256];
11 __shared__ unsigned int global_index[256];
12
13 //reset histogram
14 histo_private[threadIdx.x] = 0;
15 global_index[threadIdx.x] = 0;
16
17 __syncthreads();
18 unsigned int i = threadIdx.x + blockDim.x * blockIdx.x;
19 unsigned int stride = blockDim.x * gridDim.x;
20
21 while(i < size) {
22 if (freq[data[i]] > 0) {
23 atomicAdd(&histo_private[freq[data[i]]], 1);
24 global_index[freq[data[i]]] = data[i];
25 }
26 else
27 atomicAdd(*histo[data[i]], 1);
28 i += stride;
29 }
30 __syncthreads();
31 //contribute to global histogram
32 atomicAdd(&histo[global_index[threadIdx.x]], histo_private[threadIdx.x]);
33 }
```

**Part 2b (4 Points)** How many **shared-memory atomic operations** and **global-memory atomic operations** are being performed by all the threads in the kernel if the stream contains  $10^7$  data items and 80% of the elements are in the “hot” bins? Write down the expression for your answer. Pay attention to the code structure.

shared-memory atomics: 0.8 \* 10<sup>7</sup>

global-memory atomics: 0.2 \* 10<sup>7</sup> + 256\*1000

Name: \_\_\_\_\_

**Part 2c (4 Points)** For this part, we will use a very simple performance model with the following assumptions:

- the input stream **data** contains 32 data items and 25 (approx 80%) of the elements are in the “hot” bins
- There is 1 thread block with 32 threads (1 warp) in this execution.
- Further assume each shared-memory atomic operation requires 1 ns, whereas each global-memory atomic operation requires 100 ns.
- All atomic operations are blocking, which means the next instruction cannot execute until the atomic operation is completed.
- All other operations require 0 ns.
- You may assume the full warp starts execution at time 0, and executes every cycle.

With these assumptions, what is the execution time in nanoseconds for this thread block?

25\*1ns + 7\* 100ns + 32\*100ns

or

25\*1ns + 7\*100ns + 256\*100ns

**Your Answer Here:**

Name: \_\_\_\_\_

### Problem 3 (15 points): Multiple Prefix Sums

For this problem, we will implement a CUDA kernel for generating the prefix sums of multiple lists of data. Data from a biology experiment recorded the amount of growth of a plant specimen in millimeters in a given day for some number of plants. From this data, we want to generate the plant size in millimeters on each day for each plant. That is we need to perform a prefix sum for each plant. For this, we will devise a CUDA kernel based on the Brent-Kung approach for prefix sum. The table below provides an example of the input data for 3 plants across 4 days.

|       | Plant 0 | Plant 1 | Plant 2 |
|-------|---------|---------|---------|
| Day 1 | 30      | 10      | 1       |
| Day 2 | 20      | 5       | 5       |
| Day 3 | 15      | 20      | 4       |
| Day 4 | 17      | 4       | 6       |

The input data **input** is a matrix, with **num\_plants** as width and **num\_days** as height. The matrix will be stored as one-dimensional array in the row-major layout. We've used a grid with 2D thread blocks to solve this task. All threads in the same x dimension will process the data of one plant. Threads in the same x dimension in a block will process **2\*BLOCK\_SIZE** data.

**Part 3a (3 points):** Why does Brent-kung Prefix Sum algorithm not require double-buffering? Please answer the question in one line.

There is no read after write dependency among threads.

Your Answer Here:

Name: \_\_\_\_\_

**Part 3b (12 points)** Fill in the blanks to complete the prefix sum kernel described above.

```
0 #define BLOCK_SIZE 32
1 __global__
2 void PlantScan(float *input, float *output,
3 int num_plants, int num_days)
4 {
5 __shared__ float partialSum[BLOCK_SIZE][BLOCK_SIZE*2];
6 int tx = threadIdx.x;
7 int ty = threadIdx.y;
8 if (blockIdx.x * BLOCK_SIZE + tx < num_plants) {
9 if (2 * blockIdx.y * BLOCK_SIZE + ty < num_days) {
10 partialSum[tx][ty] =
11 input[(2 * blockIdx.y * BLOCK_SIZE + ty) *
12 num_plants + blockIdx.x * BLOCK_SIZE + tx];
13 }
14 else partialSum[tx][ty] = 0;
15 if ((2 * blockIdx.y + 1) * BLOCK_SIZE + ty < num_days) {
16 partialSum[tx][ty+BLOCK_SIZE] =
17 input[((2 * blockIdx.y + 1) * BLOCK_SIZE + ty) *
18 num_plants + blockIdx.x * BLOCK_SIZE + tx];
19 }
20 else partialSum[tx][ty] = 0;
21 }
22 else partialSum[tx][ty] = 0;
23 }
```

Name: \_\_\_\_\_

```
24 int stride = 1;
25 while(stride < 2 * BLOCK_SIZE) {
26 __syncthreads();
27
28 int index = (ty+1)*stride*2-1;
29
30 if (index < 2 * BLOCK_SIZE && index-stride >= 0
31 partialSum[tx][index] +=
32
33 partialSum[tx][index-stride];
34 stride = stride * 2;
35 }
36
37 stride = BLOCK_SIZE/2;
38 while(stride > 0){
39 __syncthreads();
40 int index = (tx + 1) * stride * 2 - 1;
41
42 if (index+stride < 2 * BLOCK_SIZE) {
43
44 partialSum[tx][index+stride]+=partialSum[tx][index];
45 }
46 stride = stride / 2;
47 }
48
49 __syncthreads();
50 ...
51 }
52 // Host Code
53 int main ()
54 {
55 ...
56 // Invoke Kernel Here
57 dim3 dimGrid(num_plants/BLOCK_WIDTH, num_days/BLOCK_WIDTH, 1);
58 dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH, 1);
59 PlantScan<<<dimGrid, dimBlock>>>
60 (input, output, num_plants, num_days);
61 ...
62 }
```

Name: \_\_\_\_\_

## Problem 4 (20 points): Sparse Matrix Multiplication

This question tests your knowledge of sparse matrix representation and operation.  
You are given a sparse matrix representation in CSR (Compressed Sparse Row) format:

Nonzero values:            `data` [2, 5, 3, 4, 1, 2, 2, 3]  
Column indices:            `col_index` [0, 3, 1, 2, 0, 1, 2, 3]  
Row pointers:            `row_ptr` [0, 2, 4, 4, 8]

**Part 4a (2 points):** Write down the dense matrix (4 x 4) of the given CSR format

For the following parts, use \* to represent zero element

**Part 4b (2 points):** Provide the COO representation of the same matrix:

COO:

Non zero values:            `data` [ ]  
Column indices:            `col_index` [ ]  
Row indices:            `row_index` [ ]

**Part 4c (2 points)** Provide the JDS representation of the same matrix:

Non zero values:            `data` [ ]  
Column indices:            `col_index` [ ]  
Row pointer:            `row_ptr` [ ]  
Row indices:            `row_index` [ ]

**Part 4d (2 points)** Provide the JDS\_T representation of the same matrix:

Non zero values:            `data` [ ]  
Column indices:            `col_index` [ ]  
Column pointer:            `col_ptr` [ ]  
Row indices:            `row_index` [ ]

**Part 4e (2 points)** For sparse matrix-vector multiply, what is the major drawback of the COO representation, and why don't the other representations suffer from the same limitation?

Name: \_\_\_\_\_

**Part 4f (10 points)** CSC (Compressed Sparse Column) format is similar to CSR, in which the non-zero values in column are stored continuously in the memory. CSC format of matrix is equivalent to the transpose of CSR format.

CSC Format Example:

$$A = \begin{bmatrix} 0 & 4 & 1 & 5 \\ 1 & 0 & 4 & 0 \\ 0 & 2 & 0 & 2 \\ 3 & 0 & 0 & 0 \end{bmatrix}$$

Nonzero value:

data [ 1, 3, 4, 2, 1, 4, 5, 2 ]

Row indices:

row\_index [ 1, 3, 0, 2, 0, 1, 0, 2 ]

Column pointer:

col\_ptr [ 0, 2, 4, 6, 8 ]

Given the information above, please fill in blank for the SpMV\_CSC kernel.

Note: data - input matrix; x - input vector; y - output vector

```
__global__ void SpMV_CSC (int num_cols, float *data, int
*col_ptr, int *row_index, float *x, float *y)
1. {
2. int col = blockIdx.y*blockDim.y+threadIdx.y;
3. if (col < num_cols) {
4. int col_start = col_ptr[_____];
5. int col_end = col_ptr[_____];
6. for (int elem = col_start; elem < col_end; elem++) {
7. float dot = data[_____]*x[_____];
8. atomicAdd(&(y[_____]), dot);
9. }
10. }
11. }
```

Answer:

A.  $\begin{bmatrix} 2 & 0 & 0 & 5 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 2 & 2 & 3 \end{bmatrix}$

Name: \_\_\_\_\_

B. COO:

values: **data** [ 2, 5, 3, 4, 1, 2, 2, 3]  
Column indices: **col\_index** [ 0, 3, 1, 2, 0, 1, 2, 3]  
Row indices: **row\_index** [ 0, 0, 1, 1, 3, 3, 3, 3]

C. JDS:

values: **data** [ 1, 2, 2, 3, 2, 5, 3, 4 ]  
Column indices: **col\_index** [ 0, 1, 2, 3, 0, 3, 1, 2 ]  
Row pointer: **row\_ptr** [ 0, 4, 6, 8, 8 ]  
Row indices: **row\_index** [ 3, 0, 1, 2 ]

D. JDS\_T:

values: **data** [ 1, 2, 3, 2, 5, 4, 2, 3 ]  
Column indices: **col\_index** [ 0, 0, 1, 1, 3, 2, 2, 3 ]  
Column pointer: **col\_ptr** [ 0, 3, 6, 7, 8 ]  
Row indices: **row\_index** [ 3, 0, 1, 2 ]

- E. Drawbacks of COO: Need atomic operation, therefore less efficient. Each thread process a portion of the data elements and use an atomic operation to accumulate result into output.

This is because the threads in SpMV kernel with COO format are no longer mapped to a particular row. COO comes with the cost of additional storage for the **row\_index** array.

F.

```
1. __global__ void SpMV_CSC (int num_cols, float *data, int
 *col_ptr, int *row_index, float *x, float *y)
2. {
3. int col = blockIdx.y*blockDim.y+threadIdx.y;
4. if (col < num_cols){
5. int col_start = col_ptr[_col_];
6. int col_end = col_ptr[_col+1_];
7. for (int elem = col_start; elem < col_end; elem++){
8. float dot = data[_elem_] * x[_col_];
9. atomicAdd(&(y[_row_index[elem]]), dot);
10. }
11. }
12. }
```

Name: \_\_\_\_\_

## **Problem 5 (20 points): Convolutional Neural Network (CNN)**

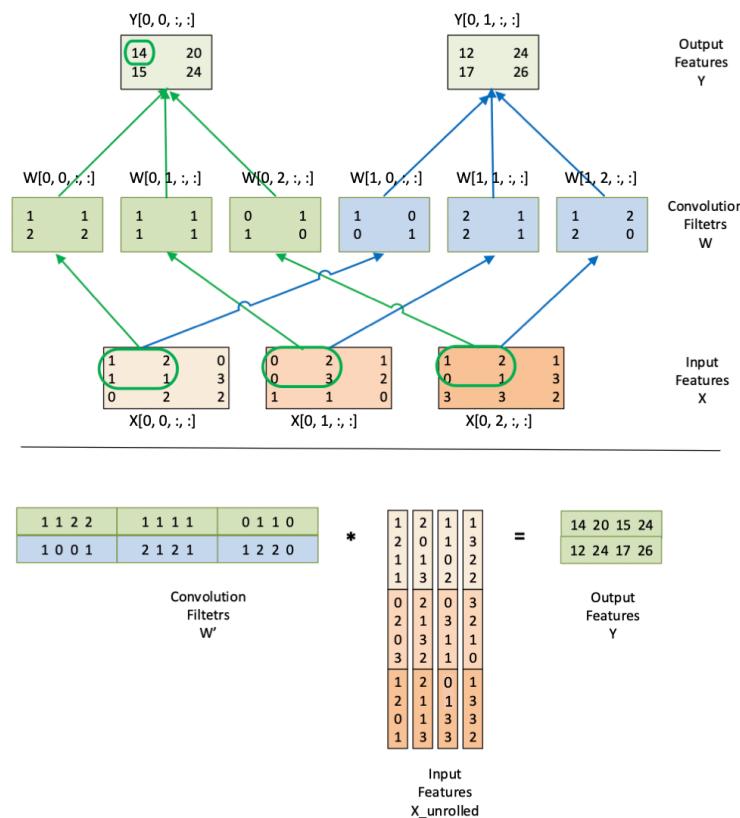
A basic convolution layer in a CNN consists of filter W, input X, and output Y. In this question, we want to accelerate the forward propagation of convolution layers in the training process.

$W$  is the convolution filter weight tensor, organized as a tensor  $W[M, C, K, K]$ , where  $M$  is the number of output feature maps,  $C$  is the number of input feature maps,  $K$  is the height and width of each filter. Tensors are stored as multi-dimensional arrays in the memory.

$X$  is the input feature map, organized as a tensor  $X[B, C, H, W]$ , where  $B$  is the number of images,  $H$  is the height of each input feature map, and  $W$  is the width of each input feature map.

$Y$  is the output feature map, organized as a tensor  $Y[B, M, H_{out}, W_{out}]$ , where  $H_{out} = H - K + 1$  is the height of each output feature map and  $W_{out} = W - K + 1$  is the width of each output feature map.

One way to implement the forward propagation in CUDA is to reduce the convolution into the general matrix multiplication (GEMM). The diagram below shows the relationship between basic convolution and how that can be done by using GEMM. Note that the diagram only shows a single mini-batch, e.g. batch 0.



Name: \_\_\_\_\_

**Part 5a (2 points):** From the lecture, we learned that before applying the GEMM, we first need to unroll the input feature map (X) into the correct shape. Your friend told you that you need to unroll the weight matrix (W) as well. Is this necessary? Why or why not?

Ans: No need for conversion. W is stored in the correct form for GEMM in the device memory.

**Part 5b (2 points):** How many times on average will be each X element be replicated after the unrolling? Provide your answer as an expression using tensor dimensions.

Ans: The size of the unrolled matrix will be  $C*K*K \times H_{out}*W_{out}$ . The size of the input feature maps is  $C*(H_{out}+K-1)*(W_{out}+K-1)$ . The ratio of the two gives the answer:  $K*K * (H_{out}*W_{out}) / ((H_{out}+K-1)*(W_{out}+K-1))$ .

**Part 5c (16 points):** After finishing the unrolling kernel and matrix multiplication kernel, you realized the performance could be better if we use only the tiled matrix multiplication kernel without actual unrolling. That is, instead of having a separate unrolling kernel, we perform unrolling when loading the tile into shared memory by correctly calculating the data indices. You need to fill in the missing parts so that the convolution layer is complete. (Note that we use multidimensional indexing notation for simplicity.)

```
// The code will be launched by using the following configuration.
dim3 gridDim(ceil(H_out*W_out/(1.0*TILE_WIDTH)),
 ceil(M/(1.0*TILE_WIDTH)),B);
dim3 blockDim(TILE_WIDTH,TILE_WIDTH,1);

// Kernel code.
01: __global__ void ConvLayerForward(int C, int K, int W_out, int H_out,
float* X, float* W, float* Y) {
02: __shared__ float tileMatA[TILE_WIDTH][TILE_WIDTH];
03: __shared__ float tileMatB[TILE_WIDTH][TILE_WIDTH];
04:
05: int b = blockIdx.z;
06:
07: int tx = threadIdx.x, ty = threadIdx.y;
08: int row = blockIdx.y * TILE_WIDTH + ty;
09: int column = blockIdx.x * TILE_WIDTH + tx;
10: int numMatAColumns = C*K*K; // This is the same as numMatBRows.
11:
12: float acc = 0.0;
13:
14: int num_iterations = ceil(numMatAColumns/(1.0*TILE_WIDTH));
15:
16: for (int i = 0; i < num_iterations; i++) {
```

Name: \_\_\_\_\_

```
17: int temp_col = i*TILE_WIDTH + tx, temp_row = i*TILE_WIDTH + ty;
18: tileMatA[ty][tx] = 0;
19: tileMatB[ty][tx] = 0;
20:
21: // Original indices in the filter tensor.
22: int W_m = row;
23: int W_c = __temp_col/(K*K)__;
24: int W_h = __temp_col%(K*K)/K__, W_w = __temp_col%(K*K)%K__;
25:
26: if (temp_col < numMatAColumns && row < M)
27: tileMatA[ty][tx] = W[W_m, W_c, W_h, W_w];
28: else
29: tileMatA[ty][tx] = 0;
30:
31: // Original indices in the input tensor.
32: int X_b = b;
33: int X_c = __temp_row/(K*K)__;
34: int X_p = __temp_row%(K*K)/K__, X_q = __temp_row%(K*K)%K__;
35: int X_h = __column/W_out__, X_w = __column%W_out__;
36:
37: if (temp_row < numMatAColumns && column < H_out*W_out)
38: tileMatB[ty][tx] = X[X_b, X_c, X_h + X_p, X_w + X_q];
39: else
40: tileMatB[ty][tx] = 0;
41:
42: __syncthreads();
43:
44: for (int q = 0; q < TILE_WIDTH; q++)
45: acc += tileMatA[ty][q] * tileMatB[q][tx];
46: __syncthreads();
47: }
48:
49: // Original indices in the output tensor.
50: int Y_b = b;
51: int Y_m = row;
52: int Y_h = column / W_out, Y_w = column % W_out;
53:
54: if (row < M && column < W_out*H_out)
55: Y[Y_b, Y_m, Y_h, Y_w] = acc;
56: }
```

Name: \_\_\_\_\_

## Problem 6 (5 points): Profiling

One fine Monday morning at Pied Piper, you get an email from Richard Hendricks! The email is as follows:

*"Greetings from Richard Hendricks!*

*I learned that you have experience optimizing CUDA code through your ECE408 coursework! I would like to take your inputs to identify performance bottlenecks in our next generation machine learning model. Why don't you educate me as I am new to it*

*Regards  
R.H."*

After reading this email, you are delighted to know that you can educate Richard and you meet him in the afternoon. Richard has already done his homework and has some profiling data and it's your time to explain the bottlenecks in his code.

**Part 6a (3 points)** I have a code base with 1 kernel and 3 different optimizations. I profiled my kernel and saw the following in the profiler's visual analysis tool. See the charts on the following page. For each of the optimization, identify as specifically as possible the utilization limiting factor (i.e., memory bound, compute latency bound, resource bound).

- a. Optimization 1: \_\_\_\_\_ **Memory bound** \_\_\_\_\_
  
- b. Optimization 2: \_\_\_\_\_ **Compute latency bound** \_\_\_\_\_
  
- c. Optimization 3: \_\_\_\_\_ **Resource or latency bound** \_\_\_\_\_

**Part 6b (2 points)** Richard asks you to guess the optimization that allowed him to improve his performance from optimization 1 to optimization 2. There are several major optimizations that could have achieved such results. Provide two below.

- a. \_\_\_\_\_
  
- b. \_\_\_\_\_

Possible answers:

- i. Shared memory
- ii. reduced the memory divergence or remove uncoalesced accesses
- iii. register tiling or reducing the redundant or repeated usage of memory operations

Name: \_\_\_\_\_



Memory operations



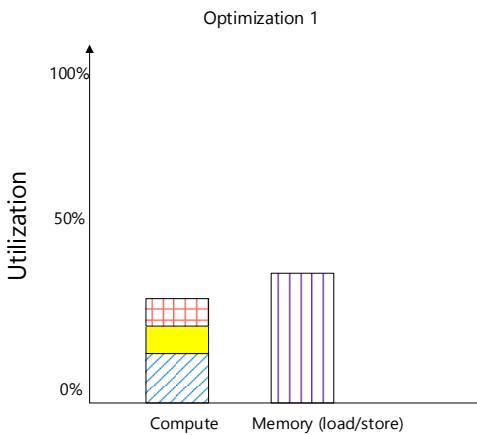
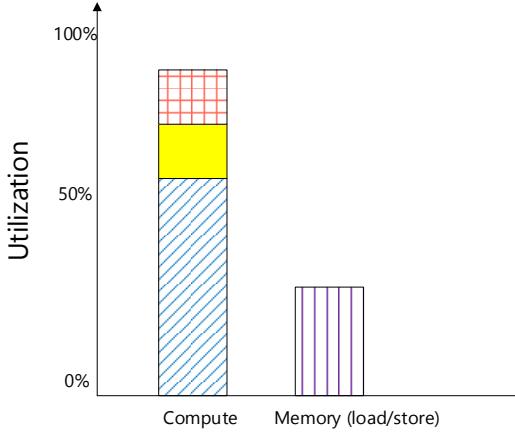
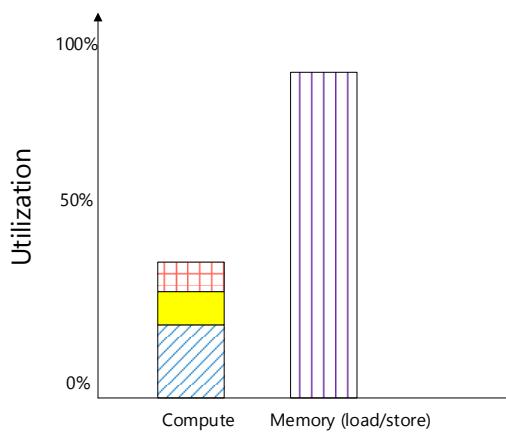
Control flow



Address generation



Arithmetic



Optimization 2

Optimization 3

Name: \_\_\_\_\_

This blank page is provided as extra space for calculations

# ECE 408 Exam 2, Spring 2018

April 23, 2018

- You are allowed one 8.0x11.5 cheat sheet with notes on both sides. The minimal font size for your text on the cheat sheet should be 8pts.
- 16
- No interactions with humans other than course staff are allowed.
- This exam is designed to take 150 minutes to complete. To allow for any unforeseen difficulties, we will be giving everyone up to 180 minutes.
- This exam is based on lectures, textbook chapters, as well as lab MPs/projects.
- The questions are randomly selected from the topics we covered.
- You can write down the reasoning behind your answers for possible partial credit.
- You must write your answers with pen in order to request regrade.

Good luck!

Name: \_\_\_\_\_

Netid: \_\_\_\_\_

UIN: \_\_\_\_\_

Question 1: (25 points) \_\_\_\_\_

Question 2: (15 points) \_\_\_\_\_

Question 3: (15 points) \_\_\_\_\_

Question 4: (15 points) \_\_\_\_\_

Question 5: (15 points) \_\_\_\_\_

Question 6: (15 points) \_\_\_\_\_

Name: \_\_\_\_\_

**Question 1 (25 points, 30 minutes):** multiple-choice and short-answer questions. If you get more than 25 points by answering all questions (1-9), your score will saturate at 25 points.

For multiple-choice questions, give a concise explanation for your answer for possible partial credit. Answer each of the short-answer questions in as few words as you can. Your answer will be graded based on completeness, correctness, and conciseness.

1. (3 points) For the Brent-Kung scan kernel based on reduction trees and inverse reduction trees, assume that we have 2048 elements, which of the following values is the total levels of the reduction tree and the inverse reduction tree?

- (A) 7
- (B) 11
- (C) 21
- (D) 25

Answer: (C)

Explanation: For N elements, the reduction tree has  $\log(N)$  levels and the inverse reduction has  $\log(N) - 1$  levels. The total is  $2 \cdot \log(N) - 1$ . For 2048 elements, the answer is  $2 \cdot \log(2048) - 1 = 2 \cdot 11 - 1 = 21$ .

2. (3 points) For the Brent-Kung scan kernel based on reduction trees and inverse reduction trees, assume that we have 2048 elements in each section and warp size is 32, how many warps in each block will have control divergence during the reduction tree phase iteration where stride is 16?

- (A) 0
- (B) 1
- (C) 16
- (D) 32

Answer: (A)

Explanation: All active threads are consecutive starting with index 0. There are a total of 1024 threads and 64 active threads in each block when the stride is 16. The active threads form entire warps 0 and 1. All other warps are completely inactive. No warps have control divergence.

Name: \_\_\_\_\_

3. (3 points) For a processor that supports atomic operations in L2 cache, assume that each atomic operation takes 5ns to complete in L2 cache and 500ns to complete in DRAM. What is the approximate hit rate of the L2 cache needed to achieve a throughput of 1/250 G operations per second?
- (A) 99%  
(B) 75%  
(C) 50%  
(D) 25%

Answer: (C)

Explanation: The desired latency is 250ns. A little more than half of the atomic operations need to hit in cache to achieve this latency. So, the hit rate should be about 50%.

4. (3 points) Given a sparse matrix of integers with R original rows, L non-zero elements in the original row with the largest number of non-zeros, and a total of N non-zeros. How many more integers are needed to represent the matrix in JDS as compared to CSR?
- (A) R  
(B) 2N  
(C) L  
(D) N

Answer: (A)

Explanation: We need R additional integers to keep track of the original row number for each row after sorting.

5. (3 points) For a sparse matrix-vector multiplication (SpMV) with 100,000 rows, a total of 300,000 non-zero elements, and a maximal of 10 non-zeros in each row, how many additional zero elements will be added when we convert the matrix from CSR to ELL?
- (A) 5  
(B) 300,000  
(C) 700,000  
(D) 1,000,000

Answer: (C)

Explanation: In ELL, all rows will be filled to 10 elements. This will make the matrix  $10 \times 100,000 = 1,000,000$  elements for ELL. CSR will have only 300,000 elements. The difference is  $1,000,000 - 300,000 = 700,000$ .

Name: \_\_\_\_\_

6. (3 points) For a sparse matrix-vector multiplication (SpMV) with 100,000 rows, 10,000 columns, a total of 300,000 non-zero elements, how many times on average will each vector element be used?
- (A) 1
  - (B) 3 on average
  - (C) 30 on average
  - (D) 100,000

Answer: (C)

Explanation: A total of N accesses will be made to the vector elements. There are C of them. So each us used  $N/C = 300,000/10,000 = 30$  on average.

7. (3 points) Keven has a 640MB array that he would like to process with GPU. He measured that the execution time of the code on CPU was 0.02 seconds. He also implemented a kernel and measured that the kernel execution on the GPU with the data in the GPU memory was 0.0004 seconds, a 50x speedup! However, he needs to transfer the data into the GPU memory. There is negligible data to be transferred back to the CPU. His system has a PCIe Gen3 x16 interconnect. What would the real speedup be?
- (A) 40x speedup
  - (B) 20x speedup
  - (C) 5x speedup
  - (D) 0.5x speedup (100% slow down)

Answer: (D)

Explanation: The data copy will take  $(640M)/(16*1000M) = 0.04$  sec to copy the input data to GPU. Thus the total speedup is  $0.02/(0.04+0.0004) \approx 0.02/(0.04) = 0.5x$  speedup, or 100% slow down

8. (3 points) For the following host code sequence:

```
1) cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),..., stream0);
2) cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),..., stream1);
3) cudaMemcpyAsync(d_A2, h_A+i+2*SegSize, SegSize*sizeof(float),..., stream2);
4) cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),..., stream0);
5) cudaMemcpyAsync(h_C+i+SegSize, d_C1, SegSize*sizeof(float),..., stream1);
6) cudaMemcpyAsync(h_C+i+2*SegSize, d_C2, SegSize*sizeof(float),..., stream2);
```

Which of the statements could be executed in parallel on the GPU

Name: \_\_\_\_\_

- (A) 1) and 2)
- (B) 2) and 3)
- (C) 2) and 4)
- (D) 4) and 5 )

Answer: (C)

Explanation: API operations in different streams can be executed in parallel.

However, there is only one PCIe copy engine in each direction. So, only 2) and 4) can go in parallel among the choices given

9. (3 points) In the following OpenACC code, which of the following is false?
- (A) Statement 1 will be executed redundantly executed by the the 32 gangs.
  - (B) The n iterations of loop i will be divided and distributed to the 32 gangs for execution.
  - (C) Statement 2 will be executed a total of n times.
  - (D) Statement 3 will be executed a total of m times.

```
#pragma acc parallel num_gangs(32)
{
 Statement 1;
 #pragma acc loop gang
 for (int i=0; i<n; i++) {
 Statement 2;
 }
 for (int j=0; j<m; j++) {
 Statement 3;
 }
}
```

Answer: (D)

Explanation: The j loop is in a parallel eregion but not marked as an acc Loop, thus the entire loop will be redundantly executed by all the 32 gangs. Satement 3 will be executed  $32*m$  times.

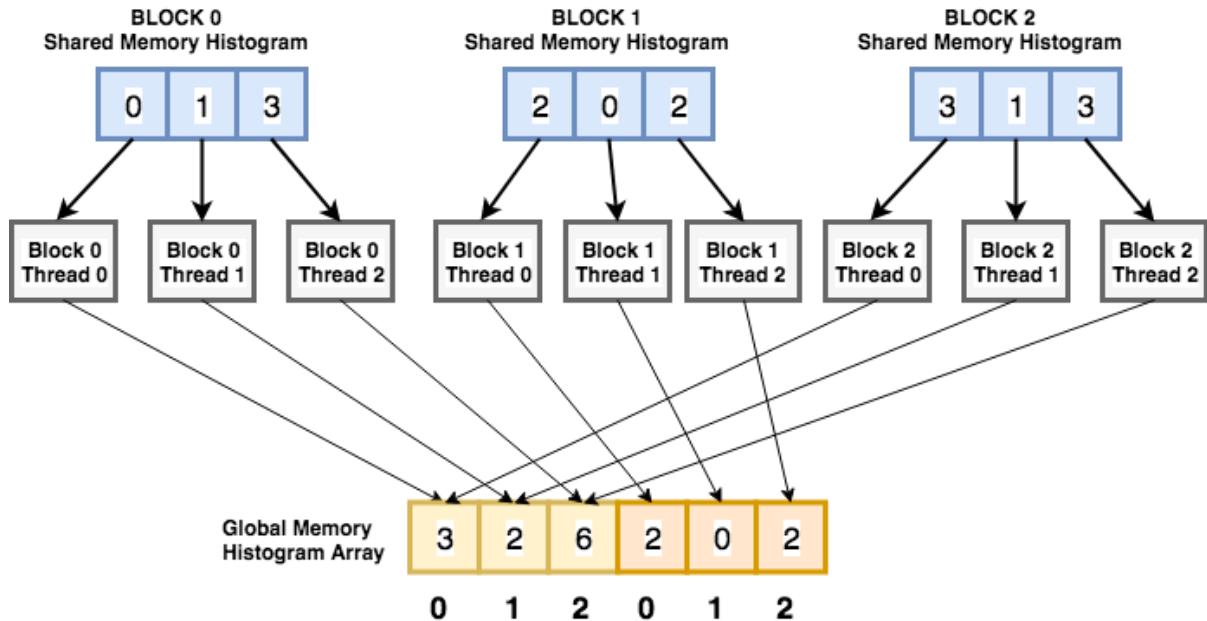
Name: \_\_\_\_\_

**Question 2 (15 points, suggested time allocation 25 minutes):** This question tests your understanding of parallel histogram computation and privatization.

You are part of a team that tallies up the voting result of a president's election. The election was between **four** different candidates. You have 300 million of votes that you need to process. Knowing GPUs can process the mass amount of data, you decided to create a **modified** version of the histogram kernel you learned in class to count how many votes each candidate got.

In the modified version of the histogram kernel, instead of having one global histogram, you have an array of global histograms so that you can have less contention on the global histograms. The global histogram array index is calculated as  $\text{blockIdx.x \% NUM\_HISTOGRAM}$ . Carefully analyze the host code as it will help. Below is a visualization of the modified kernel for a small example of three candidates, three thread blocks and an array of two global histograms.

**Block Dim: 3, Grid Dim: 3**  
**NUM\_HISTOGRAM: 2**  
**NUM\_BIN: 3**



(A) (3 Points) Complete the following kernel to implement the modified histogram. Note that each thread block will still have its own privatized histogram in the shared memory.

```
/* histo_kernel is launched with the following parameters */
#define NUM_VOTERS 3000000000 // 300 million
#define NUM_HISTOGRAM 16
#define NUM_BIN 4
```

Name: \_\_\_\_\_

```
unsigned int histogram_array[NUM_HISTOGRAM * NUM_BIN] = {0};
unsigned int histogram[NUM_BIN] = {0};

dim3 gridDim(10000);
dim3 blockDim(1000);
histo_kernel<<<DimGrid, DimBlock>>>(votes, NUM_VOTERS, histogram_array);

for(int i = 0; i < NUM_HISTOGRAM; i++) {
 for(int j = 0; j < NUM_BIN; j++) {
 histogram[j] += histogram_array[i*NUM_BIN + j];
 }
}

__global__ void histo_kernel(unsigned int *votes, long size, unsigned int
*histo_array){
 __shared__ unsigned int histo_private[__NUM_BIN_(+.5)_];

// Each element of the votes[] array contains an integer that selects one of
// the candidates. Its value ranges from 0 to NUM_BIN-1

 int i = threadIdx.x + blockDim.x * blockIdx.x;
 int stride = __blockDim.x * gridDim.x_(+.5_);

 // Reset histogram
 if(threadIdx.x < NUM_BIN)
 histo_private[threadIdx.x] = 0;
 __syncthreads();

 while (i < size) {
 atomicAdd(&(histo_private[__votes[i]_(+.5)_]), 1);
 i += stride;
 }
 __syncthreads();

 // contribute to one of the global histograms
 if(threadIdx.x < NUM_BIN){
 int histo_index = blockIdx.x % NUM_HISTOGRAM;
 atomicAdd(&(histo_array[_histo_index * NUM_BIN + threadIdx.x_(+1)_]),
 histo_private[__threadIdx.x_(+.5)_]);
 }
}
```

(B) (2 Point) How many times does the 500th thread of the 25th block( threadIdx.x = 500, blockIdx.x = 25) iterate in the **while loop**?

Name: \_\_\_\_\_

**Answer:** 30

**Explanation:**  $300000000 / (10000 * 1000)$

(C) (2 Points) How many **non-atomic global memory reads/writes** and **shared-memory/global-memory atomic operations** are being performed by **all the threads** executing the kernel?

**Non-atomic Global Memory reads:** 300,000,000

**Non-atomic Global Memory writes:** 0

**Shared-memory atomic operations:** 300,000,000

**Global-memory atomic operations:**  $10000 * 4$

Explanation:

For the following questions, consider only the atomic operations in the process of analyzing the kernel code. Assume that

- each atomic operation in the global memory has a constant total latency of 100ns.
- each atomic operation in a shared memory has a total latency of 1ns.

(D) (2 Points) If the votes were evenly spread out ( 25% each), what is the theoretical minimum runtime of the **original histogram kernel** (i.e., NUM\_HISTOGRAM = 1)?

**Answer:** 1.0075 ms

**Explanation:**  $(1000*30)* .25 * 1 \text{ ns} + 10000 * 100 \text{ ns} = 10\text{e}6 + 7.5\text{e}3 \text{ ns}$

(E) (2 Points) If the votes were evenly spread out ( 25% each), what is the theoretical minimum runtime of the **modified histogram kernel**?

**Answer:** 0.07 ms

**Explanation:**  $(1000*30) * .25 * 1 \text{ ns} + 10000 / 16 * 100 \text{ ns} = 6.25\text{e}4 + 7.5\text{e}3 \text{ ns}$

(F) (2 Points) If the vote resulted in a distribution of (90%, 5%, 5%, 0%), what is the theoretical minimum runtime of the **original histogram kernel**?

**Answer:** 1.027 ms

**Explanation:** The execution time of the privatized histogram phase of each thread block will be dominated by the atomic operations to the most popular candidate bin. The time to contribute to the global histogram will remain the same as the case with even distribution.  $(1000*30) * .9 * 1\text{ns} + 10000 * 100 \text{ ns}$

(G) (2 Points) If the vote resulted in a distribution of (80%, 5%, 5%, 10%), what is the theoretical minimum runtime of the **modified histogram kernel**?

Name: \_\_\_\_\_

**Answer:** 0.0865 ms

**Explanation:**  $(1000*30) * .8 * 1\text{ns} + 10000 / 16 * 100 \text{ ns}$

(H) (Extra Credit 1 Point) What technique could further help with a dataset that has a large concentration of identical data values in localized areas in the histogram kernel? (Hint: This technique was discussed in the textbook.)

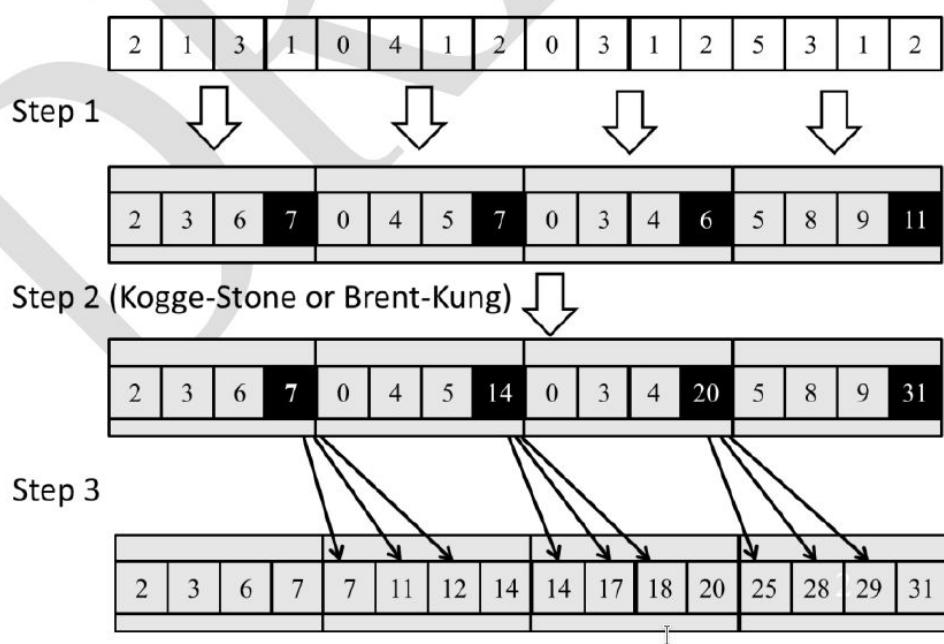
Aggregation

Name: \_\_\_\_\_

**Question 3: Scan (15 points, suggested time allocation 25 minutes):**

To further optimize the scan kernel, we can perform the scan operations more efficiently by further dividing the first kernel of the prefix-sum MP 5.2 into three steps. Note that all the three steps are performed within one kernel.

- I. In Step 1 (see the figure below), each thread operates on its own small section and perform sequential scan in the shared memory. The number of these small sections are the same as the number of threads in a block. (The figure below shows a small example where each block has 4 threads, each thread handles 4 elements, and each block is handling 16 elements).
- II. In Step 2, all threads preform Kogge-Stone scan **only** on every last element of the small sections in the shared memory.
- III. In Step 3, each threads will add the last element (partial sum of its own section) to the elements in the next section, except for the last one since it's already computed in the second step.



This algorithm only completes the scan on the data belongs each block. In order to perform a complete scan, there will be another two kernels adding partial sums to data of other blocks (like the hierarchical scan in MP5.2). This is just an attempt to make the per-block scan kernel more efficient by having each thread to handle more than one element.

For this question, we will be focusing only on the scan kernel described above. You can also assume that the number of input data elements is a multiple of the number of threads or the size of the shared memory (i.e, don't worry about the boundary conditions).

Name: \_\_\_\_\_

- (A) (6 points) Fill in the blanks to complete the scan kernel described above. (Hint: use the figure when you analyze the kernel code.)

```
#define TILE_WIDTH 1024 // Number of threads in each block
#define SHARE_LIMIT 4*1024 // Total number of elements being scanned in each block

// Number of elements in each small section to be processed by each thread
#define NUM_PER_SEC (SHARE_LIMIT/TILE_WIDTH)

__global__ void scan(float* arr)
{
 // Shared memory to hold input elements for each block
 __shared__ float scanShare[SHARE_LIMIT];

 // Explicitly store the last element of each small sub-section
 __shared__ float sectionEndShare[TILE_WIDTH];

 size_t tx = threadIdx.x;
 size_t offset = blockIdx.x * SHARE_LIMIT;
 size_t subsection_start = tx*NUM_PER_SEC;

 // Step 1

 // - Each thread will perform a scan on its own small section
 for (size_t i = 0; i < NUM_PER_SEC; i++) {
 size_t currIdx = subsection_start + i;
 scanShare[currIdx] = arr[offset + currIdx];
 if (i > 0){
 scanShare[currIdx] += scanShare[currIdx - 1];
 }
 }
 sectionEndShare[tx] = scanShare[(tx+1)*NUM_PER_SEC - 1];

 // Step 2
 // - Kogge-Stone scan on the end of each section
 for (size_t stride = 1; stride < TILE_WIDTH; stride *= 2) {
 __syncthreads();
 if (tx >= stride) {
 sectionEndShare[tx] += sectionEndShare[tx - stride];
 }
 }
 scanShare[(tx+1)*NUM_PER_SEC - 1] = sectionEndShare[tx];

 // Step 3
 // Add partial sums to necessary elements
 __syncthreads();
 for (size_t i = 0; i < NUM_PER_SEC - 1; i++) {
```

Name: \_\_\_\_\_

```
if (tx > 0){
 scanShare[subsection_start + i] += sectionEndShare[tx - 1];
}
}

// Write results back to global memory
__syncthreads();
for (size_t i = 0; i < NUM_PER_SEC; i++) {
 size_t currIdx = subsection_start + i;
 arr[offset + currIdx] = scanShare[currIdx];
}
}

// Code on host side:

// num is the number elements in the input array
int numBlocks = (num-1) / (SHARE_LIMIT) + 1;

dim3 dimBlock(TILE_WIDTH, 1, 1);
dim3 dimGrid(numBlocks, 1, 1);
scan<<<dimGrid, dimBlock>>>(dev_arr);

// Assume there will be other hierarchical scans to complete scan on the entire
array
...
```

(B) (5 points) After completing this kernel, you found out that the scan results is incorrect. It turns out that the implementation of Kogge-Stone in Step 2 is buggy. Explain what the problem is and rewrite the Kogge-Stone part such that it is correct.

Explanation:

New Kogge-Stone:

```
// - Kogge-Stone scan on the end of each section
for (size_t stride = 1; stride < TILE_WIDTH; stride *= 2) {
 float temp = sectionEndShare[tx];

 if (tx >= stride){

 }
}
```

Name: \_\_\_\_\_

```
 }

sectionEndShare[tx] = temp;
}
```

**Solution:**

```
// - Kogge-Stone scan on the end of each section
for (size_t stride = 1; stride < TILE_WIDTH; stride *= 2) {
 float temp = sectionEndShare[tx];
 __syncthreads();
 if (tx >= stride){
 temp += sectionEndShare[tx - stride];
 }
 __syncthreads();
 sectionEndShare[tx] = temp;
}
```

(C) (4 points) When reading (step 1) and writing data (step 3) from/to global memory, there are still room for improvement. In order to further optimize it, we can first load all the data in a coalesced manner and then perform scan (this is what we call corner turning technique in the class). We can then have better memory coalescing and fewer control divergence.

Fill in the blanks such that the access pattern for reading memory (step 1) is coalesced. (Writing to memory in step 3 can be done similarly.) Note that all of the following code is still in the kernel.

```
// Step 1
// - Load data into shared memory
for (size_t i = tx; i < SHARE_LIMIT; i+=TILE_WIDTH) {
 scanShare[i] = arr[offset + i];
}
__syncthreads();

// - Each thread will perform a scan on its own small section
for (size_t i = 1; i < NUM_PER_SEC; i++) {
 int currIdx = subsection_start + i;
 scanShare[currIdx] += scanShare[currIdx - 1];
}
```

(D) (3 points) Calculate the total number of arithmetic FLOPs in step 2 per block (i.e, only considering the Kogge-Stone part). Choose only one answer.

Below are some equations that you might need:

Name: \_\_\_\_\_

$$\log_2(1024) = 10, \log_2(4096) = 12$$

$$1 + 2 + \dots + N/2 = N - 1$$

- a.  $4096 * 10$
- b.  $4096 * 12$
- c.  $4096 * 1024$
- d.  $4096 * 2 - 2 - 12$
- e.  $4096 * 10 - (4096 - 1)$
- f.  $4096 * 12 - (4096 - 1)$

$\sum (N - Stride)$  for stride = 1, 2, ..., N/2

$$\sum_{i=0}^{\log_2(N)-1} (N - 2^i) = N * \log_2(N) - (N - 1) = 4096 * 12 - (4096 - 1)$$

Is it possible perform scan on the same amount of data (4\*1024 floating point numbers) using Kogge-Stone kernel directly in a single block? Why or why not?

No. It would require 4096 threads which exceeds the maximum number of threads per block.

(E) (2 points) We already learned in class that both Kogge-Stone and Brent-Kung scan kernel can speedup the scan operations. What is the main drawback of the Kogge-Stone scan kernel? What is the main drawback of the Brent-Kung scan kernel?

Kogge-Stone: The use of extra hardware resources could be very problematic. If the number of input elements is large, then the efficiency will be limited by the hardware resources.

Brent-Kung: The active threads doing “useful” work drops quickly thus wasting hardware resources. / Control divergence.

Name: \_\_\_\_\_

**Question 4. Sparse Matrix Multiplication (15 points, suggested time allocation 25 minutes):**

This question tests your knowledge of Sparse Matrix representation and operation. We first give you the kernel codes for CSR, ELL, JDS, and JDS-T formats (You may not need to read through all the kernels. They are here help you to recall what we discussed in lecture). Based on the given code, please answer the multiple-choice questions and short answer questions below. Please note that in this problem, you have to give explanation for each question for full points.

**CSR Kernel:**

```
1. __global__ void SpMV_CSR(int num_rows, float *data,
 int *col_index, int *row_ptr, float *x, float *y) {
2. int row = blockIdx.x * blockDim.x + threadIdx.x;
3. if (row < num_rows) {
4. float dot = 0;
5. int row_start = row_ptr[row];
6. int row_end = row_ptr[row+1];
7. for (int elem = row_start; elem < row_end; elem++) {
8. dot += data[elem] * x[col_index[elem]];
9. }
10. y[row] = dot;
11. }
12. }
```

**ELL Kernel:**

```
1. __global__ void SpMV_ELL (int num_rows, float *data, int *col_index, int
num_elem, float *x, float *y) {
2. int row = blockIdx.x * blockDim.x + threadIdx.x;
3. if (row < num_rows) {
4. float dot = 0;
5. for (int i = 0; i < num_elem; i++) {
6. dot += data[row+i*num_rows] * x[col_index[row+i*num_rows]];
7. }
8. y[row] = dot;
9. }
10. }
```

**JDS Kernel:**

```
1. __global__ void SpMV_JDS(int num_rows, float *data,
 int *col_index, int *jds_row_ptr,int *jds_row_perm,
 float **x, float *y) {
2. int row = blockIdx.x * blockDim.x + threadIdx.x;
3. if (row < num_rows) {
4. float dot = 0;
5. int row_start = jds_row_ptr[row];
6. int row_end = jds_row_ptr[row+1];
7. for (int elem = row_start; elem < row_end; elem++) {
```

Name: \_\_\_\_\_

```

8. dot += data[elem] * x[col_index[elem]];
9. }
9. y[jds_row_perm[row]] = dot;
}
}

```

**JDS-T Kernel:**

```

1.__global__ void SpMV_JDS_T(int num_rows, float *data,
 int *col_index, int *jds_t_col_ptr, int *jds_row_perm,
 float *x, float *y) {
2. int row = blockIdx.x * blockDim.x + threadIdx.x;
3. if (row < num_rows) {
4. float dot = 0;
5. unsigned int sec = 0;
6. while (jds_t_col_ptr[sec+1]-jds_t_col_ptr[sec] > row){
7. dot += data[jds_t_col_ptr[sec]+row]*
8. x[col_index[jds_t_col_ptr[sec]+row]];
9. sec++;
}
10. y[jds_row_perm[row]] = dot;
}
}

```

(A) (2 points) If we want to have a sparse matrix multiplication kernel which minimizes the control divergence between threads (given number of threads generated is a multiple of 32), which kernel above is our **best** option and give explanation?

**CSR**

**ELL**

**JDS**

**JDS-T**

Explanation: By padding in the ELL format which makes every row has the same number of element, the ELL format has no control divergence when the number of threads generated is a multiple of 32.

(B) (3 points) In terms of memory accessing, generally speaking, which kernel (or kernels) access the memory for the input matrix (the data array in this problem) in a coalesced manner? (Circle all possible choices and give explanation)

**CSR**

**ELL**

**JDS**

**JDS-T**

Explanation: ELL format and JDS-T format have the coalesced memory access which means adjacent thread access the adjacent memory location.

$$\begin{bmatrix} 3 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 0 & 0 & 0 & 9 \end{bmatrix}$$

For the following questions, consider the original dense form of matrix = Assuming all the kernels are invoked using:

```
dim3 BlockDim (4, 1, 1);
```

Name: \_\_\_\_\_

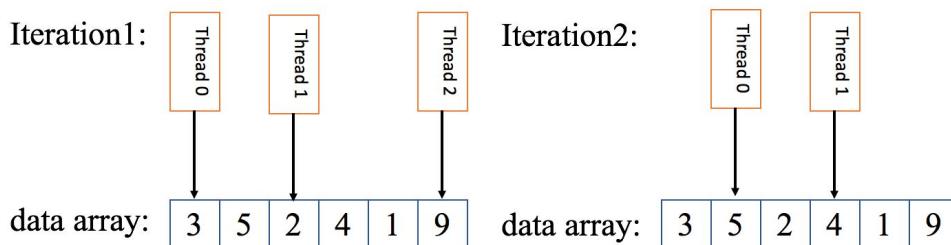
```
dim3 GridDim (1, 1, 1);
```

Draw a diagram for each of the following sparse matrix format to show how the data is stored and the access pattern of all threads in first two iterations.

We give you an example for the **CSR** format. You need to draw similar diagrams for **JDS**, and **JDS-T** formats. For each part, give explanation for full points.

(Example) (0 point) **CSR** format:

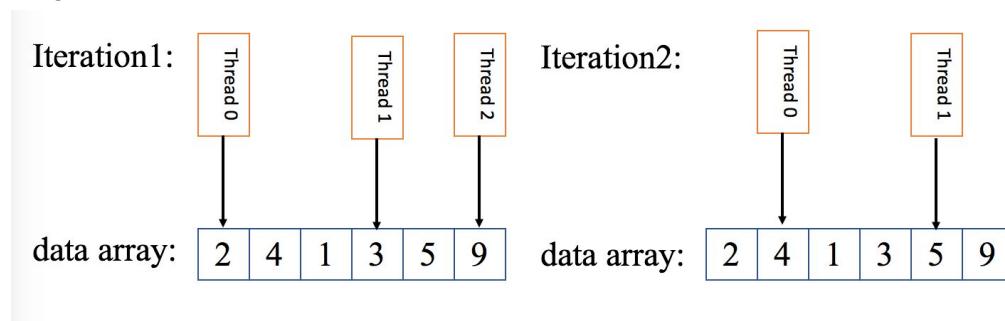
Diagram:



Explanation: For the CSR format, the data array compresses all zero elements and store non-zeros row-by-row. Each thread in CSR takes care of one row in the original matrix and for each iteration, each thread calculates one element in that row.

(C) (5 points) **JDS** format:

Diagram:

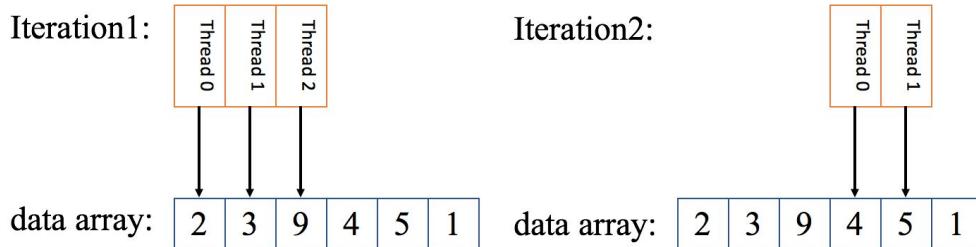


Explanation: From CSR to JDS, we sort the rows according to the length (say longest->shortest). Thus, the order of the data array would be changed accordingly. The thread access pattern is like CSR format: each thread in CSR takes care of one row in the original matrix and for each iteration, each thread calculates one element in that row.

(D) (5 points) **JDS-T** format:

Diagram:

Name: \_\_\_\_\_



Explanation: For the JDS-T format, we further transpose the data in JDS format and use a column pointer to store the start position of each row after the transpose. After this transposition, each thread still access same set of element but with memory coalesced access pattern.

Name: \_\_\_\_\_

**Question 5. Convolutional Neural Network(CNN) (15 points, suggested time allocation 25 minutes):**

A basic convolution layer consists of filter W, input X and output Y. We want to accelerate the forward propagation of convolution layers in the training process.

- W is the convolution filter weight tensor, organized a tensor  $W[M, C, K, K]$ , where
  - M is the number of output feature maps,
  - C is the number of input feature maps,
  - K is the height and width of each filter.

Tensors are stored as multi-dimensional arrays in the memory.

- X is the input feature map, organized as a tensor  $X[B, C, H, W]$ , where
  - B is the number of images in one mini-batch (recall that mini-batch is used to efficiently updates gradients while keeping relatively fast convergence),
  - H is the height of each input feature map and
  - W is the width of each input feature map.
- Y is the output feature map, organized as a tensor  $Y[B, M, H_{out}, W_{out}]$ , where
  - $H_{out} = H - K + 1$  is the height of each output feature map and
  - $W_{out} = W - K + 1$  is the width of each output feature map.

For all questions below assume the CNN is implemented using convolution.

- (A) (2 points) Consider the following 2 declarations for the block dimension:

`dim3 WHDim(W_out, H_out, 1);`

`dim3 HWDim(H_out, W_out, 1);`

Which declaration would cause the CNN to execute quicker and why?

Answer: WHDim because since the minor (vertical) dimension is mapped to the consecutive threads in the x-dimension, memory accesses will be coalesced

- (B) (3 points) Consider the following 2 declarations for the block dimension where the first one maps threads to the output feature map elements and the second one maps threads to the input feature map elements:

`dim3 outputDim(W_out, H_out, 1);`

`dim3 inputDim(W, H, 1);`

If you use shared memory in your kernel, name one advantage of each declaration.

Answer: outputDim maps to the output so it will use fewer threads, this also has less control divergence if using shared memory and strategy 2. inputDim maps to the input so it can load the shared memory in a single pass and gives inherent memory coalescing in the load as opposed to outputDim which would have to be adjusted

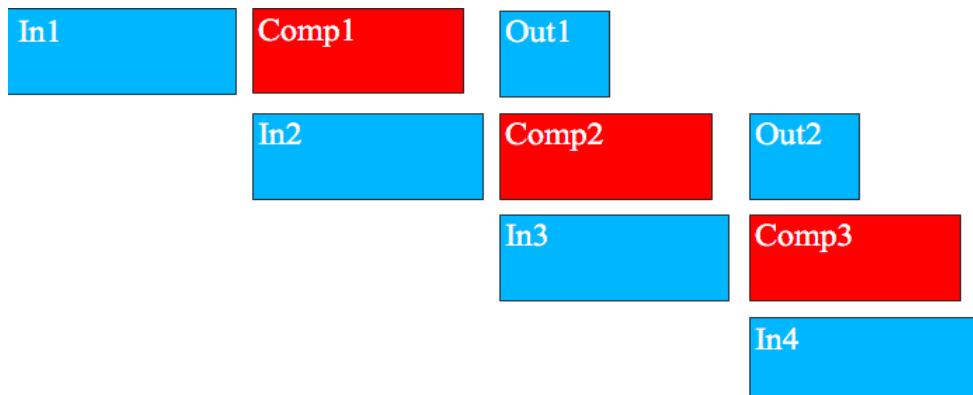
Name: \_\_\_\_\_

(C) (3 points) Consider the case where the convolution filter weight tensor W is too big to fit into constant memory. We decide to instead put it into shared memory and call it Wshared. Assuming the weights are single-precision floating-point numbers, what is the minimum amount of shared memory in bytes we need to allocate for each block and how many copies of Wshared will be allocated and loaded in the kernel execution with the following block and grid dimensions.

```
dim3 gridDim(B, M, C);
dim3 blockDim(W_out, H_out, 1);
```

Answer: Wshared needs a minimum of  $K * K * 4$  bytes large since we parallelize across batch, output feature, and input feature so we only need a single piece of the filter. We will generate  $B * M * C$  copies of Wshared which is the number of blocks.

(D) (3 points) Consider the case where we decide to use 3 CUDA Streams and are not given the memory already copied on the device for us like in the project. In the case where the memory transfer of the required input for a given computation takes longer than the computation itself and the time to transfer the output is even shorter, which of the following input memory section sizes will have the lowest execution time assuming no time overhead for launching of kernels for each section and that the weight tensor W is already in the device constant memory? Explain why. This is illustrated for you below.



- a) `sizeof(float) * B * C * H * W bytes`
- b) `sizeof(float) * B/3 * C * H * W bytes`
- c) `sizeof(float) * 1 * C * H * W bytes`
- d) all of the above are equivalent

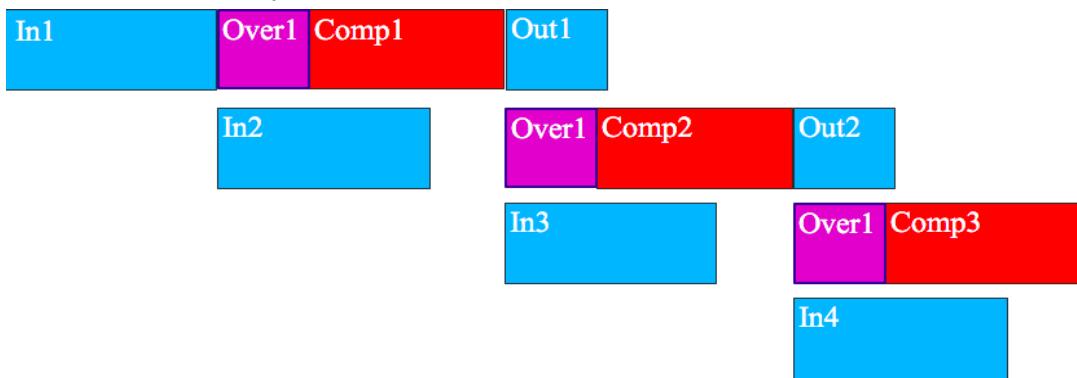
Answer: c. For the case of a, once we load all of X we can compute the entire output so this is equivalent to the case of a single stream and will take the longest. For cases b and c, the computation's time cost and output transfer time cost for each

Name: \_\_\_\_\_

stream iteration is masked by the input data transfer meaning the length of time overall will be the time it takes to load all of the input, plus the time it takes to perform the computation of 1 section, and finally the time it takes to transfer the last section. This means that the overall time is minimized by having smaller sections, so c will run faster.

Name: \_\_\_\_\_

(E) (4 points) Consider now that we have a constant cost overhead to begin a computation and it is large enough that the the computation plus the overhead for an input section takes longer than the memory transfer of the input section. Assuming B is 15 and the time to transfer the entire input is 45 seconds and the time to transfer the entire output is 15 seconds, will your answer change for certain sized overhead? Explain why and how long the time the overhead must be if yes or why not if no. This is illustrated for you below.



Answer: It can. (b) will run faster than (c) if the overhead takes longer than 4/3 seconds. (a) will run faster than (c) if the overhead takes longer than 4 seconds. (a) will run faster than (b) if the overhead takes longer than 20 seconds. From the previous scheme, now the input transfer time of the next stream is masked by the computation of the current stream, meaning the total time will be the time it takes to perform the entire computation, plus the time it takes to transfer the first input section and the last output section. The total execution time can be defined as Computation + #Sections \* Overhead + (Input + Output Transfer) / # Sections. Thus the difference between any two schemes with X and Y sections is  $(X - Y) * \text{Overhead} + (Y - X)(\text{Input} + \text{Output}) / XY$ , which results in a trade off between  $(X - Y) * \text{Overhead}$  and  $(Y - X)(\text{Input} + \text{Output}) / XY$ . If  $(X - Y) * \text{Overhead} > (Y - X)(\text{Input} + \text{Output}) / XY$ , then the overhead is too costly and Y is better than X. Putting in X as 15 and Y as 3, we get  $45 * \text{Overhead} > 60$ , or  $\text{Overhead} > 4/3$  to switch from c to b. If we put X as 15 and Y as 1, we get  $14 * \text{Overhead} > 14 * 4$  or  $\text{Overhead} > 4$  to switch from c to a. If we put X as 3 and Y as 1 we get  $2 * \text{Overhead} > 2 * 20$  or  $\text{Overhead} > 20$  to switch from b to a. Note that this answer is independent of how long the computation takes and only depends on the number of sections and time for memory transfer and overhead. This is because we are trying to balance the benefit of cutting the transfer into sections vs. the cost of adding more overhead.

Name: \_\_\_\_\_

**Question 6. Use of visual profiler to analyze the performance of kernel execution. (15 points, suggested time allocation 20 minutes):**

The profile shown on the next page was generated using nvprof and the NVIDIA Visual Profiler:

- It is annotated with two timestamps and four durations.
- It is a matrix multiplication of a [250x80] [80x1000] = [250x1000] matrix of floats, requiring approximately 40 million floating-point operations.

The times have been adjusted to make the math easier without a calculator.

You may find the following equalities useful when computing the answers

$$8 / 7 = 1.14$$

$$32/7 = 4.58$$

$$8 / 2.5 = 3.2$$

(A) (3 points) How many GFLOPS (billion floating-point operations per second) does the GPU kernel achieve?

$$4e7 \text{ flop} / 2.5e-4\text{s} = 1.6e11 \text{ flops} = 160 \text{ GFlops}$$

(B) (4 points) From the perspective of the host, the *performance* is the number of floating-point operations divided by the elapsed time between the beginning of the first cudaMemcpy to the end of the last cudaMemcpy. What is the *performance* of this matrix multiplication?

$$4e10 \text{ flops} = 4e7 \text{ flop} / 1e-3\text{s} = 40 \text{ GFlops}$$

(C) (4 points) If the kernel was optimized to produce a speedup of 2, what would the peZXs

$$4e7 \text{ flop} / ((7/8) * 1e-3) = 32/7 * 1e10 \text{ flops} = 45.7 \text{ GFlops}$$

OR: 1.14x speed up

(D) (4 points) Estimate the host-to-device performance on the CPU to GPU link, assuming the [250x80] float matrix was copied using the annotated (0.025ms) cudaMemcpy

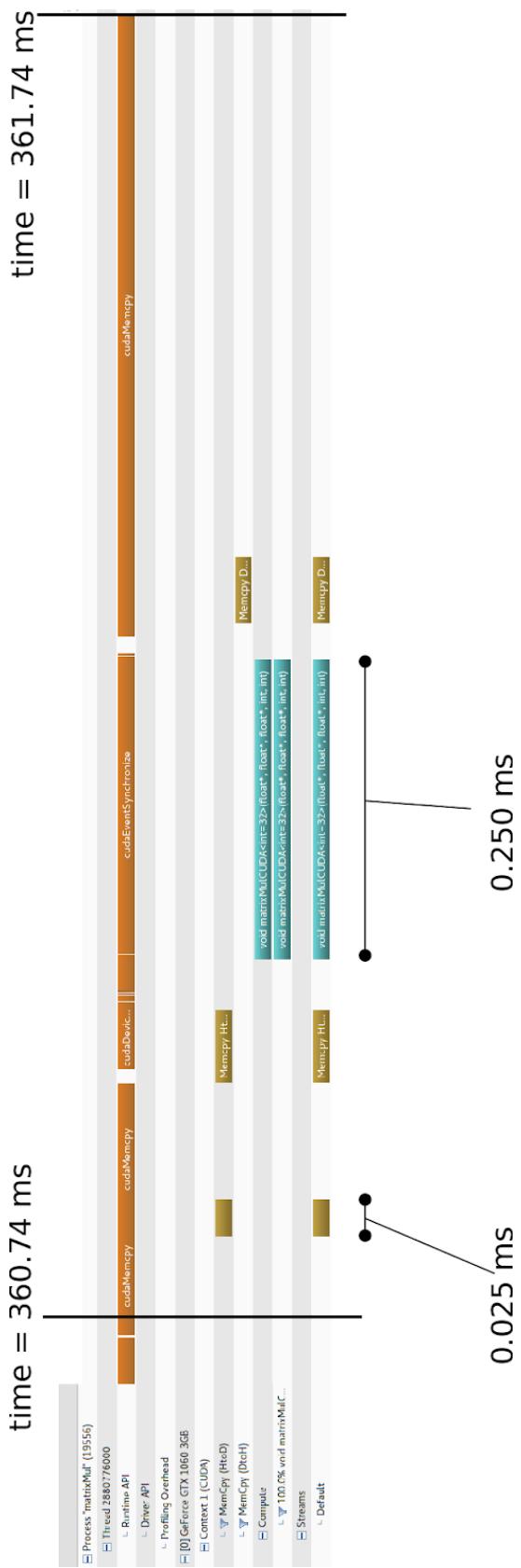
$$8e4 \text{ bytes} / 2.5e-5\text{s} = 8e9 / 2.5 = 3.2e9 \text{ B/s (or 3.2 GB/s)}$$

OR: 0.8 G floating point per second

(0 pt if no unit)

(no partial credit for this question)

Name: \_\_\_\_\_



# ECE 408 Exam 2, Fall 2017

December 12th, 2017

- You are allowed one 8.0x11.5 cheat sheet with notes on both sides. The minimal font size for your text on the cheat sheet should be 8pts.
- No interactions with humans other than course staff are allowed.
- This exam is designed to take 150 minutes to complete. To allow for any unforeseen difficulties, we will consider giving everyone up to 180 minutes.
- This exam is based on lectures, textbook chapters, as well as lab MPs/projects.
- The questions are randomly selected from the topics we covered.
- You can write down the reasoning behind your answers for possible partial credit.

Good luck!

Name: \_\_\_\_\_

Netid: \_\_\_\_\_

UIN: \_\_\_\_\_

Question 1: \_\_\_\_\_

Question 2: \_\_\_\_\_

Question 3: \_\_\_\_\_

Question 4: \_\_\_\_\_

Question 5: \_\_\_\_\_

Question 6: \_\_\_\_\_

Name: \_\_\_\_\_

**Question 1 (30 points, 40 minutes):** multiple-choice and short-answer questions. If you get more than 30 points by answering all questions (1-10), your score will saturate at 30 points. The bonus question is extra.

For multiple-choice questions, give a concise explanation for your answer for possible partial credit. Answer each of the short-answer questions in as few words as you can. Your answer will be graded based on completeness, correctness, and conciseness.

1. (3 points) For the Brent-Kung scan kernel based on reduction trees and inverse reduction trees, assume that we have 2048 elements, which of the following values is the closest to the total number of useful floating-point add operations performed in both the reduction tree phase and the inverse reduction tree phase?  
(A)  $(1024-1)*2$   
(B)  $(2048-1)*2 - 11$   
(C)  $1024*2048$   
(D)  $2048*11$

Answer: (B)

Explanation: For N elements, the reduction tree performs  $N-1$  floating-point add operations performed in the reduction tree phase and the inverse reduction performs  $N-1-\log(N)$  floating-point add operations. So the total is  $2*(N-1) - \log(N)$ .

2. (4 points) For the Brent-Kung scan kernel based on reduction trees and inverse reduction trees, assume that we have 2048 elements in each section and warp size is 32, how many warps in each block will have control divergence during the reduction tree phase iteration where stride is 64?  
(A) 0  
(B) 1  
(C) 16  
(D) 32

Answer: (B)

Explanation: All active threads are consecutive starting with index 0. There are a total of 1024 threads and 16 active threads in each block when the stride is 64. The active threads form half of warp zero. All other warps are completely inactive. Only warp 0 has control divergence.

3. (4 points) For a processor that supports atomic operations in L2 cache, assume that each atomic operation takes 5ns to complete in L2 cache and 500ns to complete in

Name: \_\_\_\_\_

DRAM. Assume that 99.9% of the atomic operations hit in L2 cache. What is the approximate throughput for atomic operations on the same global memory variable?

- (A) 1/500 G atomic operations per second
- (B) 1/50 G atomic operations per second
- (C) 1/5 G atomic operations per second
- (D) 1/13.6 G atomic operations per second

Answer: (C)

Explanation: The average latency is  $5\text{ns} * 99.9\% + 500\text{ns} * 0.1\% = 5.5\text{ns}$ . The average throughput is approximately 1/5 G atomic operations per second

4. (3 points) In Question 3, if a kernel performs 10 floating-point operations per atomic operation, what is the approximate maximal floating-point throughput of the kernel execution as limited by the throughput of the atomic operations?
- (A) 1 GFLOPS
  - (B) 2 GFLOPS
  - (C) 10 GFLOPS
  - (D) 50 GFLOPS

Answer: (B)

Explanation: 10 floating-point operations every 5.5 ns, approximately  $10/(5\text{ns}) = 2 \text{ GFLOPS}$

5. (3 points) Given a sparse matrix of integers with R original rows, L non-zero elements in the original row with the largest number of non-zeros, and a total of N non-zeros. How many integers are needed to represent the matrix in JDS-T? Assume that we keep track of the number of non-zeros in each original row, as we specified in the MP assignment.
- (A)  $R+L+N$
  - (B)  $2R+L+N+1$
  - (C)  $2R+L+2N$
  - (D)  $2R+2L+N$

Answer: (C)

Explanation: We need N integers for the non-zero elements, N integers for the column indices of the non-zero elements, L pointers to the beginning of each sorted column after transposition, R integers to track the length of each row, and R integers to track the original row index before permutation

6. (4 points) For a sparse matrix-vector multiplication (SpMV) with R rows, a total of N non-zero elements, and a maximal of L non-zeros in each row, how many times is each matrix element used?

Name: \_\_\_\_\_

- (A) 1
- (B) 2
- (C) R
- (D) L

Answer: (A)

Explanation: Each matrix element will be used only once

7. (4 points) For a sparse matrix-vector multiplication (SpMV) with R rows, C columns, a total of N non-zero elements, how many times will each vector element be used?

- (A) 1
- (B) N/R on average
- (C) N/C on average
- (D) R

Answer: (C)

Explanation: A total of N accesses will be made to the vector elements. There are C of them. So each us used N/C on average.

8. (4 points) Keven has a 320MB array that he would like to process with GPU. He measured that the execution time of the code on CPU was 0.2 seconds. He also implemented a kernel and measured that the kernel execution on the GPU with the data in the GPU memory was 0.004 seconds, a 50x speedup! However, he needs to transfer the data into the GPU memory and transfer 320MB data output data back to the host memory. His system has a PCIe Gen3 x16 interconnect. What would the real speedup be?

- (A) 40x speedup
- (B) 20x speedup
- (C) 5x speedup
- (D) 1.5x slow down

Answer: (C)

Explanation: The data copy will take  $(320M)/(16*1000M) = 0.02$  sec to copy the input data in and another 0.02 sec to copy the output data back. Thus the total speedup is  $0.2/(0.004+0.02+0.02) \approx 0.2/(0.04) = 5$ x speedup

9. (2 points) For the following host code sequence:

- 1) `cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),..., stream0);`
- 2) `cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),..., stream0);`
- 3) `cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),..., stream1);`
- 4) `cudaMemcpyAsync(h_C+i+SegSize, d_C1, SegSize*sizeof(float),..., stream1);`
- 5) `cudaMemcpyAsync(d_A2, h_A+i+2*SegSize, SegSize*sizeof(float),..., stream2);`
- 6) `cudaMemcpyAsync(h_C+i+2*SegSize, d_C2, SegSize*sizeof(float),..., stream2);`

Name: \_\_\_\_\_

Which of the statements could be executed in parallel on the GPU

- (A) 1) and 2)
- (B) 2) and 3)
- (C) 1) and 3)
- (D) 2) and 4)

Answer: (B)

Explanation: API operations in different streams can be executed in parallel. However, there is only one PCIe copy engine in each direction. So, only 2) and 3) or go in parallel

10. (2 points) When your parallel reduction kernel generates a slightly different result than a sequential reduction function for a float input array, what would be the most likely reason?

- (A) The hardware failed
- (B) There is a missing \_\_syncthreads() call
- (C) Floating-point operations are not necessarily commutative nor associative
- (D) CPU and GPUs have different precision for float numbers

Answer: The operation order has changed, and in one of the orders, one or more additions involved a very small operand and a much larger operand.

(Bonus 2 points) List the errors and typos that you reported via Piazza postings, e-mails, or in person communication with Prof. Hwu or Carl Pearson. (0.5 points for each item.)

Each one worth 1/2

**Question 2 (15 points, suggested time allocation 20 minutes):** This question tests your understanding of parallel histogram computation and privatization.

You are the owner of a supermarket and want to know the distribution of the price tags of your merchandise. So you decided to build a histogram with intervals of \$5 (e.g. \$0 <= histo[0] < \$5, \$5 <= histo[1] < \$10 and so on) on a GPU. However, the shared memory can only accommodate 256 bins for each block. As a compromise, you decide to privatize the first 256 of the bins into the shared memory. Whenever the data value doesn't fall in the first 256 bins, you

Name: \_\_\_\_\_

will have to increment the global bins. Assume that the prices are integer values and the **global histogram array is sized to accommodate the prices of all the items**. Also, assume that all global histogram elements have been initialized to zero before the kernel is launched.

(A) (3 Points) Complete the following kernel to implement the partial privatization of the histogram.

```
1. /* histo_kernel is launched with the following parameters
2.
3. dim3 gridDim(8);
4. dim3 blockDim(256);
5. */
6.
7. __global__ void histo_kernel(unsigned int *prices, long size, unsigned int *histo){
8. __shared__ unsigned int histo_private[256];
9.
10. // Reset histogram
11. histo_private[__threadIdx.x (.+.)] = 0;
12. __syncthreads();
13.
14. int i = threadIdx.x + blockIdx.x * blockDim.x;
15. // stride is total number of threads
16. int stride = blockDim.x * gridDim.x;
17.
18. while (i < size) {
19. if (prices[i] < 1280 (.+.)) atomicAdd(&(histo[prices[i] / 5]) (.+.), 1);
20. else atomicAdd(&(histo[prices[i] / 5]) (.+.), 1);
21. i += stride;
22. }
23. __syncthreads();
24.
25. // contribute to global histogram
26. atomicAdd(&(histo[threadIdx.x]) (.+.) , histo_private[threadIdx.x] (.+.));
27. }
```

(B) (4 Points) how many **non-atomic global memory reads/writes** and **shared-memory/global-memory atomic operations** are being performed by all the threads executing your kernel if there are **4096** items in total which are all priced less than \$1280?

Non-atomic Global Memory reads: 4096

Non-atomic Global Memory writes: 0

Shared-memory atomic operations: 4096

Global-memory atomic operations: 8 \* 256

Explanation:

Name: \_\_\_\_\_

(C) (4 Points) how many **non-atomic global memory reads/writes** and **shared-memory/global-memory atomic operations** are being performed by all the threads executing your kernel if there are **4000** items in total which are all priced less than \$1280?

**Non-atomic Global Memory reads:** **4000**

**Non-atomic Global Memory writes:** **0**

**Shared-memory atomic operations:** **4000**

**Global-memory atomic operations:** **8 \* 256**

Explanation:

(D) (4 Points) how many **non-atomic global memory reads/writes** and **shared-memory/global-memory atomic operations** are being performed by your kernel if there are 4000 items are priced less than \$1280 and 96 items are priced above \$1280?

**Non-atomic Global Memory reads:** **4096**

**Non-atomic Global Memory writes:** **0**

**Shared-memory atomic operations:** **4000**

**Global-memory atomic operations:** **8 \* 256 + 96**

Explanation:

Name: \_\_\_\_\_

**Question 3: Parallelization (15 points, suggested time allocation 25 minutes):**

Consider the **dense** matrix-vector multiplication  $\mathbf{Ax} = \mathbf{b}$ . The  $i^{\text{th}}$  element of  $\mathbf{b}$  is the dot product of  $\mathbf{x}$  with the  $i^{\text{th}}$  row of  $\mathbf{A}$ .

- $\mathbf{A}$  is a pointer to a  $\text{numRows} \times \text{numCols}$  row-major matrix,
- $\mathbf{b}$  is a pointer to a vector of length  $\text{numRows}$  with all entries initialized to 0, and
- $\mathbf{x}$  is a pointer to a vector of length  $\text{numCols}$ .

The following is a CPU sequential code that needs to be parallelized:

```
void mv_cpu(float *b, const float *A, const float *x, const int numRows,
 const int numCols) {
 for (int colIdx = 0; colIdx < numCols; ++colIdx) {
 for (int rowIdx = 0; rowIdx < numRows; ++rowIdx) {
 b[rowIdx] += A[rowIdx * numCols + colIdx] * x[colIdx];
 }
 }
}
```

Your colleague proposes the following three parallelizations to take advantage of GPU parallelism:

```
1. /* mv_gpu_1 launched with the following parameters
2. dim3 gridDim(10);
3. dim3 blockDim(256);
4. */
5. __global__ void mv_gpu_1(float *b, const float *A, const float *x,
 const int numRows, const int numCols) {
7. const int rowStart = blockIdx.x * blockDim.x + threadIdx.x;
8.
9. for (int r = rowStart; r < numRows; r += blockDim.x * blockDim.x) {
10. float dot = 0;
11. for (int c = 0; c < numCols; ++c) {
12. dot += A[r * numCols + c] * x[c];
13. }
14. b[r] = dot;
15. }
16. }
```

```
1. /* mv_gpu_2 launched with the following parameters
2. dim3 gridDim(16);
3. dim3 blockDim(256);
4. */
5. __global__ void mv_gpu_2(float *b, const float *A, const float *x,
 const int numRows, const int numCols) {
7. int colStart = blockIdx.x * blockDim.x + threadIdx.x;
8.
9. for (int c = colStart; c < numCols; c += blockDim.x * blockDim.x) {
10. const float v = x[c];
11. for (int r = 0; r < numRows; ++r) {
```

Name: \_\_\_\_\_

```
12. atomicAdd(&b[r], A[r * numCols + c] * v);
13. }
14. }
15. }

1. /* mv_gpu_3 launched with the following parameters
2. dim3 gridDim(4, 4);
3. dim3 blockDim(16,16);
4. */
5. template <size_t BS> // Assume that BS is 16
6. __global__ void mv_gpu_3(float *b, const float *A, const float *x,
7. const int numRows, const int numCols) {
8.
9.
10. __shared__ float b_s[BS];
11. const int tx = threadIdx.x;
12. const int ty = threadIdx.y;
13. const int colStart = blockIdx.x * blockDim.x + tx;
14. const int rowStart = blockIdx.y * blockDim.y + ty;
15.
16. for (int r = rowStart; r < numRows; r += gridDim.y * blockDim.y) {
17. if (ty == 0)
18. b_s[tx] = 0;
19. __syncthreads();
20.
21. for (int c = colStart; c < numCols; c += gridDim.x * blockDim.x) {
22. atomicAdd(&b_s[ty], A[r * numCols + c] * x[c]);
23. __syncthreads();
24. }
25.
26. if (tx == 0)
27. atomicAdd(&b[r], b_s[ty]);
28. __syncthreads();
29. }
30. }
```

For each column of the following table, choose entries from the first row of that table that apply to the kernel in question.

- The first column should contain only A-D, and the second column should contain only E-J.

Name: \_\_\_\_\_

- Each box may contain 0, 1, or more than one letter.

(Hint) It might be more efficient to focus on one of (A)-(J) and determine if it is applicable to mv\_gpu\_1 through mv\_gpu\_3 before moving to the next one.

|                                         | <b>could be fast because</b>                                                                                                                                         | <b>could be slow because</b>                                                                                                                                                                                                          |
|-----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Possible choices for this column</b> | (A) Coalesced memory accesses to A<br>(B) numRows global memory writes<br>(C) Many active threads if numCols is large<br>(D) Many active threads if numRows is large | (E) contention in shared memory atomics<br>(F) Barrier synchronization<br>(G) numRows & numCols global memory writes<br>(H) redundant loads from X<br>(I) uncoalesced memory accesses to A<br>(J) contention in global memory atomics |
| <b>mv_gpu_1</b>                         | (D)<br><br>(B)                                                                                                                                                       | (I)<br><br>(H)                                                                                                                                                                                                                        |
| <b>mv_gpu_2</b>                         | (A)<br><br>(C)                                                                                                                                                       | (J)<br><br>(G)                                                                                                                                                                                                                        |
| <b>mv_gpu_3</b>                         | (D)<br><br>(A)<br><br>(C)                                                                                                                                            | (F)<br><br>(E)<br><br>(J)<br><br>(H)                                                                                                                                                                                                  |

**Question 4. Sparse Matrix Multiplication (10 points, suggested time allocation 25 minutes):**

This question tests your knowledge of Sparse Matrix representation and operation. For your convenience, we are enclosing an example that illustrates how a dense matrix is transferred to the sparse matrix in ELL format. Recall that we take CSR, pad elements to make all rows of equal length, and transpose the padded matrix.

Name: \_\_\_\_\_

**Dense Matrix:**

|       |   |   |   |   |
|-------|---|---|---|---|
| Row 0 | 3 | 0 | 1 | 0 |
| Row 1 | 0 | 0 | 0 | 0 |
| Row 2 | 0 | 2 | 4 | 1 |
| Row 3 | 1 | 0 | 0 | 1 |

**Sparse Matrix in ELL:**

data:

|   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | * | 2 | 1 | 1 | * | 4 | 1 | * | * | 1 | * |
|---|---|---|---|---|---|---|---|---|---|---|---|

col\_index:

|   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | * | 1 | 0 | 2 | * | 2 | 3 | * | * | 3 | * |
|---|---|---|---|---|---|---|---|---|---|---|---|

num\_elem = 3

num\_rows = 4

(A) (4 points) In the following ELL kernel, fill in the missing indexing expressions for accessing data (input matrix), x (input vector) and y (output vector).

```
1. __global__ void SpMV_ELL (int num_rows, float *data, int *col_index, int
num_elem, float *x, float *y) {
2. int row = blockIdx.x * blockDim.x + threadIdx.x;
3. if (row < num_rows) {
4. float dot = 0;
5. for (int i = 0; i < num_elem; i++) {
6. dot += data[_____] * x[_____];
7. }
8. y[row] = dot;
9. }
10. }
```

row+i\*num\_rows

col\_index[row+i\*num\_rows]

(B) (4 points) After we transform the matrix into ELL layout as the example shows, and launch the kernel in (A), circle one answer for each question below and **justify your choice**.

(1) Is there any control divergence (assuming num\_rows is a multiple of 32)?

Answer: **Yes** or **No**

Explanation:

Name: \_\_\_\_\_

No. With zero-padding, all rows are of the same length. All threads iterate the same number of times in the dot product loop. Thus, control flow divergence no longer occurs in warps.

(2) Is memory access coalesced?

Answer: **Yes** or **No**

Explanation:

Yes. As all the elements are arranged into column major order, all adjacent threads are accessing adjacent memory locations which enables memory coalescing.

(C) (2 points) State what the hybrid ELL-COO format is and give a situation when hybrid ELL-COO method performs better than the ELL format.

In the situation where one or a small number of rows have an exceedingly large number of nonzero elements, the ELL format will result in excessively number of padded elements. To solve this, we can use the hybrid ELL-COO method which “take away” some of the excessive elements to reduce zero-paddings.

Name: \_\_\_\_\_

**Question 5. Convolution Neural Network ( 15 points, suggested time allocation 15 minutes):**

A basic convolution layer consists of filter W, input X and output Y. We want to accelerate the forward propagation of convolution layers in the training process.

W is the convolution filter weight tensor, organized a tensor  $W[M, C, K, K]$ , M is the number of output feature maps, C is the number of input feature maps, K is the height and width of each filter. Tensors are stored as multi-dimensional arrays in the memory.

X is the input feature map, organized as a tensor  $X[B, C, H, W]$ , where B is the number of images in one mini-batch (recall that mini-batch is used to efficiently updates gradients while keeping relatively fast convergence), H is the height of each input feature map and W is the width of each input feature map.

Y is the output feature map, organized as a tensor  $Y[B, M, H_{out}, W_{out}]$ , where  $H_{out} = H - K + 1$  is the height of each output feature map and  $W_{out} = W - k + 1$  is the width of each output feature map.

- (A) You first start with a naïve implementation for the first convolutional layer given that an output feature map of  $H_{out} \times W_{out}$  ( $28 \times 28$ ) can fit in one block (so that each thread is processing one element in the output). You need to fill in the missing parts so that the convolution layer is complete.

```
// assume that the kernel will be launched with the following configuration
01: dim3 gridDim(B, M, C);
02: dim3 blockDim(W_out, H_out, 1);
03:
04: __global__ void convLayerForward_Naive(int C, int K, int W_out, int H_out,
 float* X, float* W, float* Y){
05: int b = _____;
06: int m = _____;
07: int c = _____;
08: int h = _____;
09: int w = _____;
10:
11: float acc = 0;
12: for (int p = 0; p < K; p++) // KxK filter
13: for (int q = 0; q < K; q++)
14: acc += X[b, c, _____, _____] * W[m, c, p, q];
15:
16: Y[b, m, h, w] = acc;
17: }
```

**Answer:**

```
// assume that the kernel will be launched with the following configuration
01: dim3 gridDim(B, M, C);
```

Name: \_\_\_\_\_

```
02: dim3 blockDim(W_out, H_out, 1);
03:
04: __global__ void convLayerForward_Naive(int C, int K, int W_out, int H_out,
 float* X, float* W, float* Y){
05: int b = blockIdx.x;
06: int m = blockIdx.y;
07: int c = blockIdx.z;
08: int h = threadIdx.y;
09: int w = threadIdx.x;
10:
11: float acc = 0;
12: for (int p = 0; p < K; p++) // KxK filter
13: for (int q = 0; q < K; q++)
14: acc += X[b, c, h + p, w + q] * W[m, c, p, q];
15:
16: Y[b, m, h, w] = acc;
17: }
```

- (B) You then find out that the x, y, z dimension in the thread blocks can be mapped arbitrarily. Suppose now you change blockDim to blockDim(H\_out, W\_out, 1), which line(s) in the kernel of part (1) need to be changed in order to be an correct implementation?

New gridDim:

```
01: dim3 gridDim(B, M, C);
02: dim3 blockDim(H_out, W_out, 1);
```

Line(s) need to be changed (You may not need to fill in all the lines):

Line \_\_\_\_\_, change to: \_\_\_\_\_.

Answer:

Line 8, change to: int h = threadIdx.x.

Line 9, change to: int w = threadIdx.y.

- (C) Comparing the kernel in part (2) to the kernel in part (1), would you expect the performance of kernel in part (2) to increase, decrease or stay the same? Explain why.  
**The performance will drop because memory coalesce is not achieved. When threadIdx.x is mapped to w consecutive addresses are accessed within a warp.**

Name: \_\_\_\_\_

**Question 6. Scan (15 points, suggested time allocation, 25 minutes):** You are a new hire at a Parallelism for Cheap Inc. The company specialises in creating parallel algorithms for machines with cheaper hardware. Your boss heard you took Applied Parallel Programming with the Wen-Mei Hwu and wants you show your expertise. Your boss wants you to make some changes to the Brent-Kung algorithm to further improve the efficiency of the hardware use.

You reason that you can further improve the execution efficiency by using fewer threads to do more work. For example, instead of using 1024 threads in each block to process 2048 elements in each section, you would like to use 64 threads in each block. To do this you break up the work at each level of the reduction and post-scan steps into parts.

At each level, all threads in the block will process one part of their section of input elements before moving onto the next part. For example, at the first level of the reduction tree, all 64 threads will process the first 128 elements (64 pairwise additions) as part 0. They will then move to process the next 128 elements as part 1. So the threads will iterate through 16 parts for the first level of the reduction tree.

- (A) (5 Points) With 2048 elements in each section, about how many times more computations will each thread be doing compared to a regular Brent-Kung thread on average? Answer should be in the form 1 times, 5 times, 100 times etc. Explain for full credit.

**Answer:** For the new kernel, 64 threads will do the same amount of work that used to be done by 1024 threads, so each thread will be doing 16 times more work on average.

- (B) (5 Points) Below is the skeleton code for implementing this strategy. The given code is very similar to your MP. Fill in the missing conditionals and indexing to make the code run as described in part A. Note that you may leave a blank empty if you feel nothing should go there. No explanation is necessary.

```
01: #define BLOCK_SIZE 64
02: #define SECTION_SIZE 2048
03: __global__ void scan(float *input, float *output, int len) {
04: __shared__ float shared[SECTION_SIZE];
05: int bx = blockIdx.x;
06: int tx = threadIdx.x;
07: int i = bx * SECTION_SIZE + tx;
08:
09: for(int part = 0; part < SECTION_SIZE / BLOCK_SIZE; part++) {
10: if(i + part * BLOCK_SIZE < len)
11: shared[tx + part * BLOCK_SIZE] = input[i + part * BLOCK_SIZE];
12: else
13: shared[tx + part * BLOCK_SIZE] = 0;
14: }
15:
16: for(unsigned int stride = 1; stride < SECTION_SIZE; stride *= 2) {
17: __syncthreads();
18: for(int part = 0; part < _____; part++){
19: int index = (tx+1+_____) * (stride *2) - 1 _____;
```

Name: \_\_\_\_\_

```
20: if(index < SECTION_SIZE)
21: shared[index] = shared[index] + shared[index - stride];
22: }
23: }
24:
25: for(unsigned int stride = (SECTION_SIZE)/4; stride > 0; stride /= 2) {
26: __syncthreads();
27: for(int part = 0; part < _____; part++){
28: int index = (tx+1+_____) * (stride *2) - 1 _____;
29: if(index + stride < SECTION_SIZE)
30: shared[index + stride] = shared[index + stride] + shared[index];
31: }
32: }
33:
34: __syncthreads();
35: for(int part = 0; part < SECTION_SIZE / BLOCK_SIZE; part++){
36: if(i + part * BLOCK_SIZE < len)
37: output[i + part * BLOCK_SIZE] = shared[tx + part * BLOCK_SIZE];
38: }
39: }
```

Answer:

```
#define BLOCK_SIZE 512
#define SECTION_SIZE 4096
__global__ void scan(float *input, float *output, int len) {
 __shared__ float shared[SECTION_SIZE];
 int bx = blockIdx.x;
 int tx = threadIdx.x;
 int i = bx * blockDim.x * (SECTION_SIZE / BLOCK_SIZE) + tx;

 for(int block = 0; block < SECTION_SIZE / BLOCK_SIZE; block++) {
 if(i + block * BLOCK_SIZE < len)
 shared[tx + block * BLOCK_SIZE] = input[i + block * BLOCK_SIZE];
 else
 shared[tx + block * BLOCK_SIZE] = 0;
 }

 for(unsigned int stride = 1; stride < SECTION_SIZE; stride *= 2) {
 __syncthreads();
 for(int part = 0; part < SECTION_SIZE/(BLOCK_SIZE*2); part++){
 int index = (tx+part*BLOCK_SIZE+1)*(stride*2) - 1;
 // (tx+1)*(stride*2) + part*BLOCK_SIZE * (stride*2)- 1
 if(index < SECTION_SIZE)
 shared[index] = shared[index] + shared[index - stride];
 }
 }

 for(unsigned int stride = (SECTION_SIZE)/4; stride > 0; stride /= 2) {
 __syncthreads();
 for(int part = 0; part < SECTION_SIZE/(BLOCK_SIZE*2); part++){
 int index = (tx+part*BLOCK_SIZE+1)*(stride*2) - 1;
```

Name: \_\_\_\_\_

```
// (tx+1)*(stride*2) + part*BLOCK_SIZE * (stride*2)- 1
if(index + stride < SECTION_SIZE)
 shared[index + stride] = shared[index + stride] + shared[index];
}
}

__syncthreads();
for(int block = 0; block < SECTION_SIZE / BLOCK_SIZE; block++){
 if(i + block * BLOCK_SIZE < len)
 output[i + block * BLOCK_SIZE] = shared[tx + block * BLOCK_SIZE];
}
}
```

- (C) (5 Points) Does this strategy optimally use the hardware efficiently in terms of active threads and control divergence? Explain why or why not. If it does not, how could you improve the strategy? Your answer must begin with yes or no followed by an explanation.

Answer: Yes. By separating the computations into parts, as we proceed through the reduction and post scan steps we turn off half the parts, but all of the threads will still do the same amount of work. This leads to having as many threads as possible being active at any given iteration, thus optimally using hardware efficiently. In addition, the threads that are active at any given iteration will all be adjacent minimizing control divergence.

For 2048 elements, The average number of active threads is

$(64 + 64 + 64 + 64 + 64 + 32 + 16 + 8 + 4 + 2 + 1) + (2-1) + (4-1) + (8-1) + (16-1) + (32-1) + (64-1) + 64 + 64 + 64 + 64$  which is approximately  $64*10 + 2*32 + 2*32 = 768$   
 $(16 + 8 + 4 + 2 + 1) * 64 + 32 + 16 + 8 + 4 + 2 + 1 + (16 + 8 + 4 + 2 + 1) * 64 - 1*5 + 31 + 15 + 7 + 3 + 1 / ((16 + 8 + 4 + 2 + 1 + 16 + 8 + 4 + 2 + 1 + 6 + 5) * 64)$   
which is approximately  $(64 * 64) / (73 * 64) = 64/73$ , threads are active about 88% of the time.

The average number of active threads is  $768/21 = 36$

The average portion of threads that are active is  $36/64$

For regular Brent-Kung, the average number of active threads is

$(1024 + 512 + \dots + 1 + (2-1) + (4-1) + (8-1) + \dots + (1024-1))/21$  which is approximately  
 $2*2048/21 = 195$   
 $(1024 + 512 + 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 + (2-1) + (4-1) + (8-1) + (16-1) + (32-1) + (64-1) + (128-1) + (256-1) + (512-1) + (1024-1)) / (21 * 1024)$   
which is approximately  $(2 * 2048) / (21 * 1024) = 4/21$ , threads are active about 19% of the time.

So, on average, only  $195/1024 = 19\%$  of the threads are active

Name: \_\_\_\_\_

The new Brent-Kung has 4x better execution efficiency