UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

ECE 408/CS 483 MIDTERM 1 REVIEW

Electrical & Computer Engineering

Fall 2024

ADAPTED BY HRISHI SHAH (from Thomas Bae)

10/13/2024

EXAM DETAILS

- Tuesday, October 15th, 2024, from 7:00 PM to 9:00 PM
- Closed book exam and no cheat sheets allowed
- Coverage:
  - Lecture 1 through 12 (CNN and unrolling), **including** guest lecture on profiling with Nsight
  - Labs 1 through 4 (3D Convolution)
  - Project Milestone 1

HOW TO PREPARE

- Reread lectures and write a few takeaways (also attempt end of lecture questions)
- Read through textbook for more difficult topics
- NVIDIA Documentation for basic parallel programming ideas
- Check your mistakes on previous labs and quizzes
- Make sure to understand your code, even if its from lecture
- Past exams (several provided in files section of Canvas), particularly SP24 one
- Make your own review sheet (can’t use on exam)

KEY TOPICS

- CUDA Parallel Execution Model
- CUDA Memory Model
- Tiling, Locality, and DRAM (Memory coalescing)
- Branch Divergence
- Shared Memory and Synchronization
- Matrix Multiplication
- Convolution
- AI/ML/DL
- Profiling

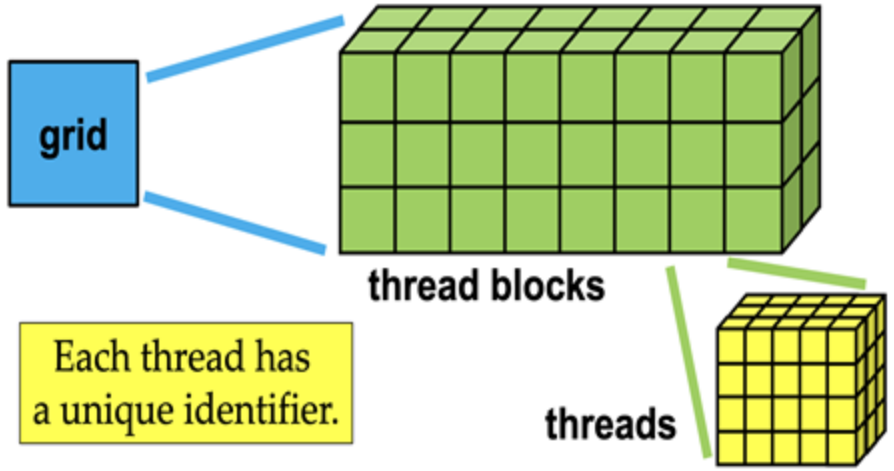
FUNDAMENTALS – GRID BLOCK THREAD

- Grid (of blocks)
- Blocks (of threads)
- Thread
  - unit of execution (kernel code)
  - runs **parallel** with other threads in a warp

Grid size refers to the # of blocks, and **block size** refers to the # of threads in a block.

In hardware, the threads (and blocks) are not really 2D or 3D, but unrolled into 1D

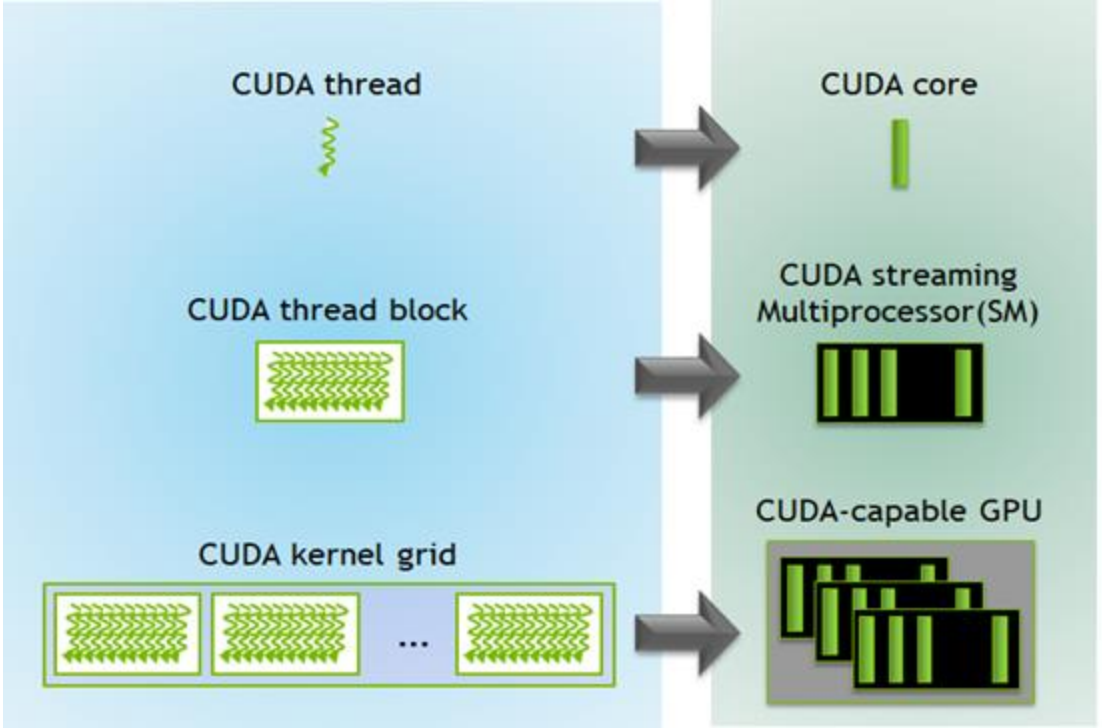
What is a warp? Why do we care?



- Unique Identifier
  - blockIdx.x / blockIdx.y / blockIdx.z
  - threadIdx.x / threadIdx.y / threadIdx.z

FROM A HARDWARE LEVEL...

- Each architecture in GPU consists of several **Streaming Multiprocessors.**
- Several thread blocks are assigned to a Streaming Multiprocessor (SM).
- The GPU houses the grid for the kernel.

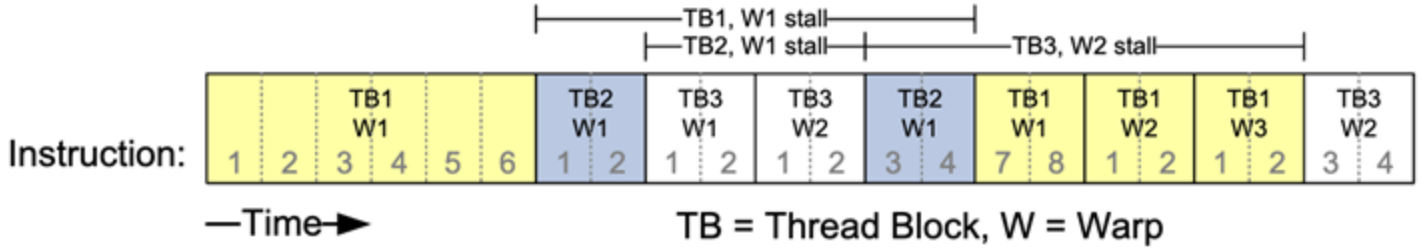


WARPS? BUT I THOUGHT WE HAD BLOCKS…

Warp is a **unit of thread scheduling** in the GPU. Streaming Multiprocessors (SM) executes **each block in 32-thread warps**.

SM implements a **zero-overhead** strategy. It selects which ever warp is ready to go!

All threads in a warp execute the same instruction when selected.



Example execution timing of an SM

How does a block get divided into warps? What does that look like?

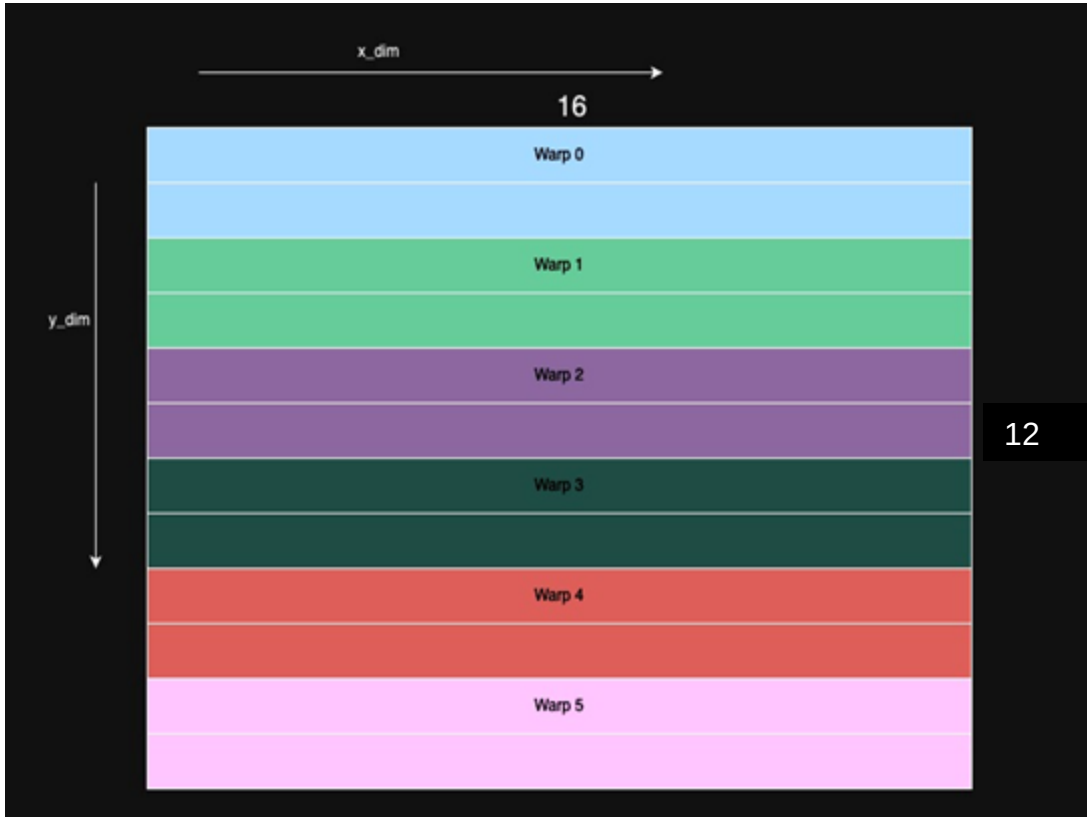
HOW DO WE DIVIDE BLOCKS INTO WARPS?

Remember, blocks are unrolled into **1D** in hardware and warps are assigned as such.

Blocks are unrolled in the order x, y, z

So, **threads 0-31** in the unrolled block will be assigned to the first warp, **32-63** assigned to second warp, etc.

What are some problems or pitfalls with warps? (Think concurrent execution)



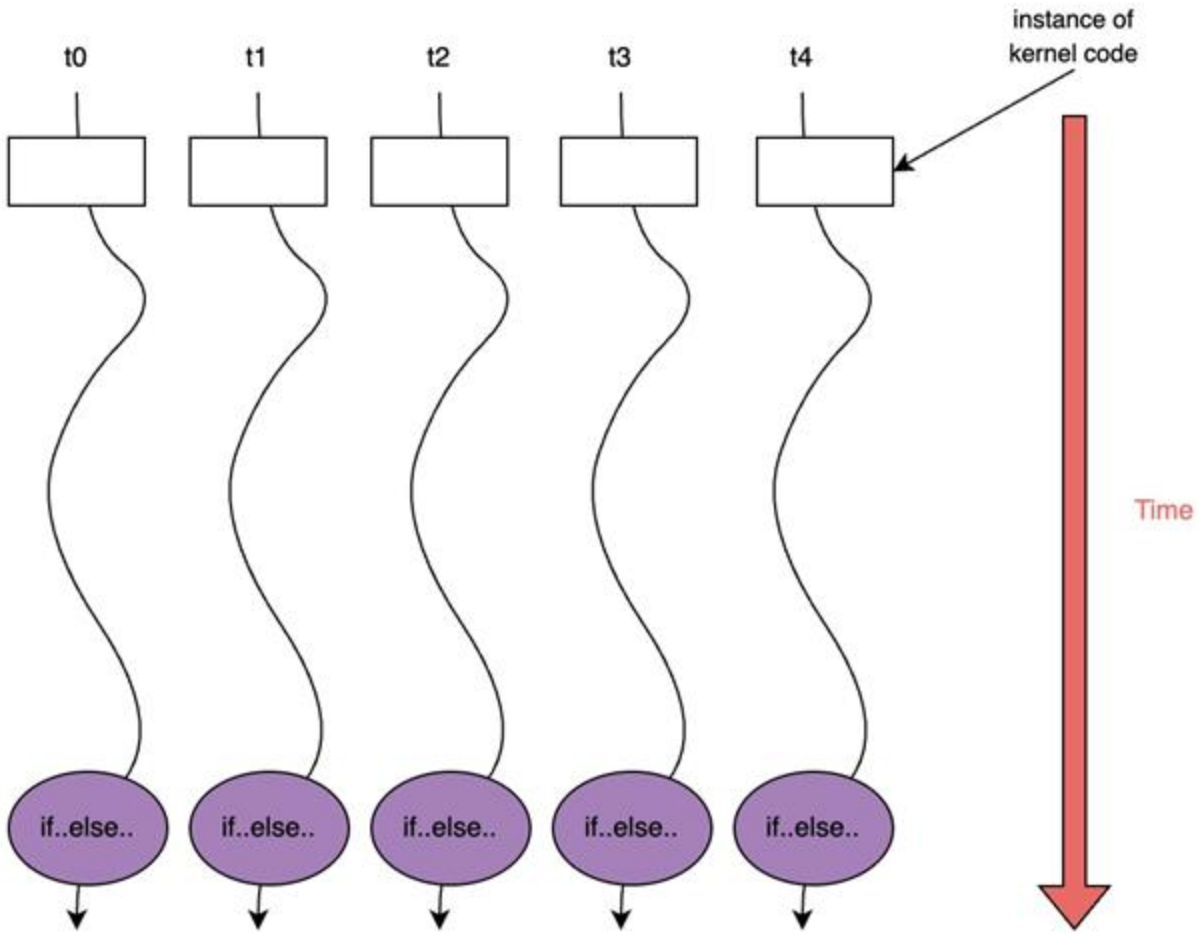
UNDERSTANDING CONTROL DIVERGENCE

Control divergence happens when threads in a **warp** take different **paths**.

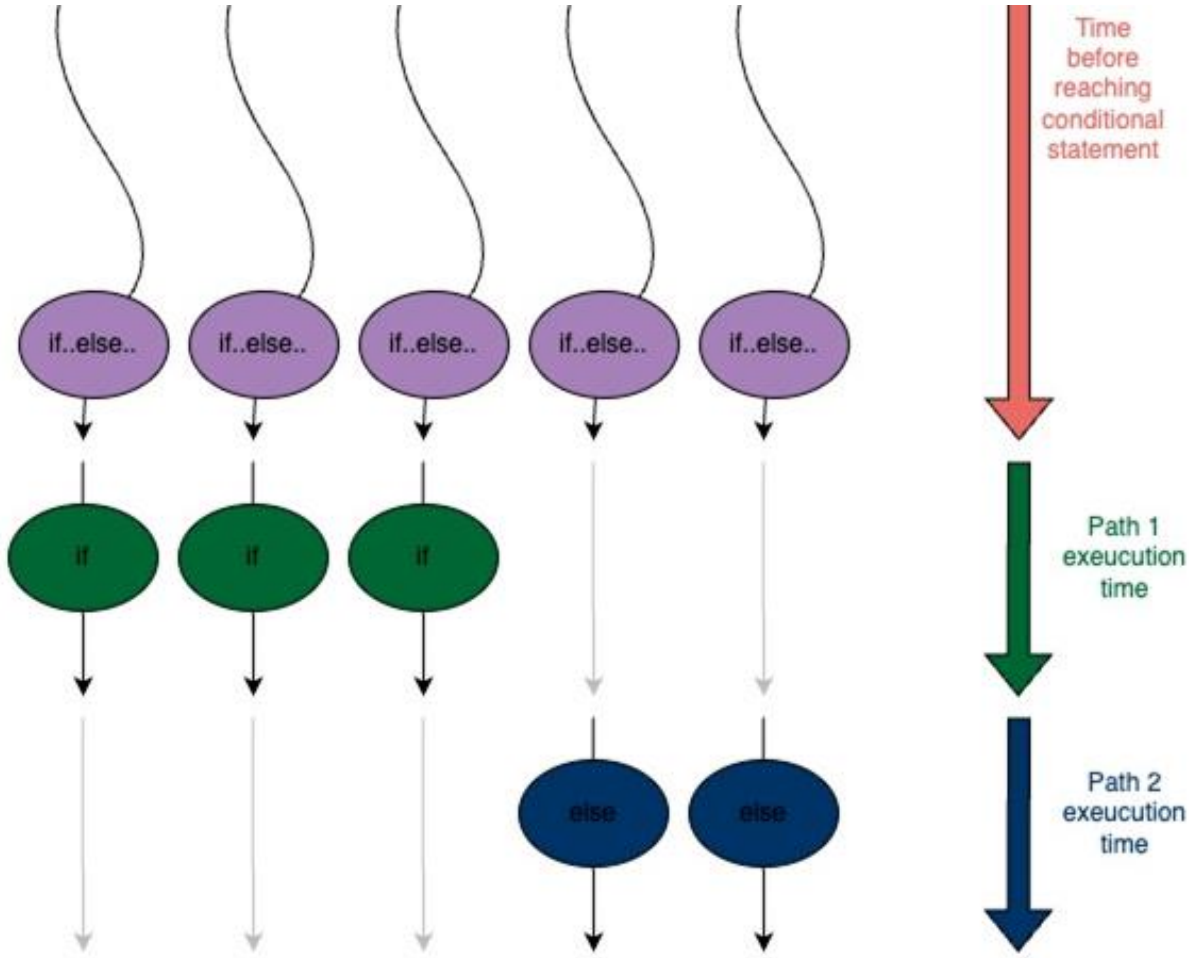
How does the warp take different paths? Why is this bad?

For example…

```
//Assume block dim is (32, 1, 1)
if (threadIdx.x < 4){
    //execute path 1
} else {
    //execute path 2
}
```



UNDERSTANDING CONTROL DIVERGENCE (cont'd)



It becomes not-so parallel…?

Threads take different paths, which means…. **sequential execution**!

If the execution time of path 1 is 50 seconds and path 2 is 5 seconds, which threads are being affected the most by control divergence?

Let's try another example…

```
//Assume block dim is (32, 4, 1)
if (threadIdx.x < 48){
    //execute path 1
} else {
    //execute path 2
}
```

```
__global__ void A(int[] io, int size) {
    unsigned int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < size) {
        io[index] *= 3;
    }
}

__global__ void B(int[] io) {
    unsigned int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (blockIdx.x < 16) {
        io[index] *= 3;
    } else {
        io[index] -= 3;
    }
}

__global__ void C(int[] io, int control) {
    unsigned int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (control != 0) {
        io[index] *= control;
    } else {
        io[index] = 0;
    }
}
```

```
__global__ void D(int[] io) {
    unsigned int index = threadIdx.x + blockIdx.x * blockDim.x;
    io[index] *= threadIdx.x > 31 ? 1 : -1;
}

__global__ void E(int[] io, int control) {
    unsigned int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (control < 0) control = 1;
    if (control * threadIdx.x < 0) {
        if (io[index] == 0) io[index] = control;
    }
    io[index] *= control;
}
```

PRACTICE PROBLEM

Suppose we are performing 1D vector addition in which each block is responsible for adding 3* BLOCK_DIM set of elements. Assume the vector size is 2500 and BLOCK_DIM is 32.

- What is the grid size?
- Write the kernel function below where each thread is responsible for every BLOCK_DIM element.

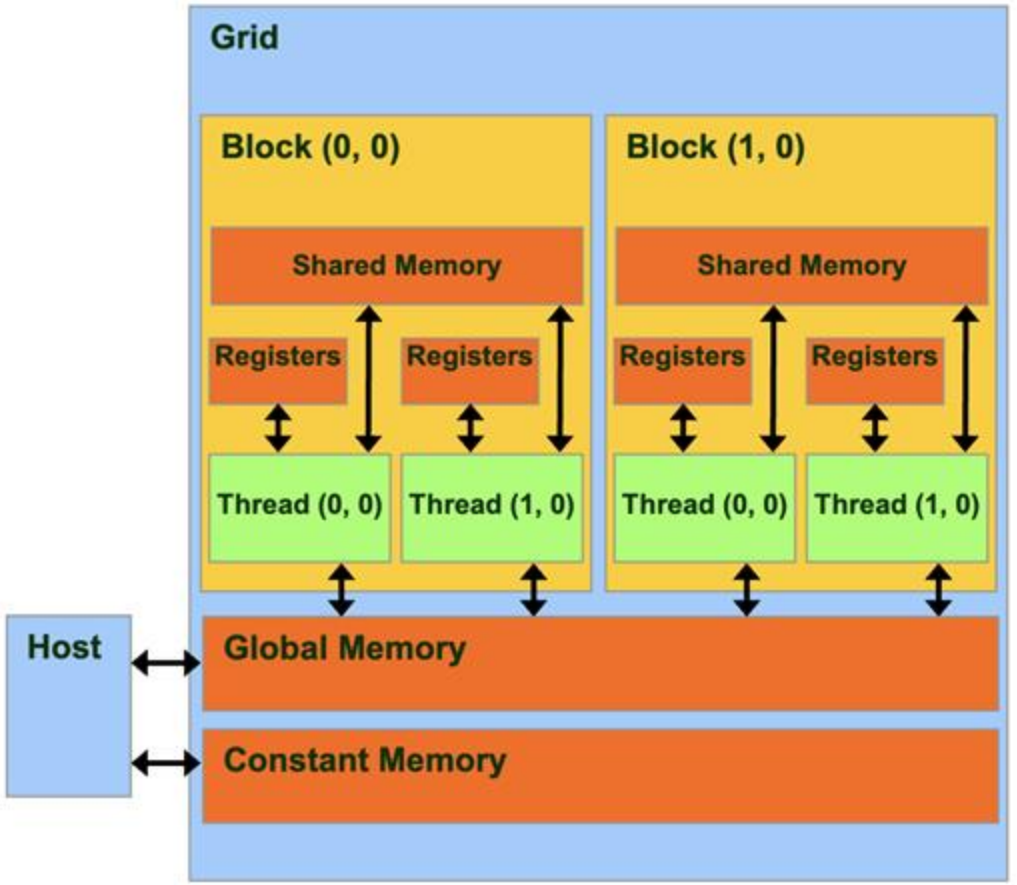
```
__global__ void vector_addition(float* in1, float* in2, float* out, int size){


}
```

Which block is responsible for adding the 167th element? Which thread of that block?

CUDA MEMORY MODEL

- per-thread registers (~1 cycle)
  - all local variables (if there are enough registers) get stores in the register
- per-block memory
  - **shared memory** (~5 cycles)
    - limited size
    - programmable L1 cache
- per-grid memory
  - **global memory (~500 cycles)**
  - **constant memory** (~5 cycles w/ caching)
    - read-only

Why do we care so much about memory usage and bandwidth in CUDA?



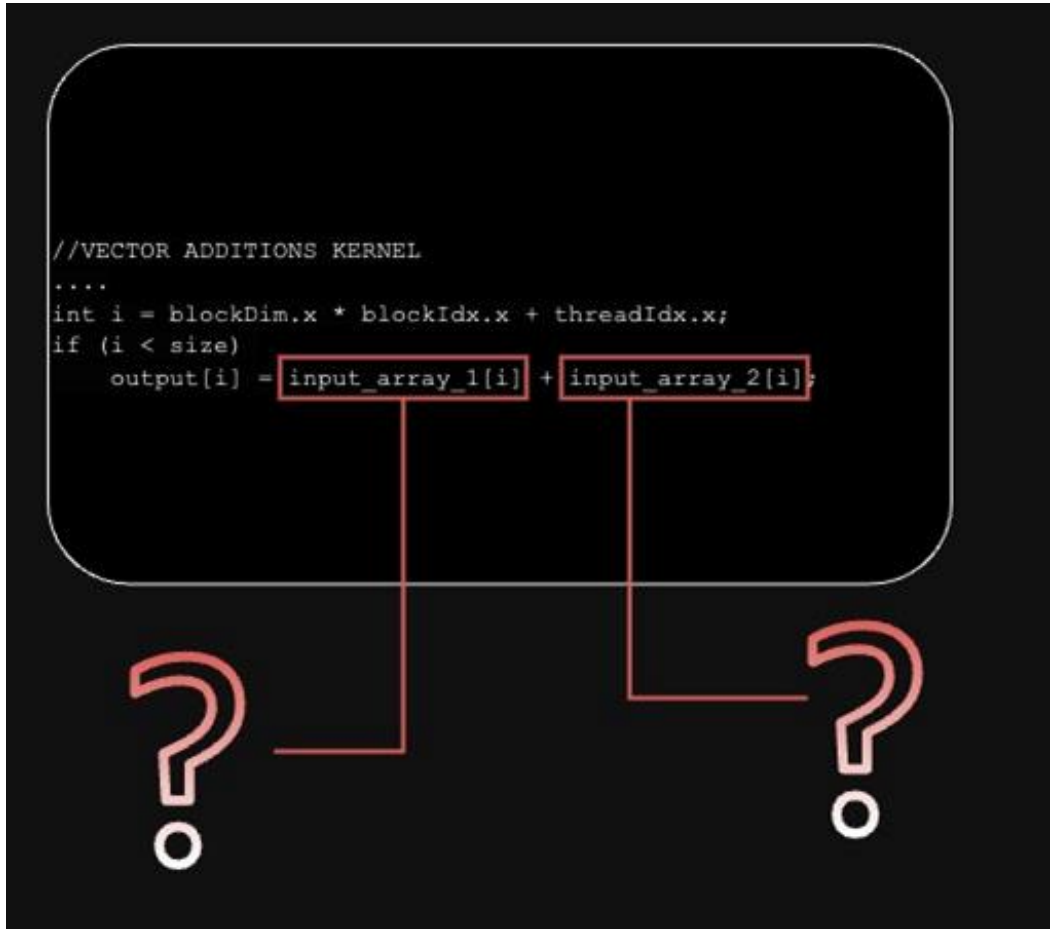
CUDA MEMORY MODEL (cont'd)

Where are we retrieving these from?

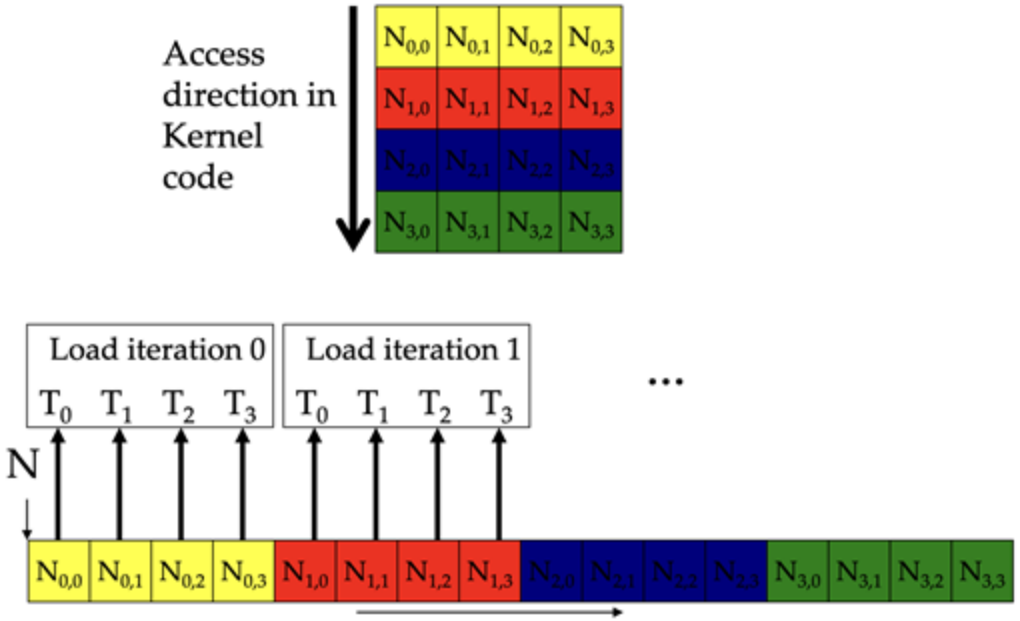
Assuming this is the most basic form of vector addition code, these are retrieved from **global memory(~500 cycles)**.

But.. imagine all these threads asking for data from global memory at the same time.. we are limited by **bandwidth.**
(Imagine matrix multiply kernel)

Is there a better way to go about this? How do we minimize global memory access?

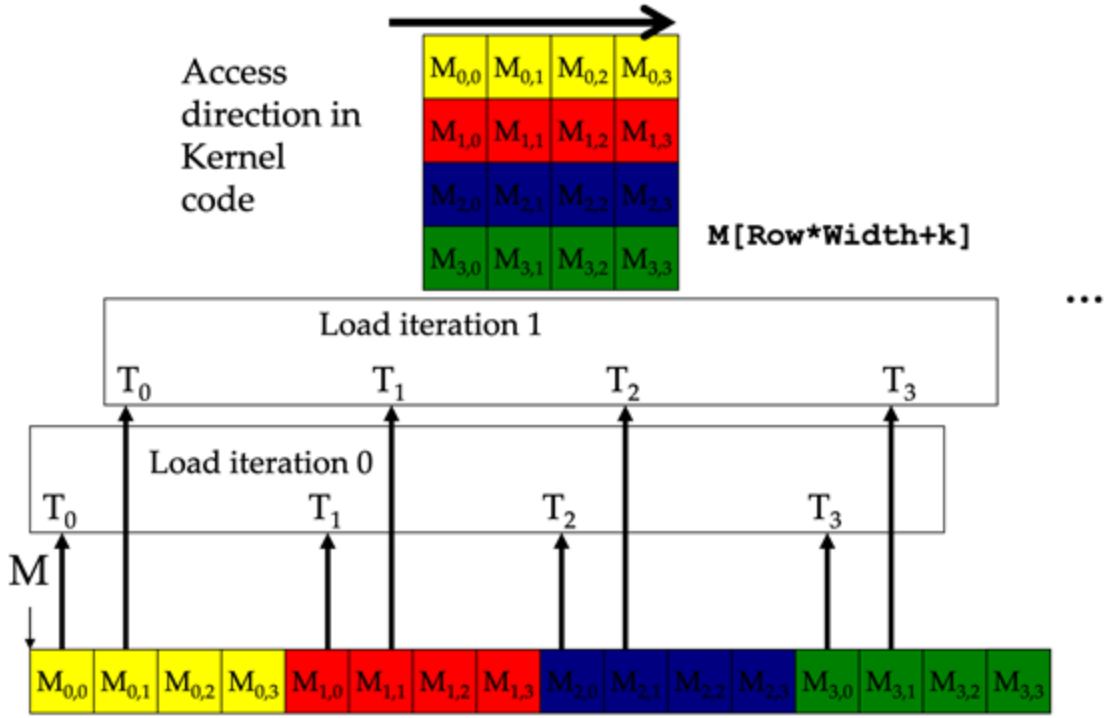


MEMORY COALESCING



VS.

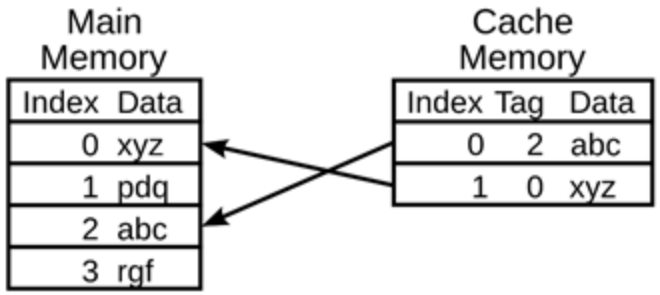
Each thread loads in an entire col each iteration



What is a DRAM burst? Which scenario utilizes a set of DRAM bursts more efficiently?

DRAM BURST AND CACHING

When we make a call to global memory, DRAM gives you back a **whole sequential line of data in bursts**. These lines of data are stored in the cache.

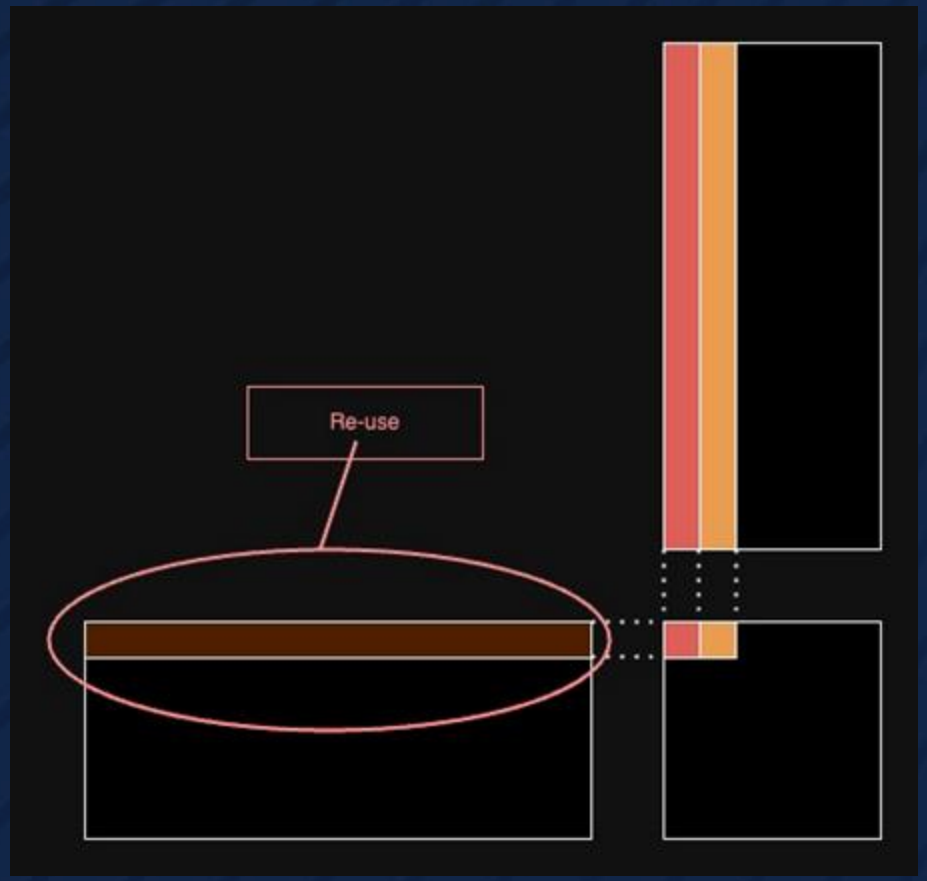


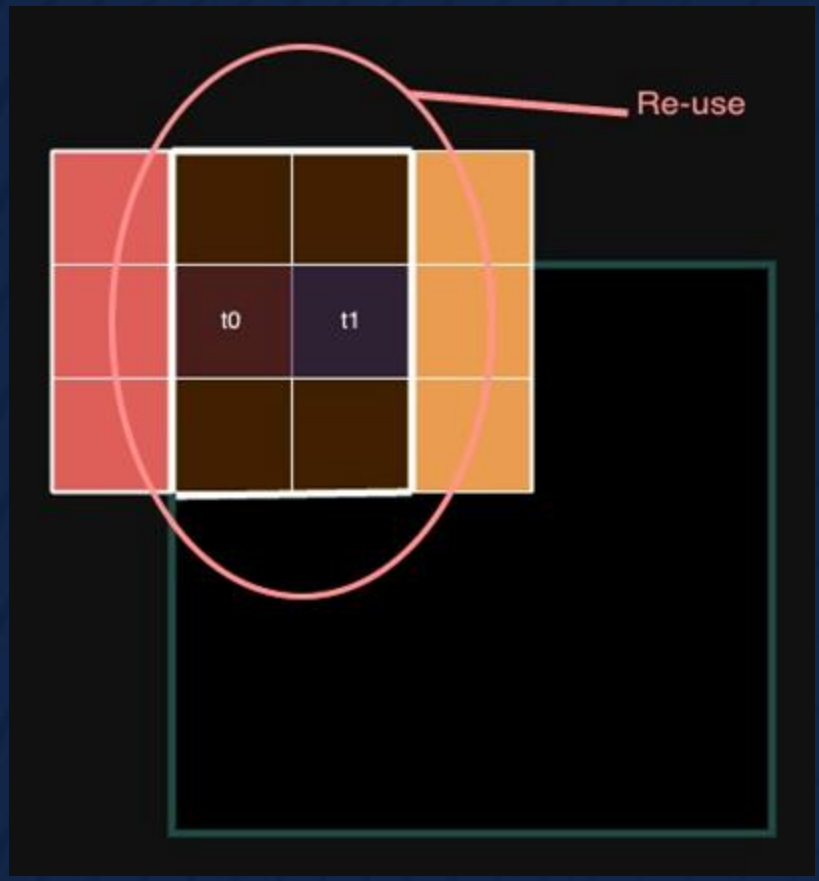
What's a cache?

- array of **cache lines**
- a global memory read produces a line
- cache stores those lines
- **spatial locality and memory coalescing**
  - threads should access nearby/consecutive elements in memory to maximize bandwidth!
- **any subsequent requests for elements that are already present in the cache doesn't have to go to global memory (faster access)**

Can we 100% guarantee memory coalescence?

SHARED MEMORY FOR IN-TILE REUSE

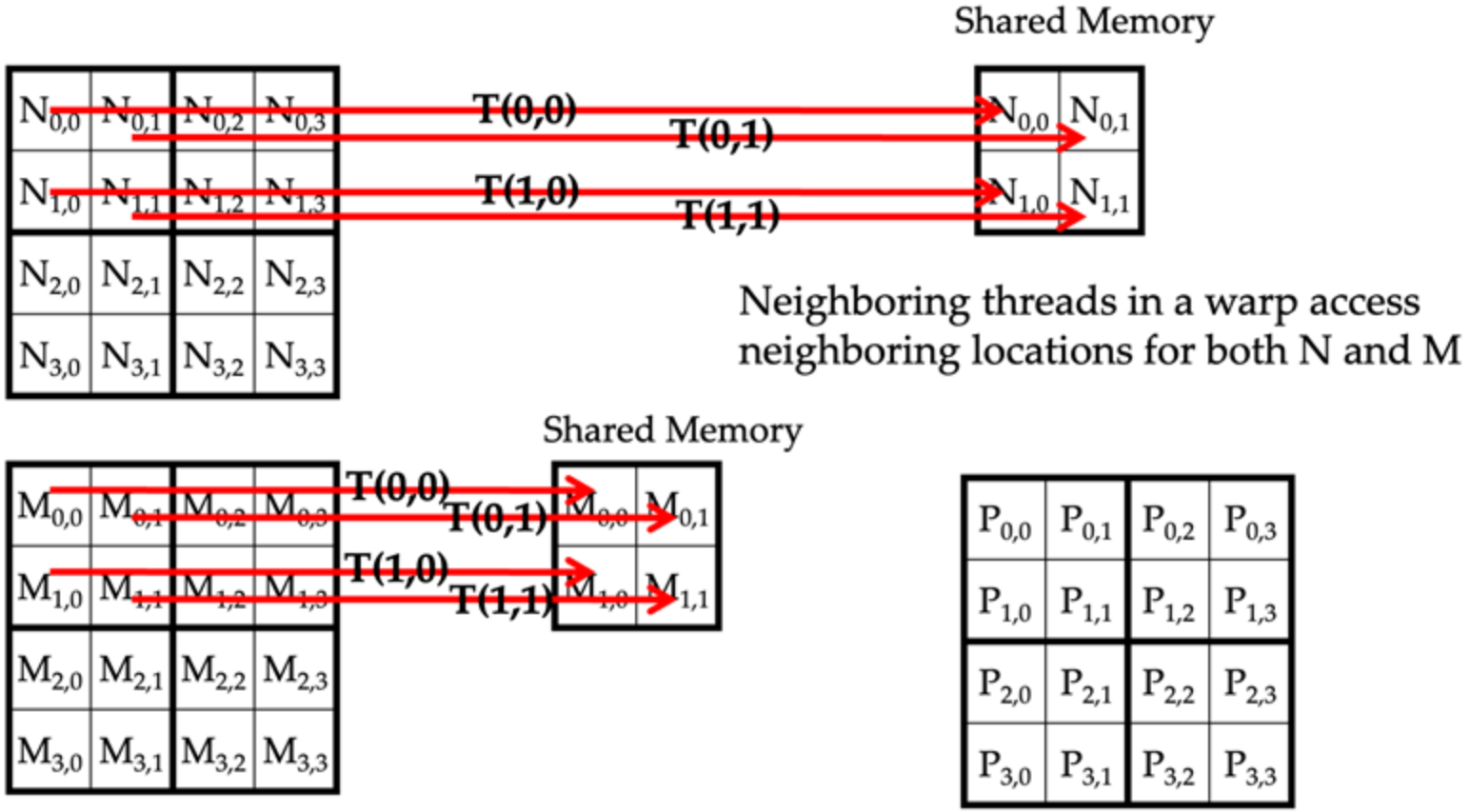




TILING (cont'd)

We want to use a "tile" in shared memory (any dimensional array in **shared memory**) so that we can have quicker access!

We can reduce the memory bandwidth usage.



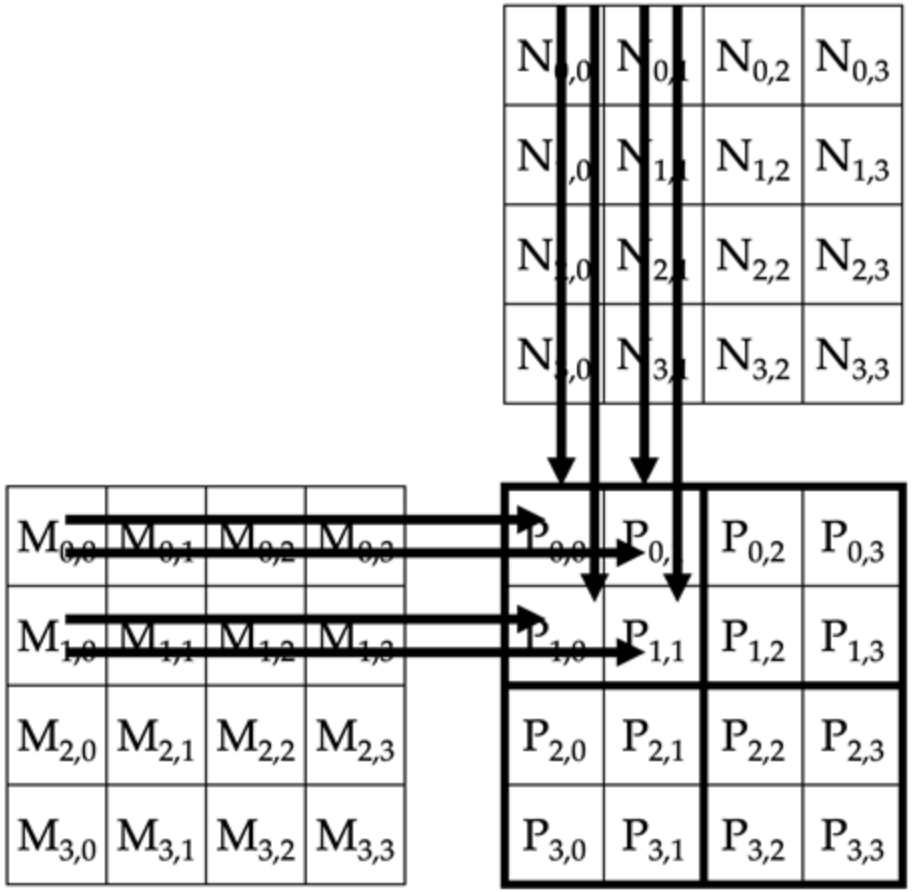
MORE PRACTICE

1. 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[4*i+2];
```

1. 240 GB/s
2. 180 GB/s
3. 120 GB/s
4. 60 GB/s
5. None of them are correct.

BRIEF MATRIX MULTIPLICATION OVERVIEW



Each element/thread of the output will read in the corresponding row and column of the **M** and **N** input matrix respectively.

As we discussed.. a lot of reuse happening. Let's do better!

Solution: we use a "tile" in shared memory so that we can have quicker access for elements that can be reused.

But that means we have to read elements first from global -> shared, then shared -> kernel. Is this better?

Reminder: ~5 cycles vs. ~500 cycles (factor of 10)

THREAD SYNCHRONIZATION

Thread barrier is formed to allow a**ll threads to reach a point** before moving on. Threads will idle (stuck in a loop) until **all threads** have called the function __syncthreads().

In context of tiled matrix multiplication:

- First __syncthreads() is needed to make sure all threads have loaded in their respective elements into shared mem.
- Second call is to make sure all threads are done with current iteration before loading in a new batch of elements. (Why?)

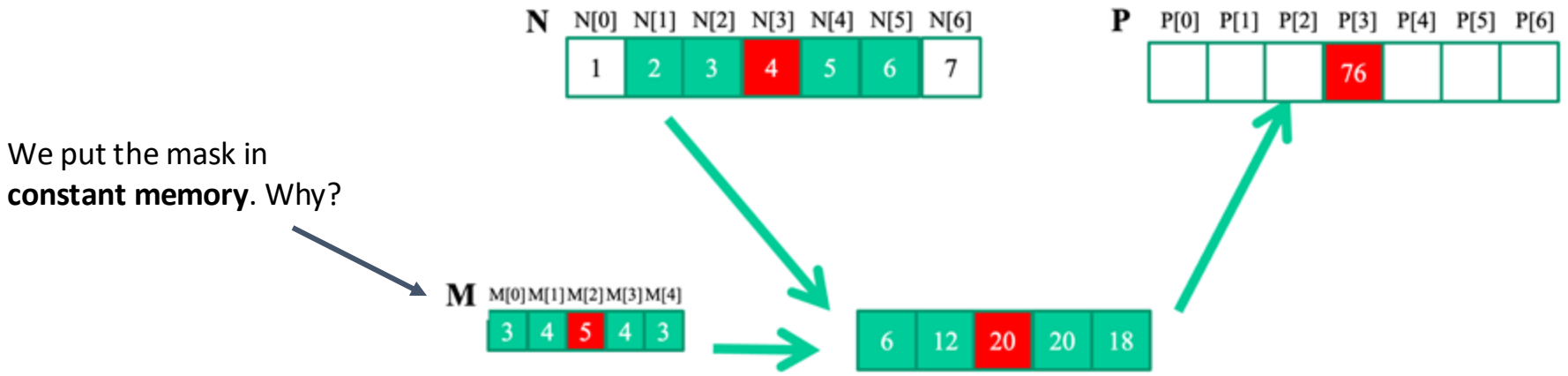
Is __syncthreads() good for performance?

```
//tiled matrix multiplication
for (int q = 0; q < ceil((float)numBRows/TILE_WIDTH); ++q){
    if (row < numARows && (q*TILE_WIDTH+tx) < numAColumns){
      subTileM[ty][tx] = A[row*numAColumns + (q*TILE_WIDTH+tx)];
    }else{
      subTileM[ty][tx] = 0;
    }
    if (col < numBColumns && (q*TILE_WIDTH+ty) < numBRows){
      subTileN[ty][tx] = B[(q*TILE_WIDTH+ty)*numBColumns+col];
    }else{
      subTileN[ty][tx] = 0;
    }
    __syncthreads();
    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += subTileM[ty][k] * subTileN[k][tx];
    __syncthreads();
  }
```

CONVOLUTION

Convolution is an array operation where **each output data element is a weighted sum of a collection of neighboring input elements**.

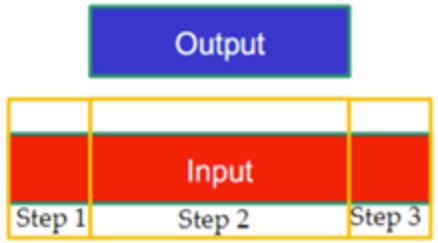
We often use a filter/mask as the weight. (**not required**)



3 DIFFERENT SHARED MEMORY STRATEGIES

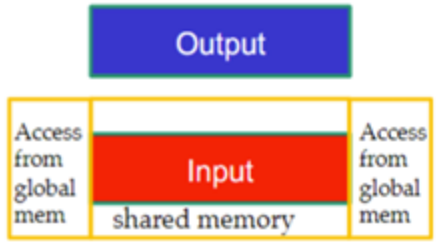
Example: we have an 1D array of 128 that we want to perform a convolution on.

- Block size: 8
- Mask size: 5



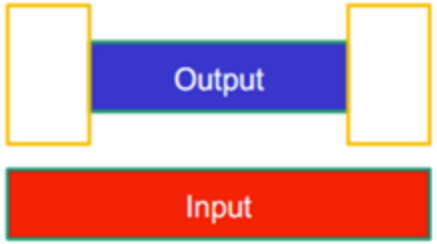
Strategy 1

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

CONVOLUTION – STRATEGY 1 – MULTIPLE STEPS

We want to have our shared memory tile (input) to **ACCOUNT** for our halo cells, but we **don't want to use extra threads to load it in**, **if we don't have to**.

- Grid size = ceil(128/8.0)
- Block size = 8
- Shared memory tile size = (8 + 5 - 1) = 12

We only have 8 threads but have to load in 12 elements. How do we go about this?

- We increase the work of every thread into **three (or even two!) parts** (for loading our shared memory tile)
  - load left halo cells
  - load actual elements
  - load right halo cells
  - **or alternatively, just load left half, then right half** (when mask radius < block size/2??)

Benefit:

- we can handle a large mask well because we don't have to turn off any threads in the computing stage

Drawbacks:

- overhead with loading shared memory (minimal)
- code complexity

What if this is 2-D? how does our loading look like? What about 3-D?

CONVOLUTION – STRATEGY 2 – EXTRA THREADS

We still want our shared memory tile to **ACCOUNT** for halo cells. But this time around, we would have to **utilize extra threads to load it all in one step**.

- Grid size = ceil(128/8.0)
- Block size = 8 + 5 - 1 = 12
- Shared memory input tile size = 8 + 5 - 1 = 12

Since block size == shared memory size, one-for-one loading for each thread! but.. we only need to computation for 8 elements (output size)

Solution: 4 out of 12 threads (right and left most 2) don't write back to output.

Benefits:
- reduce loading overhead
- simpler code

Drawbacks:
- Having certain threads "turn off" is **nonoptimal**
  - also causes control divergence

CONVOLUTION – STRATEGY 3 – GLOBAL MEMORY READS

We want to have our shared memory tile (input) to **NOT ACCOUNT** for our halo cells and retrieve halo cells **directly from global memory**.

- Grid size = ceil(128/8.0)
- Block size = 8
- Shared memory tile size = 8

This is pretty simple.. right? While doing our convolution, if it's a halo cell, fetch from global memory!

Benefit: no unnecessary threads turned off
Drawback:

- As mask gets big, large amount of time spend fetching from global memory
- **control divergence in computing stage**

If the mask size is 1, is there a difference in the number of global memory loads for strategy 3 and strategy 2?

THINGS TO THINK ABOUT…

How many times are we using/reusing halo elements?

Does this impact the way we choose strategy?

What are our limitations?

What kind of boundary checks do you need to do for each strategy? Where?

See lecture 8 and 9 for code implementation of each strategy.

CONVOLUTION PRACTICE

Suppose we have an output tile size of 16 x 16 and a mask of 9 x 5.

If we are implementing strategy 2:

- For an internal tile, how many total global memory reads throughout the execution of the kernel?
- What is the average number of times we use an element in shared memory?

If we are implementing strategy 3:

- How many total global memory reads now?
- What is the average number of times we use an element in shared memory?

Which strategy would you implement to perform this convolution? Why?

CONVOLUTION PRACTICE

Strategy 2

```
[[ 1.  2.  3.  4.  5.  5.  5.  5.  5.  5.  5.  5.  5.  5.  5.  5.  4.  3.  2.  1.]
 [ 2.  4.  6.  8. 10. 10. 10. 10. 10. 10. 10. 10. 10. 10. 10. 10.  8.  6.  4.  2.]
 [ 3.  6.  9. 12. 15. 15. 15. 15. 15. 15. 15. 15. 15. 15. 15. 15. 12.  9.  6.  3.]
 [ 4.  8. 12. 16. 20. 20. 20. 20. 20. 20. 20. 20. 20. 20. 20. 20. 16. 12.  8.  4.]
 [ 5. 10. 15. 20. 25. 25. 25. 25. 25. 25. 25. 25. 25. 25. 25. 25. 20. 15. 10.  5.]
 [ 6. 12. 18. 24. 30. 30. 30. 30. 30. 30. 30. 30. 30. 30. 30. 30. 24. 18. 12.  6.]
 [ 7. 14. 21. 28. 35. 35. 35. 35. 35. 35. 35. 35. 35. 35. 35. 35. 28. 21. 14.  7.]
 [ 8. 16. 24. 32. 40. 40. 40. 40. 40. 40. 40. 40. 40. 40. 40. 40. 32. 24. 16.  8.]
 [ 9. 18. 27. 36. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 36. 27. 18.  9.]
 [ 9. 18. 27. 36. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 36. 27. 18.  9.]
 [ 9. 18. 27. 36. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 36. 27. 18.  9.]
 [ 9. 18. 27. 36. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 36. 27. 18.  9.]
 [ 9. 18. 27. 36. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 36. 27. 18.  9.]
 [ 9. 18. 27. 36. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 36. 27. 18.  9.]
 [ 9. 18. 27. 36. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 36. 27. 18.  9.]
 [ 9. 18. 27. 36. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 36. 27. 18.  9.]
 [ 8. 16. 24. 32. 40. 40. 40. 40. 40. 40. 40. 40. 40. 40. 40. 40. 32. 24. 16.  8.]
 [ 7. 14. 21. 28. 35. 35. 35. 35. 35. 35. 35. 35. 35. 35. 35. 35. 28. 21. 14.  7.]
 [ 6. 12. 18. 24. 30. 30. 30. 30. 30. 30. 30. 30. 30. 30. 30. 30. 24. 18. 12.  6.]
 [ 5. 10. 15. 20. 25. 25. 25. 25. 25. 25. 25. 25. 25. 25. 25. 25. 20. 15. 10.  5.]
 [ 4.  8. 12. 16. 20. 20. 20. 20. 20. 20. 20. 20. 20. 20. 20. 20. 16. 12.  8.  4.]
 [ 3.  6.  9. 12. 15. 15. 15. 15. 15. 15. 15. 15. 15. 15. 15. 15. 12.  9.  6.  3.]
 [ 2.  4.  6.  8. 10. 10. 10. 10. 10. 10. 10. 10. 10. 10. 10. 10.  8.  6.  4.  2.]
 [ 1.  2.  3.  4.  5.  5.  5.  5.  5.  5.  5.  5.  5.  5.  5.  5.  4.  3.  2.  1.]]
```

Strategy 3

```
[[15. 20. 25. 25. 25. 25. 25. 25. 25. 25. 25. 25. 25. 25. 20. 15.]
 [18. 24. 30. 30. 30. 30. 30. 30. 30. 30. 30. 30. 30. 30. 24. 18.]
 [21. 28. 35. 35. 35. 35. 35. 35. 35. 35. 35. 35. 35. 35. 28. 21.]
 [24. 32. 40. 40. 40. 40. 40. 40. 40. 40. 40. 40. 40. 40. 32. 24.]
 [27. 36. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 36. 27.]
 [27. 36. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 36. 27.]
 [27. 36. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 36. 27.]
 [27. 36. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 36. 27.]
 [27. 36. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 36. 27.]
 [27. 36. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 36. 27.]
 [27. 36. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 36. 27.]
 [27. 36. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 45. 36. 27.]
 [24. 32. 40. 40. 40. 40. 40. 40. 40. 40. 40. 40. 40. 40. 32. 24.]
 [21. 28. 35. 35. 35. 35. 35. 35. 35. 35. 35. 35. 35. 35. 28. 21.]
 [18. 24. 30. 30. 30. 30. 30. 30. 30. 30. 30. 30. 30. 30. 24. 18.]
 [15. 20. 25. 25. 25. 25. 25. 25. 25. 25. 25. 25. 25. 25. 20. 15.]]
```

MACHINE LEARNING

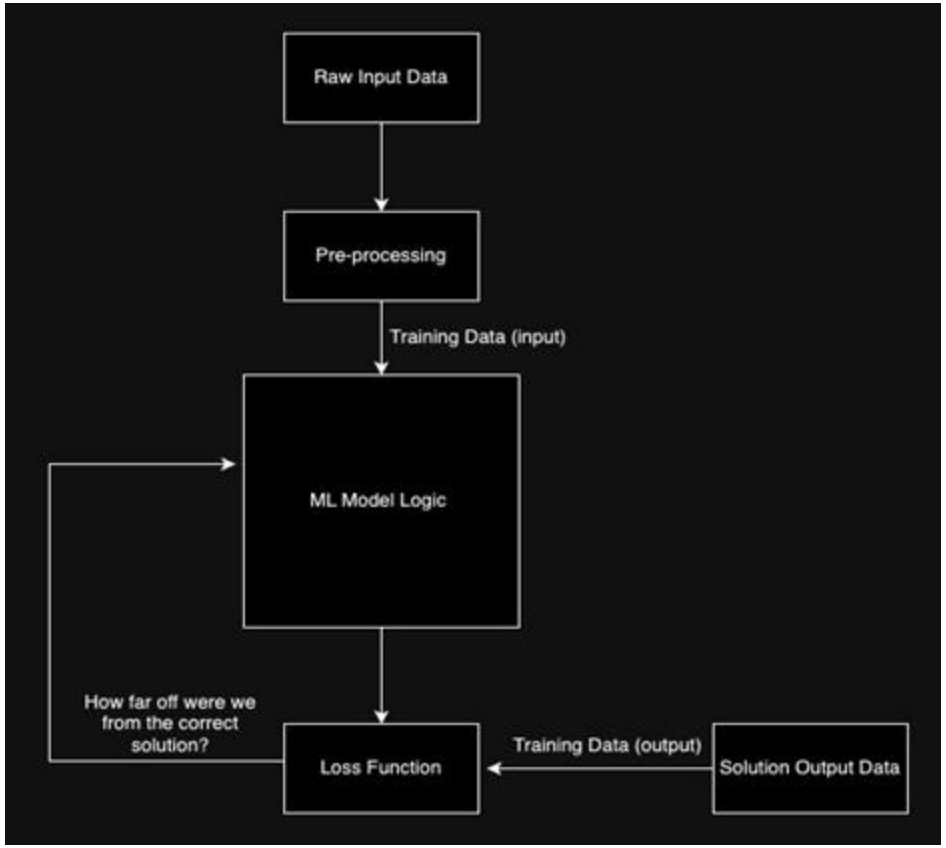
Machine learning is the process of **allowing the computers to develop their own algorithms and logic** based on a set of inputs and outputs in a training data set.

We oftentimes know the general type of logic applied in learning, but it's hard/impossible to fully understand it. (Think thousands and thousands of parameters – e.g., weight matrices)

Types of models (logic):

- linear regression (e.g., Numerical relationships)
- classification (e.g., category selection, Naive Bayes)
- language based (e.g., translation)
- neural networks (non-linear relationships... not simple math)

What are some limitations of ML?

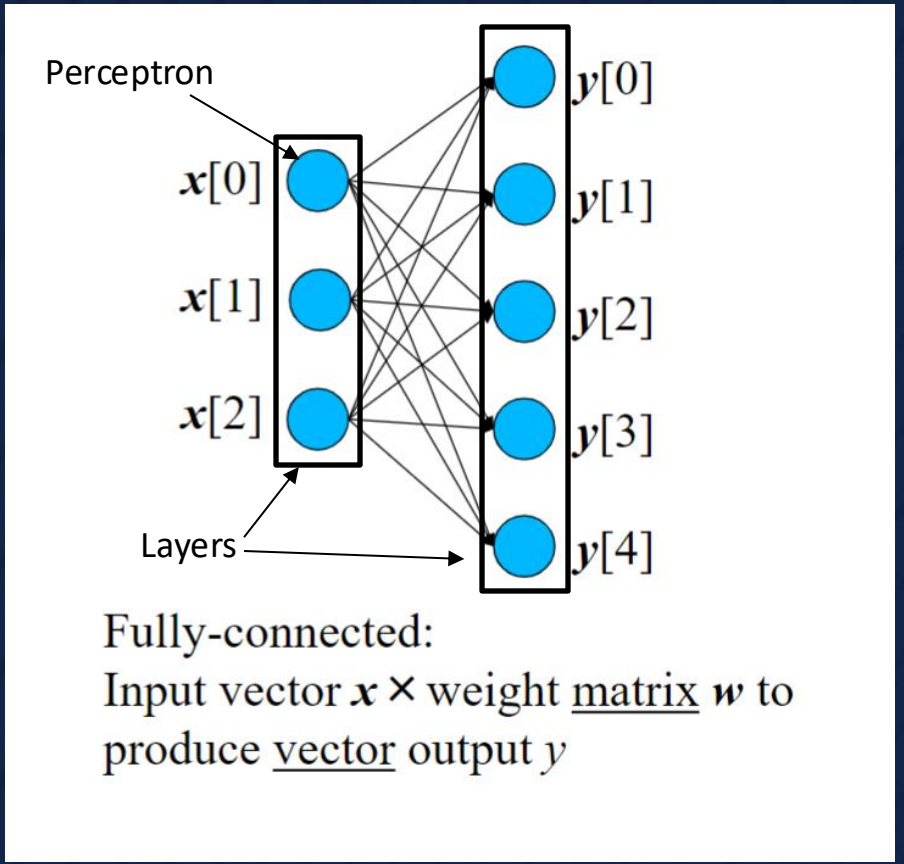


NEURAL NETWORKS

Jack of all trades! Neural networks, in our case, are formed using nodes in layers, where all nodes connect to nodes to the next layer. (A group of perceptrons)

Input elements in an input vector are multiplied with weight matrices and adjusted with bias to form an output vector.

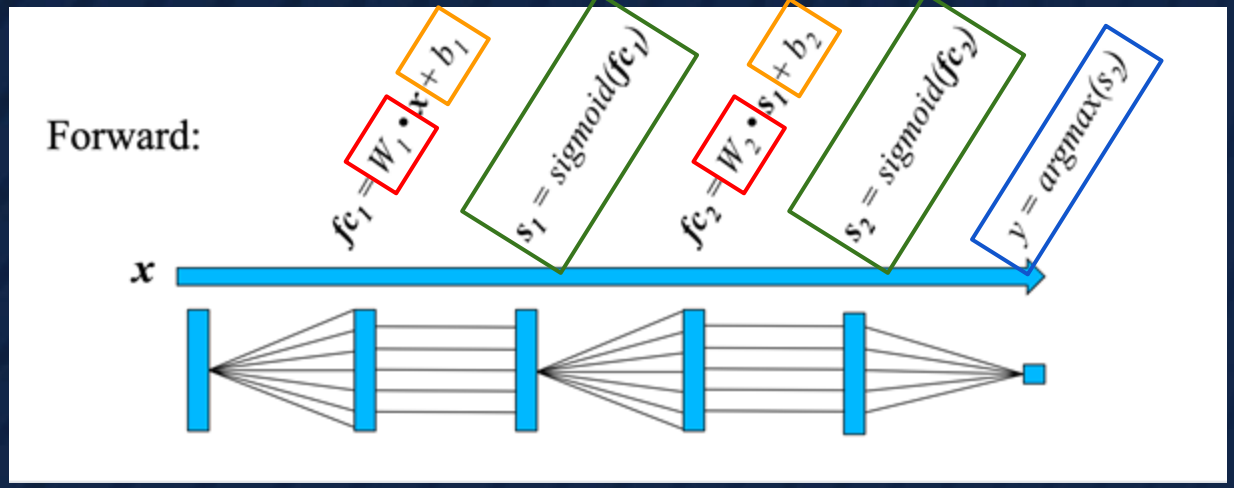
Neural networks first send data through a network of layers (forward propagation). Once the outputs are generated, it calculates how much it messed up through a loss function. It will then go back starting from the last layer and edit weight matrices through a process called gradient descent.



Forward propagation:

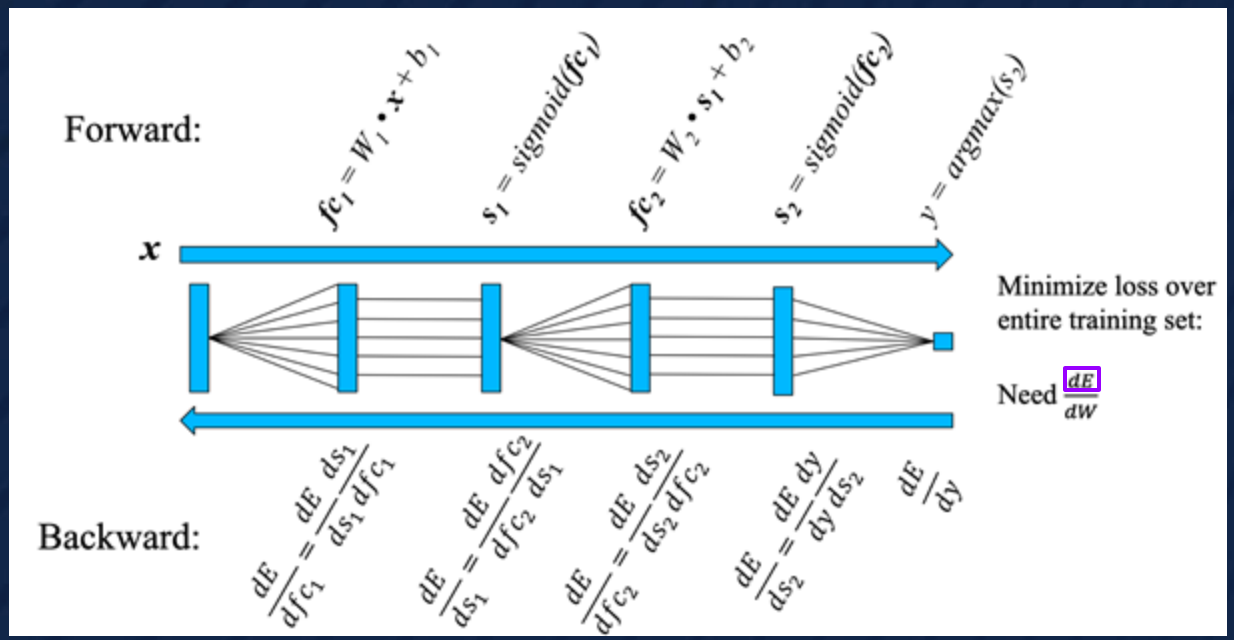
Parameters
- Weight Matrices
- Bias vectors

Functions
- Activation Function
- argmax function



Backward propagation:

We perform gradient descent starting from the output layer by calculating the derivative of the loss function in respect to a layer's parameters.



Okay... so what is an **activation function** and why do we need it?

Without an activation function, this process is no different from a linear model. Activation functions allow a NN model to capture **complex relationships** between the inputs and outputs.

It also allows the model to filter certain values to retain certain thresholds set by the function. (E.g., ReLU functions, discard negatives? What about sigmoid functions? Step functions? What purposes do they serve?)

MACHINE LEARNING PRACTICE

If we have an input image represented by 200 x 200 pixels, and we have 3 layers, each consisting of 12 nodes, how many parameters will there be in total?

Can we parallelize the forward propagation process? How?

WHERE DOES CONVOLUTION COME IN?

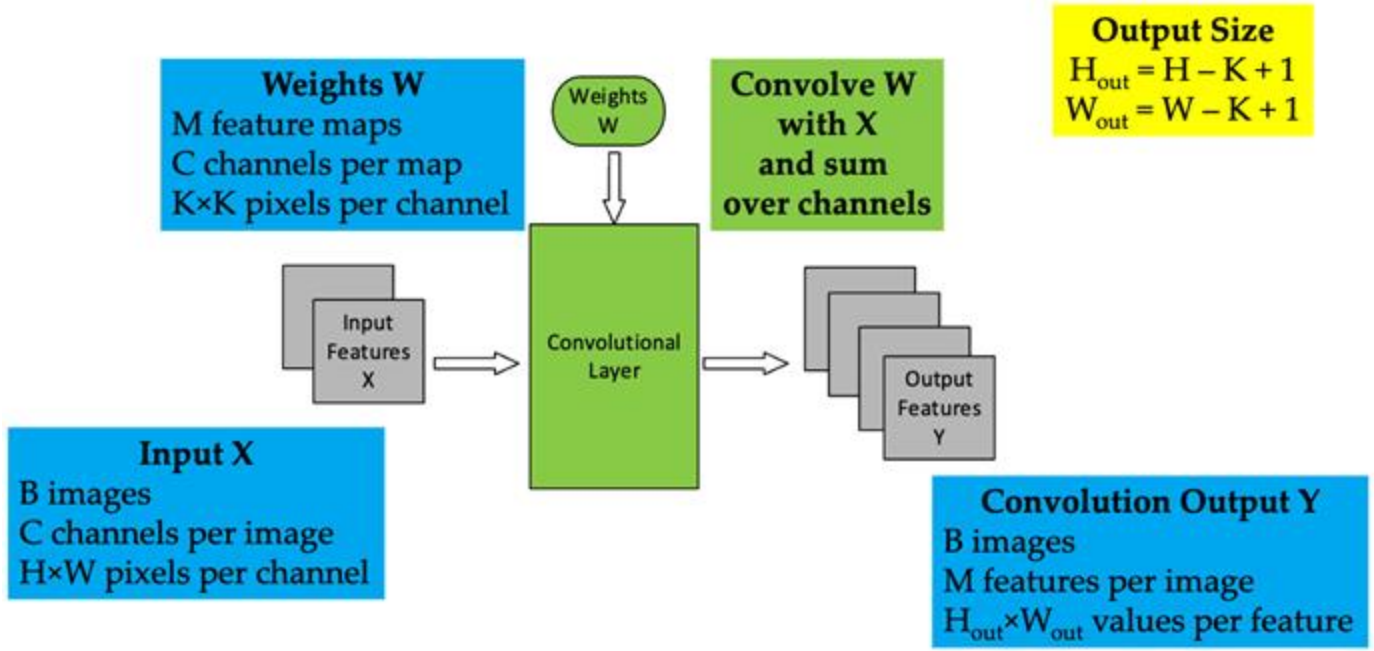
What happens when the input is so large, and the neural network structure is fixed?

What if its too small?

Convolution in the context of a neural network is to edit/modify the input data to better suit the model's need. It is oftentimes added as a **layer**(s) to our model in the early stages.

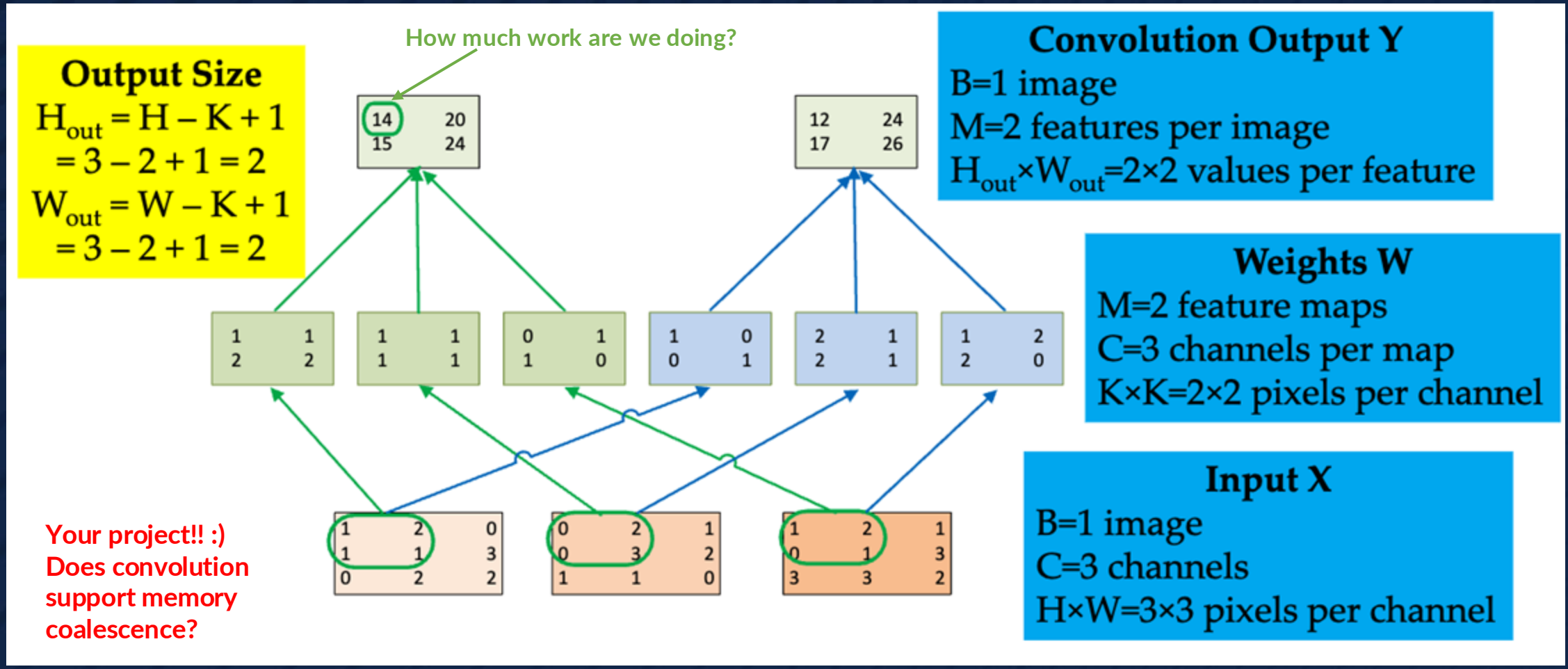
If we have a pixel image as an input to our CNN... we can..

- find patterns and relationships **within the input data**
- reduce input data noise by performing **subsampling/pooling**
- reduces the amount of total learnable parameters due to **shared filters/masks**
- and more...



Does Convolutional Layer also learn through gradient descent?

FORWARD PASS EXAMPLE



CONVOLUTION AS MATRIX MULTIPLICATION – UNROLLING

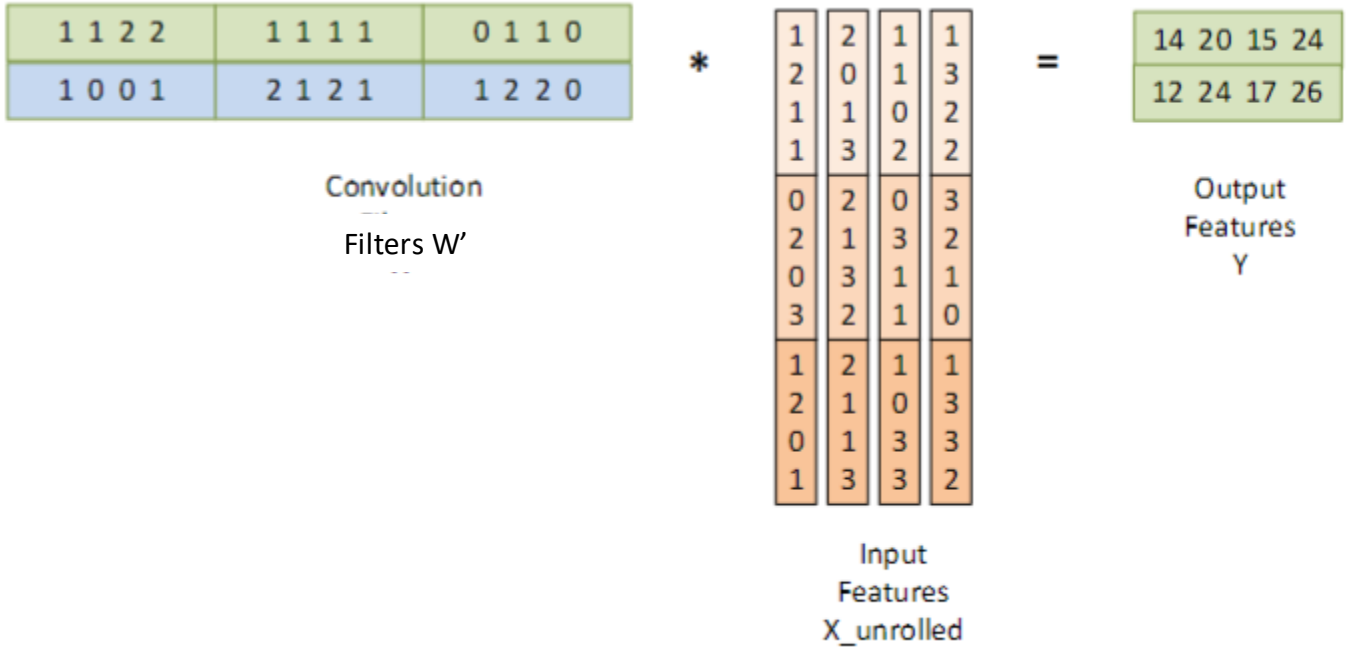
Since convolution does not support coalesced memory accesses well, perhaps we can look towards an operation that does?

Computing convolution as a **matrix multiplication** by **unrolling** our feature map supports **coalescing**.

This however requires you to store certain features twice, but the overhead of doing so is often outweighed using effective patterns for accessing.

Example here is 3 input channels (with 2x2 masks) resulting in 2 output features

W' row corresponds to one output feature
W' horizontal sections are masks for each a channel
4 numbers per section mean 2x2 mask



X_unrolled vertical sections are each a channel
X_unrolled columns are input features across channels
Four places (columns) to place our 2x2 masks (so 3x3 inputs)

UNROLLING PRACTICE

One Image – 3 Channels

5	6	3
3	7	9
1	5	3

2	7	9
9	3	2
2	5	5

1	4	9
2	8	6
9	9	3

Unroll Channel 0

Unroll Channel 1

Unroll Channel 2

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

PROJECT MILESTONE 1 – BRIEF OVERVIEW

Already went over convolution, so milestone 1 should hopefully be a breeze.

Combination of skills from all your lecture, especially the ones on **machine learning** and **convolution** for milestone 1.

Be prepared to understand what each parameter means, and how we took provided sequential code and found ways to parallelize (Lecture 12).

Batch (B) – Number of images in input

Map_out (M) – Number of output feature maps

Channel (C) – Number of input features to map

Mask (W) – Convolution weights

K – Mask height and width both (since mask is square)

Also have input, output, height, and width

```
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];
        }
}
```

Computed by the grid

Computed by a thread

PROFILING – GUEST LECTURE

Hopefully, everyone watched the guest lecture…

Prior to profiling, you **must clean up errors** that may not cause your regular programs to terminate but can cause profiling issues.

Generate code with **--generate-line-info/-lineinfo, NOT** --profile, --debug, or --device-debug.

Cuda Memcheck – Should run if profiling crashes or misbehaves to detect bad memory behavior

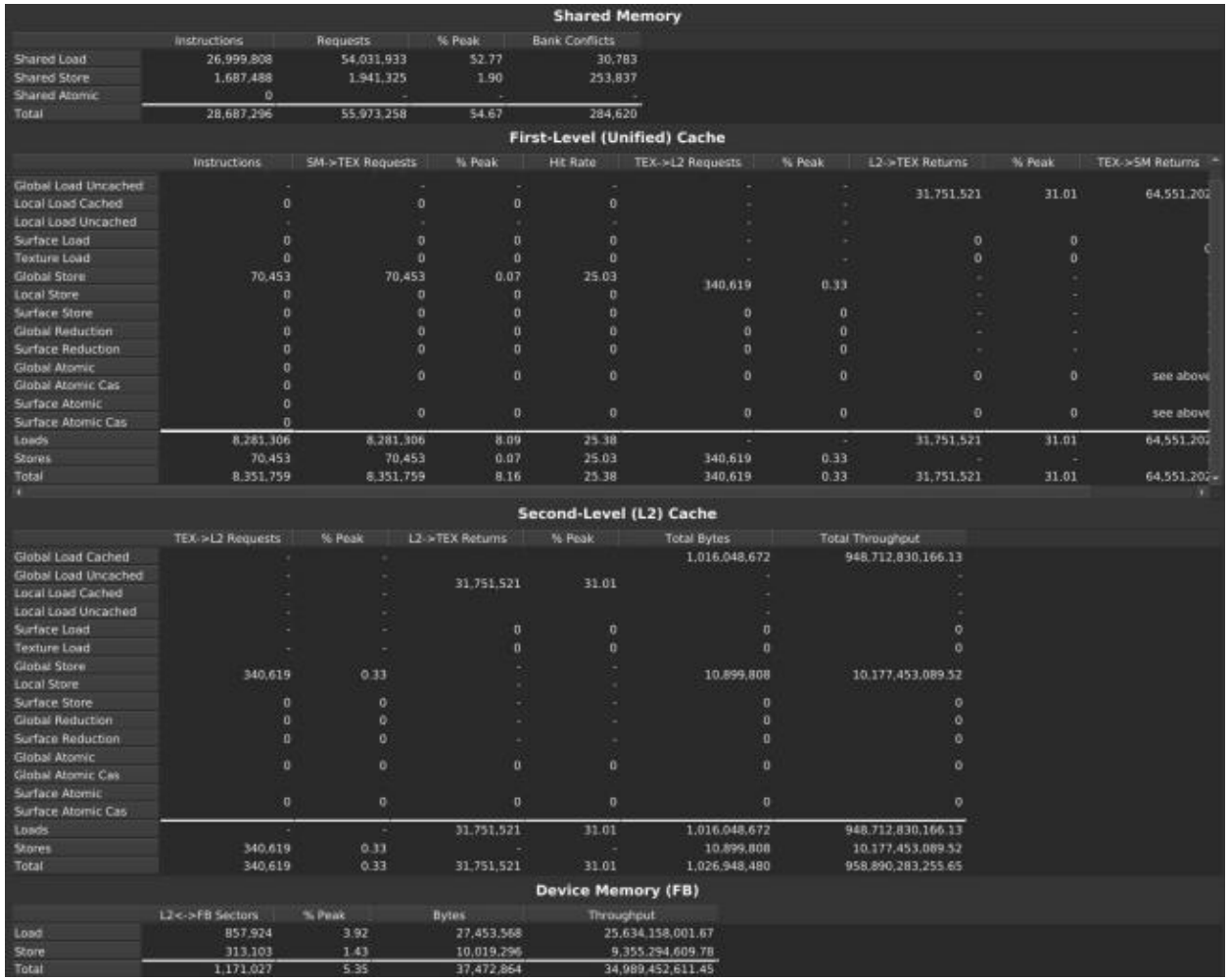
Nsight Compute – Kernel-level profiling

- Evaluates how fast your kernel executes
- Useful for calculating speed of light and other performance related metrics

Nsight System – System-level profiling

- How effectively does my system deliver to the GPU?
- How fast is data moving to/from the GPU?
- What is the time division for CPU vs GPU execution?

We will not cover all this right now, the lecture does



Shared Memory

	Instructions	Requests	% Peak	Bank Conflicts
Shared Load	26,999,808	54,031,933	52.77	30,783
Shared Store	1,687,488	1,941,325	1.90	253,837
Shared Atomic	0	-	-	-
Total	28,687,296	55,973,258	54.67	284,620

First-Level (Unified) Cache

	Instructions	SM->TEX Requests	% Peak	Hit Rate	TEX->L2 Requests	% Peak	L2->TEX Returns	% Peak	TEX->SM Returns
Global Load Uncached	-	-	-	-	-	-	31,751,521	31.01	64,551,202
Local Load Cached	0	0	0	0	-	-			
Local Load Uncached	-	-	-	-	-	-			
Surface Load	0	0	0	0	-	-	0	0	
Texture Load	0	0	0	0	-	-	0	0	
Global Store	70,453	70,453	0.07	25.03	340,619	0.33	-	-	
Local Store	0	0	0	0			-	-	
Surface Store	0	0	0	0	0	0	-	-	
Global Reduction	0	0	0	0	0	0	-	-	
Surface Reduction	0	0	0	0	0	0	-	-	
Global Atomic	0								
Global Atomic Cas	0	0	0	0	0	0	0	0	see above
Surface Atomic	0								
Surface Atomic Cas	0	0	0	0	0	0	0	0	see above
Loads	8,281,306	8,281,306	8.09	25.38	-	-	31,751,521	31.01	64,551,202
Stores	70,453	70,453	0.07	25.03	340,619	0.33	-	-	
Total	8,351,759	8,351,759	8.16	25.38	340,619	0.33	31,751,521	31.01	64,551,202

Second-Level (L2) Cache

	TEX->L2 Requests	% Peak	L2->TEX Returns	% Peak	Total Bytes	Total Throughput
Global Load Cached	-	-			1,016,048,672	948,712,830,166.13
Global Load Uncached	-	-	31,751,521	31.01	-	-
Local Load Cached	-	-			-	-
Local Load Uncached	-	-			-	-
Surface Load	-	-	0	0	0	0
Texture Load	-	-	0	0	0	0
Global Store	340,619	0.33	-	-	10,899,808	10,177,453,089.52
Local Store			-	-		
Surface Store	0	0	-	-	0	0
Global Reduction	0	0	-	-	0	0
Surface Reduction	0	0	-	-	0	0
Global Atomic	0	0	0	0	0	0
Global Atomic Cas						
Surface Atomic	0	0	0	0	0	0
Surface Atomic Cas						
Loads	-	-	31,751,521	31.01	1,016,048,672	948,712,830,166.13
Stores	340,619	0.33	-	-	10,899,808	10,177,453,089.52
Total	340,619	0.33	31,751,521	31.01	1,026,948,480	958,890,283,255.65

Device Memory (FB)

	L2<->FB Sectors	% Peak	Bytes	Throughput
Load	857,924	3.92	27,453,568	25,634,158,001.67
Store	313,103	1.43	10,019,296	9,355,294,609.78
Total	1,171,027	5.35	37,472,864	34,989,452,611.45

Questions?