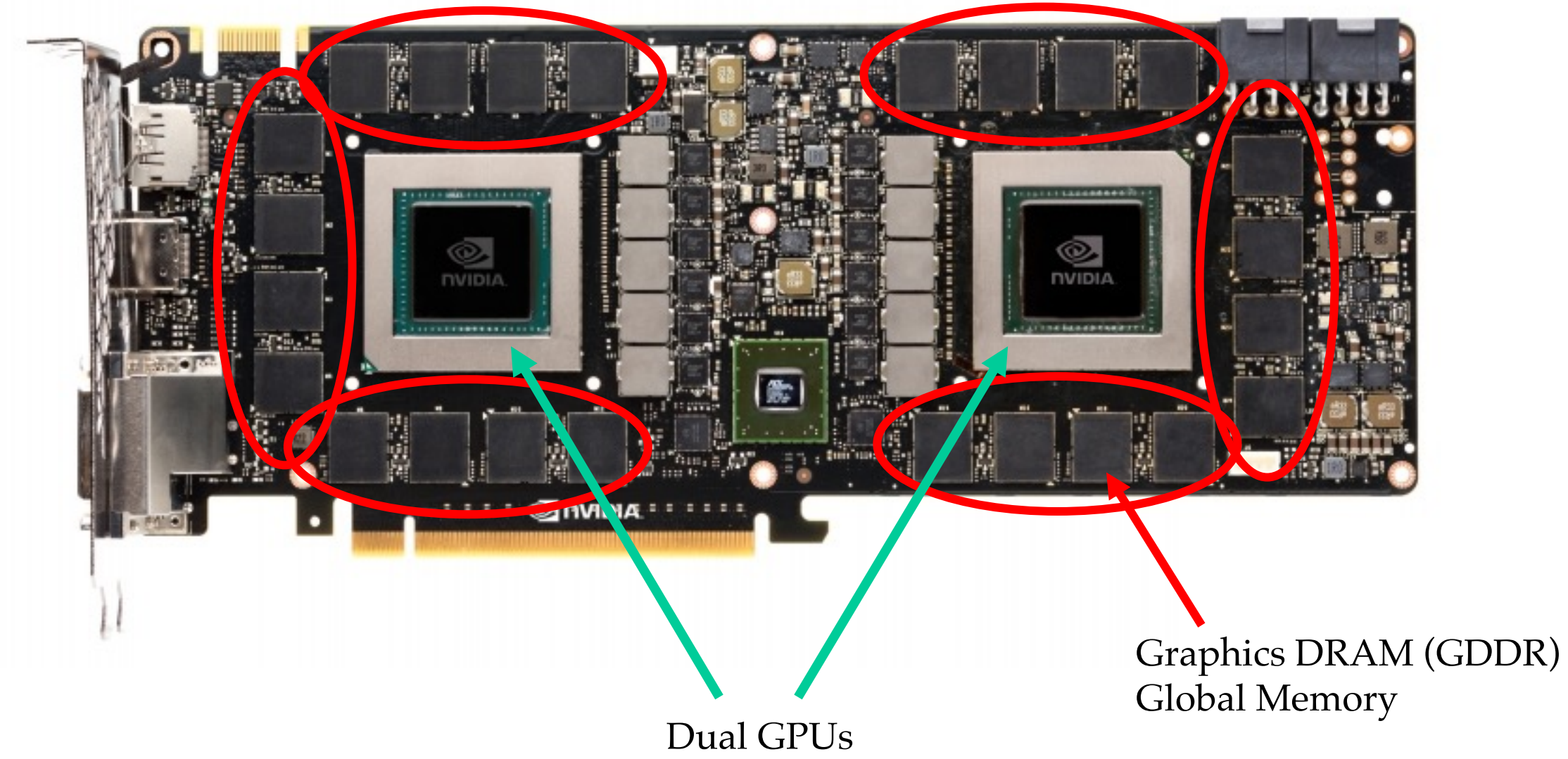
ECE408/CS483/CSE408 Fall 2022

Applied Parallel Programming

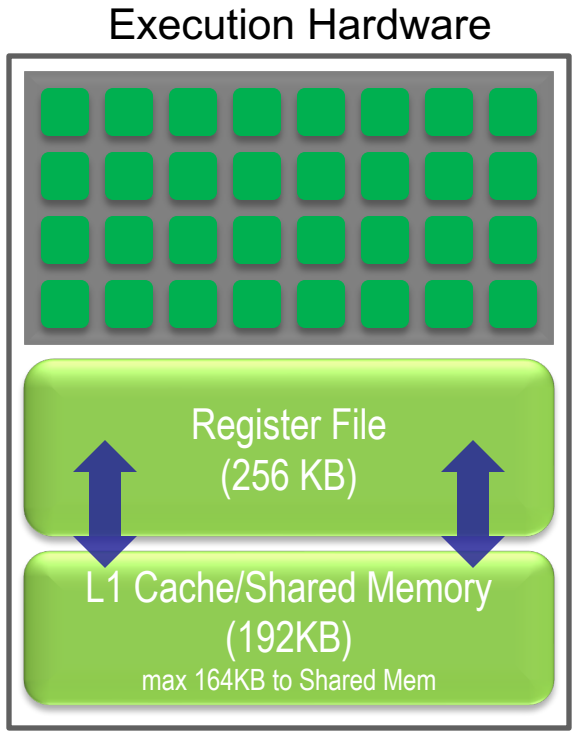
Lecture 9:
Tiled Convolution Analysis

Course Reminders

- Lab1-3 Grades are being posted on Canvas this week
- Lab 4 out, it is due on Friday
- Midterm 1 is on Tuesday, October 11th
  - On-line, everybody will be taking it at the same time
    - Thursday, Oct. 7th 7:00pm-8:20pm US Central time
    - Friday, Oct. 9th 8:00am-9:20am Beijing time
- Project Milestone 1: Baseline CPU implementation is due Friday October 14th
  - Project details to be posted next week on course wiki


Dual GPUs
Graphics DRAM (GDDR)
Global Memory

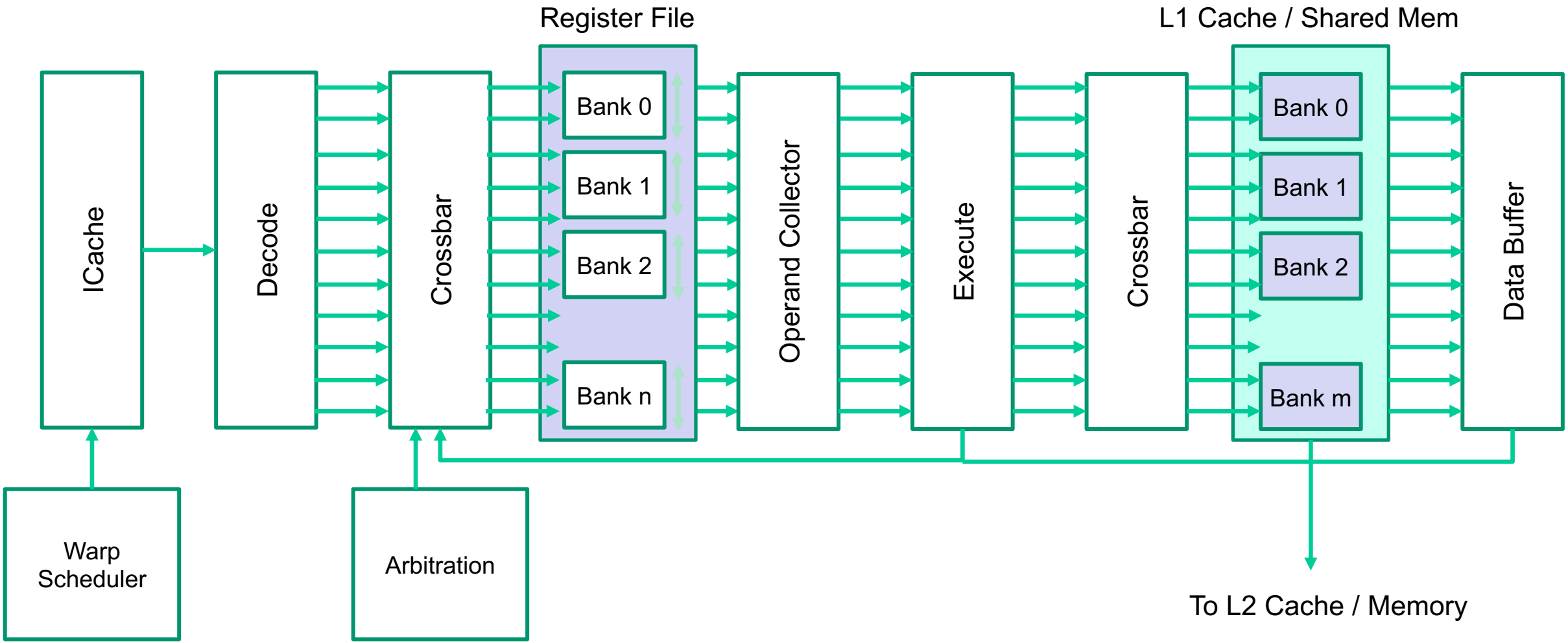
Ampere SM Memory Architecture



Ampere SM Memories
(GPU from 2020)

- **registers (~1 cycle)**
- **shared memory (~5 cycles)**
- **cache/constant memory (~5 cycles)**
- **global memory (~500 cycles)**

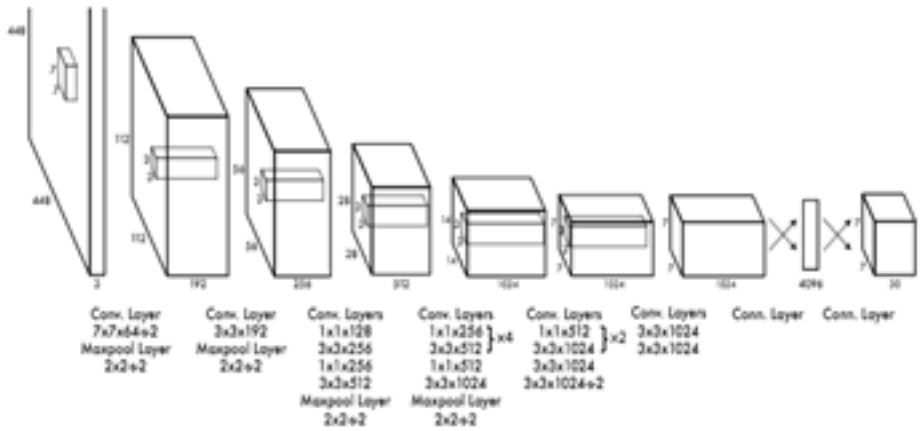
Overall Data Parallel Pipeline

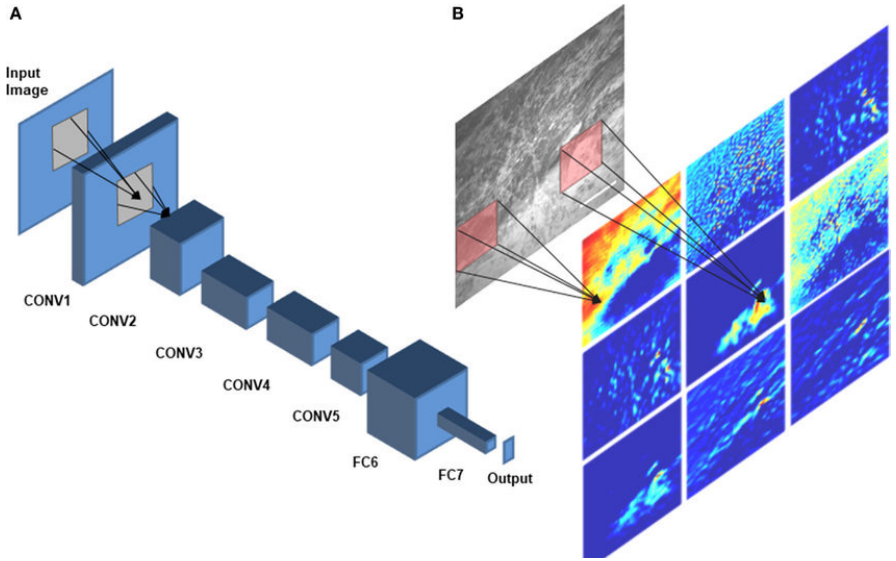


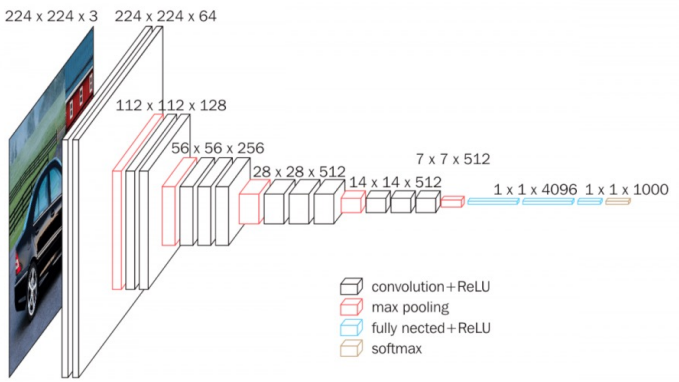
Memory Hierarchy Considerations

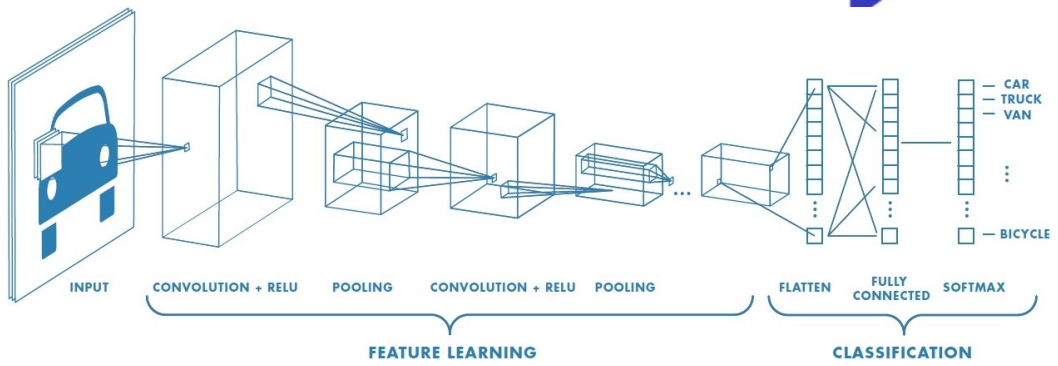
- Register file is highly banked
  - But we can have bank conflicts that cause pipeline stalls
- Shared memory is highly banked
  - But we can have bank conflicts that cause pipeline stalls
- Global memory has multiple channels, banks, pages
  - Relies on bursting
  - Coalescing is important. Need programmer involvement.
- L1 Cache is non-coherent

Convolutions for Image-based Deep Learning



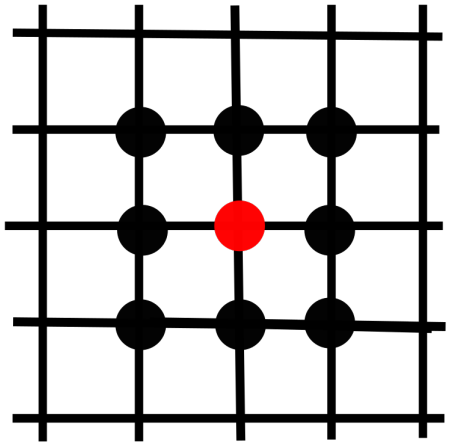




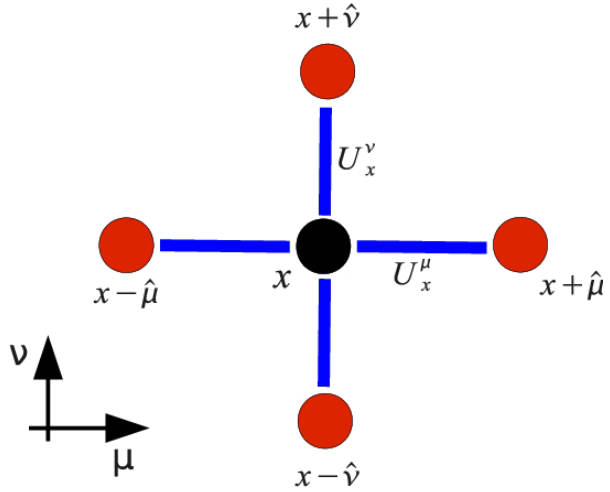


Stencil Patterns

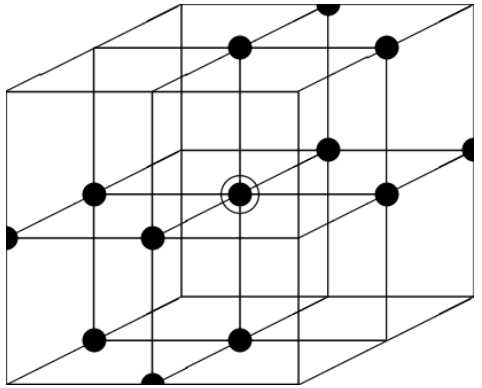
- Numerical data processing algorithms which update array elements according to some fixed pattern, called a *stencil*
  - Convolution is just one such example



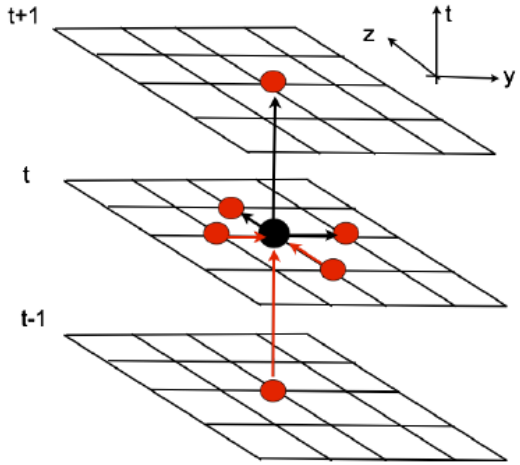
2D convolutional kernel (9-point 2D stencil)



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

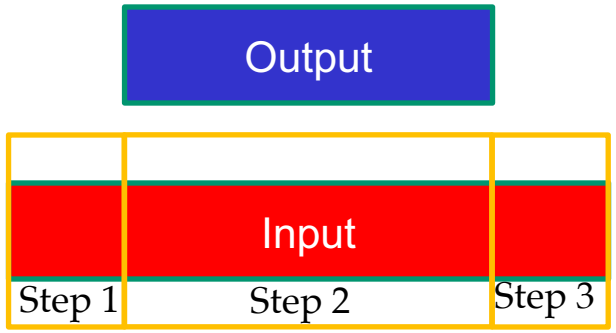


Finite difference stencil for 3D explicit time-marching (13-point 3D stencil)



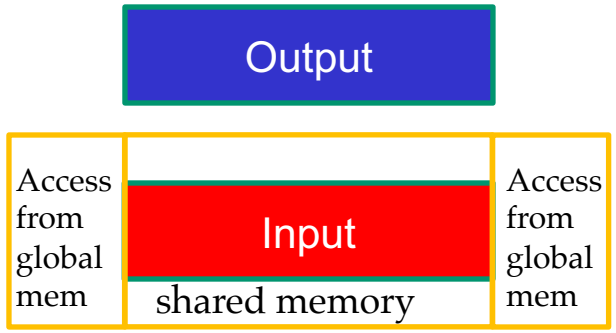
The Wilson-Dslash operator (4D stencil)

Review: Three Tiling Strategies



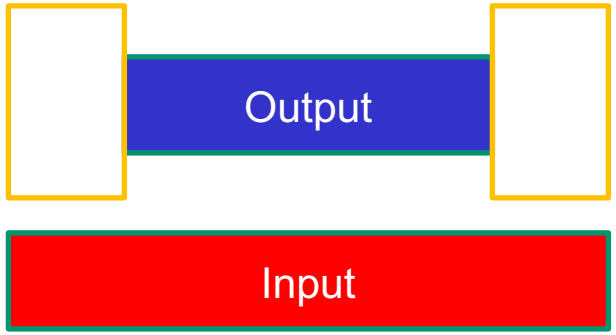
Strategy 1

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



Strategy 3

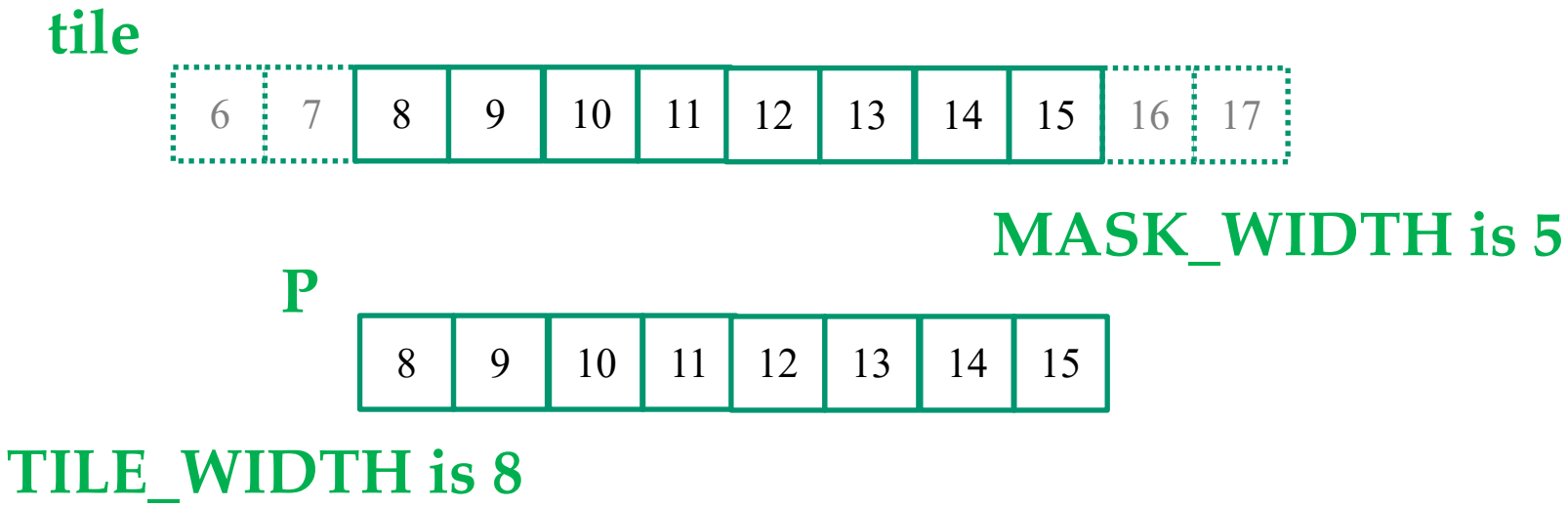
1. Block size covers **output** tile
2. Load only "core" of input tile
3. Access halo cells from global memory



Strategy 2

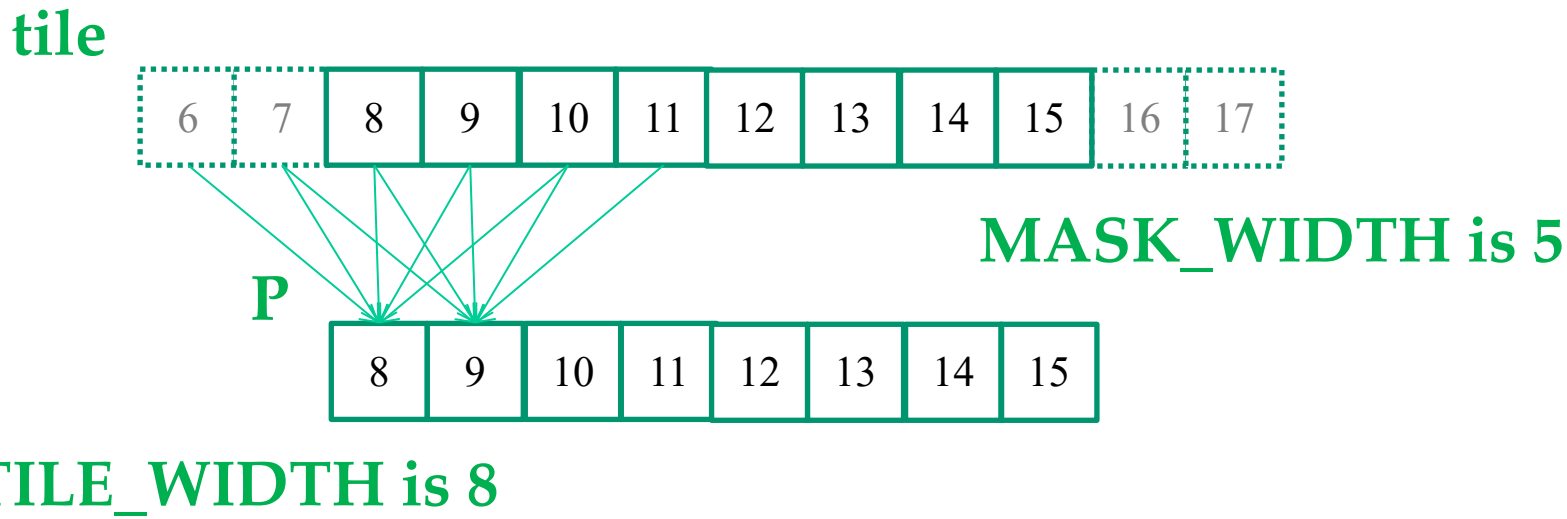
1. Block size covers **input** tile
2. Load input tile in one step
3. Turn off some threads when calculating output

A Small 1D Convolution Example



- 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

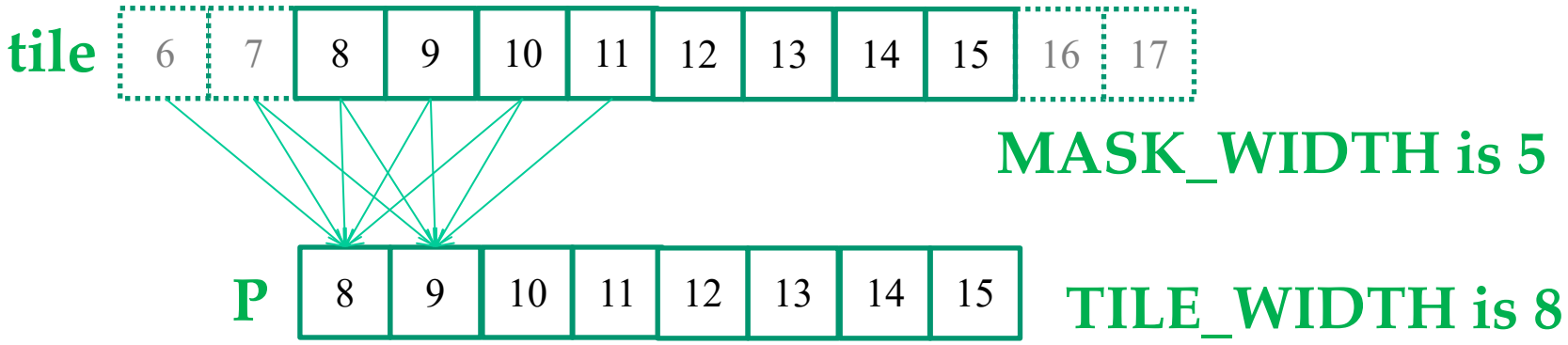
- 8+(5-1) = 12 unique elements of input array N loaded
- 8*5 global memory accesses potentially replaced by shared memory accesses
- This gives a bandwidth reduction of **40/12=3.3**
- This is independent of the size of N

In General, for 1D convolution kernels

- Load (TILE_WIDTH + MASK_WIDTH – 1) elements from global memory to shared memory
- Replace (TILE_WIDTH * MASK_WIDTH) global memory accesses with shared memory accesses
- This leads to bandwidth reduction of

(TILE_SIZE * MASK_WIDTH) / (TILE_SIZE + 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 uses 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

  1 + 2 + 3 + 4 + 5 * (8-5+1) + 4 + 3 + 2 + 1

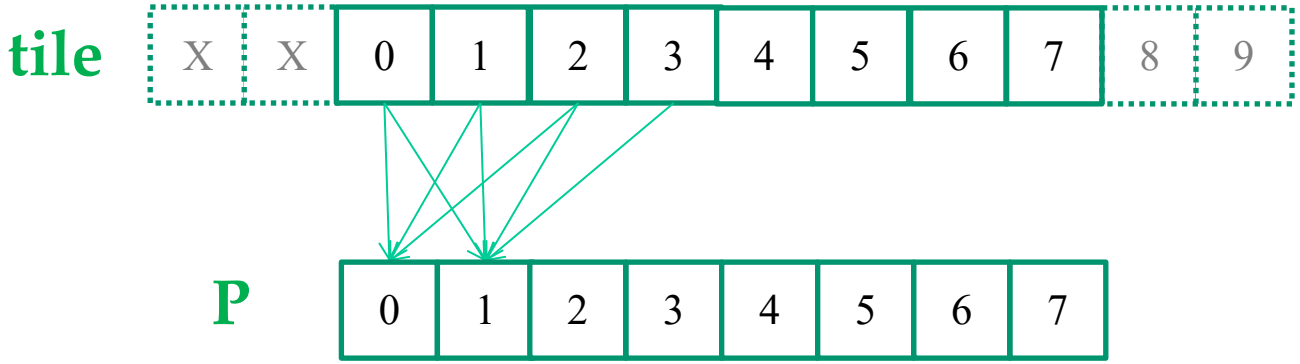
  = 10 + 20 + 10

  = 40

- 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 TILE_WIDTH + (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

1 + 2 + … + Mask_Width-1+ Mask_Width * (TILE_WIDTH -Mask_Width+1) + Mask_Width-1+… + 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

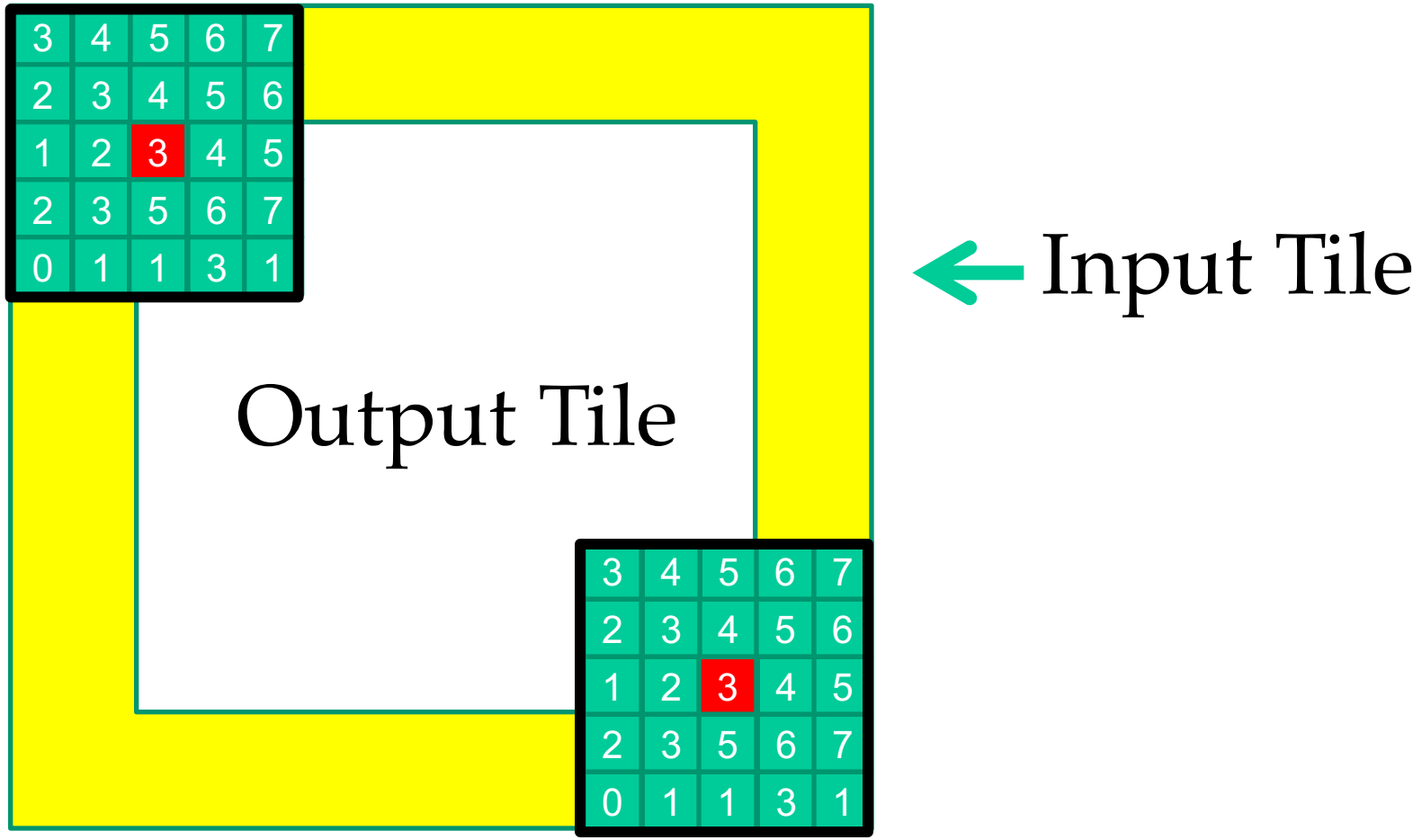
Bandwidth Reduction for 1D

- The reduction is

(TILE_SIZE * MASK_WIDTH) / (TILE_SIZE + 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: Strategy 2 loading of 2D Tile



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

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

Bandwidth Reduction for 2D Convolution Kernel

The reduction is:

$(TILE\_WIDTH * MASK\_WIDTH)^2 / (TILE\_WIDTH + MASK\_WIDTH - 1)^2$

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

Untiled1D Convolution Kernel

```
__global__
void convolution_1D_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;
}
```

2B/FLOP for Untiled Convolution

- **How much global memory per floating point operation (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

- Say we have a hypothetical GPU from 2010:
  - **1,000 GFLOP/s** for GPU, and
  - **150 GB/s** memory bandwidth.

- Dividing memory bandwidth by **2B/FLOP**,

$$\frac{150\ \text{GB/s}}{2\ \text{B/FLOP}} = \mathbf{75\ GFLOP/s = 7.50\%\ of\ peak}.$$

- **Need** at least **100/7.50 = 13.3× reuse** to make full use of compute resources

In 2021, Need 52.1× Reuse

- **In 2021**, the **GRID K520** offers
  - nearly **5,000 GFLOP/s**, but only
  - **192 GB/s** memory bandwidth

- Dividing memory bandwidth by **2B/FLOP**,

$$\frac{192\ \text{GB/s}}{2\ \text{B/FLOP}} = \mathbf{96\ GFLOP/s = 1.92\%\ of\ peak}$$

- **Need** at least **100/1.92 = 52.1× reuse** to make full use of compute resources

Need Really Big Mask to Balance Resources

% of peak compute for **1D** tiled convolution with **TILE_WIDTH 1024**

MASK_WIDTH	2010	2020
5	37%	9.6%
9	67%	17%
15	100%	28%
55	100%	100%

Need Really Big Mask to Balance Resources

% of peak compute for **2D** tiled convolution with **TILE_WIDTH 32×32**

MASK_WIDTH	2010	2020
3	60%	15%
5	100%	37%
7	100%	67%
9	100%	almost 100%

ANY MORE QUESTIONS?
READ CHAPTER 7