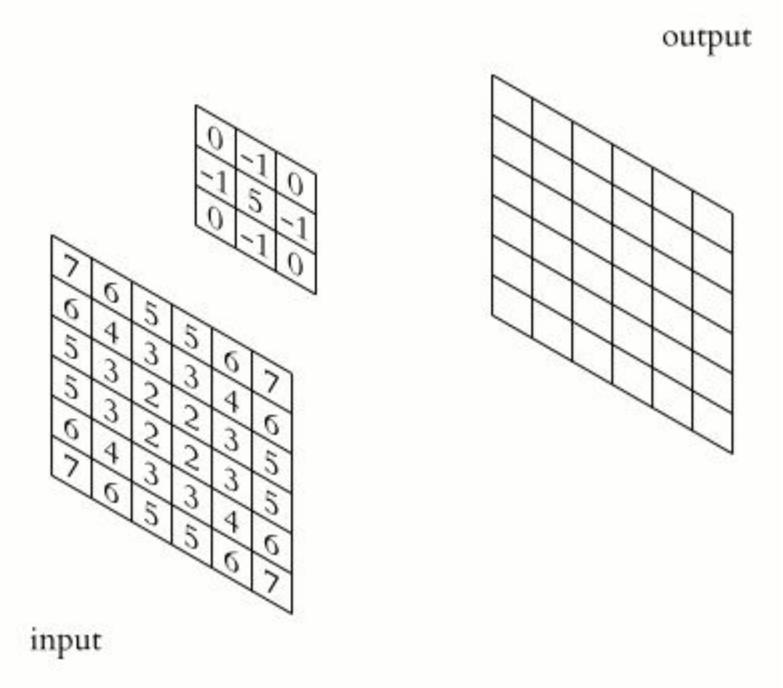
Convolution

Profiling and Optimizations

What is a Convolution?



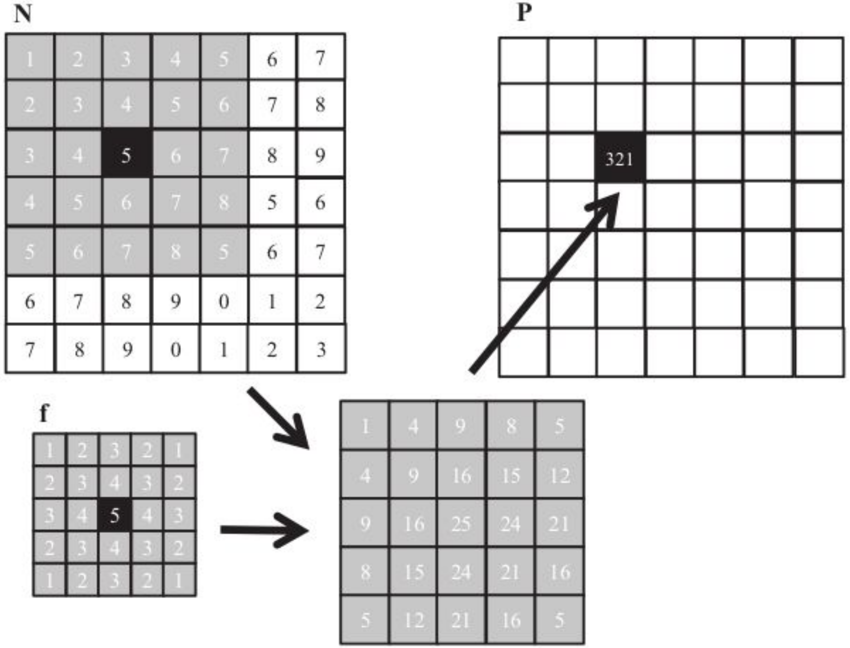
0	-1	0
-1	5	-1
0	-1	0

7	6	5	5	6	7
6	4	3	3	4	6
5	3	2	2	3	5
5	3	2	2	3	5
6	4	3	3	4	6
7	6	5	5	6	7

Formal Definition

If we assume that the dimension of the filter is $(2r_x + 1)$ in the x dimension and $(2r_y + 1)$ in the y dimension, the calculation of each P element can be expressed as follows

$$P_{y,x} = \sum_{j=-r_y}^{r_y} \sum_{k=-r_x}^{r_x} f_{y+j,x+k} \times N_{y,x}$$



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				

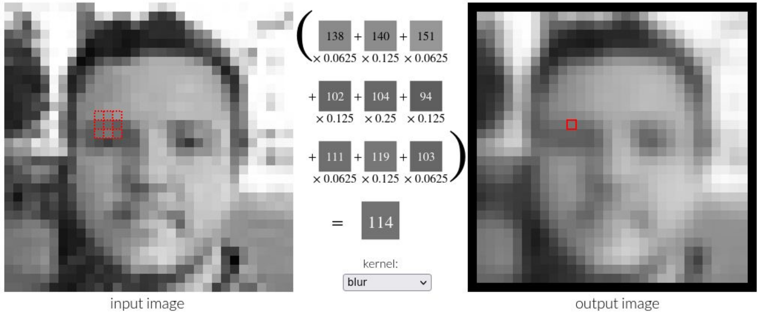
f

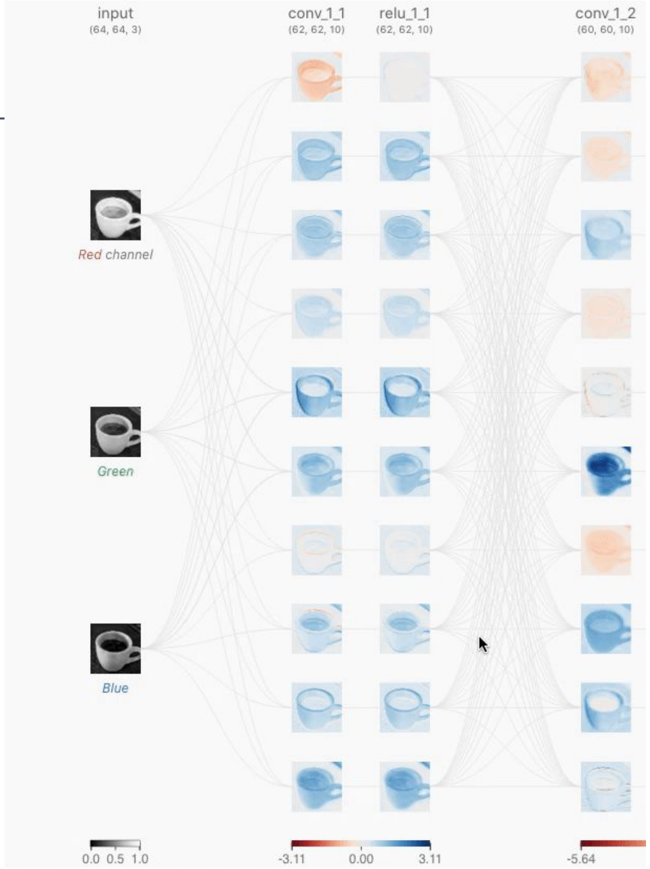
1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

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

Why Convolutions matter?

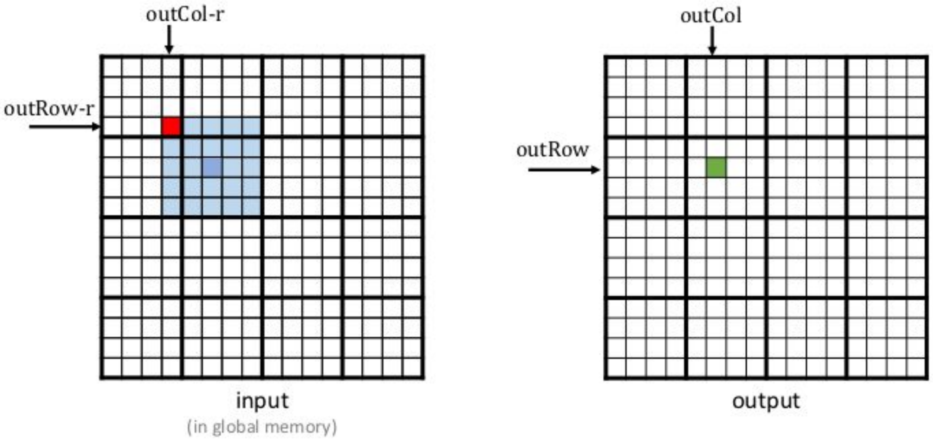
- Image processing
  - Blurring, sharpening, embossing, edge detection
  - A good explanation can be found here
- Artificial Intelligence and Machine Learning
  - Convolutional neural network
  - You can visualize the process here





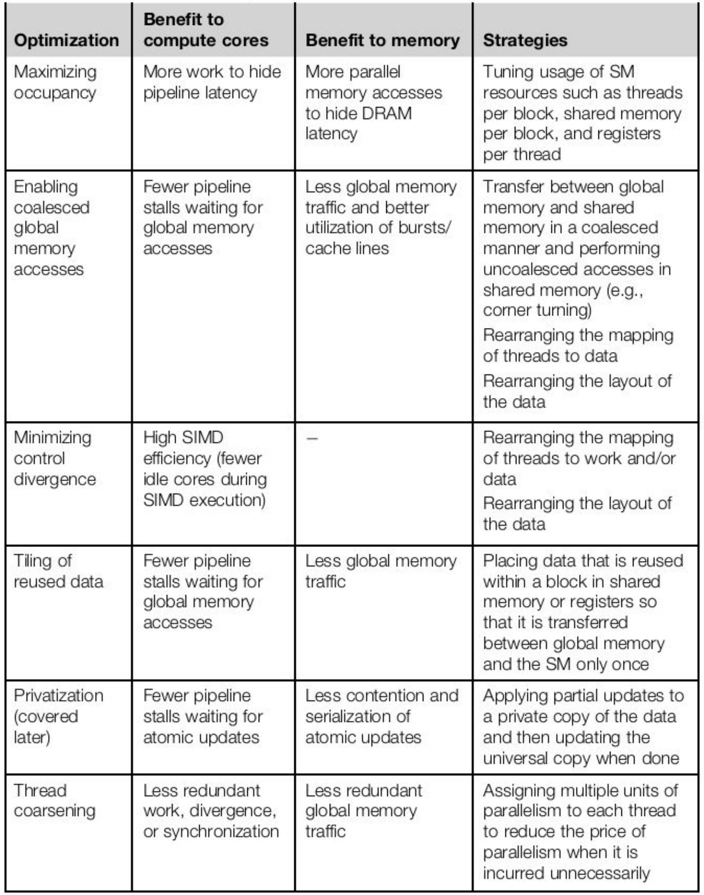
Naive Implementation

- Every thread is assigned to calculate an output pixel
  - Thread's x and y indices are the same as the thread’s x and y indices.
- The ratio of floating-point arithmetic calculation to global memory accesses is only about 0.25 OP/B to global memory (DRAM)
  - 2 operations for every 8 bytes loaded



Problems and Optimizations

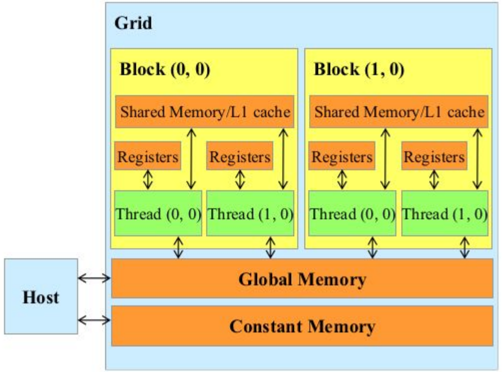
- Constant Memory
- Coarsening
- Tiling
- Caching



Optimization	Benefit to compute cores	Benefit to memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread
Enabling coalesced global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/ cache lines	Transfer between global memory and shared memory in a coalesced manner and performing uncoalesced accesses in shared memory (e.g., corner turning) Rearranging the mapping of threads to data Rearranging the layout of the data
Minimizing control divergence	High SIMD efficiency (fewer idle cores during SIMD execution)	–	Rearranging the mapping of threads to work and/or data Rearranging the layout of the data
Tiling of reused data	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic	Placing data that is reused within a block in shared memory or registers so that it is transferred between global memory and the SM only once
Privatization (covered later)	Fewer pipeline stalls waiting for atomic updates	Less contention and serialization of atomic updates	Applying partial updates to a private copy of the data and then updating the universal copy when done
Thread coarsening	Less redundant work, divergence, or synchronization	Less redundant global memory traffic	Assigning multiple units of parallelism to each thread to reduce the price of parallelism when it is incurred unnecessarily

Optimizations - Constant Memory

- **Constant Memory**
  - A constant memory cannot be modified by threads during kernel execution
  - The size of the constant memory is quite small (64KB) and efficient
    - It is a read only cache
      - Less complex hardware to manage it
    - **ATTENTION:** it is near the device memory, and access to different parts of it will be serialized
      - good performance only when each thread accesses the same data
  - Since everything is in constant memory cache, access in global memory is not needed, thus the Arithmetic Intensity becomes 0.5 OP/B (2 operations for every 4 bytes loaded)

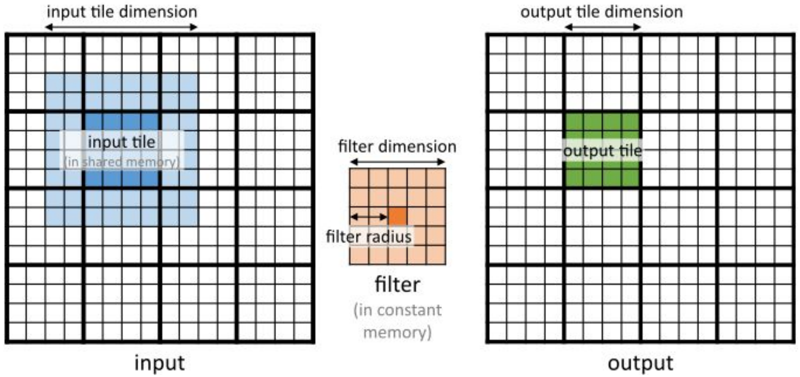


Optimizations - Coarsening

- **Coarsening**
  - We use the same caching and tiling approach as before
  - We increase efficiency by coarsening threads
    - Each block will process multiple tile
    - Thus each block's thread will produce multiple output
    - A stride access has to be defined

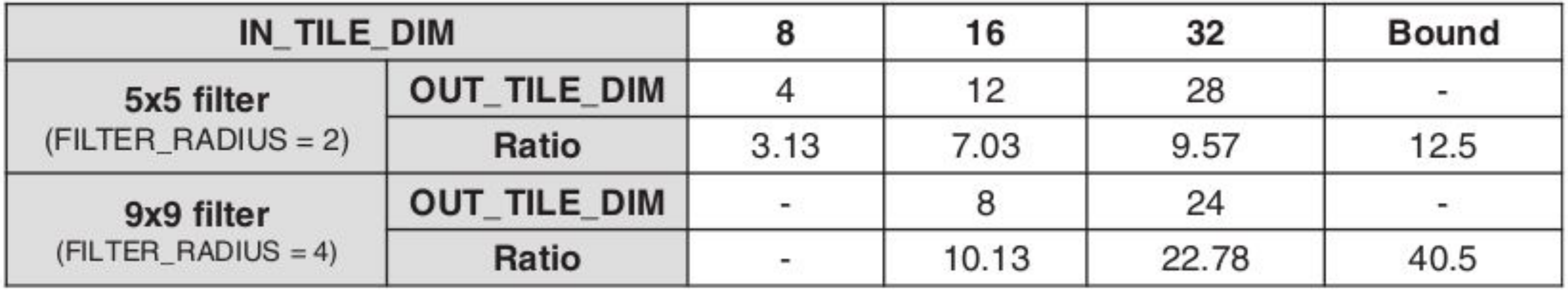
Optimizations - Tiling

- **Tiling** (shared memory)
  - It still leverages constant memory
  - Threads load the input tile into the shared memory
  - They calculate output tiles throughout the shared memory.
- Threads will be dimensioned based on the input tile sizes
  - Thus during computation, the halo threads will be disabled



Optimizations Analysis

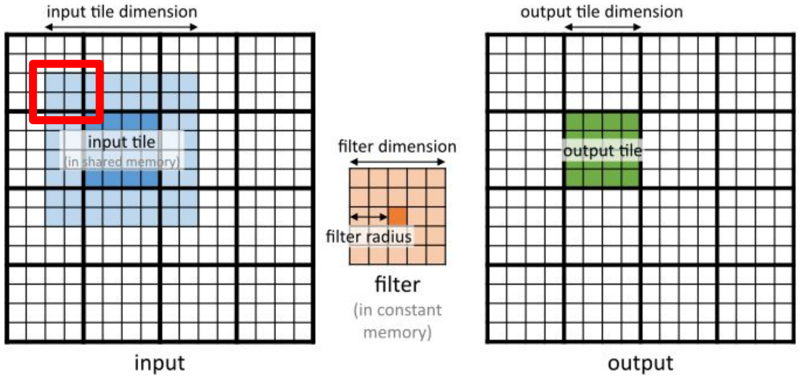
- The Arithmetic Intensity analysis becomes more complex
  - Operations: $OUT\_TILE\_DIM^2 * (2 * FILTER\_RADIUS + 1)^2 * 2$
  - Since floating point operations are executed 4 bytes are considered
  - Memory Access: $IN\_TILE\_DIM^2 * 4 = (OUT\_TILE\_DIM + 2 * FILTER\_RADIUS)^2 * 4$
- Asymptotically if $OUT\_TILE\_DIM >> FILTER\_RADIUS$
  - $Operations/Memory\ Access = (2 * FILTER\_RADIUS+1)^2 / 2$
- For example



IN_TILE_DIM		8	16	32	Bound
5x5 filter (FILTER_RADIUS = 2)	**OUT_TILE_DIM**	4	12	28	-
	Ratio	3.13	7.03	9.57	12.5
9x9 filter (FILTER_RADIUS = 4)	**OUT_TILE_DIM**	-	8	24	-
	Ratio	-	10.13	22.78	40.5

Optimizations - Caching

- **Caching Halo cells**
  - Similar to the previous approach about tiling
  - Halo cells are data that come from other tiles
  - Upon running it is very likely that these data are already available in the L2 cache
- We load in shared memory only the internal part of the tile
- We rely on the fact that halo cells are already available in L2 from previous requests



Code Hands-on

Bonus

- cuDNN can be used to perform convolutions
  - You have a reference to the CNN sub-library at the following link
- It can be used more in general to implement Convolutional neural network
- cuBLAS could be used to perform a matrix multiplication
  - Between each input tile and the filter

Thank you for your attention!

Gianmarco Accordi
gianmarco.accordi@polimi.it