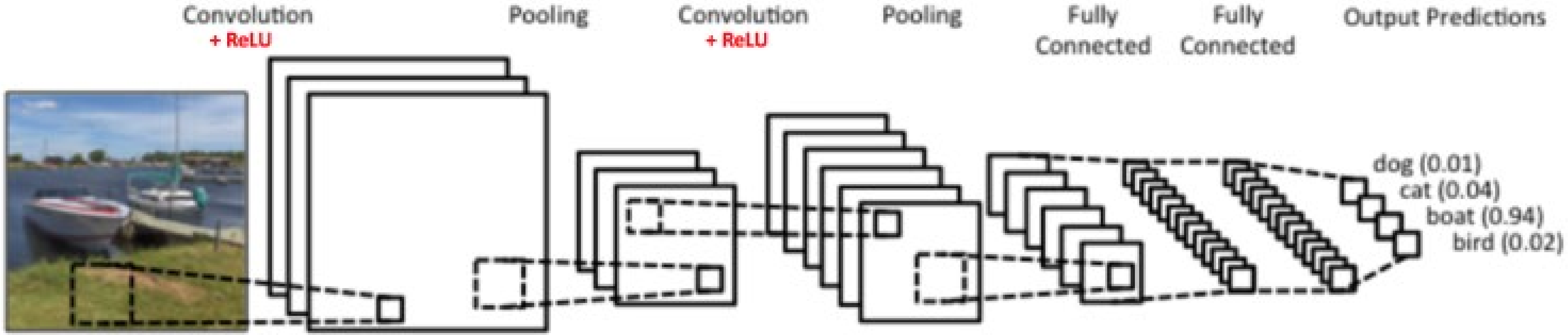
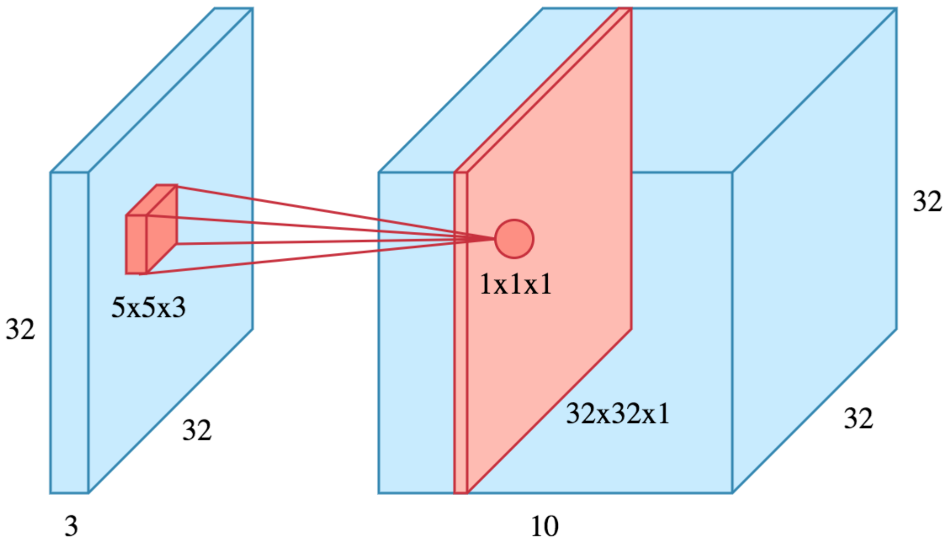
Accelerating ZF-Net Forward Propagation by CUDA

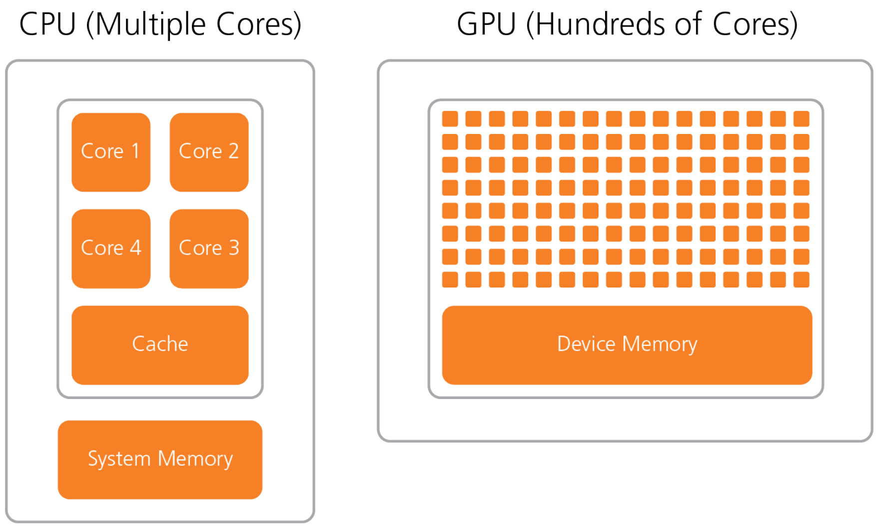
Wenqi Jiang (wj2285)   Hanzhou Gu(hg2498)





CPU convolution: For each point in the output, do convolution

```
for i in output row:
    for j in output col:
        for k in output layer:
            do convolution
```



GPU convolution: Have many cores, each core correspond to an point of output.

```
in every thread:
    do convolution
```

No disgusting for loop any more!

2000 times faster than CPU in our experiment!

Challenge: Fast Memories are very limited!

Shared Memory per Block: 48KB

Constant Memory in total: 64KB

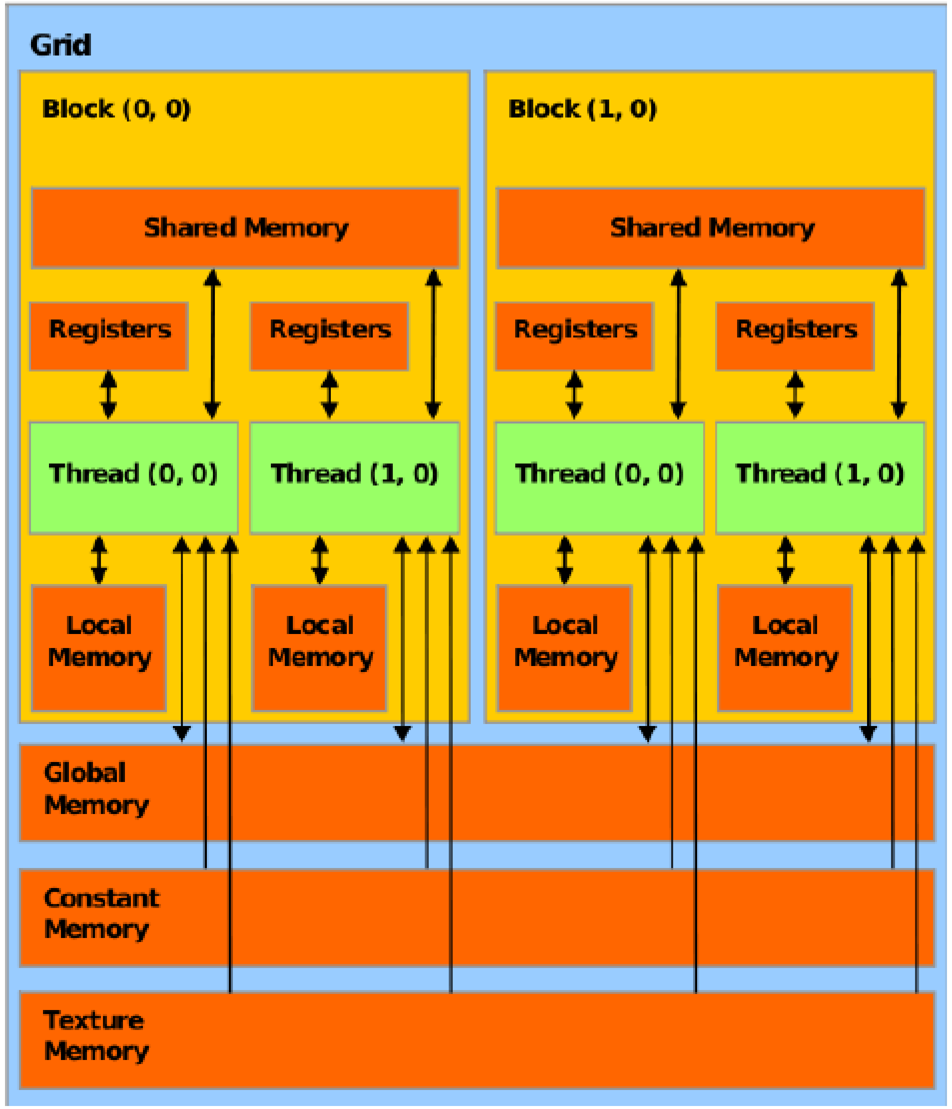
Layer 2:

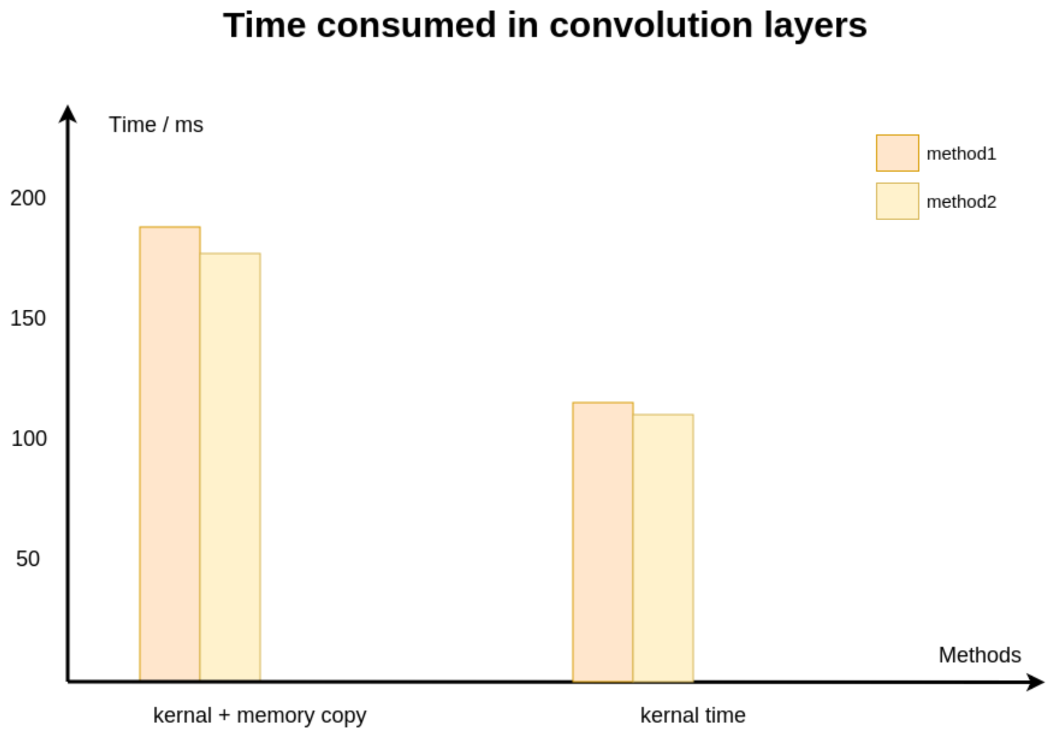
Input: 169 KB (but we can use many blocks)

Filter: 4 x 384 x 3 x 3 x 256 = 3,456 KB

Natural idea: Store input to shared memory;

Leave filter in global memory



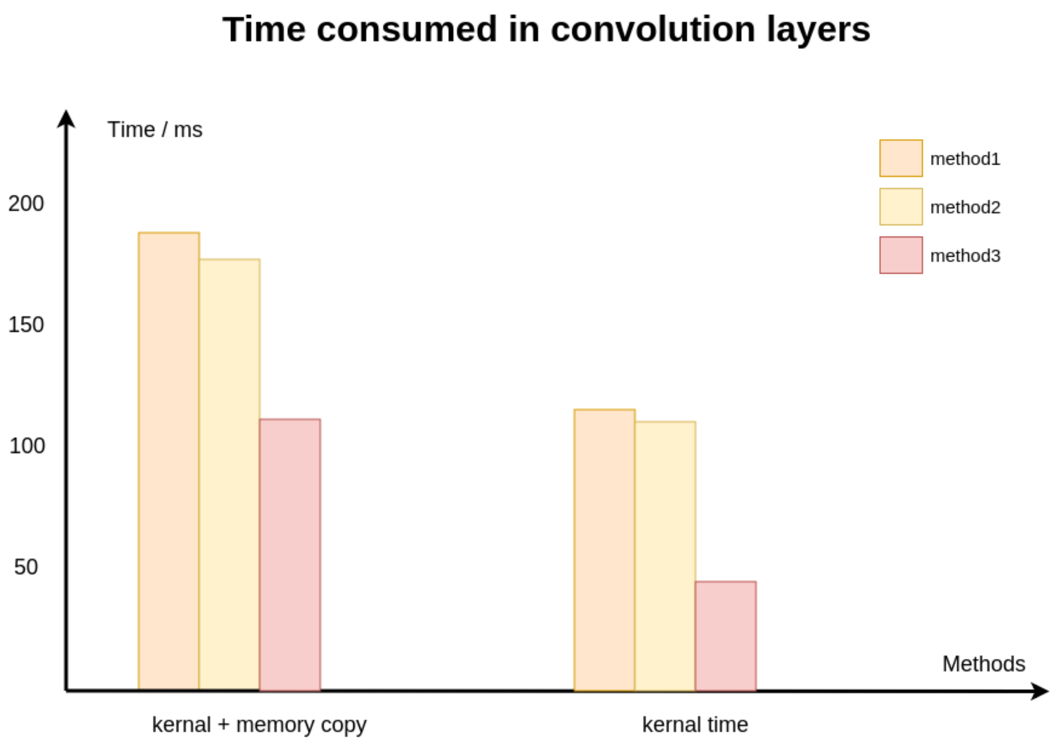


Not good enough!

Input size is far more smaller than filter

In the majority of time, GPU is loading filters rather than loading input

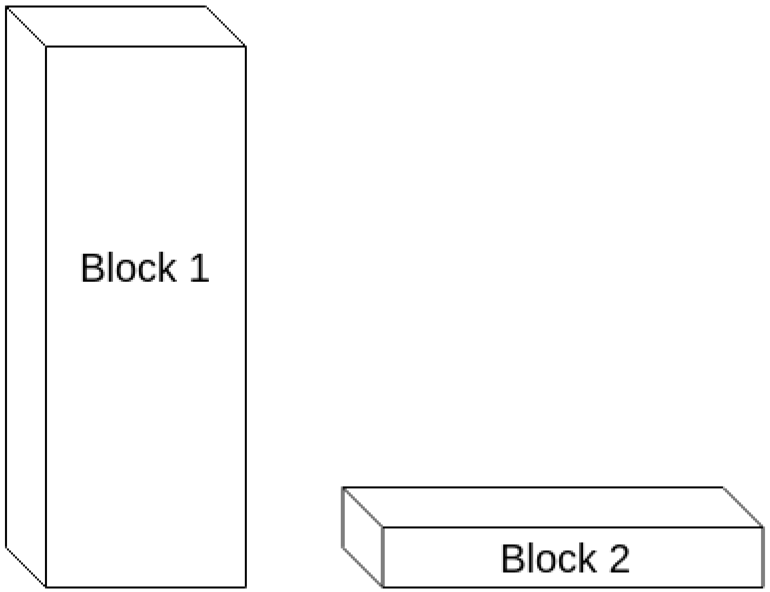
Advanced idea: load filter to Shared Memory iteratively (since Shared memory is very limited)



Much better!

Compare to naïve parallel:

2 x faster (kernel + memory copy)
3 x faster (kernel)

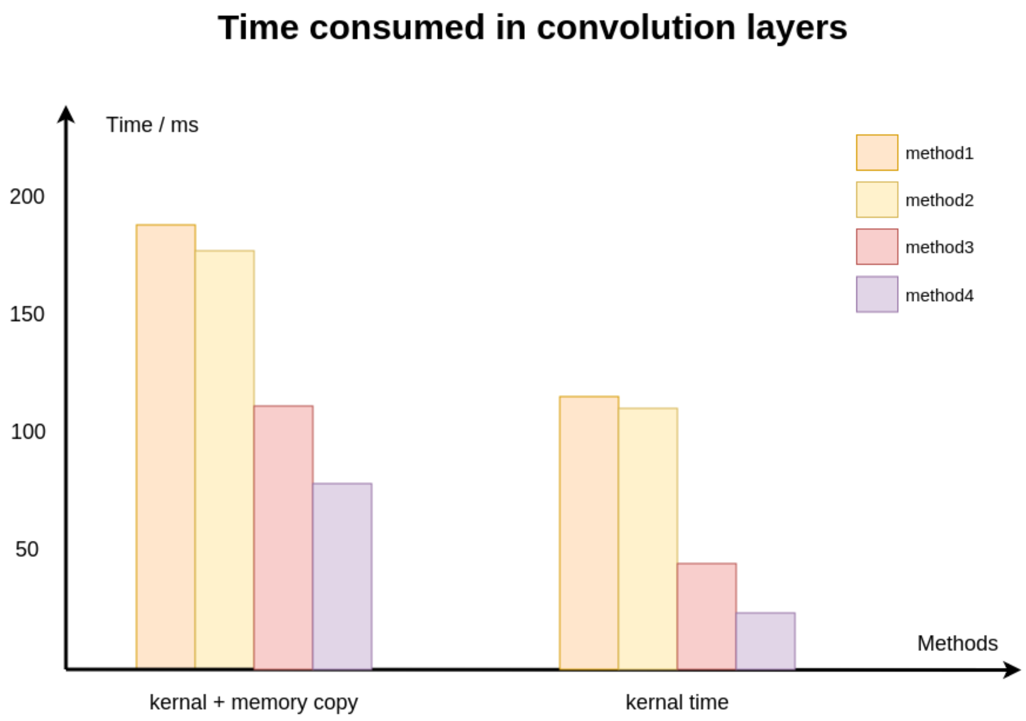


Magic! Changing the block size can significantly influence the algorithm performance!

Philosophy of designing high performance parallel algorithm: put data in fast memories, use them as much as possible.

Block setting 1: filter in shared memory can only be used very few times

Block setting 2: filter in shared memory can be used many times, which speeds up the algorithm



Compare to naïve parallel:

10.0 x faster (kernel)
2.4 x faster (kernel + memory copy)

Compare to Block setting 1:

3.9 x faster (kernel)
1.4 x faster (kernel + memory copy)

Fully-connected Layers: Matrix Multiplication of arbitrary size

Serial

```
for I in matrix1. length:
    for j in matrix2.length:
      for k in matrix1.width:
          do multiplication
```

Naïve Parallel

```
threads number:=
     size of output matrix
for each iteration:
     do multiplication of two cell
```

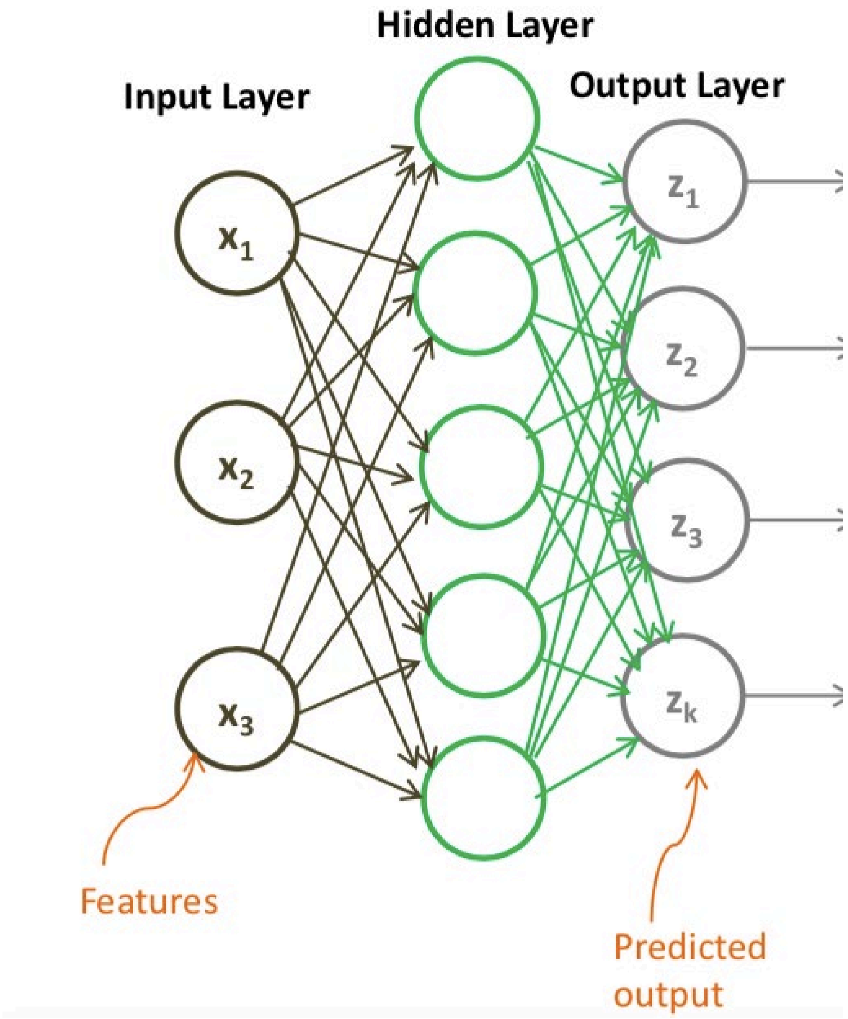
Optimization 1:

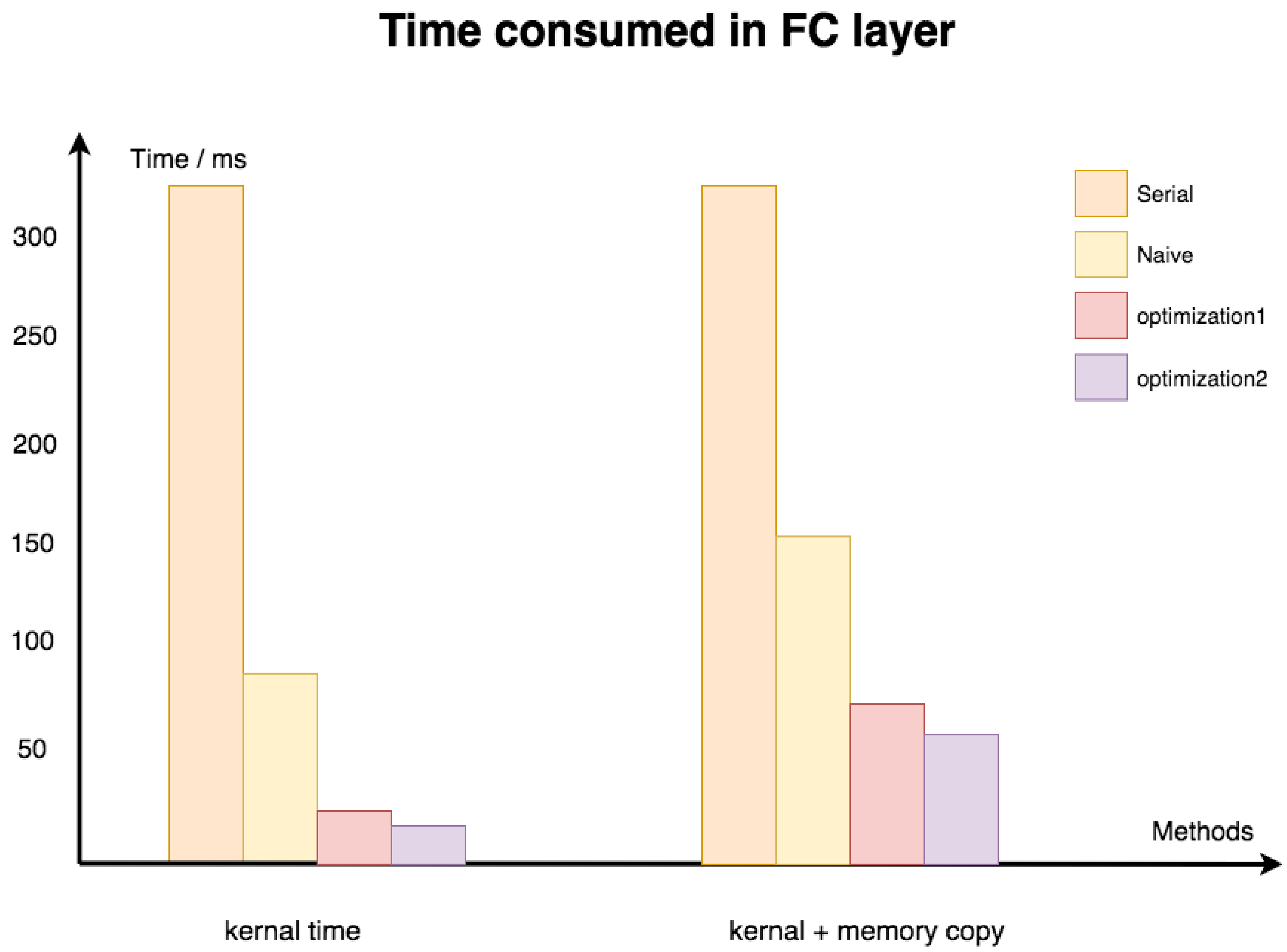
```
     for each block:
          for each iteration:
               copy input to shared memory
               do tiled multiplication
```

Optimization 2:

```
     for each block:
          for each iteration:
               prefeching the tile in to register
               copy input to shared memory
               do tiled multiplication
```

Parallel computing much faster





Max Equation

- **The max() equation returns the largest value from a set of values.**

Enumerated set of values (S)

For all x that are elements of set x

$$\max_{x \in S}(\underbrace{x_1, x_2, x_3, x_4, x_5}_{})$$

S : Set of Discrete Values
R: Set of Continuous Real Values
∈ : Symbol for Element of a Set
x_i : An Instance of an Element of a Set
≥ : Greater than or equal to for all elements in a set

$$S(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{j} e^{y_j}}$$

- **Condition is met where element x_j is the maximum in $x \in S$, when:**

x_j is greater than or equal to all elements x_i in set S

$$x_j \geq x_i,\ x \in S$$

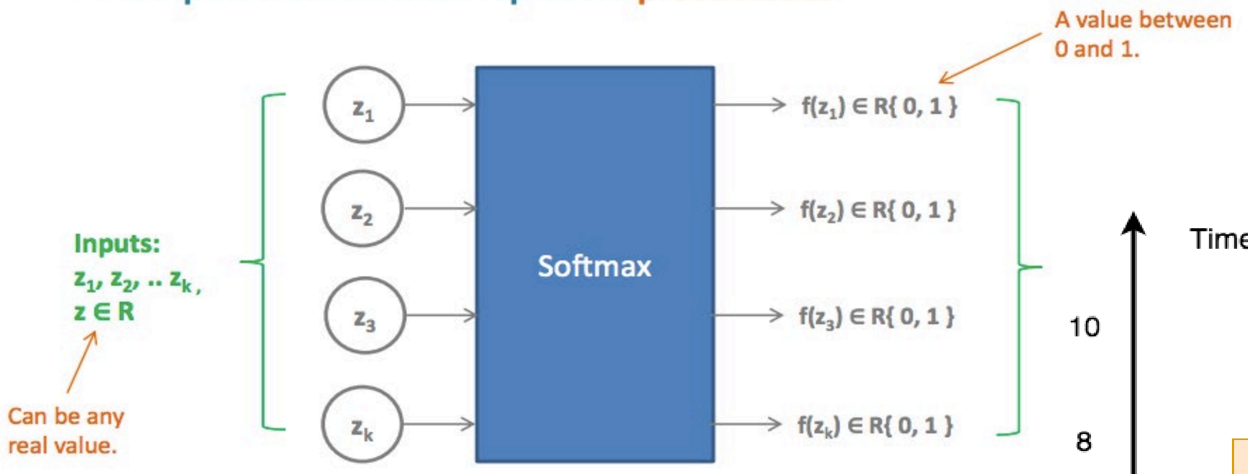
Serial Softmax:
batch size 128
for each row:
calculate and compare SM

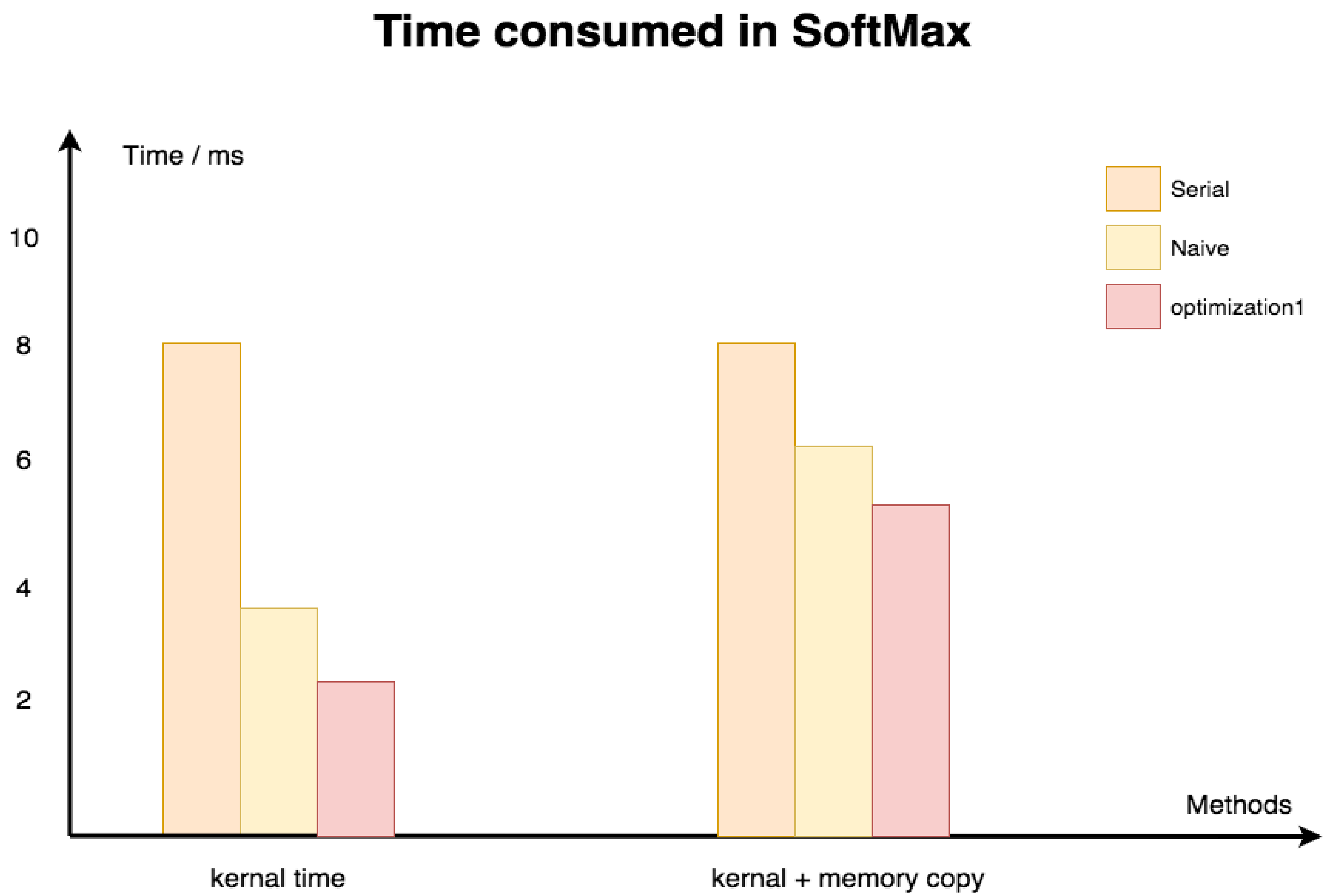
Parallel Max and Sum:
batch size 128
halving thread in each iteration:
do reduction

Parallel findMaxElement:
batch size 128
halving thread in each iteration
do reduction

SoftMax Equation

- **The softmax() equation takes as input a set of real values, and outputs a new set of values between 0 and 1, and where the values add up to one.**
  - **Typically used in squashing the outputs of a neural network.**
  - **Inputs can be any real values of any range (e.g., > 1.0).**
  - **Outputs from softmax represent probabilities.**





Overall Result

Best optimized algorithm compare to naïve parallel:

3 x faster (kernel + memory copy)
2.2 x faster (kernel)

Best optimized algorithm compare to serial:

696 x faster (kernel + memory copy)
3905 x faster (kernel)