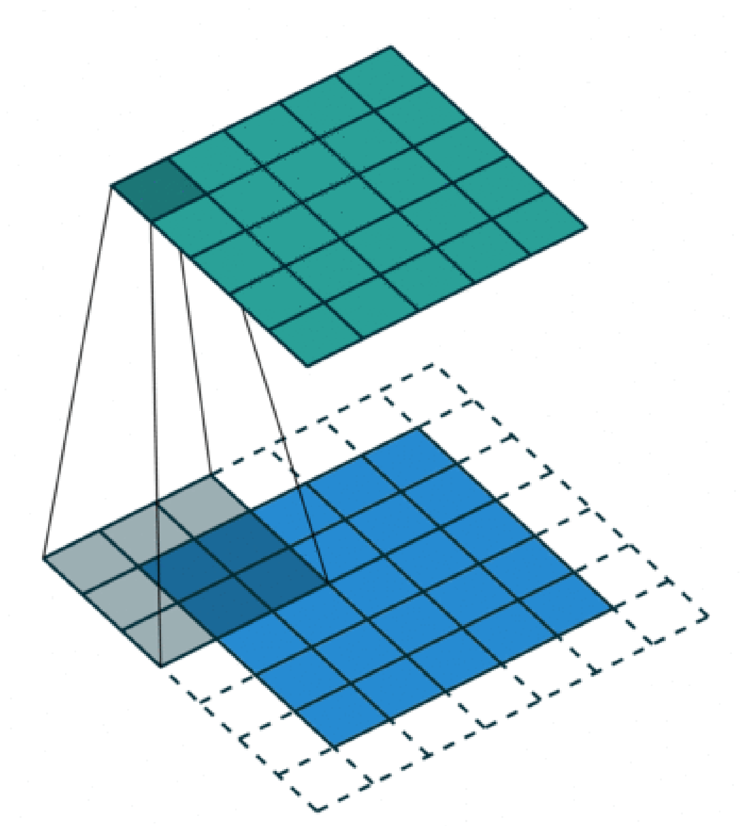
# 2D Convolution Performance Analysis

## The Algorithm

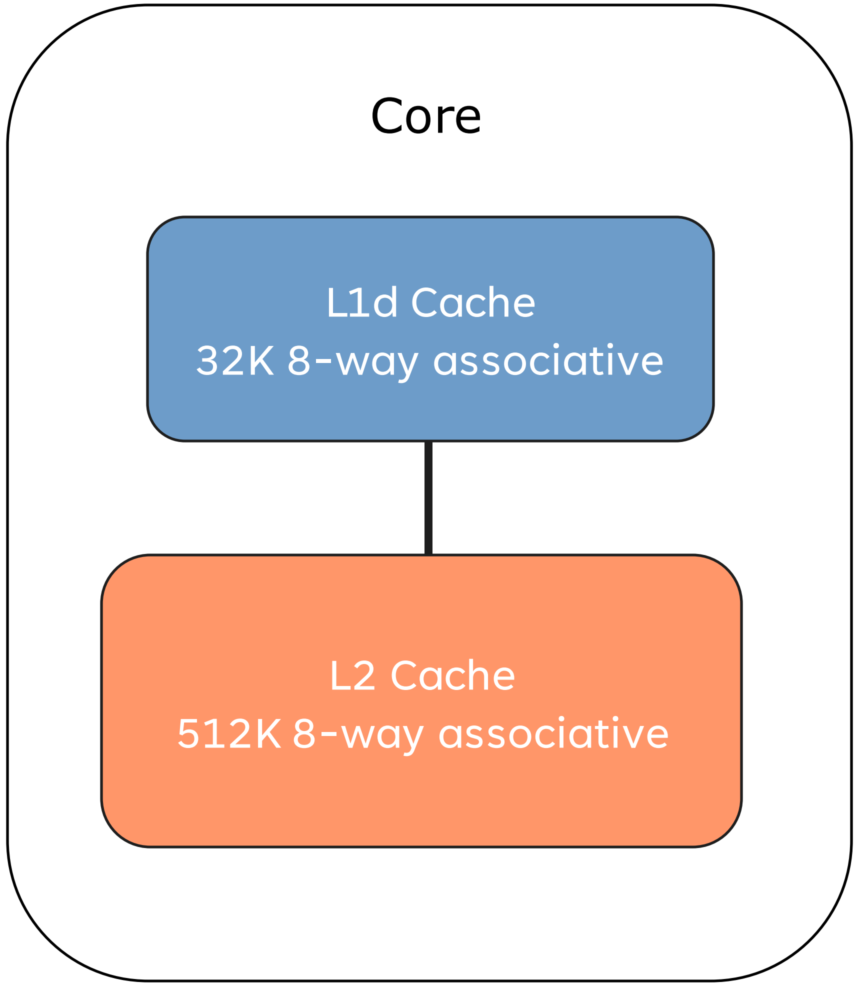
- 2D Convolution is an highly used algorithm in image filtering and Convolutional Neural Networks.
- Highly Parallelizable.
- CPU bound.

Given that the algorithm is parallelizable, let’s try to implement a SIMT version and analise the speedup obtained.



## System

- Architecture : x86_64
- CPU: AMD Ryzen 5 3500U
- Thread(s) per core: 2
- Cores : 4
- Frequency: 2.2 Ghz



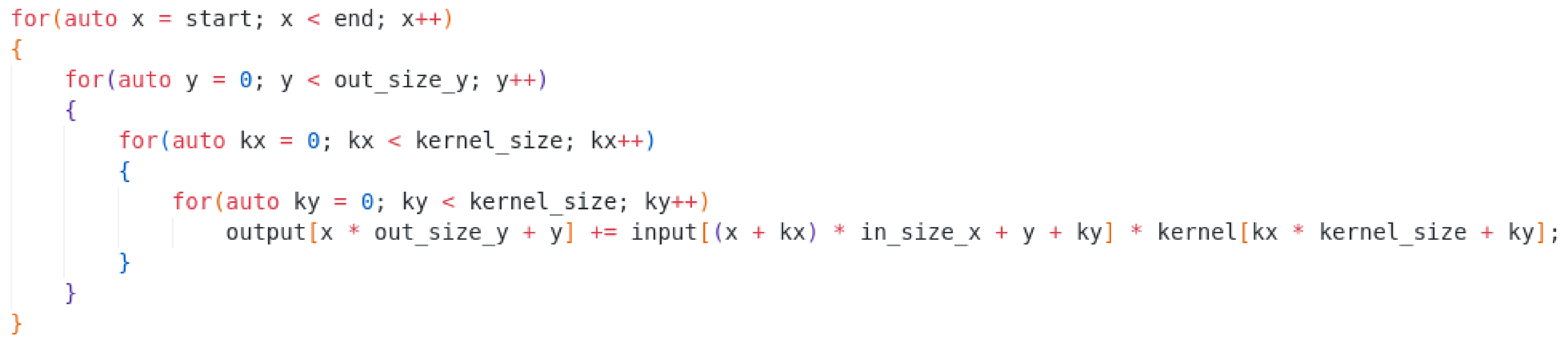
## Measurements

- We measured Wall clock time using std::crono

## Performance indexes

- Cumulative Speedup
- Execution time

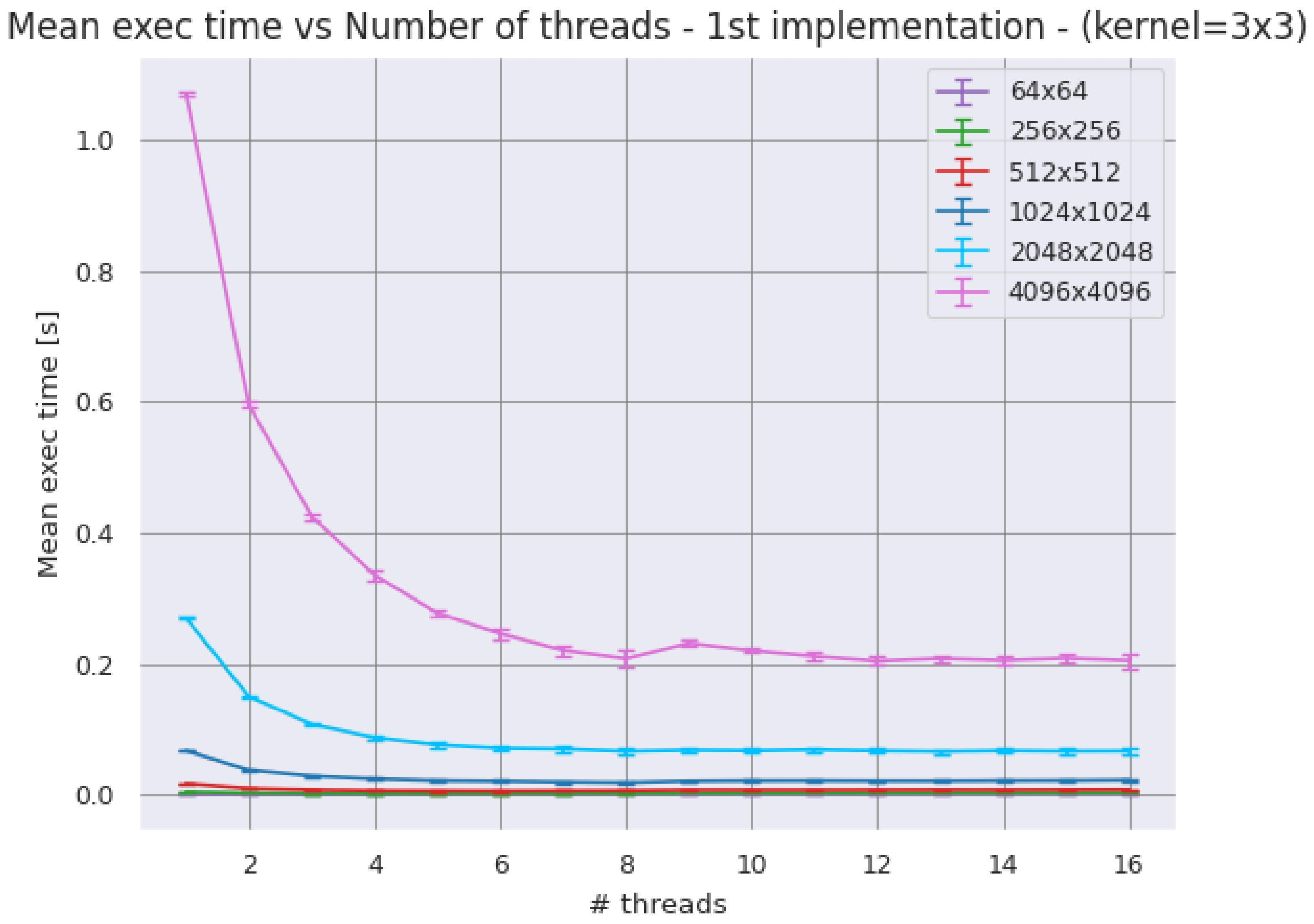
# 1st Implementation



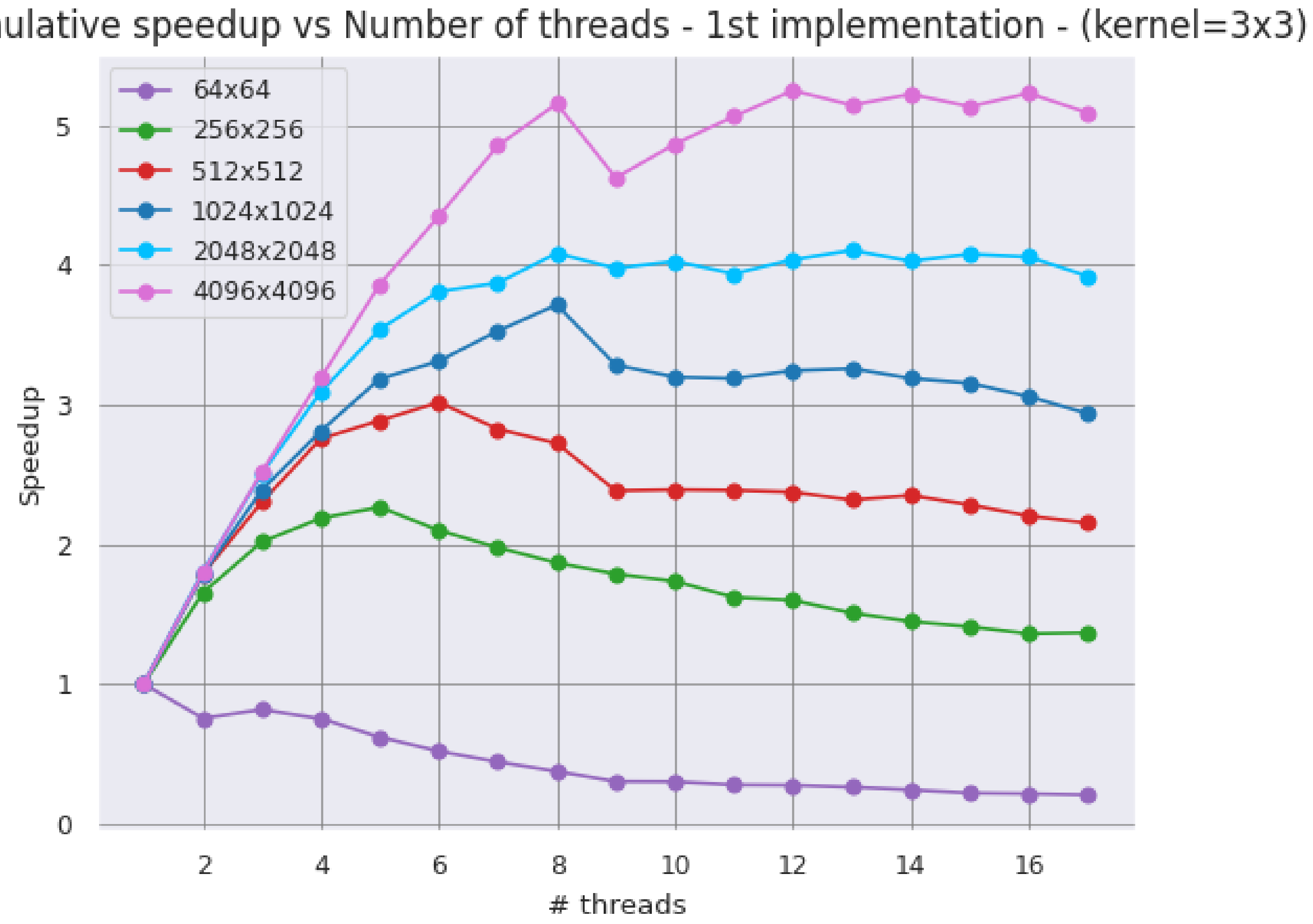
```
for(auto x = start; x < end; x++)
{
    for(auto y = 0; y < out_size_y; y++)
    {
        for(auto kx = 0; kx < kernel_size; kx++)
        {
            for(auto ky = 0; ky < kernel_size; ky++)
                output[x * out_size_y + y] += input[(x + kx) * in_size_x + y + ky] * kernel[kx * kernel_size + ky];
        }
    }
}
```

- Thread's task: compute a costant number of output's elements
- **output[]** is shared among all threads

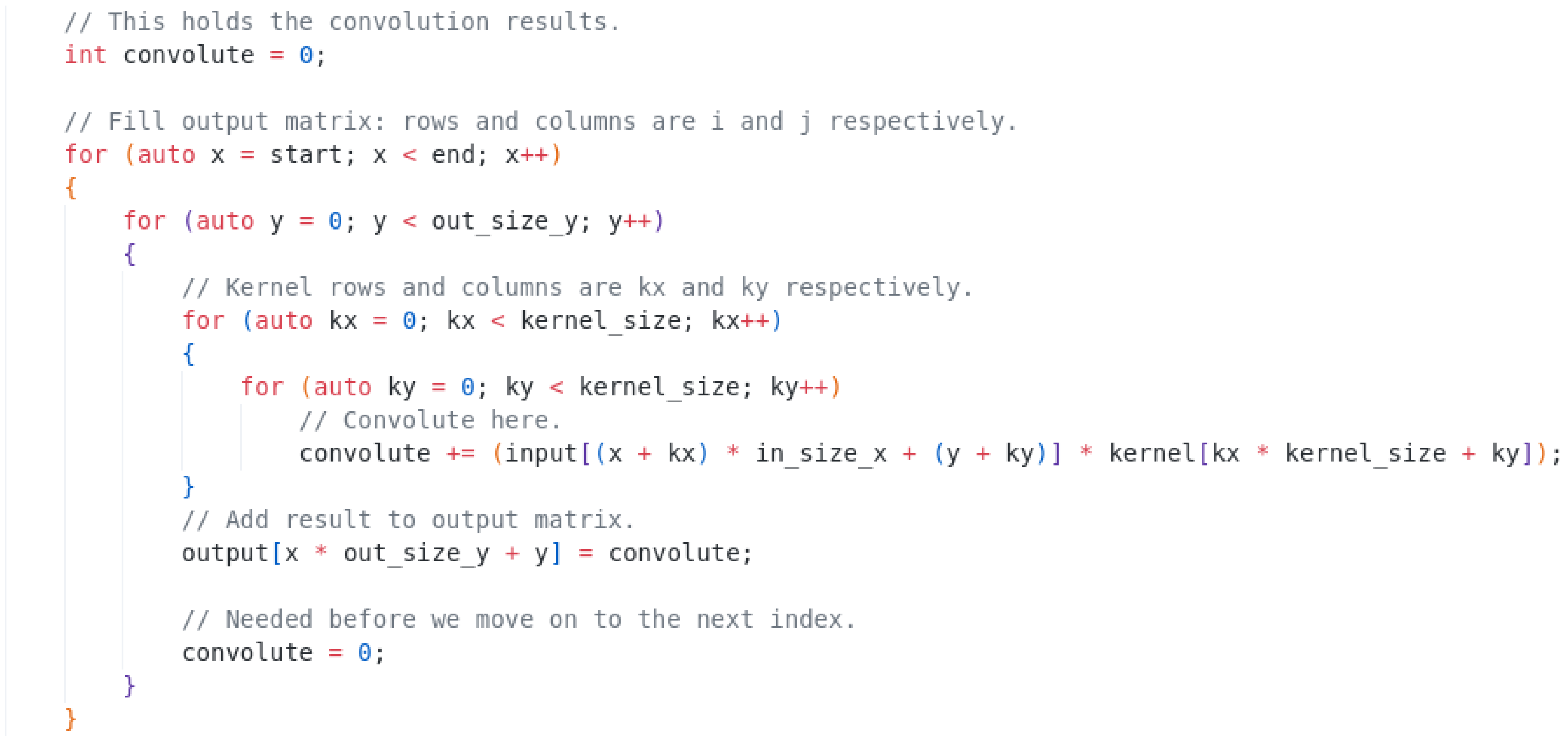
# 1st Implementation: execution time



## 1st Implementation: cumulative speedup



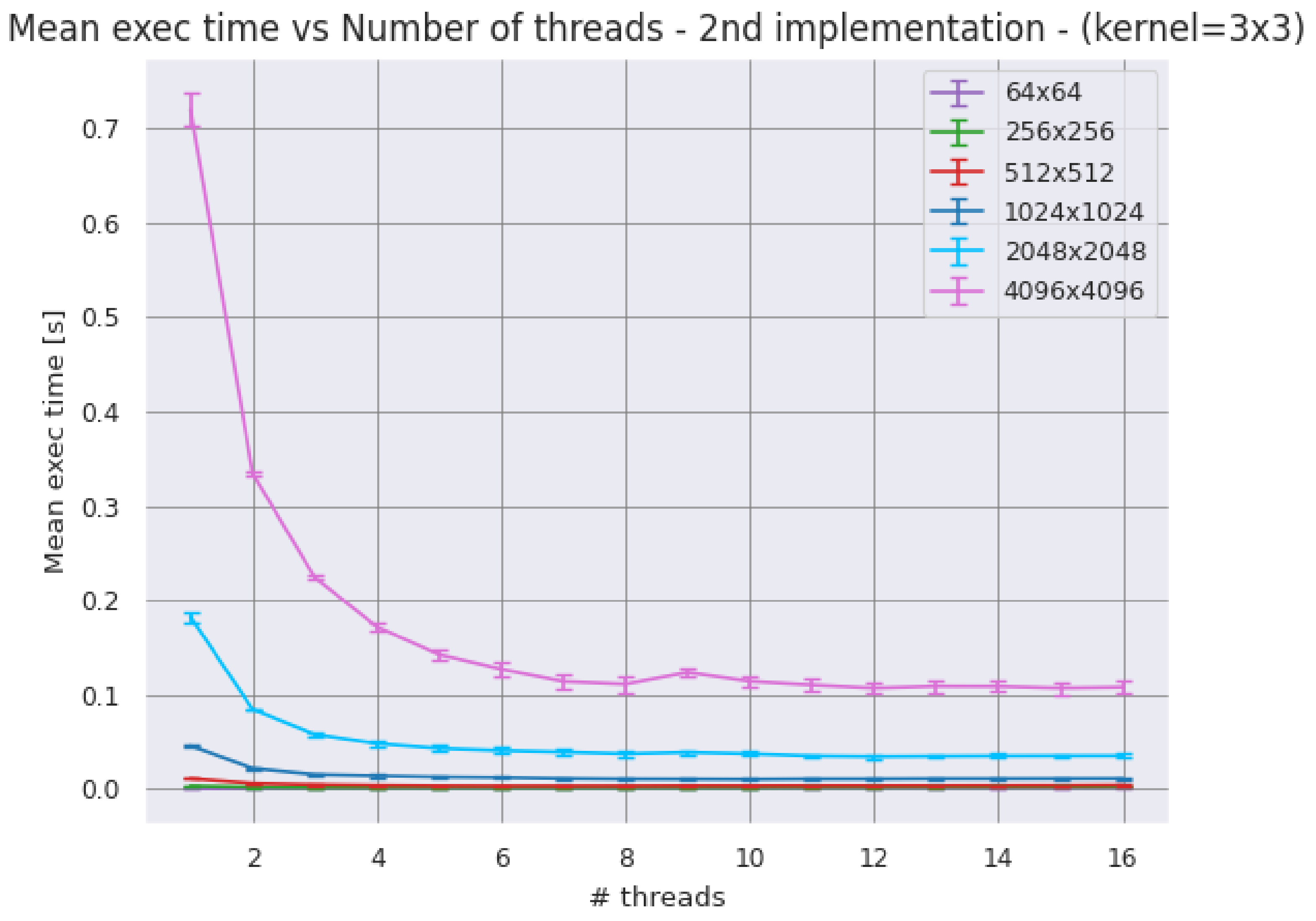
## 2nd Implementation



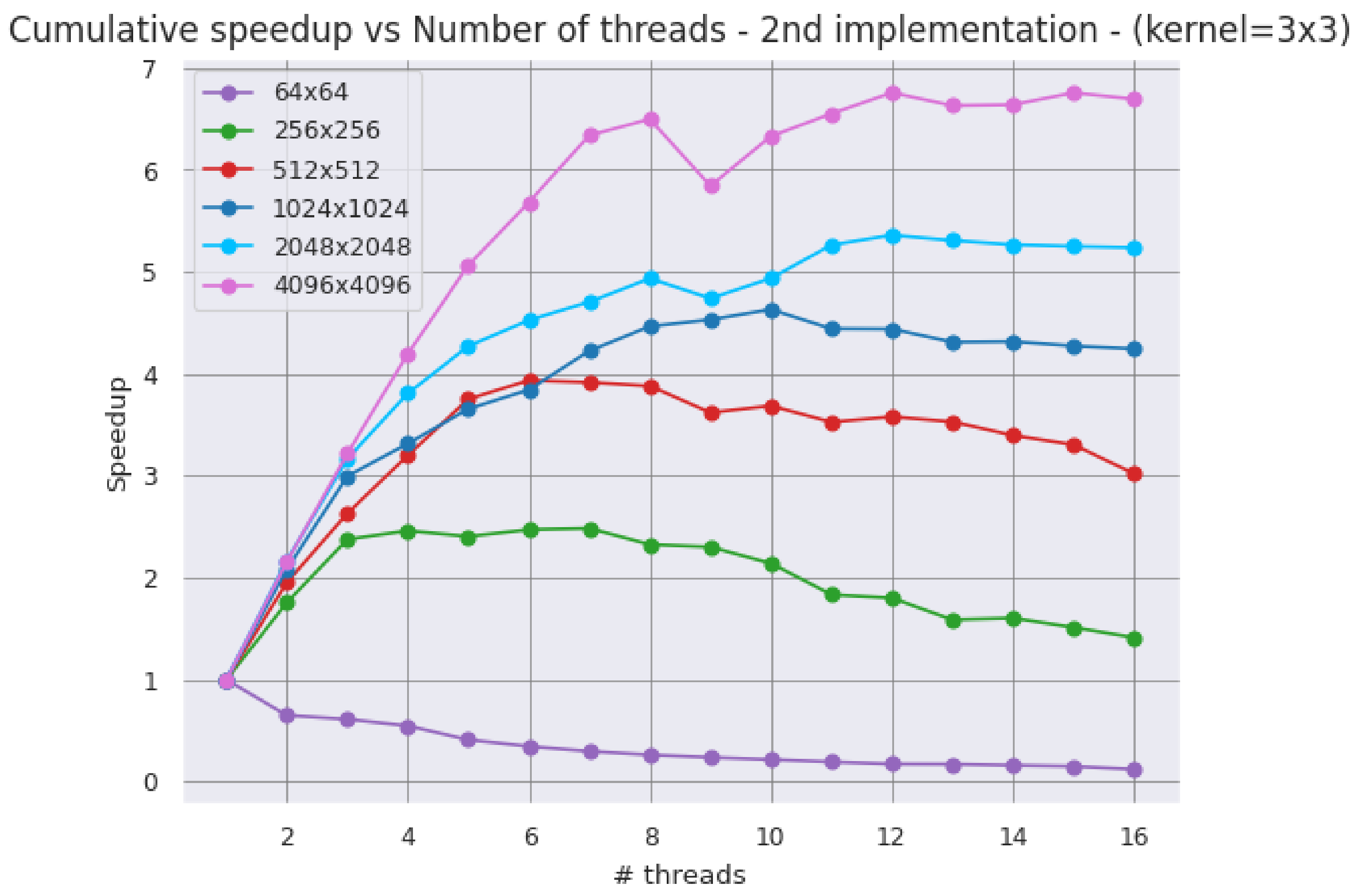
```
// This holds the convolution results.
int convolute = 0;

// Fill output matrix: rows and columns are i and j respectively.
for (auto x = start; x < end; x++)
{
    for (auto y = 0; y < out_size_y; y++)
    {
        // Kernel rows and columns are kx and ky respectively.
        for (auto kx = 0; kx < kernel_size; kx++)
        {
            for (auto ky = 0; ky < kernel_size; ky++)
                // Convolute here.
                convolute += (input[(x + kx) * in_size_x + (y + ky)] * kernel[kx * kernel_size + ky]);
        }
        // Add result to output matrix.
        output[x * out_size_y + y] = convolute;

        // Needed before we move on to the next index.
        convolute = 0;
    }
}
```

- **convolute** is private for each thread.

# 2nd Implementation: execution time

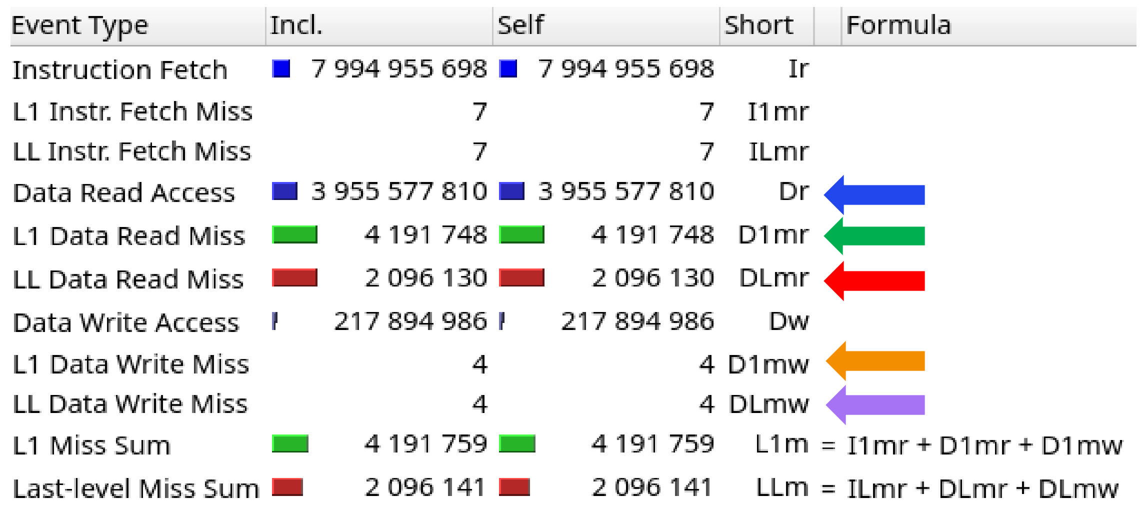


# 2nd Implementation: cumulative speedup



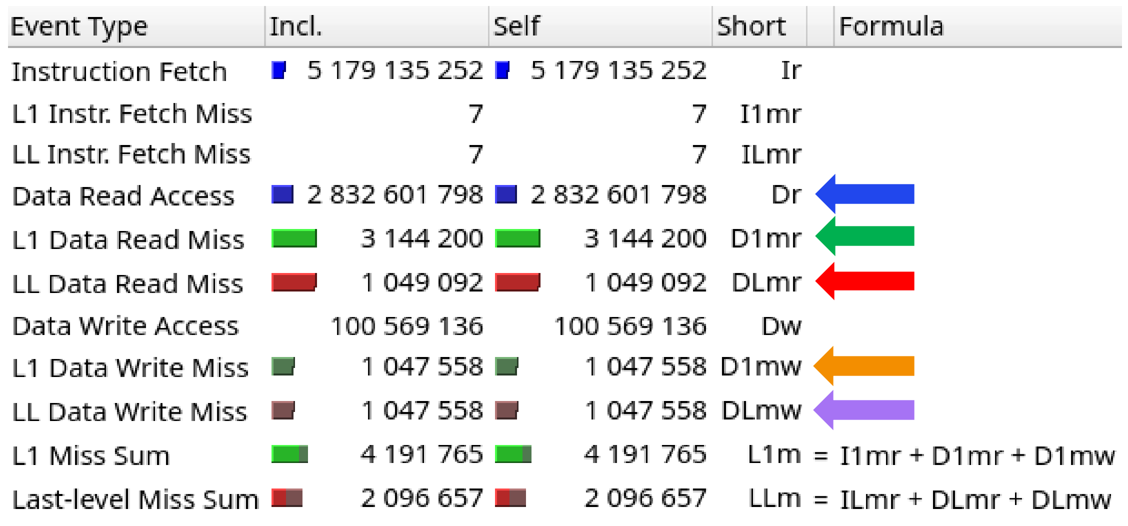
# 2nd Implementation: analysis

convolute(void*): **1st** implementation perfomance



| Event Type | Incl. | Self | Short | Formula |
|---|---|---|---|---|
| Instruction Fetch | 🟦 7 994 955 698 | 🟦 7 994 955 698 | Ir | |
| L1 Instr. Fetch Miss | 7 | 7 | I1mr | |
| LL Instr. Fetch Miss | 7 | 7 | ILmr | |
| Data Read Access | 🟦 3 955 577 810 | 🟦 3 955 577 810 | Dr | |
| L1 Data Read Miss | 🟩 4 191 748 | 🟩 4 191 748 | D1mr | |
| LL Data Read Miss | 🟥 2 096 130 | 🟥 2 096 130 | DLmr | |
| Data Write Access | 217 894 986 | 217 894 986 | Dw | |
| L1 Data Write Miss | 4 | 4 | D1mw | |
| LL Data Write Miss | 4 | 4 | DLmw | |
| L1 Miss Sum | 🟩 4 191 759 | 🟩 4 191 759 | L1m | = I1mr + D1mr + D1mw |
| Last-level Miss Sum | 🟥 2 096 141 | 🟥 2 096 141 | LLm | = ILmr + DLmr + DLmw |

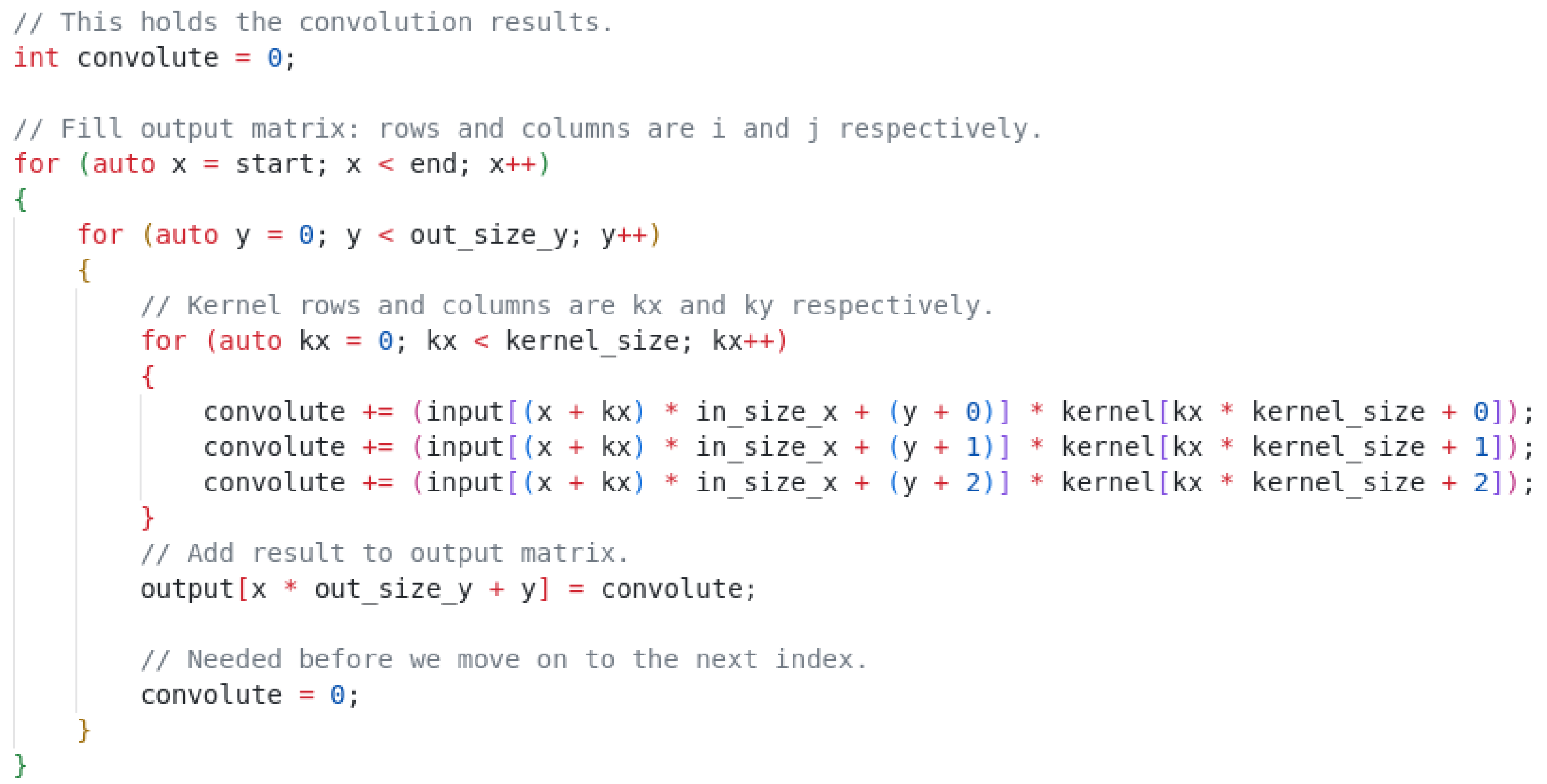
convolute(void*): **2nd** implementation perfomance



| Event Type | Incl. | Self | Short | Formula |
|---|---|---|---|---|
| Instruction Fetch | 🟦 5 179 135 252 | 🟦 5 179 135 252 | Ir | |
| L1 Instr. Fetch Miss | 7 | 7 | I1mr | |
| LL Instr. Fetch Miss | 7 | 7 | ILmr | |
| Data Read Access | 🟦 2 832 601 798 | 🟦 2 832 601 798 | Dr | |
| L1 Data Read Miss | 🟩 3 144 200 | 🟩 3 144 200 | D1mr | |
| LL Data Read Miss | 🟥 1 049 092 | 🟥 1 049 092 | DLmr | |
| Data Write Access | 100 569 136 | 100 569 136 | Dw | |
| L1 Data Write Miss | 🟩 1 047 558 | 🟩 1 047 558 | D1mw | |
| LL Data Write Miss | 🟥 1 047 558 | 🟥 1 047 558 | DLmw | |
| L1 Miss Sum | 🟩 4 191 765 | 🟩 4 191 765 | L1m | = I1mr + D1mr + D1mw |
| Last-level Miss Sum | 🟥 2 096 657 | 🟥 2 096 657 | LLm | = ILmr + DLmr + DLmw |

- L1 Data Read Misses have decreased by 25%.
- L3 Data Read Misses have decreased by roughly 50%.
- L1 Miss sum and L3 Miss sum have not changed.

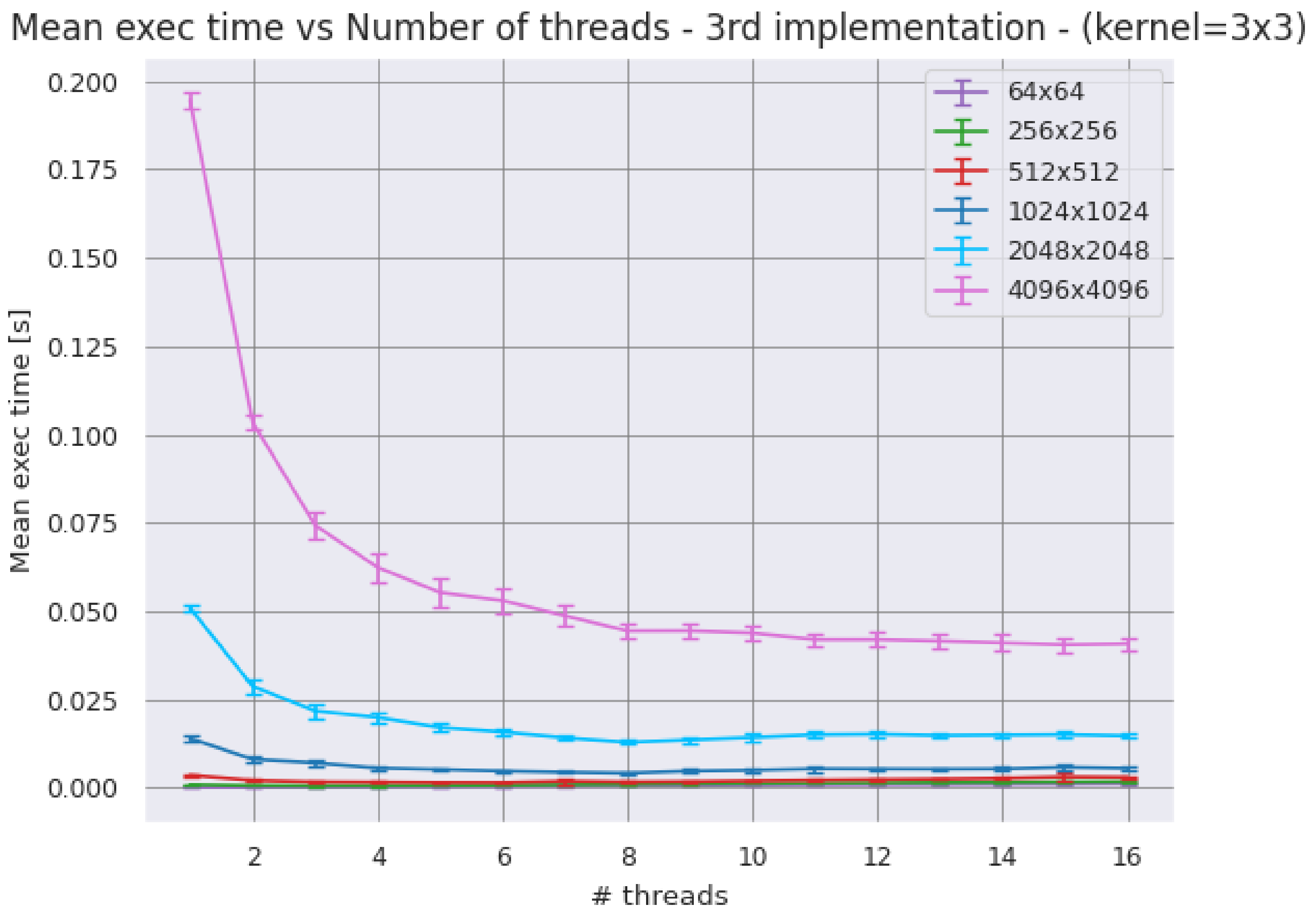
# 3rd Implementation



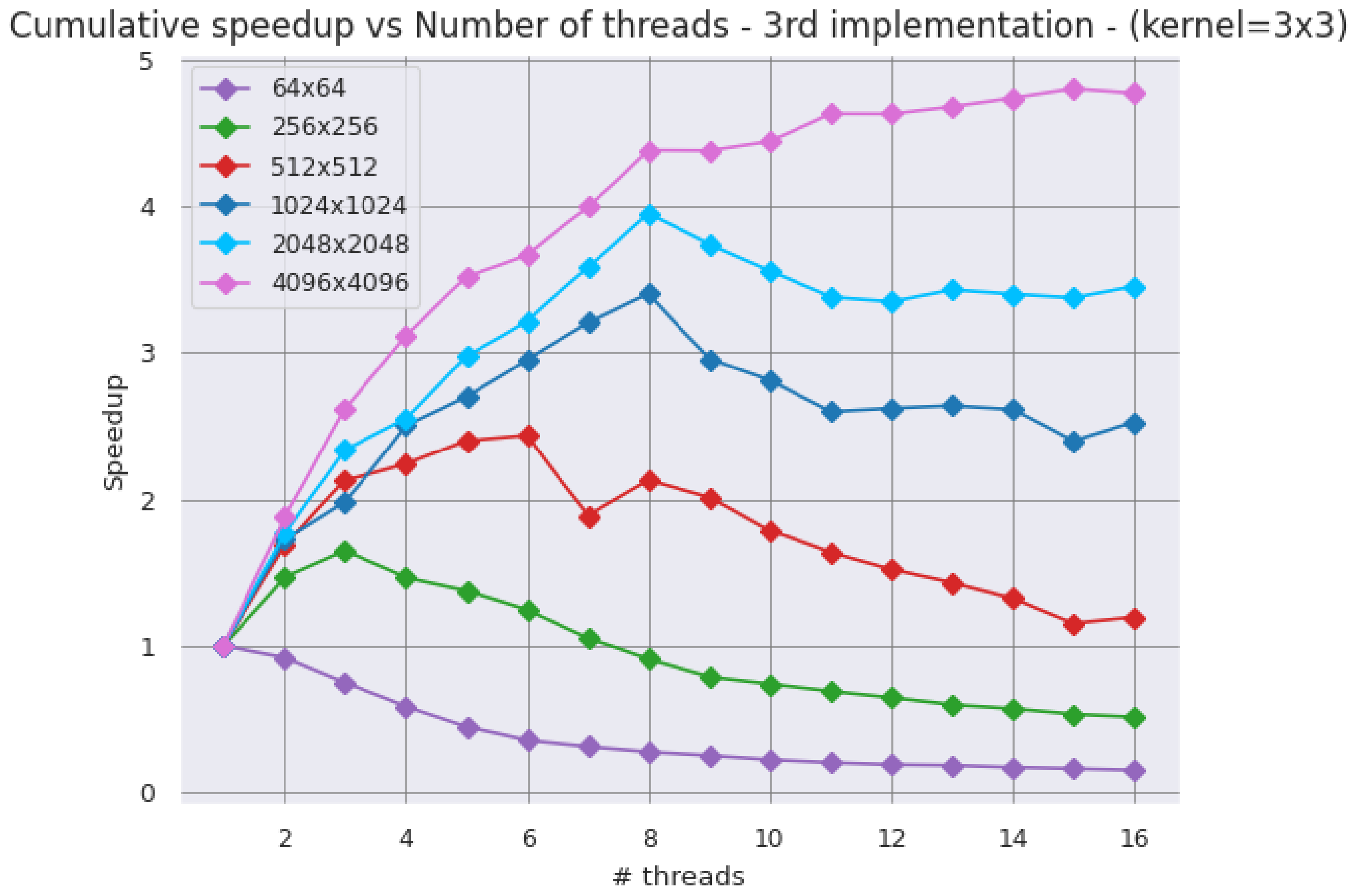
```
// This holds the convolution results.
int convolute = 0;

// Fill output matrix: rows and columns are i and j respectively.
for (auto x = start; x < end; x++)
{
    for (auto y = 0; y < out_size_y; y++)
    {
        // Kernel rows and columns are kx and ky respectively.
        for (auto kx = 0; kx < kernel_size; kx++)
        {
            convolute += (input[(x + kx) * in_size_x + (y + 0)] * kernel[kx * kernel_size + 0]);
            convolute += (input[(x + kx) * in_size_x + (y + 1)] * kernel[kx * kernel_size + 1]);
            convolute += (input[(x + kx) * in_size_x + (y + 2)] * kernel[kx * kernel_size + 2]);
        }
        // Add result to output matrix.
        output[x * out_size_y + y] = convolute;

        // Needed before we move on to the next index.
        convolute = 0;
    }
}
```

- **Loop unrolling**
- Variables classified as **registers**

# 3rd Implementation: execution time

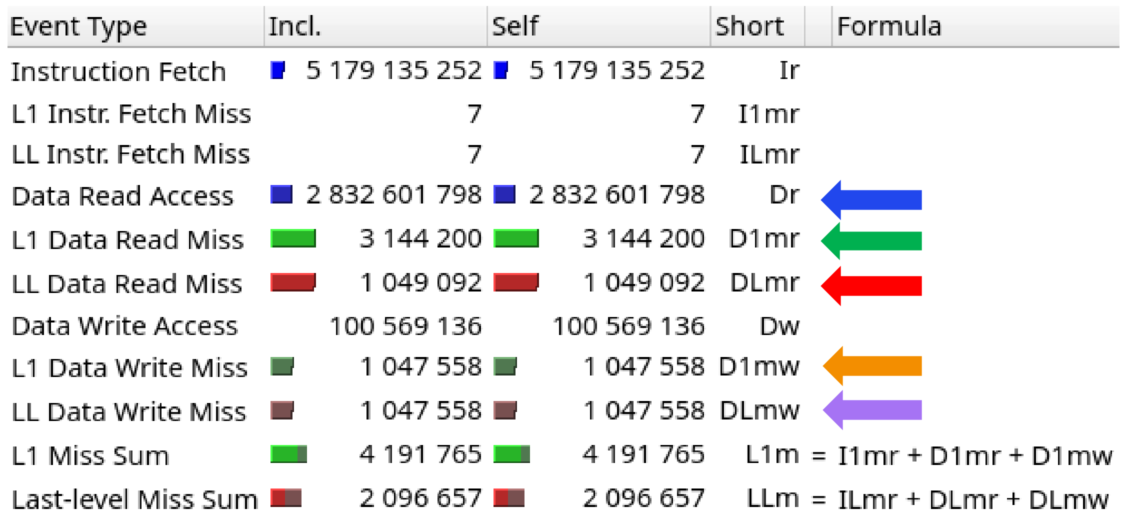


## 3rd Implementation: cumulative speedup



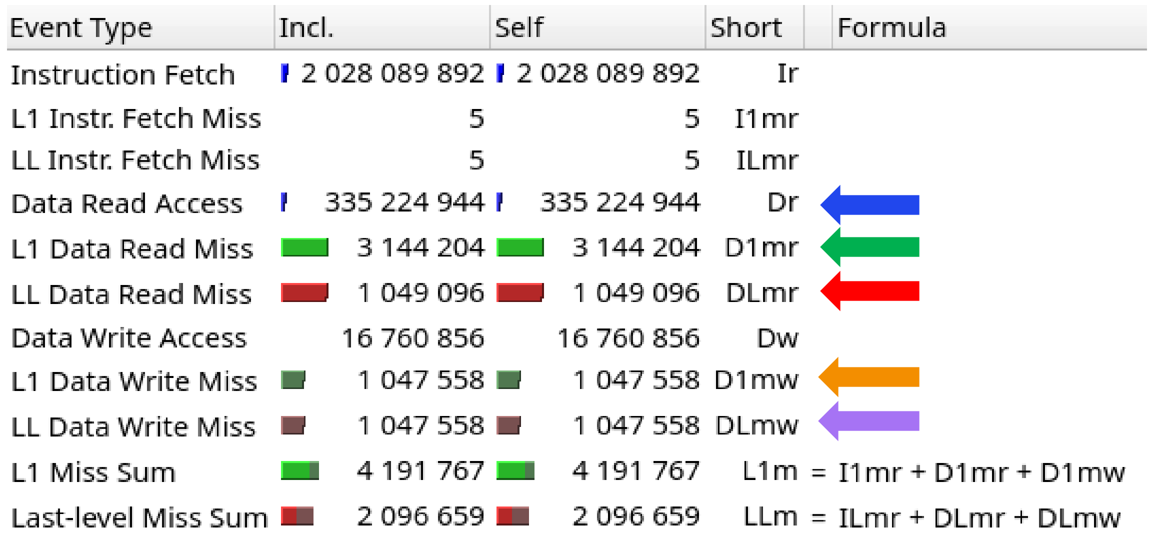
# 3rd Implementation: analysis

convolute(void*): **2nd** implementation perfomance



| Event Type | Incl. | Self | Short | Formula |
|---|---|---|---|---|
| Instruction Fetch | 5 179 135 252 | 5 179 135 252 | Ir | |
| L1 Instr. Fetch Miss | 7 | 7 | I1mr | |
| LL Instr. Fetch Miss | 7 | 7 | ILmr | |
| Data Read Access | 2 832 601 798 | 2 832 601 798 | Dr | |
| L1 Data Read Miss | 3 144 200 | 3 144 200 | D1mr | |
| LL Data Read Miss | 1 049 092 | 1 049 092 | DLmr | |
| Data Write Access | 100 569 136 | 100 569 136 | Dw | |
| L1 Data Write Miss | 1 047 558 | 1 047 558 | D1mw | |
| LL Data Write Miss | 1 047 558 | 1 047 558 | DLmw | |
| L1 Miss Sum | 4 191 765 | 4 191 765 | L1m | = I1mr + D1mr + D1mw |
| Last-level Miss Sum | 2 096 657 | 2 096 657 | LLm | = ILmr + DLmr + DLmw |

convolute(void*): **3nd** implementation perfomance



| Event Type | Incl. | Self | Short | Formula |
|---|---|---|---|---|
| Instruction Fetch | 2 028 089 892 | 2 028 089 892 | Ir | |
| L1 Instr. Fetch Miss | 5 | 5 | I1mr | |
| LL Instr. Fetch Miss | 5 | 5 | ILmr | |
| Data Read Access | 335 224 944 | 335 224 944 | Dr | |
| L1 Data Read Miss | 3 144 204 | 3 144 204 | D1mr | |
| LL Data Read Miss | 1 049 096 | 1 049 096 | DLmr | |
| Data Write Access | 16 760 856 | 16 760 856 | Dw | |
| L1 Data Write Miss | 1 047 558 | 1 047 558 | D1mw | |
| LL Data Write Miss | 1 047 558 | 1 047 558 | DLmw | |
| L1 Miss Sum | 4 191 767 | 4 191 767 | L1m | = I1mr + D1mr + D1mw |
| Last-level Miss Sum | 2 096 659 | 2 096 659 | LLm | = ILmr + DLmr + DLmw |

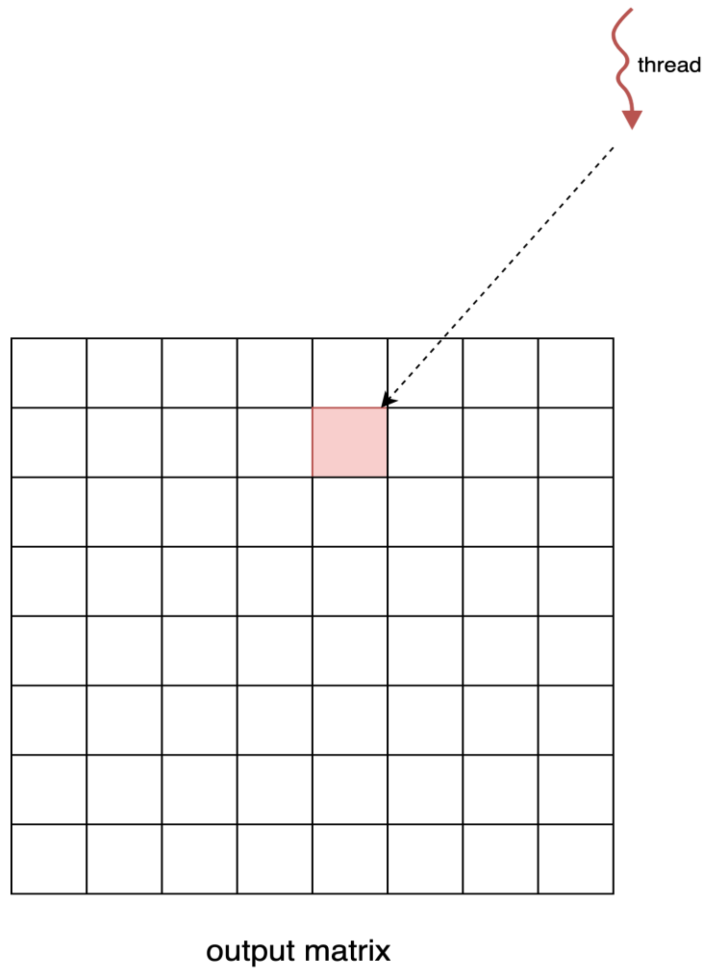
- Data Read Access have decreased by 89%.
- L1 Miss sum and L3 Miss sum have not changed.

## GPU Implementation

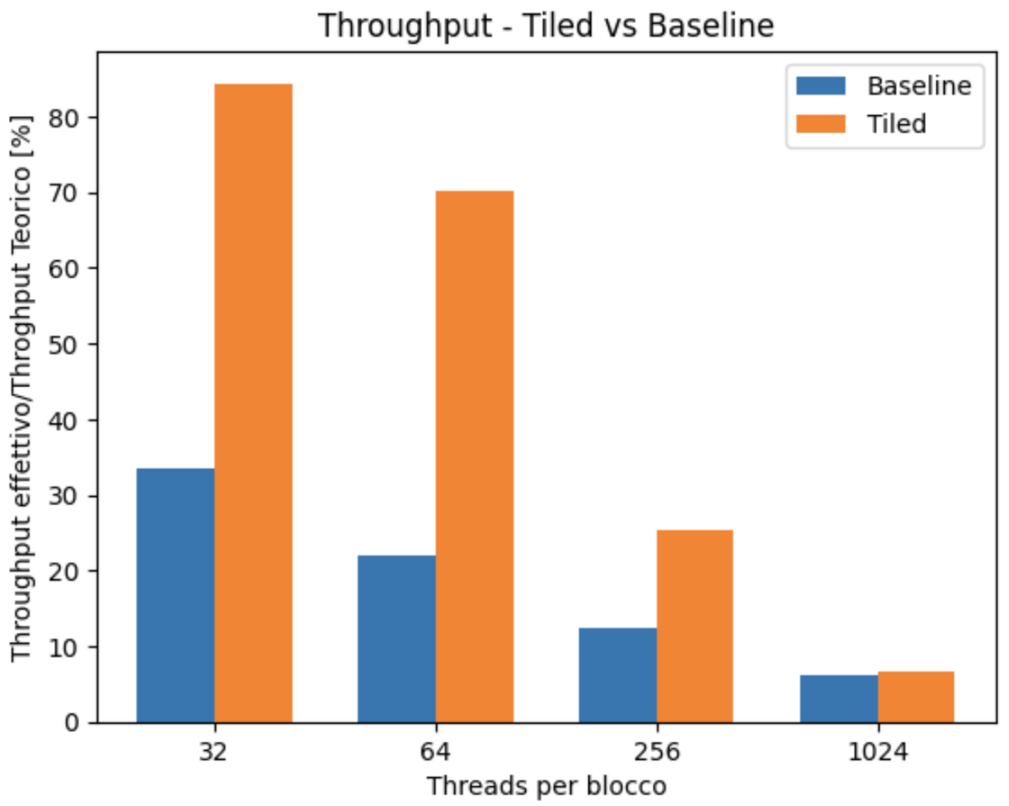
Each thread computes one element.

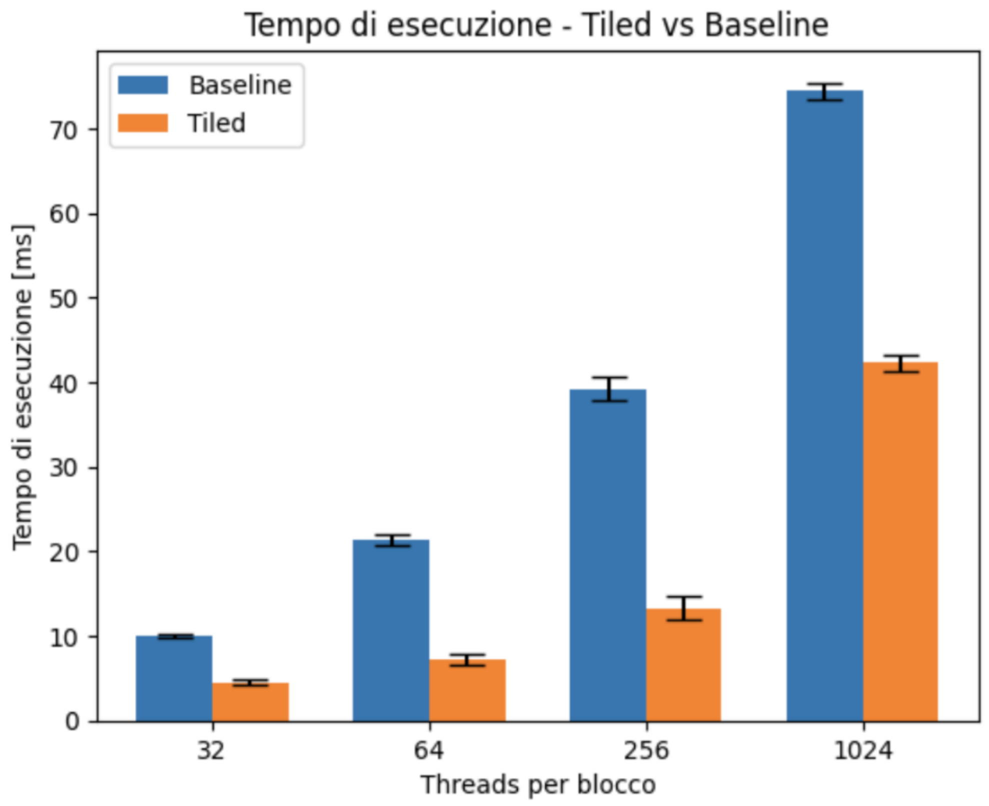
Two versions:

- Baseline
- Tiled (exploits shared memory)



# Performace comparison





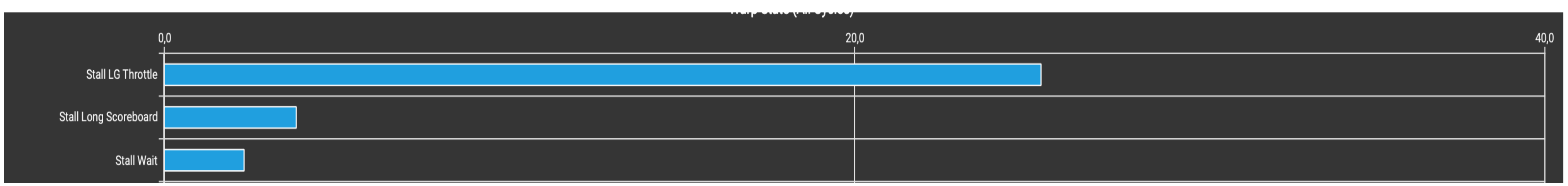
Observations:

– Shared version always performs better.

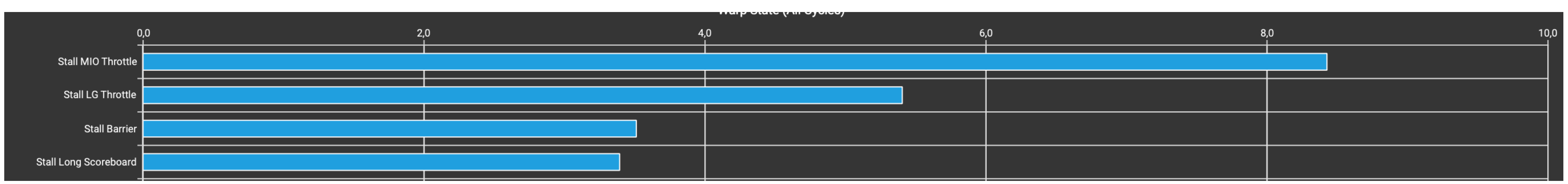
– Increasing block size worsen performance.

# Why Shared performs better

Base:



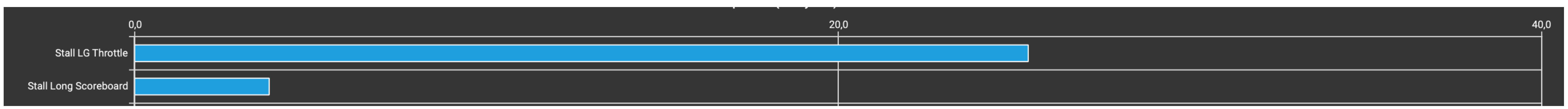
Shared:



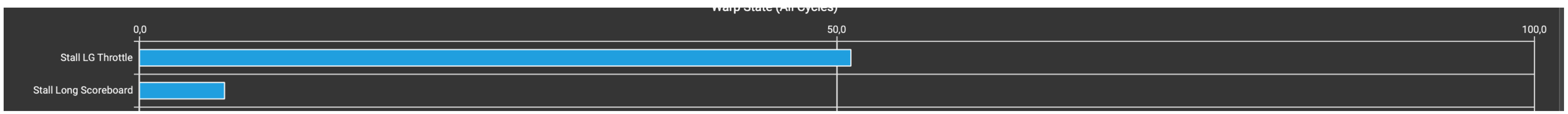
We get much less stalls on global memory loads instruction

# Why increasing threads in a block worsen performance

– 64 Threads per block:



– 256 Threads per block:



- Number of stalled warp cycles doubles, that is in response to contention on global memory.

## Conclusions

- **CPU**
  - If the **input size is small**, the advantages of **multithreading** are **nullified by** the **overhead** resulting from thread scheduling.
  - Massive memory access, such as in convolution, may not allow for achieving speedups that are proportional to the number of threads.

- **GPU**
  - The **performance improves** as the **block dimension decreases** when there is no need to exchange information between threads, as in our algorithm.
  - Exploiting shared memory maximizes throughput.