UNIVERSITÀ DI SIENA 1240

Efficient Convolution

Multi-core implementation

Design of Applications, Systems and Services project

Filippo Guerranti

filippo.guerranti@student.unisi.it

LinkedIn GitHub

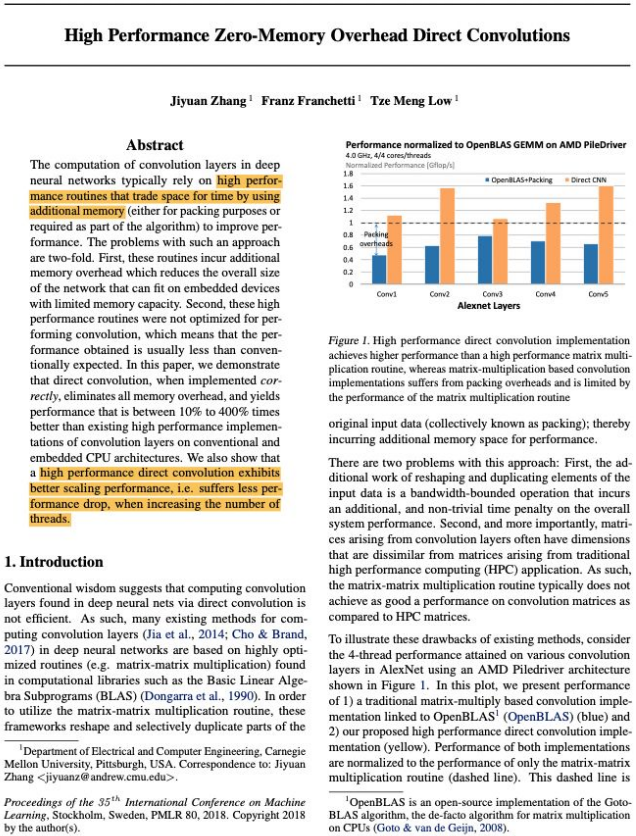
Mirco Mannino

mirco.mannino@student.unisi.it

LinkedIn GitHub

Outline

- **Theoretical introduction**
  - im2col
  - Direct convolutions*
- **Technical implementation**
  - class `Tensor`
- **Experimental results**
  - conclusions



High Performance Zero-Memory Overhead Direct Convolutions

Jiyuan Zhang [1] Franz Franchetti [1] Tze Meng Low [1]

Abstract

The computation of convolution layers in deep neural networks typically rely on high performance routines that trade space for time by using additional memory (either for packing purposes or required as part of the algorithm) to improve performance. The problems with such an approach are two-fold. First, these routines incur additional memory overhead which reduces the overall size of the network that can fit on embedded devices with limited memory capacity. Second, these high performance routines were not optimized for performing convolution, which means that the performance obtained is usually less than conventionally expected. In this paper, we demonstrate that direct convolution, when implemented *correctly*, eliminates all memory overhead, and yields performance that is between 10% to 400% times better than existing high performance implementations of convolution layers on conventional and embedded CPU architectures. We also show that a high performance direct convolution exhibits better scaling performance, i.e. suffers less performance drop, when increasing the number of threads.

1. Introduction

Conventional wisdom suggests that computing convolution layers found in deep neural nets via direct convolution is not efficient. As such, many existing methods for computing convolution layers (Jia et al., 2014; Cho & Brand, 2017) in deep neural networks are based on highly optimized routines (e.g. matrix-matrix multiplication) found in computational libraries such as the Basic Linear Algebra Subprograms (BLAS) (Dongarra et al., 1990). In order to utilize the matrix-matrix multiplication routine, these frameworks reshape and selectively duplicate parts of the

[1] Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, USA. Correspondence to: Jiyuan Zhang <jiyuanz@andrew.cmu.edu>.

Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018. Copyright 2018 by the author(s).

Figure 1. High performance direct convolution implementation achieves higher performance than a high performance matrix multiplication routine, whereas matrix-multiplication based convolution implementations suffers from packing overheads and is limited by the performance of the matrix multiplication routine

original input data (collectively known as packing); thereby incurring additional memory space for performance.

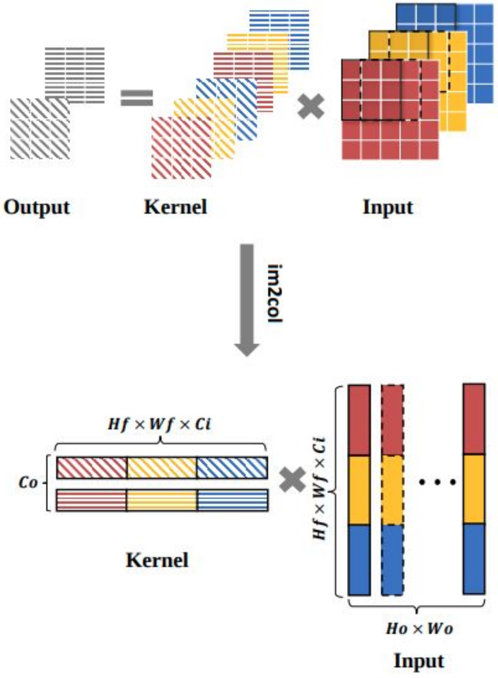
There are two problems with this approach: First, the additional work of reshaping and duplicating elements of the input data is a bandwidth-bounded operation that incurs an additional, and non-trivial time penalty on the overall system performance. Second, and more importantly, matrices arising from convolution layers often have dimensions that are dissimilar from matrices arising from traditional high performance computing (HPC) application. As such, the matrix-matrix multiplication routine typically does not achieve as good a performance on convolution matrices as compared to HPC matrices.

To illustrate these drawbacks of existing methods, consider the 4-thread performance attained on various convolution layers in AlexNet using an AMD Piledriver architecture shown in Figure 1. In this plot, we present performance of 1) a traditional matrix-multiply based convolution implementation linked to OpenBLAS[1] (OpenBLAS) (blue) and 2) our proposed high performance direct convolution implementation (yellow). Performance of both implementations are normalized to the performance of only the matrix-matrix multiplication routine (dashed line). This dashed line is

[1] OpenBLAS is an open-source implementation of the GotoBLAS algorithm, the de-facto algorithm for matrix multiplication on CPUs (Goto & van de Geijn, 2008).

* Zhang, J., Franchetti, F. and Low, T.M. (2018). High Performance Zero-Memory Overhead Direct Convolutions. *Proceedings of the 35th International Conference on Machine Learning*, in *Proceedings of Machine Learning Research* 80:5776-5785

im2col: vectorized convolutions



APPROACH

1) Transform input tensor into a 2D-matrix.
   `[Ci x Hi x Wi] -> [(Hf*Wf*Ci) x (Ho*Wo)]`
2) Transform kernel tensor into a 2D-matrix.
   `[Hf x Wf x Cf] -> [(Co) x (Hf*Wf*Ci)]`
3) Matrix multiplication between Kernel-2D and Input-2D.
4) Add Bias to all elements of the result.
5) Transform 2D-matrix output into a tensor.
   `[(Co) x (Ho*Wo)] -> [(Co x Ho x Wo)]`

DISADVANTAGES

- Additional memory requirements.
- Sub-optimal matrix matrix multiplication.

ADVANTAGES

- High performance libraries for matrix multiplication (BLAS, cuBLAS, …).

* See the appendix for a better understanding of the notation

Direct convolutions: parallelized convolutions

Algorithm 1 Naive Convolution Algorithm

Input: Input $\mathcal{I}$, Kernel Weights $\mathcal{F}$, stride s;
Output: Output $\mathcal{O}$
for $i = 1$ **to** C_i **do**
  for $j = 1$ **to** C_o **do**
    for $k = 1$ **to** W_o **do**
      for $\ell = 1$ **to** H_o **do**
        for $m = 1$ **to** W_f **do**
          for $n = 1$ **to** H_f **do**
            $\mathcal{O}_{j,k,\ell} \mathrel{+}= \mathcal{I}_{i,k\times s+m,\ell\times s+n} \times \mathcal{F}_{i,j,m,n}$

Algorithm 2 Reorder Convolution Algorithm

Input: Input $\mathcal{I}$, Kernel Weights $\mathcal{F}$, stride s;
Output: Output $\mathcal{O}$
for $\ell = 1$ **to** H_o **do**
  for $n = 1$ **to** H_f **do**
    for $m = 1$ **to** W_f **do**
      for $i = 1$ **to** C_i **do**
        for $k = 1$ **to** W_o **do**
          for $j = 1$ **to** C_o **do**
            $\mathcal{O}_{j,k,\ell} \mathrel{+}= \mathcal{I}_{i,k\times s+m,\ell\times s+n} \times \mathcal{F}_{i,j,m,n}$

Algorithm 3 Parallelized Direct Convolution Algorithm

Input: Input $\mathcal{I}$, Kernel Weights $\mathcal{F}$, stride s;
Output: Output $\mathcal{O}$
for $j' = 1$ **to** $C_o/C_{o,b}$ **in Parallel do**
  for $i' = 1$ **to** $C_i/C_{i,b}$ **do**
    for $\ell = 1$ **to** H_o **do**
      for $k' = 1$ **to** $W_o/W_{o,b}$ **do**
        for $n = 1$ **to** H_f **do**
          for $m = 1$ **to** W_f **do**
            for $ii = 1$ **to** $C_{i,b}$ **do**
              for $kk = 1$ **to** $W_{o,b}$ **do**
                for $jj = 1$ **to** $C_{o,b}$ **do**
                  $\mathcal{O}_{j'C_{o,b}+jj,k'W_{o,b}+kk,\ell} \mathrel{+}= \mathcal{I}_{i'C_{i,b}+ii,sk'W_{o,b}+kk+m,\ell s+n} \times \mathcal{F}_{i'C_{i,b}+ii,j'\times C_{o,b}+jj,m,n}$

APPROACH

1) Use `Algorithm 1` as basis.
2) Reorder for loops to take advantage of hardware architecture and cache memory.
3) Select the loop to be divided into ranges which will be computed by different threads, in order to parallelize the operation.

DISADVANTAGES

- Tuning based on specific architecture
- Non-direct use of high performance libraries

ADVANTAGES

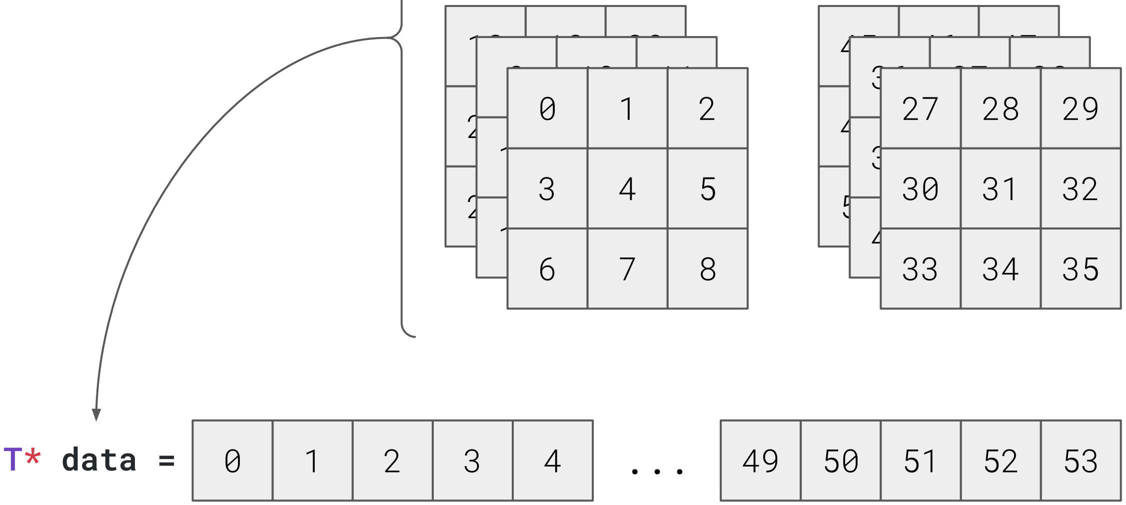
- Almost zero-memory overhead

* See the appendix for a better understanding of the notation

Class Tensor: attributes

```
private:
    // Main class members
    T* data;
    uint32_t nElements;
    uint32_t nChannels;
    uint32_t height;
    uint32_t width;
    // Secondary class members
    uint32_t size;
    std::vector<uint32_t> shape;
    bool valid;
```

```
nElements: 2
nChannels: 3
height:    3
width:     3
```



Class Tensor: methods* - Constructors

```
public:
    // Default constructor
    Tensor();
    // 3D constructor
    Tensor(const uint32_t& nChannels_, const uint32_t& height_, const uint32_t& width_,
           const tensor::init& init);
    // 4D constructor
    Tensor(const uint32_t& nElements_, const uint32_t& nChannels_, const uint32_t& height_, const uint32_t&
width_,
           const tensor::init& init);
    // Copy constructor
    Tensor(const Tensor<T>& other);
    // Move constructor
    Tensor(Tensor<T>&& other);
```

* The list of all the methods can be found in the GitHub repository.

Class Tensor: methods* - Operator "at"

```cpp
public:
    // 3D operator at() const
    const T& at(const int32_t& C_idx, const int32_t& H_idx,
                const int32_t& W_idx) const;
    // 3D operator at() non-const
    T& at(const int32_t& C_idx, const int32_t& H_idx,
          const int32_t& W_idx);

    // 4D operator at() const
    const T& at(const int32_t& E_idx, const int32_t& C_idx,
                const int32_t& H_idx, const int32_t& W_idx) const;
    // 4D operator at() non-const
    T& at(const int32_t& E_idx, const int32_t& C_idx,
          const int32_t& H_idx, const int32_t& W_idx);
```

```cpp
private:
    // 3D operator _at() const
    const T& _at(const int32_t& C_idx, const int32_t& H_idx,
                 const int32_t& W_idx) const;
    // 3D operator _at() non-const
    T& _at(const int32_t& C_idx, const int32_t& H_idx,
           const int32_t& W_idx);

    // 4D operator _at() const
    const T& _at(const int32_t& E_idx, const int32_t& C_idx,
                 const int32_t& H_idx, const int32_t& W_idx) const;
    // 4D operator _at() non-const
    T& _at(const int32_t& E_idx, const int32_t& C_idx,
           const int32_t& H_idx, const int32_t& W_idx);
```

* The list of all the methods can be found in the GitHub repository.

Class Tensor: methods* - Convolution

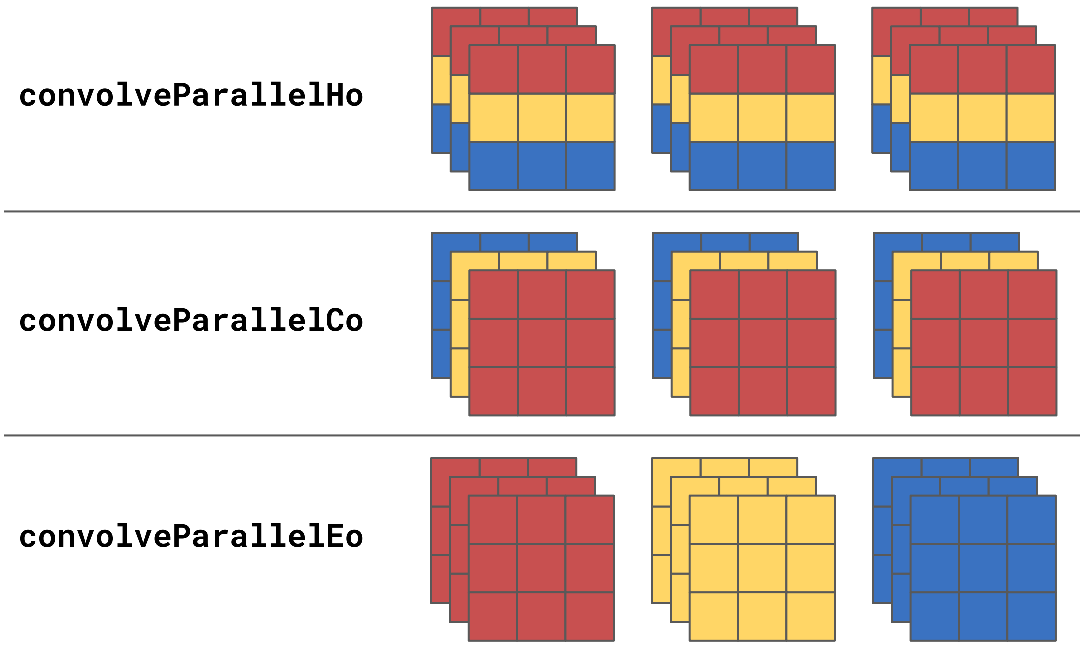
```
public:
    // Convolution operator (parallel) - dimension: output height
    Tensor<T>& convolveParallelHo(const Tensor<T>& kernel, const int32_t stride, const int32_t padding, const uint32_t nThreads) const;
    // Convolution operator (parallel) - dimension: output nChannels
    Tensor<T>& convolveParallelCo(const Tensor<T>& kernel, const int32_t stride, const int32_t padding, const uint32_t nThreads) const;
    // Convolution operator (parallel) - dimension: output nElements
    Tensor<T>& convolveParallelEo(const Tensor<T>& kernel, const int32_t stride, const int32_t padding, const uint32_t nThreads) const;

    // Convolution Naive (sequential)
    Tensor<T>& convolveNaive(const Tensor<T>& kernel, const int32_t stride, const int32_t padding) const;

    // Convolution operator that select automatically dimension for parallelization
    Tensor<T>& convolve(const Tensor<T>& kernel, const int32_t stride, const int32_t padding, const uint32_t nThreads) const;
    // Convolution operator that select automatically dimension for parallelization and number of threads
    Tensor<T>& convolve(const Tensor<T>& kernel, const int32_t stride, const int32_t padding) const;
```

* The list of all the methods can be found in the GitHub repository.

Class Tensor: convolve parallel

N. of threads: 3

- Thread 1
- Thread 2
- Thread 3

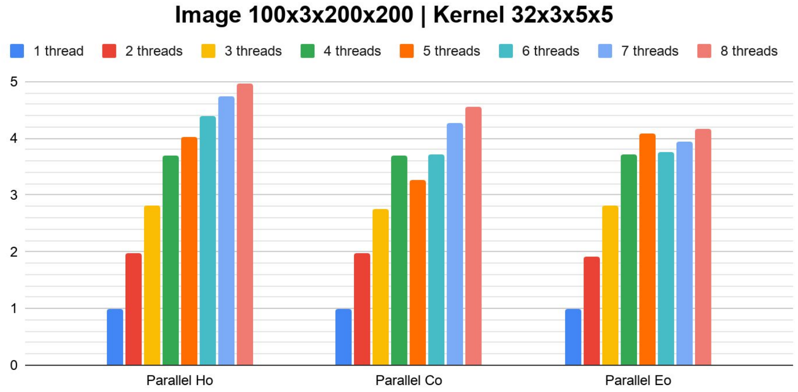


- The tensor represented in the images is the **output tensor** of dimension [3x3x3x3]
- Each thread performs the convolution operation through the private method **`convolveThread`**

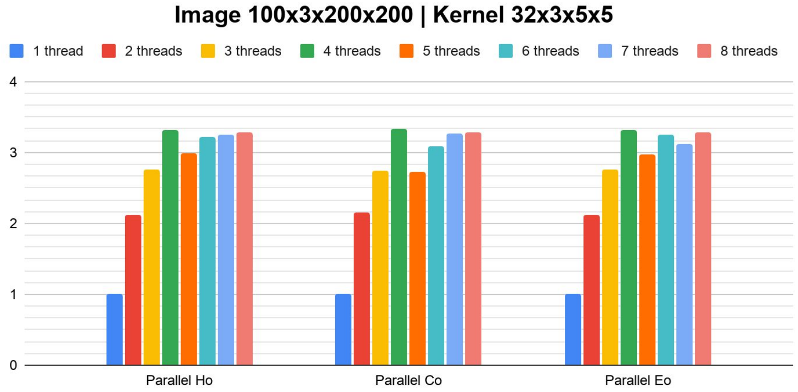
Speed-up: w.r.t. Naive impl. for different thread number

Dimension of kernel tensor: [32x3x5x5]

Dimension of input tensor: [100x3x200x200]



Apple Silicon M1*



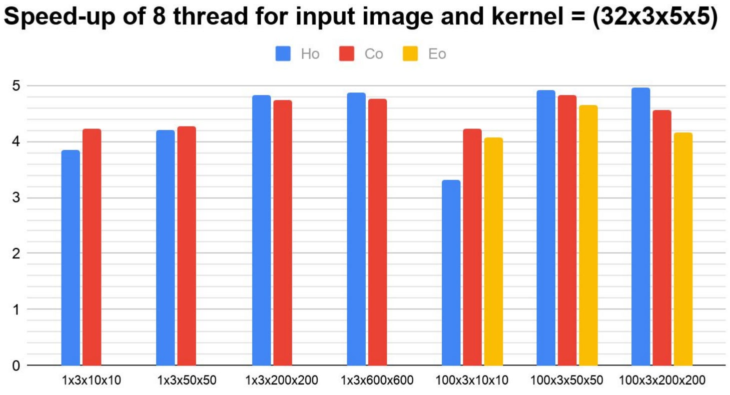
Intel Core i710510u*

* See the appendix for hardware specifications

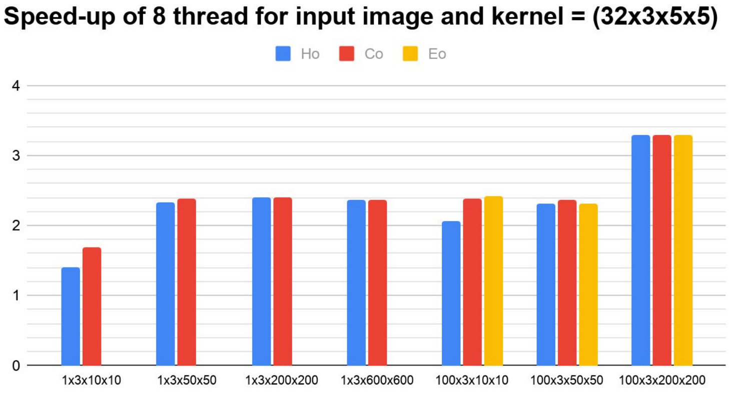
Speed-up: w.r.t. Naive impl. for 8 threads and different inputs

Dimension of kernel tensor: [32x3x5x5]

Dimension of input tensor: see x axis on the charts



Apple Silicon M1*



Intel Core i710510u*

* See the appendix for hardware specifications

Conclusions

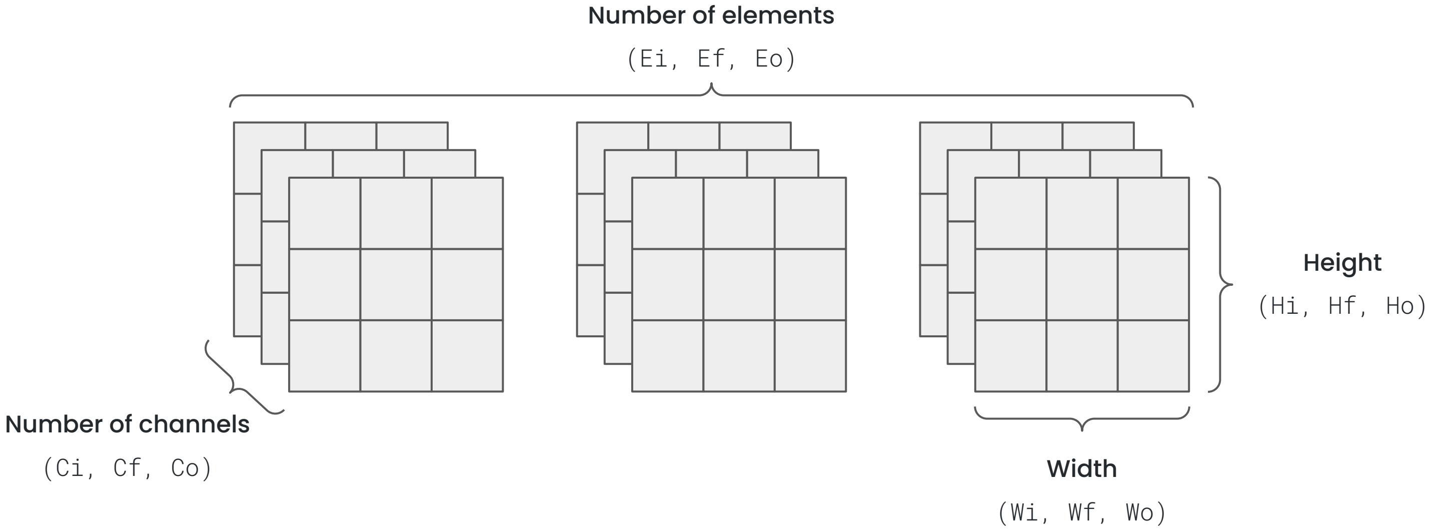
- Generally, increasing the number of threads entails a higher speed-up.
- The **Apple Silicon M1** shows a *"linear"-like* behavior, due to the presence of 8 physical cores even if the lasts four provide a weaker speed-up w.r.t. the firsts four (see appendix).
- The **Intel Core i710510u** architecture shows a *"linear"-like* behavior in the firsts four cores, whereas in the lasts four, being hyperthreaded (see appendix), there is a sort of "plateau".
- With small tensors (`nElements = 1`, or `(height = width) <= 50`) there is not deterministic behaviour between the different parallelization techniques (`parallelHo, parallelCo, parallelEo`).
  - It is difficult to determine the best parallelization technique w.r.t. the input and kernel tensors, since they all show similar results.

Appendix: notations

(Wi, Hi, Ci, Ei) — **input** dimensions

(Wf, Hf, Cf, Ef) — **kernel (filter)** dimensions

(Wo, Ho, Co, Eo) — **output** dimensions



Appendix: hardware specifications

CPU architectures	Apple Silicon M1	Intel Core i710510u
N. physical cores	8	4
N. logical cores	8	8

The 8 physical cores* of the Apple Silicon M1 are divided into:

- 4 x high-performance + 4 x high-efficiency

The Intel Core i710510u has 4 physical cores* which are hyperthreaded*.

* Sources: Apple chip | Intel chip | Hyperthreading