ARCHIVIO ISTITUZIONALE DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Dissecting FLOPs Along Input Dimensions for GreenAI Cost Estimations

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
Asperti, A., Evangelista, D., Marzolla, M. (2022). Dissecting FLOPs Along Input Dimensions for GreenAI Cost Estimations. Springer [10.1007/978-3-030-95470-3_7].

Availability:
This version is available at: https://hdl.handle.net/11585/857696 since: 2022-02-14

Published:
DOI: http://doi.org/10.1007/978-3-030-95470-3_7

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

16 September 2026

This is the final peer-reviewed accepted manuscript of:

Asperti, A., Evangelista, D., Marzolla, M. (2022). Dissecting FLOPs Along Input Dimensions for GreenAI Cost Estimations. In: , et al. Machine Learning, Optimization, and Data Science. LOD 2021. Lecture Notes in Computer Science, vol 13164. Springer, Cham, pp. 86–100

The final published version is available online at ***https://dx.doi.org/10.1007/978-3-030-95470-3_7***

Rights / License:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/)

When citing, please refer to the published version.

Dissecting FLOPs along input dimensions for GreenAI cost estimations

Andrea Asperti[1], Davide Evangelista[2], and Moreno Marzolla[1]

[1] University of Bologna
Department of Informatics: Science and Engineering (DISI)
[2] University of Bologna
Department of Mathematics

Abstract. The term GreenAI refers to a novel approach to Deep Learning, that is more aware of the ecological impact and the computational efficiency of its methods. The promoters of GreenAI suggested the use of Floating Point Operations (FLOPs) as a measure of the computational cost of Neural Networks; however, that measure does not correlate well with the energy consumption of hardware equipped with massively parallel processing units like GPUs or TPUs. In this article, we propose a simple refinement of the formula used to compute floating point operations for convolutional layers, called α-FLOPs, explaining and correcting the traditional discrepancy with respect to different layers, and closer to reality. The notion of α-FLOPs relies on the crucial insight that, in case of inputs with multiple dimensions, there is no reason to believe that the speedup offered by parallelism will be uniform along all different axes.

1 Introduction

Artificial Intelligence, especially in its modern incarnation of Deep Learning, has achieved remarkable results in recent years, matching – and frequently trespassing – human capabilities in a number of different tasks. These techniques usually require the deployment of massive computational resources, with huge implications in terms of energy consumption. To make a couple of examples the hyper-realistic Generative Adversarial Network for face generation in [19] required training on 8 Tesla V100 GPUs for 4 days; the training of BERT [12], a well known generative model for NLP, takes about 96 hours on 64 TPU2 chips. Researchers at the University of Massachusetts [26] have recently performed a life cycle assessment relative to the training of large state-of-the-art AI models, discovering that the process can emit a quantity of carbon dioxide roughly equivalent to the lifetime emissions of five medium cars. Other authors reached similar conclusions [20].

Until a few years ago, the ecological impact of artificial intelligence was entirely neglected by researchers and industry, who were mostly focused on improving performance at any cost. However, this has changed in recent years, with a growing awareness that this trend of research is not sustainable any more [28], and an increased attention towards energetic efficiency [27].

The GreenAI paper [25] summarizes well the goal and objectives of the new philosophy: it promotes a new practice in Deep Learning, that is more focused on the social costs of training and running models [2,7,15], encouraging the investigation of increasingly efficient models [21,5].

To this aim, it is essential to identify widely acceptable and reliable metrics to assess and compare the cost and efficiency of different models. Several metrics are investigated and discussed in [25]; in conclusion, the number of Floating Point Operations (FLOPs) is advocated and promoted, since it is easily computed for Neural Networks while offering a hardware independent, schematic but meaningful indication of the actual computation cost of the model [20].

Unfortunately, the mere computation of FLOPs does not cope well with the massively parallel architectures (GPU and TPU) typically used in Deep Learning [17]. Efficient implementation of neural networks on these architectures depends both on complex algorithms for General Matrix Multiplication (GEMM) [18] and sophisticated load balancing techniques [13] splitting the workload on the different execution units. As we shall see, these algorithms usually perform better for specific layers and, especially, *along specific axes* of the input dimension of these layers.

Our claim is that it is possible to study the performance of neural layers (especially, convolutions) as "black boxes", measuring the execution time for a number of different configurations, and separately investigating the execution time for increasing dimensions along different axis.

As a result, we propose a simple correction to the formula used to compute FLOPs for convolutional layers, that provides better estimations of their actual cost, and helps to understand the discrepancy with respect to the cost of different layers.

Organization of the article This paper has the following structure. In Section 2 we briefly discuss some possible metrics for measuring the efficiency of models; we particularly focus on FLOPs, discussing their computation for some basic operations relevant for Neural Networks. In Section 3 we introduce the GEMM (GEneral Matrix Multiply) operation, that helps to understand the canonical computation of FLOPs for the Convolution layers. In Section 4 we present some experiments which show that, if Convolutions are executed on GPU, FLOPs are not a good measure for efficiency. That is the motivation for introducing a correction, that we call α-FLOPs, defined and discussed in Section 5. Section 6 offers more experimental results, validating the formula with respect to growing input dimensions along specific axes.

2 Measures of Efficiency

In this section we review some of the metrics that can be used to measure the efficiency of an AI algorithm, following the discussion of [25].

Carbon Emission As already remarked in the introduction, the present work is motivated by the need to reduce the energy consumption of training large state-of-the-art AI models. Unless a significant fraction of such energy comes from renewable sources, reducing the power required for AI training means that less carbon dioxide is released into the atmosphere. Unfortunately, precise quantification of carbon emission associated with computational tasks is impractical, since it depends both on the hardware hosting the computation, and also on the local energy production and distribution infrastructure.

Number of parameters The number of parameters of a Deep Learning model is an interesting and hardware-independent measure of the complexity of models. Unfortunately, the number of parameters alone is poorly correlated with the total training time, since parameters may refer to different operations. For example, Convolutional Layers have relatively few parameters, relative to the kernel of the convolution; this does not take into account the actual cost of convolving the kernel over the input.

Execution time The total running time is a natural measure of efficiency: faster algorithms are better. Execution time depends on the number of instructions executed and hence is strictly correlated with the total energy consumption [24]; therefore, it is a good proxy of power usage when direct energy measurement is impractical. There are a couple of important considerations that must be made when considering execution time as a metric: (*i*) it requires an implementation of the algorithm being measured, which may take time and effort to be developed; (*ii*) execution time is hardware- and language-dependent, since it depends on both the underlying hardware and on the efficiency of the compiler/interpreter.

FLOPs The number of FLoating Point OPerations (FLOPs) is a metric that is widely used in the context of numerical computations [23,14,29,22]. It is defined as the total count of elementary machine operations (floating point additions and multiplications) executed by a program. Floating point operations have a latency of several CPU cycles on most current processor architectures [10,9,3], although the use of pipelining, multiple-issue and SIMD instructions significantly increase the throughput. In general, floating point operations have higher latency than most of the other CPU instructions (apart from load/stores from/to main memory, where memory access is the bottleneck); therefore, they tend to dominate the execution time of numerical algorithms. For this reason, the number of floating point operations is used as a proxy for the execution time of a program.

As an example, suppose that v and w are n-dimensional arrays. Then, the inner product between v and w

$$\langle v; w \rangle = \sum_{i=1}^{n} v_i w_i \tag{1}$$

requires n multiplications and $n-1$ additions, for a total of $2n-1$ FLOPs. Similarly, the matrix-vector product between an $m \times n$ matrix A and an n-dimensional vector v requires m inner product, for a total of $2mn - m$ FLOPs.

Since operations similar to (1), where a sequence of multiplications are added together, are very common, modern CPUs supports FMA (Fused Multiply-Add) instructions, where a multiplication followed by an addition are executed as a single operation and require less time than two separate instructions. For this reason, the definition of FLOPs is usually modified to be the total number of FMA operations needed for a full iteration of an algorithm. With this definition (that it is usually followed by some authors), the inner product of two n-dimensional arrays requires n FLOPs, while the product between an $m \times n$ matrix with an n-dimensional vector requires nm FLOPs. Nonetheless, since we are interested in measuring the performance under massively parallel architectures, through this paper we will follow the classical definition of FLOPs.

3 Computation of FLOPs for basic layers

The basic operation that dominates training of Neural Network models is the dense matrix-matrix product. This operation is often referred in the technical literature as GEMM (for *GEneral Matrix Multiply*), owing its name to the `xGEMM` family of functions provided by the Basic Linear Algebra Subprograms (BLAS) library [6]. BLAS is a widely used collection of subroutines implementing basic operations involving vectors and matrices, such as vector addition, dot product, vector-matrix multiplication and so on; these functions act as building blocks on which more complex linear algebra computations can be programmed. Being at the core of many applications, the performance of BLAS primitives are critical, so most hardware vendors provide their own optimized implementations, e.g., `cuBLAS` for nVidia GPUs [11], and `clBLAS` for OpenCL devices [8], including various brands of GPUs and multicore processors.

A GEMM operation takes the general form:

$$\mathbf{C} \leftarrow \alpha\mathbf{AB} + \beta\mathbf{C} \tag{2}$$

where $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are matrices of compatible size, and α, β are scalars. The matrix-matrix product $\mathbf{C} \leftarrow \mathbf{AB}$ is a special case of (2) where $\alpha = 1$, $\beta = 0$.

Assuming that the size of $\mathbf{A}$ is $m \times k$ and the size of $\mathbf{B}$ is $k \times n$, then the size of $\mathbf{C}$ must be $m \times n$ and the direct computation of (2) using vector dot products requires:

- $2mkn + mn$ FLOPs for the matrix product $\alpha\mathbf{AB}$, assuming that dot products are implemented with an inner loop involving a multiply-accumulate operation like $s \leftarrow s + x_i y_i$
- mn FLOPs for the computation of $\beta\mathbf{C}$
- mn additional FLOPs for the computation of the matrix sum $\alpha\mathbf{AB} + \beta\mathbf{C}$

from which we get that a total count of $2mkn + mn + mn + mn = mn(2k + 3)$ FLOPs are required for the general GEMM. Neglecting lower-order terms we can approximate the operation count with $2mkn$.

We can apply this result for the layers of a Neural Network. Consider a Dense layer, with input and output dimensions D_{in} and D_{out}, respectively. We need to compute the product between the weight matrix of size $D_{out} \times D_{in}$ and the input, plus a bias term B of dimension D_{out}; therefore, the number of FLOPs is

$$2D_{in}D_{out} - D_{in} + D_{out}$$

As above, we omit the lower order terms as they are asymptotically negligible. As a consequence, we will consider a Dense layer to have a number of FLOPs equal to

$$2D_{in}D_{out} \tag{3}$$

The case of a convolutional layer is slightly more complex. Let us consider the case of a 2D convolution. Let (W_{in}, H_{in}, C_{in}) the dimension of the input (written with the notation (*Width, Height, Channels*)), $(W_{out}, H_{out}, C_{out})$ the dimension of the output (depending on the stride and number of kernels), and let K_1, K_2 be the dimensions of the kernel. Then, the number of FLOPs is given by

$$2 \cdot \underbrace{K_1 \cdot K_2 \cdot C_{in}}_{\text{kernel dim}} \cdot \underbrace{W_{out} \cdot H_{out}}_{\text{input dim}} \cdot \underbrace{C_{out}}_{\text{output dim}} \tag{4}$$

In the following, we shall frequently consider the case of convolutions with stride 1 in "same" padding modality. In this case, $W_{in} = W_{out}$ and $H_{in} = H_{out}$, so we shall drop the subscripts, and just write W and H. Moreover, in the frequent case kernels are squared, we drop the subscripts in K_1, K_2 and just write K.

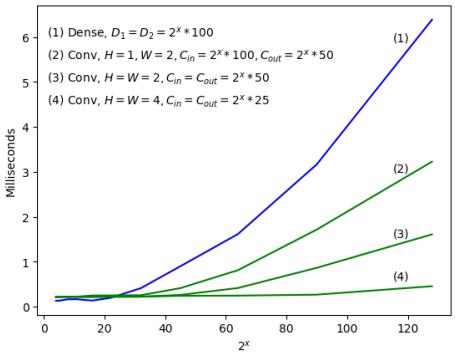
4 The problem of convolutions

A dense layer of dimension $D_{in} \times D_{out}$ is the same as a unary convolution ($K = 1$) with $C_{in} = D_{in}$, $C_{out} = D_{out}$ and $H = W = 1$; it is easy to experimentally check that both require the same time to be evaluated. However, as soon as we distribute the total number of FLOPs of equation (4) across different dimensions, we observe a significant speedup, that has no justification in terms of FLOPs. This raises concerns about the use of FLOPs for estimating running time (and hence energy consumption). In this section we provide empirical evidence of this phenomenon.

In Figure 1, we compare the time required to evaluate a dense layer with several different convolutional layers with a same amount of FLOPs computed according to (4); the execution time has been measured on an NVIDIA Quadro T2000 graphics card and a Intel Core i7-9850H CPU. Times are averaged over 2000 experiments for each scenario.

327.68 M FLOPs	
Dense layer	
(D_1, D_2)	time (ms)
$(12800, 12800)$	6.401
Convolutional layers	
$(W, H, C_{in}, C_{out}, K_1, K_2)$	time (ms)
$(1, 1, 12800, 12800, 1, 1)$	6.392
$(1, 2, , 6400, 12800, 1, 1)$	3.224
$(2, 2, 6400, 6400, 1, 1)$	1.626
$(4, 4, 3200, 3200, 1, 1)$	0.454

(a)



(b)

Fig. 1: Comparison of execution times for Dense and Convolutional layers with the *same* amount of FLOPs. In Table (a) we provide numerical values for layers with 327.68 Million FLOPs; in the right we show the execution time of similar configurations for increasing dimensions. All layers for a given value of 2^x (i.e. along any vertical section) have the same amount of FLOPs.

In particular, in Table 1a we evaluate a scenario of maximum size compatible with our hardware, corresponding to a Dense layer of size 12800×12800 ($163,852,800$ parameters), and compare it with several different convolutional layers with the same total amount of FLOPs. The dense layer takes about 6.4 milliseconds (ms), while a unary convolution with $C_{in} = C_{out} = 3200$ on an input of spatial dimension 4×4 just takes 0.46 ms, approximately 16 times faster.

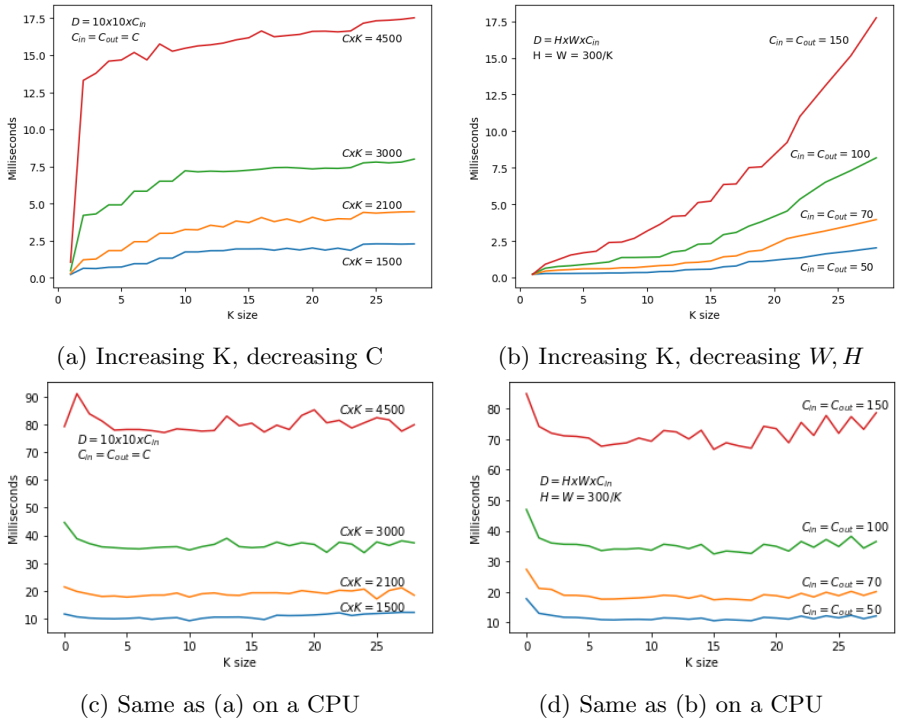
In Figure 1b, we repeat the same experiment, varying the total amount of flops with powers of 2. For the dense layer we go from dimension 100×100 to dimension $(100 \times 2^7) \times (100 \times 2^7)$.

In the following experiments, we keep the number of FLOPs constant while we increase some dimensions and proportionally decrease others. If (4) had a good correlation with time, we should observe straight horizontal lines.

In all experiments, we consider four different amounts of FLOPs identified by different colors: 2025×10^6 (red line in Figure 2), 900×10^6 (green line), 490×10^6 (orange line) and 225×10^6 (blue line). We progressively increase K from 1 to 30. In the first experiment, we compensate it by enlarging the input and output dimension of channels (C_{in} and C_{out}), keeping a constant (small) spatial dimension 10×10.

In the second test we compensate the growing convolutions by reducing the spatial dimensions, starting from an initial dimension of 300×300. Channels are constant, in this case. Result are reported in Figure 2.

In the case of the first experiment (Figure 2a), apart from the exceptional performance of 1×1 convolutions already discussed in [17], we observe the expected constant behavior. However, we have a completely different result in the



(a) Increasing K, decreasing C

(b) Increasing K, decreasing W, H

(c) Same as (a) on a CPU

(d) Same as (b) on a CPU

Fig. 2: Execution time vs different input dimensions, keeping the number of FLOPs constant. In plot (a) we increase K and proportionally decrease C_{in} and C_{out}. In plot (b) we increase K and proportionally decrease W and H. We would expect *constant lines*, but this is not the case. In plots (c) and (d) we repeat the experiment on a (single core) CPU, instead of a GPU.

case of the second experiment (Figure 2b). Here the execution time increases with the kernel dimension, possibly at a quadratic rate; this growth should have been compensated by the simultaneous decrease along both spatial dimensions, but clearly this is not the case.

By comparing the results of the two experiments, we can draw another conclusion. Remember that the number of FLOPs along lines of the same color is the same; therefore, the nonlinear behaviour in Figure 2b is not due to an *overhead* but, on the contrary, there is an important *speed up* of the computation of increasing relevance for small kernels. In other words, the formula computing FLOPs is *overestimating* the total number of operations, presumably because it does not take into consideration the fact that convolutions can be *easily parallelized* along spatial dimensions (but not quite so along kernel dimensions).

The goal of the work is to derive a simple correction to the formula for computing FLOPs explaining the observed behaviours. The correction might

depend on the specific hardware, but it should be possible to evaluate the relevant parameters in a simple way.

5 α-FLOPs

In this section we introduce our correction to the formula for computing FLOPs, that we call α-FLOPs. Instead of FLOPs, that count the total number of floating point operations, α-FLOPs provide an estimation of the "perceived" FLOPs, that are less than FLOPS due to parallelism. The crucial idea is that when we run in parallel an algorithm with a multidimensional input there is no reason to suppose that the total number of operations have similar latency along different dimensional axes. Our proposal is to adjust the formula for computing FLOPs by multiplying it by the following scaling factor:

$$\alpha_K(S) = \left(\frac{S_K + \beta_K(S - S_K)}{S}\right)^{\gamma_K} \tag{5}$$

where $S = W \times H$, and $0 < \beta_K \ll 1, 0 < \gamma_K \leq 1$, and $1 \leq S_K \leq S$ ($S_1 = 1$) are parameters (moderately) depending from K. We call α-FLOPs the correction to the usual formula for FLOPs by the previous factor.

The parameters β_K and γ_K can be easily evaluated by regression on a given GPU/TPU. Although they are hardware dependent, some preliminary investigations seem to suggest that fluctuations are smaller than expected.

For the purposes of this article, using an Invida Quadro T2000 GPU we obtained good predictions just distinguishing two cases: $K = 1$ and $K > 1$. For $K = 1$, $\beta_K = 0.02$ and $\gamma_K = .99$; for $K > 1$, $\beta_K = 0.001$ and $\gamma_K = .56$.

Before discussing the main properties of $\alpha_K(S)$, let us have a look at the prediction of the execution time (dashed line) for the problematic experiments shown above. More examples will be presented in Section 6.

The experiment in Figure 1 is replicated, with the time predicted by means of α-FLOPs, in Figure 3b. In the Table on the left, we give the computed and predicted times for the convolutional configurations (1-4) with 327.68M FLOPs.

Similarly, in Figure 4 we show the predicted execution time for the experiments of Figure 2.

5.1 Main properties of the α-correction

Before discussing our intuition behind (5), let us point out some of its distinctive properties. First of all the equation can be rewritten in the following, possibly more readable form:

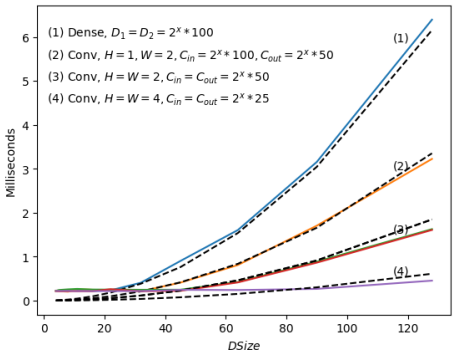
$$\alpha_K(S) = \left((1 - \beta_K) \times \frac{S_K}{S} + \beta_K\right)^{\gamma_K} \tag{6}$$

From that, the following properties can be observed:

1. $\alpha_K(S) < 1$ for any K and S. This is evident, given the constraint $S_k < S$.

327.68 M FLOPs		
Convolutional layers		
config.	time	predicted
(1)	6.392	6.154
(2)	3.224	3.351
(3)	1.626	1.847
(4)	0.454	0.611

(a)



(b)

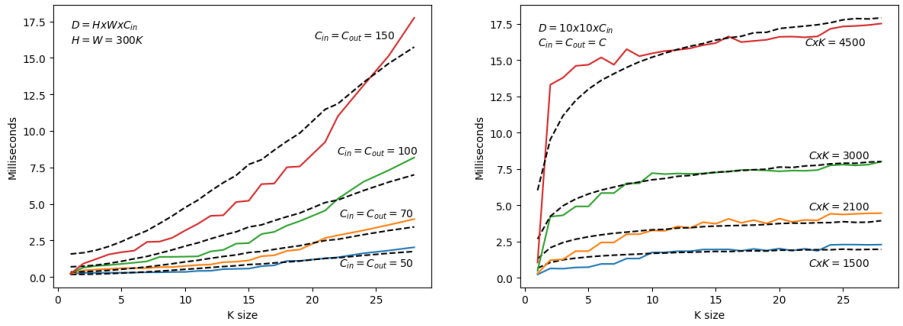
Fig. 3: Predicted execution time by means of α-FLOPs for the same convolutional configurations of Figure 1; in (b) predictions are depicted as dashed lines.

.

2. If $\beta_K = 1$, then $\alpha_K(S) = 1$, independently from γ_K and S. In this case, we recover the original expression for FLOPs, that is hence a subcase with no additional speedup.
3. $\alpha_1(1) = 1$ independently from β_K and γ_K. This is due to the fact that $S_1 = 1 = S$. The case $S = 1, K = 1$ is important since, as discussed at the beginning of Section 3, it gives the relation between convolutional and dense layers, and for a dense layer we want no correction. Moreover, the fact that the fundamental equation $\alpha_1(1) = 1$ holds independently from β and γ improves the stability of the property.
4. The formula with $\gamma = 1$ already gives reasonable approximations. However, it tends to underestimate the execution time for large S, in a more sensible way for increasing values of K. By raising the correction to a power smaller than 1 we mitigate this phenomenon without giving up the appealing properties provided by β.
5. The parameter S_K increases slowly with K. The point is to take into account a plausible overhead for growing dimensions of the kernel, especially when passing from $K = 1$ to $K > 1$. This constant can be possibly understood as a minimum expected spatial dimension for kernels larger than 1. It does not make much sense to apply a kernel of dimension 3×3, on an input of dimension 1×1, and it is hard to believe that reducing the dimension of the input below the dimension of the kernel may result in any speedup. However, fixing $S_K = K$ does not seem to be the right solution.

5.2 Rationale

We now provide a possible explanation for the α-FLOP formula (6). Let us consider a computational task requiring a given amount of work W. Let $\beta \in [0, 1]$ be the fraction of that work that can be executed in parallel; therefore, the



(a) Increasing K, decreasing C (b) Increasing K, decreasing W, H

Fig. 4: Predicted execution time by means of α-FLOPS, depicted as dashed lines, for the same convolutional configurations of Figure 2

sequential portion of the task is $(1-\beta)W$. Let us scale the problem by a factor $N > 1$; in a purely sequential framework, the amount of work would become NW. However, Gustafson's law [16] suggests that when we scale the size of a parallel task, *the sequential part tend to remain the same*. This means that the amount of work *actually* done by the parallel program is $(1-\beta)W + \beta NW$. The ratio between the *actual* amount of work done by the parallel version versus the *expected* amount of work done by the serial version is:

$$\frac{(1-\beta)W + \beta NW}{NW} = \frac{1-\beta}{N} + \beta \tag{7}$$

where we readily recognize the backbone of equation (6). We already discussed the small adjustments we had to do to this formula to fit it to the empirical observations.

Gustafson's law describes the theoretical speedup of a parallel task in terms of growing resources, on the reasonable assumption that programmers tend to set the problem size to fully exploit the available computing power. Gustafson's law was meant to address a shortcoming of a similar law formulated by Amdahl [1], that on the other hand assumes a fixed workload that does not depend on the number of execution units used.

In our case, computational resources are fixed and we focus on different input dimensions. Our assumption is that suitable programs and load balancing techniques will optimize the use of resources, eventually resulting in different speedups along different spatial dimensions.

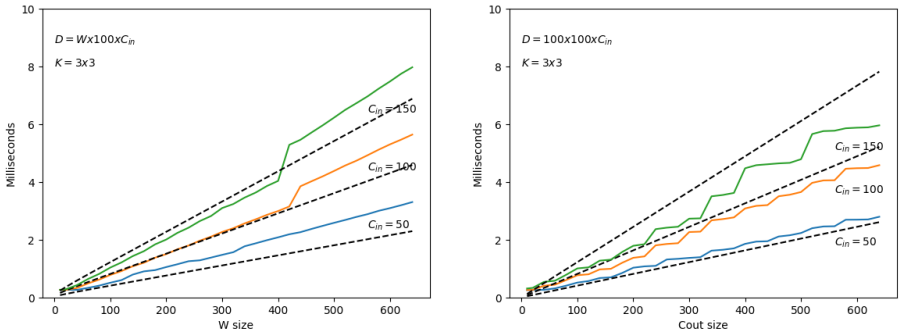
6 Additional experimental results

We conducted several experiments to assess the rate of grow of the execution time along different input dimensions. Data providing the base for this paper are available on Github (`https://github.com/asperti/alpha_flops_dataset`),

together with analysis tools and plotting facilities, including the predictions by means of α-FLOPs. Additional data are currently being collected.

All experiments discussed in this Section involve convolutions where we progressively increase the dimension of a specific input axis x, keeping a constant dimension for the others axes X_c. For each experiment, we draw the execution time for three different dimensions of an auxiliary axis x_{aux} in X_c. We found this more understandable than plotting three dimensional surfaces, that would be quite difficult to draw and decipher.

In the case of the plot in Figure 5a, x is W, and x_{aux} is C_{in}; $H = 100$ and the Kernel dimension is 3×3. For Figure 5b, x is C_{out}, and x_{aux} is C_{in}; $H = W = 100$ and the kernel dimension is 3×3. For Figure 6a, x is C_{out}, and x_{aux} is H; $C_{in} = 50$ and the kernel dimension is 1×1. Finally, for Figure 6b, x is C_{in}, and x_{aux} is K; $H = W = 10$ and $C_{out} = 1000$.



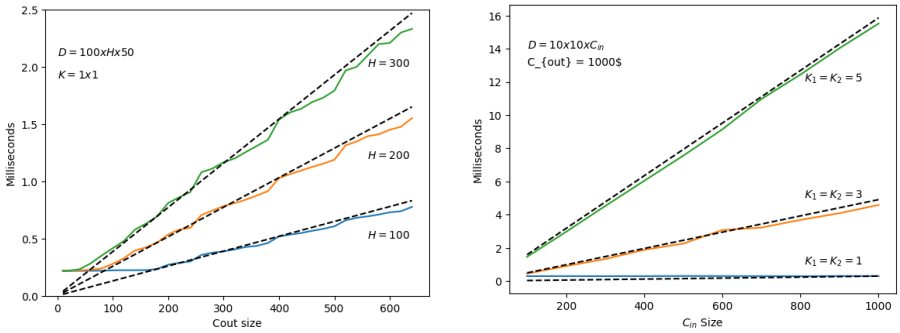
(a) Increasing W for different values of C_{in} and a kernel of dimension 3x3

(b) Increasing C_{out} for different values of $C_{in} and a kernel of dimension 3x3$

Fig. 5: Execution time and predictions by means of α-FLOPs (dashed lines)

6.1 Dense layers vs batchsize

We already observed that a dense layer can be assimilated to a convolutional layer with kernel 1×1 and spatial dimension 1. In this perspective, it is plausible to conjecture that the batchsize can be assimilated to a spatial dimension. Indeed, the general wisdom across Deep Learning researchers and practitioners is that, for making predictions over a large set of data – e.g., over the full training or test set – it is convenient to work with a batchsize as large as possible, compatibly with the resource constraints of the underlying hardware, e.g., memory. This has no justification in terms of FLOPs, since the total number of operations is always the same; however, using a large batchsize is much more efficient.



(a) Increasing C_{out} for different values of H, with a kernel of dimension 1x1

(b) Increasing C_{in} for different kernels 1x1, 3x3 and 5x5

Fig. 6: Execution time and predictions by means of α-FLOPS (dashed lines)

To test this behaviour, we take large dense layers (at the limit of our hardware capacity), and then apply them to inputs with increasing batch size.
The results are summarized in Figure 7. Under the FLOPs assumption, the lines should be straight lines *departing from the origin*. In terms of α-FLOPs we start with the cost relative to batchsize 1, and then slowly grow along the batchsize dimension, reflecting the experimental behaviour.

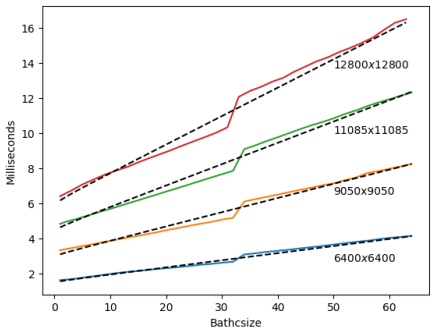


Fig. 7: Computational time for dense layer increasing the batchsize. The jump between 32 and 33 is probably due to some discretization in the software.

7 Conclusions

In this paper we introduced the notion of α-FLOPs that is meant to provide a simple numerical correction to the mismatch between FLOPs and execution time in case of hardware equipped with massively parallel processing units like GPUs or TPUs. Since this kind of hardware is the norm for AI applications based on Deep Neural Networks, α-FLOPS may become an important tool to compare the efficiency of different networks on a given hardware.

The definition of α-FLOPs is based on the crucial observation that, in case of an input with multiple dimensions, the computational speedup offered by parallelism is typically far from uniform along the different axes. In particular,

we provided extensive empirical evidence that growing spatial (and batchsize) dimensions in convolutional layers has less impact than growing different dimensions. The idea of dissecting the cost along the different input dimensions was inspired by recent investigations of the first author on computational complexity over finite types [4].

The notion of α-FLOPs lays between the number of parameters of the layer, and the traditional notion of FLOPs; in a sense, it can be understood as a revaluation of the former as a measure of cost: if it is true that, in the case of convolutions, the number of parameters does not take into account the actual cost of the convolution, the traditional notion of FLOPs seems to largely overestimate it.

Much work is still ahead. On the experimental side, we are currently collecting more data, on architectures with different computing capabilities. On the theoretical side, it would be interesting to provide a low-level algorithmic justification of α-FLOPs. The formula itself, that was derived empirically, can be eventually fine-tuned and possibly improved, both in view of additional observations, and of a better understanding of the phenomenon. In particular, we mostly focused on the spatial dimension, since it is the axis most affected by parallelism, but the dependency along different axes does eventually deserve additional investigation.

In this article, we mostly focused on convolutional and dense layers, since they are the most computationally intensive layers in Neural Networks. Extending the work to additional layers, or more sophisticated forms on convolutions, like Depth-Separable Convolutions, is another major research direction.

References

1. Gene M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, page 483–485, 1967.
2. Lasse F. Wolff Anthony, Benjamin Kanding, and Raghavendra Selvan. Carbontracker: Tracking and predicting the carbon footprint of training deep learning models. *CoRR*, abs/2007.03051, 2020.
3. Arm cortex-r8 mpcore processor, 2018. `https://developer.arm.com/documentation/100400/0002/floating-point-unit-programmers-model/instruction-throughput-and-latency?lang=en`, Last accessed on 2021-04-26.
4. Andrea Asperti. Computational complexity via finite types. *ACM Trans. Comput. Log.*, 16(3):26:1–26:25, 2015.
5. Andrea Asperti, Davide Evangelista, and Elena Loli Piccolomini. A survey on variational autoencoders from a green AI perspective. *SN Comput. Sci.*, 2(4):301, 2021.
6. An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.*, 28(2):135–151, June 2002.
7. Qingqing Cao, Aruna Balasubramanian, and Niranjan Balasubramanian. Towards accurate and reliable energy measurement of NLP models. *CoRR*, abs/2010.05248, 2020.

8. clblas. `http://clmathlibraries.github.io/clBLAS/`. Last accessed on 2021-04-26.
9. AMD corporation. Software optimization guide for amd family 19h processors (pub), November 2020. `https://www.amd.com/system/files/TechDocs/56665.zip`, Last accessed on 2021-04-25.
10. Intel Corporation. Intel® Xeon scalable processor® instruction throughput and latency, August 2017. `https://software.intel.com/content/dam/develop/public/us/en/documents/intel-xeon-scalable-processor-throughput-latency.pdf`, Last accessed on 2021-04-25.
11. cublas. `https://docs.nvidia.com/cuda/cublas/index.html`. Last accessed on 2021-04-26.
12. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.
13. Mohamed Gadou, Tania Banerjee, Meena Arunachalam, and Sanjay Ranka. Multiobjective evaluation and optimization of cmt-bone on multiple cpu/gpu systems. *Sustainable Computing: Informatics and Systems*, 22:259–271, 2019.
14. Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 1586–1595. IEEE Computer Society, 2018.
15. Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S. Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. Chasing carbon: The elusive environmental footprint of computing. *CoRR*, abs/2011.02839, 2020.
16. John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 1988.
17. Yunho Jeon and Junmo Kim. Constructing fast network through deconstruction of convolution. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 5955–5965, 2018.
18. Chetan Jhurani and Paul Mullowney. A gemm interface and implementation on nvidia gpus for multiple small matrices. *Journal of Parallel and Distributed Computing*, 75:133–140, 2015.
19. Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.
20. Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning. *CoRR*, abs/1910.09700, 2019.
21. Sean MacAvaney, Franco Maria Nardini, Raffaele Perego, Nicola Tonellotto, Nazli Goharian, and Ophir Frieder. Efficient document re-ranking for transformers by precomputing term representations. In *Proceedings of the 43rd International ACM*

SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020, pages 49–58. ACM, 2020.

22. Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.
23. David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2017.
24. Rance Rodrigues, Arunachalam Annamalai, Israel Koren, and Sandip Kundu. A study on the use of performance counters to estimate power in microprocessors. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 60(12):882–886, 2013.
25. Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green AI. *Commun. ACM*, 63(12):54–63, 2020.
26. Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for modern deep learning research. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 13693–13696. AAAI Press, 2020.
27. Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 2019.
28. Aimee van Wynsberghe. Sustainable ai: Ai for sustainability and the sustainability of ai. *AI and Ethics*, 2021.
29. Tom Veniat and Ludovic Denoyer. Learning time/memory-efficient deep architectures with budgeted super networks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 3492–3500. IEEE Computer Society, 2018.