From the algorithm to the targets, optimization flow for high performance computing on embedded GPUs

Mickaël Seznec

université
PARIS-SACLAY

Thèse de doctorat

NNT: 2021UPASG074

From the algorithm to the targets, optimization flow for high performance computing on embedded GPUs

De l'algorithme à l'implémentation, flot d'optimisations pour le calcul haute performance sur GPU embarqués

Thèse de doctorat de l'Université Paris-Saclay

École doctorale n° 580, sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat: Traitement du signal et des images
Unité de recherche: Université Paris-Saclay, CNRS, CentraleSupélec, Laboratoire des signaux et systèmes, 91190, Gif-sur-Yvette, France
Référent: faculté des sciences d'Orsay

Thèse présentée et soutenue à Paris-Saclay, le 25/10/2021, par

Mickaël SEZNEC

Composition du jury :

Frédéric CHAMPAGNAT Ingénieur de recherche, ONERA	Président
Michael KRAJECKI Professeur, Université Reims Champagne-Ardenne	Rapporteur, Examinateur
Cristina SILVANO Professeure, Politecnico di Milano	Rapportrice, Examinatrice
Julien DEMOUTH Ingénieur de recherche, NVIDIA	Examinateur
Jean-François NEZAN Professeur, INSA Rennes	Examinateur

Direction de la thèse :

Nicolas GAC Maître de conférences, Université Paris-Saclay	Directeur
François ORIEUX Maître de conférences, Université Paris-Saclay	Co-encadrant
Alvin Sashala NAIK Ingénieur de recherche, Thales Research & Technology	Co-encadrant

Abstract

Current digital processing algorithms require more computing power to achieve more accurate results and process larger data. In the meantime, hardware architectures are becoming more specialized, with highly efficient accelerators designed for specific tasks. In this context, the path of deployment from the algorithm to the implementation becomes increasingly complex. It is, therefore, crucial to determine how algorithms can be modified to take advantage of new hardware capabilities. Our study focused on graphics processing units (GPUs), a massively parallel processor. Our algorithmic work was done in the context of radio-astronomy or optical flow estimation and consisted of finding the best adaptation of the software to the hardware. At the level of a mathematical operator, we modified the traditional image convolution algorithm to use the matrix units and showed that its performance doubles for large convolution kernels. At a broader method level, we evaluated linear solvers for the combined local-global optical flow to find the most suitable one on GPU. With additional optimizations, such as iteration fusion or memory buffer re-utilization, the method is twice as fast as the initial implementation, running at 60 frames per second on an embedded platform (30 W). Finally, we also pointed out the interest of this hardware-aware algorithm design method in the context of deep neural networks. For that, we showed the hybridization of a convolutional neural network for optical flow estimation with a pre-trained image classification network, MobileNet, that was initially designed for efficient image classification on low-power platforms.

Contents

List of Figures

List of Tables

Introduction

The manuscript you are about to read results from a joint research effort between the Laboratoire des Signaux et Systèmes (L2S) and Thales Research & Technology (TRT).

As an innovative company, Thales always strives for fast, accurate, and robust algorithmic solutions to various data processing problems. This goal is achieved through the combination of the right software with the right hardware. Even if both of them can be chosen or designed separately, splitting up the process may cause the implementation of an algorithm to perform poorly on its execution platform.

In a context where various hardware architectures are available off-the-shelf, it is crucial to identify the main advantages each offers. These features introduce new trade-offs during the design of the algorithm. Knowing which programs or operations are favored at execution time makes it possible to conceive methods that use the hardware to its full potential.

This adaption of the software to the hardware is all the more critical as new innovative computer architectures bring promises of low-power, high-performance computations at the cost of specialization in the programs they execute, as shown on fig. 1. To push performance forward and stay on the leading edge of data processing systems, a company like Thales must be able to adapt its algorithms to leverage the power offered by new accelerators.

In this manuscript, I will focus on two questions: what are the opportunities for adapting to specific hardware architectures offered by algorithms? What are the impacts of specialization of the algorithm in terms of performance and quality of the result?

My work during this Ph.D. program was to identify potential performance gain in several computer vision algorithms: radio-astronomy image reconstruction, optical flow estimation, or image convolution. With these use-cases, I aimed at understanding what portion of the algorithm could be modified to become a better fit for a specific type of processor, the GPU. The proportion of the optimized code ranges from a single operator to a whole numerical method, but in all cases, the goal was to conserve similar results with better performance.

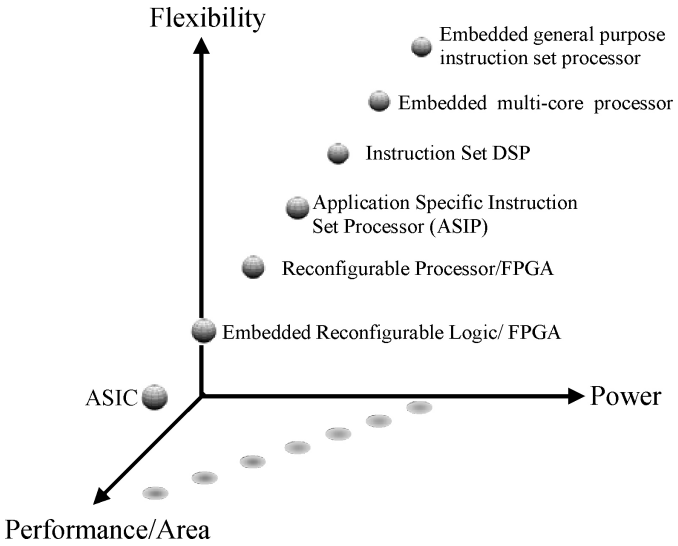


Figure 1: Representation of several hardware platforms in terms of flexibility, performance and power. From (Lee et al., 2009).

This manuscript will guide you through the different aspects of my research effort and is structured as follows:

The first chapter exposes the different aspects that govern the development of an algorithm. It provides an overview of the diverse hardware architectures available on the market and the reasons for this heterogeneity. Then, based on the optical flow estimation problem, it provides an overview of the main features of a computer vision algorithm and the challenges faced for efficient execution. This chapter ends by explaining the current methodology used for deploying such algorithms to the hardware platforms.

In chapter 2, I explore the replacement of operators of a larger algorithm with a GPU-dedicated version. The study begins in the context of radio-astronomy, where an observation of the sky is the source of an inverse problem. The initial image is sought to be restored to remove alterations from the capturing instruments. This digital image processing is limited by the time to perform image convolutions. Several methods for computing this operation on GPU are explored and compared, focusing on numerical precision. Then, we move to a broader context and present a new algorithm that computes 2D convolutions efficiently by relying on matrix-multiplication units of the GPU.

Chapter 3 extends the scope of the optimization process. In the context of optical flow, a method, Combined Local-Global, was selected at Thales for implementation on GPU. My analysis begins at the method level to choose a linear equations solver that fits GPUs' characteristics. The second step dives into

operation-level optimizations to obtain maximum performance. The optical flow algorithm can run in real-time on an embedded GPU with this combination of solver choice and hardware-aware optimizations.

Chapter 4 focuses on deep convolutional neural networks. This type of algorithm has become a predominant technique to solve many compute vision tasks. In this section, we review different network architectures designed for estimating the optical flow. Just like any other type of workload, DNNs may be tailored for a particular architecture. We then experiment with hybrid networks to benefit from previous work from the literature and adapt it to real-time, low-power devices.

This manuscript contains chapters based on research articles, either published or under-review in several international journals or conferences. The verbatims of these articles are directly included in this document. In order of appearance, these publications are:

- Seznec, M., Gac, N., Ferrari, A., & Orieux, F. (2018, October). A Study on Convolution using Half-Precision Floating-Point Numbers on GPU for Radio Astronomy Deconvolution, In *2018 IEEE International Workshop on Signal Processing Systems (SiPS)*. 2018 IEEE International Workshop on Signal Processing Systems (SiPS), Cape Town, IEEE. https://doi.org/10.1109/SiPS.2018.8598342

- Seznec, M. Gac, N. Orieux, F. & Naik, A. S. The Im2Tensor Algorithm for Efficient 2D Convolutions on GPU Tensor Cores. Journal Article. (Under review by the Journal of Real-Time Image Processing)

- Seznec, M. Gac, N. Orieux, F. & Naik, A. S. Real-Time Optical Flow Processing on Embedded GPU: an Hardware-Aware Algorithm to Implementation Strategy. Journal Article. (Under review by the SIAM Journal on Scientific Computing)

Chapter 1

Efficient deployment for high-performance architectures

The development of a software solution is a long and challenging process. The industry often divides the task between several teams with specific areas of expertise to manage the complexity of the process. We can distinguish two main groups of skills: algorithm design and implementation. The former develops software solutions, algorithms to answer a given problem. It usually uses a high-level environment, such as Python or Matlab, on a desktop or server computer and aims at validating a specification in terms of the accuracy of the solution. The second group intervenes afterward once the algorithm is developed. It implements the solution on the desired hardware target, often with size, weight, and power (SWaP) constraints. This team employs low-level tools, such a C, VHDL, or CUDA, to interact closely with the hardware and take the best of its performance.

Figure 1.1 explains this traditional way of splitting the concerns between algorithm and implementation. The main issue with this development scheme is the apparition of late implementation failures. Sometimes, the solution designed by the algorithm team cannot attain desired performance on the final hardware. This impossibility may be caused by a too complicated algorithm or a mismatch between the software characteristics and what can be executed on the hardware platform.

Let us give an example with the resolution of a linear equations system. Many solvers have been developed to solve this task. Gauss-Seidel is one of them; it is quite a performant algorithm but sequential. Conversely, the Jacobi solver converges more slowly but is parallel (Saad, 2003). This example is detailed later in the manuscript and shows that in a single CPU setting, Gauss-Seidel would outperform Jacobi, but this hierarchy is inversed on a GPU. When the implementation team understands that it will never implement the Gauss-Seidel solver as

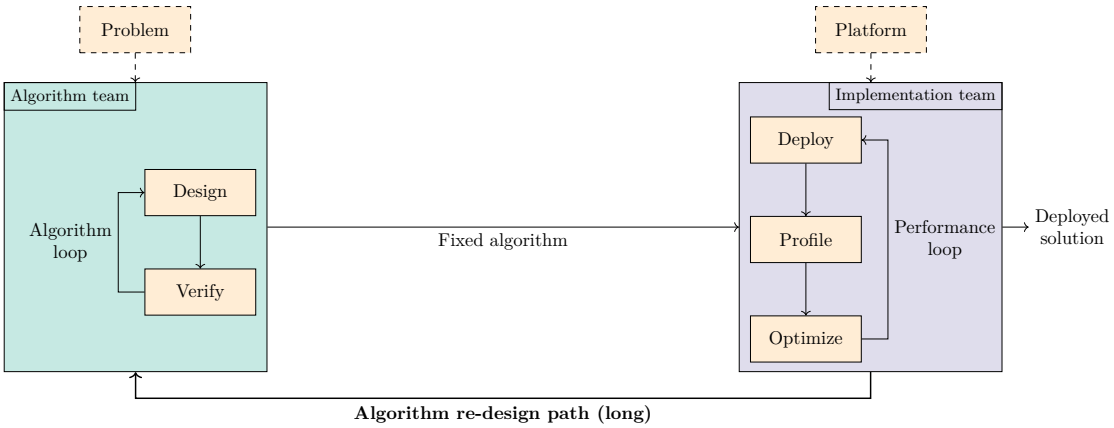


Figure 1.1: A tradionnal development path. The algorithm design team creates an algorithm that solves a given problem. The implementation team then tries to implement the solution on the target hardware.

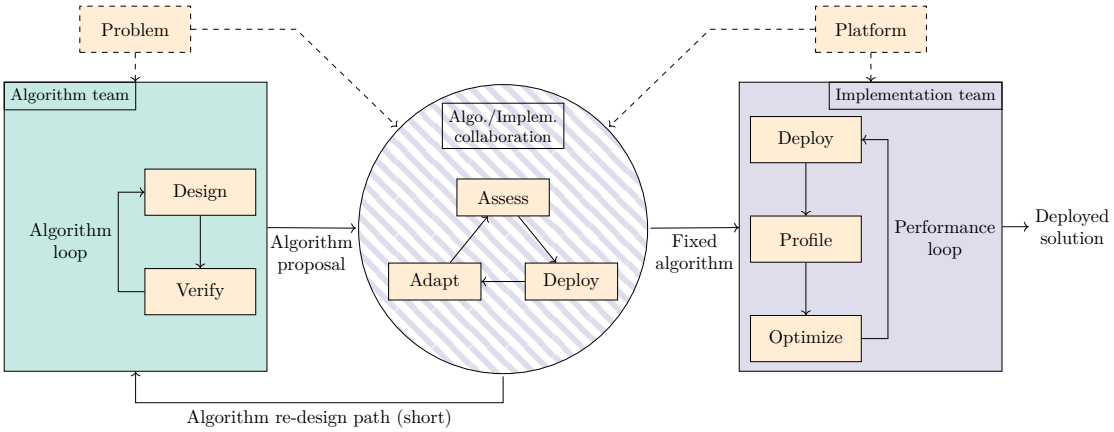


Figure 1.2: Addition of a collaborative phase between algorithm and implementation teams compared to fig. 1.1. This hardware-in-the-loop scheme finds critical inadequations between software and hardware early and permits faster algorithm re-designs. It also enables new types of optimizations that require mutual understanding of the software and the hardware.

efficiently as required, the solution must pass through the hands of the algorithm team once again, which wastes precious time.

On fig. 1.2, we look into another type of process for algorithm deployment. This scheme proposes an intermediate work between algorithm development and implementation. It requires expertise from both domains to understand whether a solution is feasible. In this scenario, the algorithm team still performs preliminary tests to design an algorithm, which we call phase one. The output of phase one is not fixed. During the algorithm/implementation stage, or phase two, the algorithm is adapted to the hardware as far as possible. This second phase would be responsible for changing the linear solver from Gauss-Seidel to Jacobi to come back to our example. The third phase can now extend the deployment with additional optimizations, thanks to the know-how of the implementation team.

The additional second phase is an adapter between the far-apart domains of algorithm design and platform optimization. It should work hand-in-hand with both teams to alleviate inadequations between the solution and the platform and allows more serene work for the implementation team. Non-feasibilities are also detected earlier than in the traditional process of fig. 1.1, thanks to having the hardware in the loop sooner. Minor issues can even be fixed directly in the second stage with combined knowledge in software and hardware.

This chapter details the stakes of acknowledging the hardware architecture early in a solution's design. In section 1.1, we present some aspects of algorithms the software must deal with, with a focus on two image processing methods, 2D convolution and optical flow estimation. They serve at introducing algorithmic concepts such as complexity and arithmetic intensity. Section 1.2 explains the reasons behind the current split of platform architectures between the two different development worlds, algorithm design, and deployment. The last section presents the implementation and optimization side of the deployment. It focuses on the tools needed for deployment and how to profile and efficiently optimize.

1.1 Examples of algorithm design for image processing

As presented in figs. 1.1 and 1.2, an algorithm team is responsible for the construction of a solution that answers a problem with a set of specifications. This section details two algorithms, image convolution, and optical flow estimation, used in different industrial contexts. They allow us to understand the characteristics of a software solution and the leeway available during the design of these algorithms.

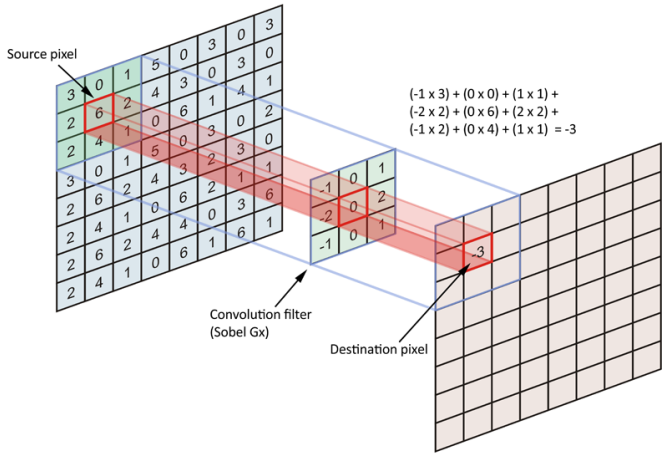


Figure 1.3: Operations required to compute one pixel of a convolution. Source: Tim Harley CC BY-NC 4.0.

1.1.1 Image convolution

The 2D-convolution is a fundamental tool for processing images. We use it here to introduce several aspects of computer algorithms that are useful for the comprehension of this manuscript. A digital image I is made of a collection of pixels $\{I[x,\, y],\, 0 \leq x < I_w \;\text{ and }\; 0 \leq y < I_h\}$. I_h and I_w are the height and the width of the image, respectively. $I[x, y]$ is a scalar value corresponding to the intensity of the pixel at coordinates $(x,\, y)$.

The convolution between two images I and K is noted $I * K$ and

$$(I * K)[i, j] = \sum_{x=0}^{K_w} \sum_{y=0}^{K_h} I[i - \lfloor \frac{K_w}{2} \rfloor + x,\, j - \lfloor \frac{K_h}{2} \rfloor + y] \cdot K[x, y]. \tag{1.1}$$

K is called the convolution filter, or kernel, with dimensions K_h, K_w.

At coordinates $(i,\, j)$, we can think of K being super-imposed over I, as in fig. 1.3. The convolution is then a sum of all kernel coefficients multiplied by the image coefficient they overlap. These operations must then be repeated for each destination pixel. For now, we do not account for border conditions.

In reality, eq. (1.1) expresses the cross-correlation between I and K. It is for simplicity called a convolution. In the proper sense, the correlation is the cross-correlation of I and K^T, the transposition of K.

Time complexity

The first characteristic we want to know about this operation is its complexity. How many operations must we do to compute a convolution? This metric is called

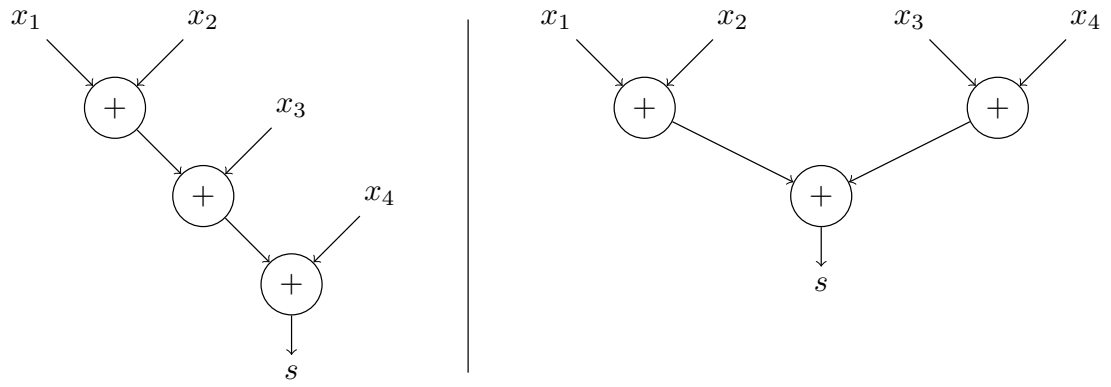


Figure 1.4: The sum of four numbers computed serially (left) or in parallel (right).

the algorithmic complexity or time complexity. It is often expressed using the big $\mathcal{O}$ notation to show predominant terms only. In this example, to compute a single destination pixel, the complexity is $\mathcal{O}(K_w K_h)$ because of the two sum loops. Since we need to re-iterate this operation on all output pixels, the final time complexity is

$$T = \mathcal{O}(I_w I_h K_w K_h). \tag{1.2}$$

The algorithmic complexity is a valuable tool to assess algorithms. It does not account, however, for independent operations. For example, in the case of convolution, all pixels of the output can be produced in parallel, as there are no dependencies between them. To measure this, we can use the step complexity S.

Step complexity

Let us introduce S with a simple example. Figure 1.4 shows the sum of (x_1, x_2, x_3, x_4) with two different methods. The first iterates over all variables in order to compute $(((x_1 + x_2) + x_3) + x_4)$. The second relies on the associativity of the sum to compute $((x_1+x_2)+(x_3+x_4))$. Computing this four-sum has time complexity T equals 3, but in one case, the depth of the computation graph is three, while in the other, it is only two. The depth of the graph is what we call the step complexity, S. A sequential machine computes the graph in a time proportional to T, while parallel architectures leverage the independent operations to compute the graph proportionaly to S.

In the case of 2D convolutions, since all pixels can be computed in parallel, the step complexity of the convolution is the step complexity of the operations required for one pixel, as in eq. (1.1). Because the step complexity of a sum of n elements is $\mathcal{O}(\log(n))$, we finally have $S = \mathcal{O}(\log(K_w K_h))$.

In an ideal scenario, the time to compute a convolution on a parallel computer

is then proportional to S, i.e., $\log(K_w K_h)$. While a sequential machine would do it in $\mathcal{O}(I_w I_h K_w K_h)$. This large gap justifies the need to find formulations of algorithms that expose parallelism and to have adequate machines to run them efficiently.

Arithmetic intensity

So far, we have analyzed algorithms in terms of arithmetic operations. Another essential component that dictates the runtime of a program is memory access. In fact, over the last decades, memory bandwidth has not progressed as fast as the number and efficiency of arithmetic units (Hennessy & Patterson, 2011). As a result, recent hardware architectures can usually perform many more operations than get data from memory in the same amount of time. The arithmetic intensity then measures the ratio of operations required with respect to the number of bytes of data needed

$$\text{AI} = \frac{\text{number of operations}}{\text{bytes moved}}. \tag{1.3}$$

In the case of the convolution, for one pixel, we need to move pixels from the kernel and a sub-image, accounting for $\mathcal{O}(K_w K_h)$ movements. In the meantime, the number of operations is $\mathcal{O}(K_w K_h)$, the arithmetic intensity is then $\text{AI} = \mathcal{O}(1)$. It is relatively low compared to other algorithms (see fig. 1.5, convolution being a type of stencil), but implementation techniques can improve this number. For example, re-using coefficients by storing them in cache enhances the overall AI.

Knowing the arithmetic intensity is critical to understanding how an algorithm will perform on a given architecture. As we will see in section 1.3.2, the execution is often limited by a bottleneck, memory, or arithmetic. Evaluating the limiting factor is essential for efficient optimization.

Fourier transforms

Algorithms can sometimes be seen in different forms. For example, we showed that the sum of a set of variables could be a sequential or parallel procedure. Similarly, convolution can be viewed as a multiplication in the Fourier domain. This technique, referred to as the convolution theorem, offers opportunities for faster execution of the operation.

The convolution theorem states that the convolution of I by K equals the inverse Fourier transform $\mathcal{F}^{-1}$ of the point-wise product, $\odot$, of the Fourier transforms $\mathcal{F}(I)$ and $\mathcal{F}(K)$

$$I * K = \mathcal{F}^{-1}(\mathcal{F}(I) \odot \mathcal{F}(K)). \tag{1.4}$$

This form is attractive as there exists fast algorithms to compute a Fourier transform as well as its inverse (Nussbaumer, 1981), in $\mathcal{O}(n \log(n))$. In fact, the

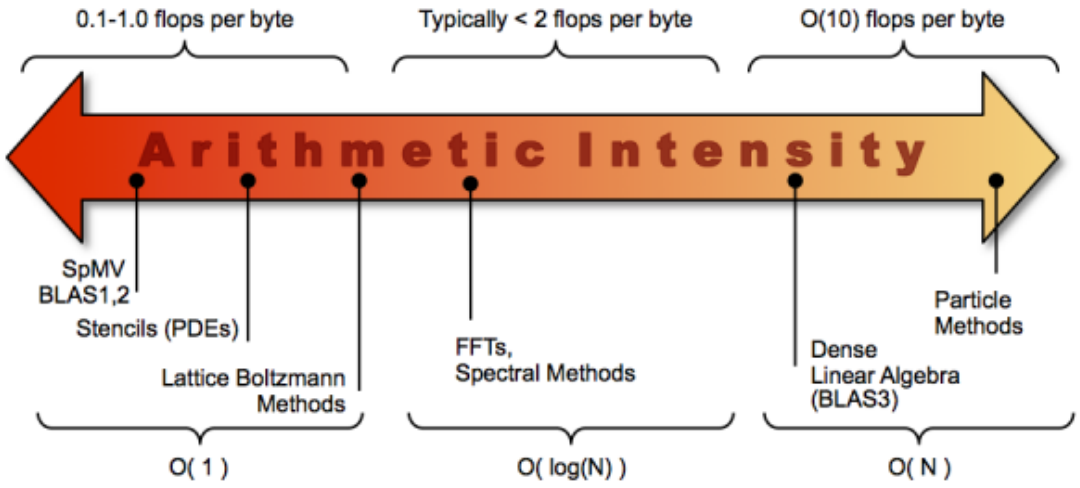


Figure 1.5: The arithmetic intensity of various algorithms. Source: Lawrence Berkeley National Laboratory.

time complexity of convolution in this form is

$$T_{\text{Fourier}} = \mathcal{O}(I_h I_w \log(I_h I_w)) \tag{1.5}$$

Compared to eq. (1.2), we see that if the kernel is large enough, so that $K_h K_w > \log(I_h I_w)$, the Fourier version becomes faster.

In the end, for an operation as essential as the image convolution, multiple ways exist for computing it. Depending on the hardware that executes the algorithm, one convolution formulation might be more performant than another. The step complexity is a crucial metric on parallel hardware, while a sequential platform is more sensitive to algorithmic complexity. Furthermore, the arithmetic intensity plays a significant role in the limitations of the algorithm's runtime, whether it is memory or operations limited.

Choosing the right way to express an operation is essential to find the best compromise regarding arithmetic intensity, algorithmic, and step complexity. Chapter 2 gives further details about actual implementation of image convolutions on GPUs.

1.1.2 Optical flow estimation

The previous section presented a fundamental operation of image processing, the 2D-convolution. On its own, this function does not produce very significant results. It is still the backbone of many computer vision techniques. Optical flow estimation

is one of them. This task consists of finding the displacement of pixels from one image to another, usually in a video stream. Figure 1.6 shows the displacement of three pixels. Images 1 & 2 are known, and optical flow estimation aims to find the displacement vectors.

Applications

Optical flow is an essential building block of more advanced computer vision applications. Super-resolution, for example, relies on optical flow to register moving pixels in a video sequence. The aggregation of sources of information from several frames increases the accuracy of the image (Baker & Kanade, 1999). Particle image velocimetry is another example that leverages optical flow to estimate the movement of particles in a moving fluid (Ruhnau et al., 2005).

In object tracking, the optical flow serves to estimate the motion of the tracked object. It is a valuable tool to update the position of the object in the next frames (Smeulders et al., 2014). Finally, the optical flow may be used to estimate stereo disparity with binocular imaging. Knowing the displacement of one object between the two cameras may then lead to depth estimation (Beauchemin & Barron, 1995).

Databases and accuracy

Several benchmarks have emerged to offer a comparison baseline for optical flow estimation. The first one, known as the *Middlebury* database, features a dozen of sequences, real and synthetic (Baker et al., 2011). For computer-generated images, it is usually simple to have access to the optical flow. These 3D scenes provide all the information required, with the camera and objects' movements. For real-life captures, the flow was re-constructed by tracking fluorescent dot patterns visible under UV illumination on objects.

The second broadly used database is the MPI (Max Planck Institute) Sintel (Butler et al., 2012). It has been constructed from an open-sourced animation

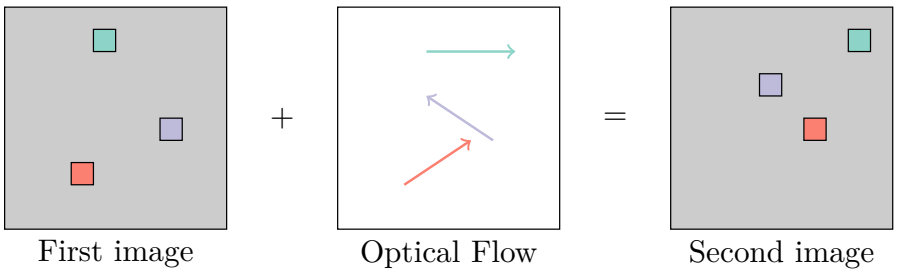


Figure 1.6: Optical flow example on three moving pixels.

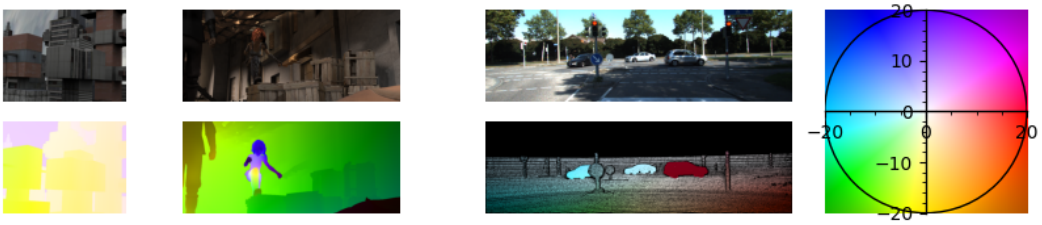


Figure 1.7: From left to right, *Middlebury*, Sintel, KITTI, and a calibration pattern. The frame is on the top, and its corresponding flow on the bottom.

movie. The database is entirely synthetic but orders of magnitude larger than *Middlebury*. It also features longer sequences, several levels of details, and optical effects such as motion blur.

Finally, the KITTI benchmark is a suite of datasets aimed at several computer vision tasks for autonomous driving: optical flow, odometry, tracking, for example. It has had two major releases in 2012 and 2015 (Geiger et al., 2012; Menze & Geiger, 2015). This time, the data originates from real-life captures. The optical flow was derived from a laser scanner mounted on top of the car that generated 3D point clouds.

Figure 1.7 shows examples from the three databases. The flow is represented in color. The hue indicates the flow direction, and the color intensity gives the norm of the vector.

To compare algorithms with these databases, the end-point error (EPE) is often used. It measures the norm between the estimated flow and the reference. The AEPE (Average EPE) summarizes this metric over all vectors by taking the mean

$$\text{AEPE}(\bar{\boldsymbol{w}}) = \frac{1}{HW} \sum_{x,y} \|\boldsymbol{w}_{x,y} - \boldsymbol{w}^*_{x,y}\|. \tag{1.6}$$

With $\boldsymbol{w}_{x,y}$ the estimated flow on coordinates (x, y) and $\boldsymbol{w}^*_{x,y}$ the reference flow. The goal is then to find an algorithm that minimizes this metric on the images of a database.

Variational methods

Algorithms made for optical flow estimation exist since the 1980s, with pioneer work from (Lucas & Kanade, 1981) and (Horn & Schunck, 1981). They have paved the way for two families of results: sparse and dense. Sparse methods only generate displacement vectors for key pixels of the image, usually where the confidence in the result is best. Conversely, dense optical flow estimation provides

a displacement for every pixel. If required, techniques exist to "densify" a sparse flow, see for example (Leordeanu et al., 2013).

In this section, we explain the basics of variational optical flow. This introduction gives an overview of the work we need to deploy to estimate optical flow. The mathematical notations are as follows: a lowercase $a \in \mathbb{R}$ is a coefficient, while $\boldsymbol{a} \in \mathbb{R}^n$ is a vector. $\bar{a}$ is a field, indexed with parenthesis: $\bar{a}(x, y, t)$, for example. $\bar{\boldsymbol{a}}$, is a vector field.

Many methods of estimation relies the illumination constancy assumption. It assumes that a pixel's value does not change between two frames

$$\bar{f}(x + u_{x,y},\, y + v_{x,y},\, t + 1) = \bar{f}(x,\, y,\, t) \tag{1.7}$$

$$\boldsymbol{w}_{x,\,y} = (u_{x,y},\, v_{x,y})^T. \tag{1.8}$$

Where $\bar{f}$ represents a video stream, the first two dimensions are spatial, and the third one is temporal. u and v are the horizontal and vertical components of the flow.

This equation has two unknowns: u and v. It is not possible to solve it as is. A remedy is to add constraints to the flow in the form of a penalization term. Our solution should then both solve eq. (1.7) while respecting constraints, such as being smooth. This can be summarized as an energy function minimization

$$E(\bar{\boldsymbol{w}}) = \int_{x,y} D(\bar{f},\, \boldsymbol{w}_{x,y},\, x,\, y) + R(\boldsymbol{w}_{x,y},\, x,\, y)\,\mathrm{d}x\,\mathrm{d}y. \tag{1.9}$$

For example, (Horn & Schunck, 1981) set

$$D_{\mathrm{HS}}(\bar{f},\, \boldsymbol{w}_{x,y},\, x,\, y) = \|f(x + u_{x,y},\, y + v_{x,y},\, t + 1) - f(x,\, y,\, t)\|^2 \tag{1.10}$$

$$R_{\mathrm{HS}}(\boldsymbol{w}_{x,y},\, x,\, y) = \|\nabla_{x,y}\boldsymbol{w}_{x,y}\|^2 \tag{1.11}$$

Algorithmic techniques and challenges

Solving eq. (1.9) is expensive. After discretization, it means finding the solution to a system with as many equations as they are pixels in the image. Furthermore, new and more accurate optical flow algorithms add complexity to the model, increasing the cost of computing a solution.

For example, (Farnebäck, 2003) uses a quadratic regularization instead of the usual linearization of eq. (1.10). (Brox et al., 2004) add a gradient conservation term to the intensity convervation, (Zach et al., 2007) use the l_1 norm instead of l_2. (Bruhn et al., 2005) adjoin a neighbor condition to the data term of eq. (1.10).

Another challenge for optical flow algorithms is the handling of large displacements. Usually, the presented method only finds movements of a few pixels maximum. To overcome this limitation, a multi-scale strategy is employed, as shown

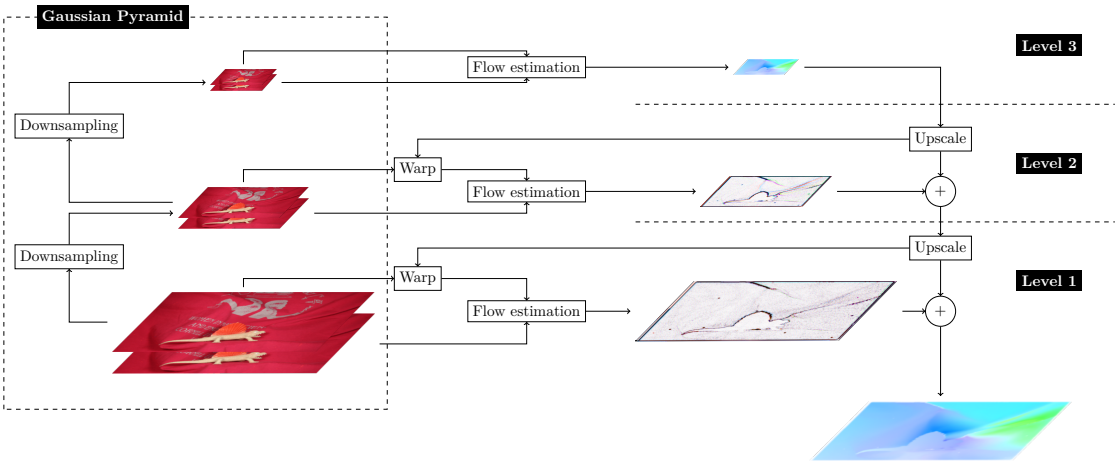


Figure 1.8: Diagram of a multi-scale optical flow estimation. First, input images generate a downsampled Gaussian pyramid. A first estimation is made at the top level and is used to warp the second image one scale below. From there, an estimation of the residual flow is done again and the procedure continues until reaching the first level.

on fig. 1.8. The idea is to process the image at lower resolutions. The solution is then used to "warp" the image. This operation moves the pixels according to the optical flow. After warping, the two images should almost match each other. We can then re-apply the optical flow algorithm to find any residual displacement (Le Besnerais & Champagnat, 2005).

This pyramidal framework is used in other areas of image processing, for example, denoising (Lebrun et al., 2014) or segmentation (Sharon et al., 2000). The benefits of scale change are twofold: comprehension of the image at another level, to find large displacement, for example, and faster processing times at higher scales. Even though smaller image sizes require less computation, they also offer less parallelism. On GPUs, the speed-up of higher levels might be compromised by this lack of parallel work, as we will see in 3.1.

1.2 High-performance architectures

This section introduces the current landscape of computer architectures to understand the gap between environments used for algorithm development and efficient deployment. Software research and development are often conducted on traditional CPU architectures, but this type of hardware has had difficulty increasing its performance in recent years. This stagnation has led to the emergence of competing architectural designs. Different approaches are currently used for reaching

high-performance with embedded requirements. This section reviews some of them and focuses on GPUs, predominantly used for our work.

1.2.1 Context and current hardware designs

Throughout the 20th century, computers' performance has seen an unprecedented expansion. The origins of this improvement are multiple. First, the size of transistors crammed onto integrated circuits has fallen and allowed doubling their number per chip every two years, according to Moore's law (Moore, 1965). Other factors such as increasing manufacturable die size and improving manufacturing processes with fewer defects supported this growth.

In addition to hardware size, several other aspects contributed to more powerful processors. A boost in the clock frequency naturally resulted in faster CPUs. Improved hardware designs with "smarter" architectures also helped in the efficiency of computers by exploiting ILP (Instruction-Level Parallelism) (Hwu & Patt, 1986), with deep pipelining and branch prediction (A. Seznec et al., 2002), for example. In the meantime, data throughput and access latency have improved with better memories and cache-based architectures.

This pace of innovation has, however, slowed down in the last decades. Figure 1.9 shows an inflection in performance happening in the mid-'00s. Current development is now facing several "walls" (Sutter et al., 2005). The power wall, for example, is a capping of the chips' energy consumption. It is caused by a large amount of heat dissipated by transistors. Cooling systems have reached an efficiency limit, and more power dissipation would lead to excessive temperatures. The power wall is, in turn, a cause for the frequency wall, as faster clock speeds would over-heat circuits. Finally, the memory wall is the delay in the improvement of memories compared to processors. From 1980 to 2010, the typical CPU performance has been multiplied by 10,000 while RAM latency access has only improved by a factor of 10 (Hennessy & Patterson, 2011; Wulf & McKee, 1995).

Figure 1.10 shows the power and frequency growth stop of typical computers. The number of transistors per chip is, however, still currently rising. To leverage this hardware availability, the number of processors per chip began to grow around 2005. Multi-core CPUs are, in fact, one form of exploitation of programs' concurrency that has become crucial in modern computers.

Making full use of a large number of available transistors is a central aspect of computer design. For that, exploiting the parallelism of programs is a must. Let us now take a look at several examples of modern design that leverage parallel architectures for high-performance.

Vector architectures Doing the same operation on different data is a simple way to take advantage of parallelism. This type of architecture is classified as

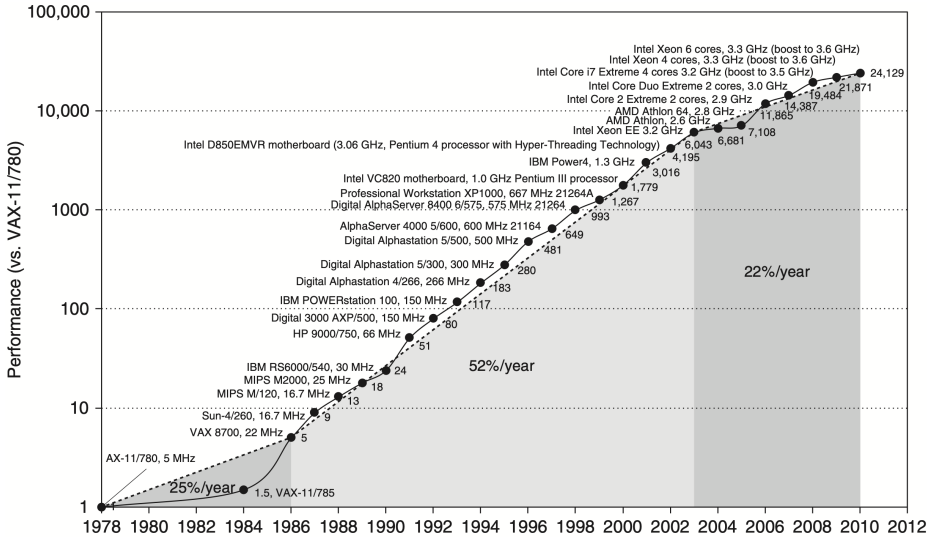


Figure 1.9: Performance of CPUs as evaluated by the SPEC benchmarks (log scale). Over the span of 20 years since 1986, the performance has increased by 52% per year. Afterwards, this development has slowed down. From (Hennessy & Patterson, 2011).

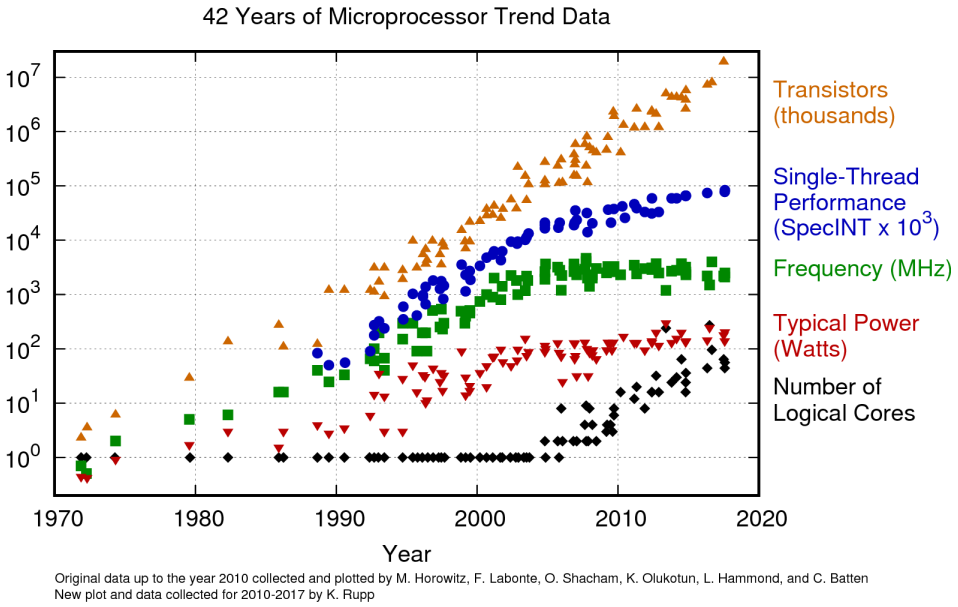


Figure 1.10: Plot of computer characteristics since 1970 (log scale). While the frequency, power and single-thread performance are stalling since the 2010's, the numbers of transistors and logical cores continue to grow. From (Rupp, 2015).

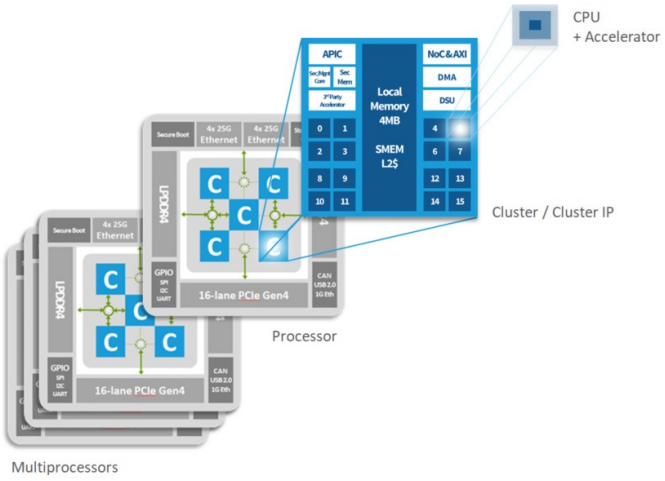


Figure 1.11: Overview of Kalray's MPPA Coolidge manycore architecture. One full processor is made of five compute clusters, each of which features sixteen CPUs. Credits: Kalray.

SIMD (Single Instruction, Multiple Data) according to Flynn's taxonomy (Flynn, 1966).

Many recent CPUs feature such vector instructions: Intel's SSE and AVX, ARM NEON or RISC-V's "P" standard extensions (Lomont, 2011; Reddy, 2008; Waterman et al., 2011). The idea is to provide an extended-length register to gather pieces of data. Then, a single instruction executes the same operation on all registers. This kind of design reaches high performance without deviating too much from the usual scalar CPU architecture. Compilers may automatically transform code that was written for SISD (Single-Instruction Single-Data) machines to leverage the vector units (Maleki et al., 2011).

The simplicity and efficiency of this architecture make it attractive for power-constrained embedded systems (Moloney et al., 2014; Petreto et al., 2018).

Manycore processors This kind of design is an extension of multi-core computers. Instead of replicating a more traditional high-performance CPU core, many small and efficient processing units are grouped to form a processor. Multi-core architectures usually feature up to 8 or 16 independent high-performance cores, while many-core designs push the design further with dozens of simpler cores. The challenge for this type of design lies in the efficiency of inter-core communication and the programmability of the hardware.

Figure 1.11 presents an overview of an MPPA (Massively-Parallel Processor Array) architecture from Kalray. The synchronization and communication between cores are handled by a NoC (Network on Chip), which is an efficient type of

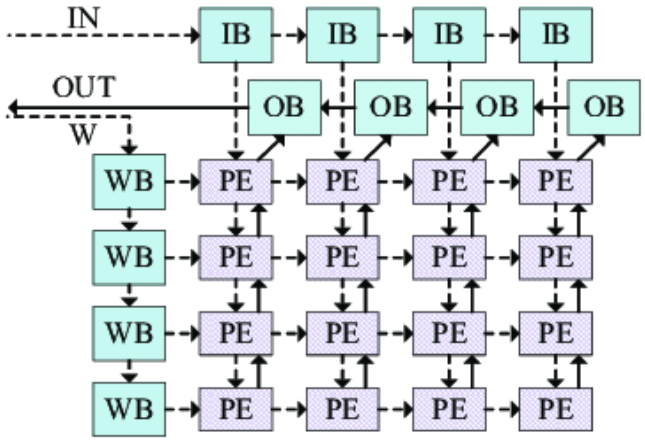


Figure 1.12: A systolic array for computing. The communication grid between PEs (Processing Elements) is well-suited for linear algebra operations. From (Wei et al., 2017).

memory design for embedded chips (Monchiero et al., 2006). It is programable via a dedicated language or through the more generic OpenCL and achieves high performance-per-Watt (de Dinechin et al., 2013).

Dedicated architectures Sometimes, the specificities of a domain of application require special hardware to perform dedicated operations. In signal processing, for example, Fourier transforms are found everywhere, so many DSPs (Digital Signal Processors) feature dedicated hardware to speed up this operation.

Recent architectures now include custom accelerators for operations found in artificial intelligence and deep learning algorithms. For example, a systolic array serves for computing matrix operations. Figure 1.12 shows an FPGA design where many small processors (PEs) collaborate through an optimized data path to compute linear algebra arithmetic operations.

NVIDIA's NVDLA is another example of deep-learning acceleration, focussed on convolutional neural networks. Is it specifically designed to handle the different steps of processing the image with convolution, activation, and pooling (Zhou et al., 2018). The specialization of the architecture allows an excellent energy efficiency for the inference of neural networks (Farshchi et al., 2019).

In the end, these types of architecture continue the growth of computers' compute power. Figure 1.13 shows that in all conditions, with high or low power requirements, innovative designs attain high performance-per-Watt. It is then crucial to count on them to reach the best performance for industrial applications. This new attractiveness is the source of a widening gap between the development

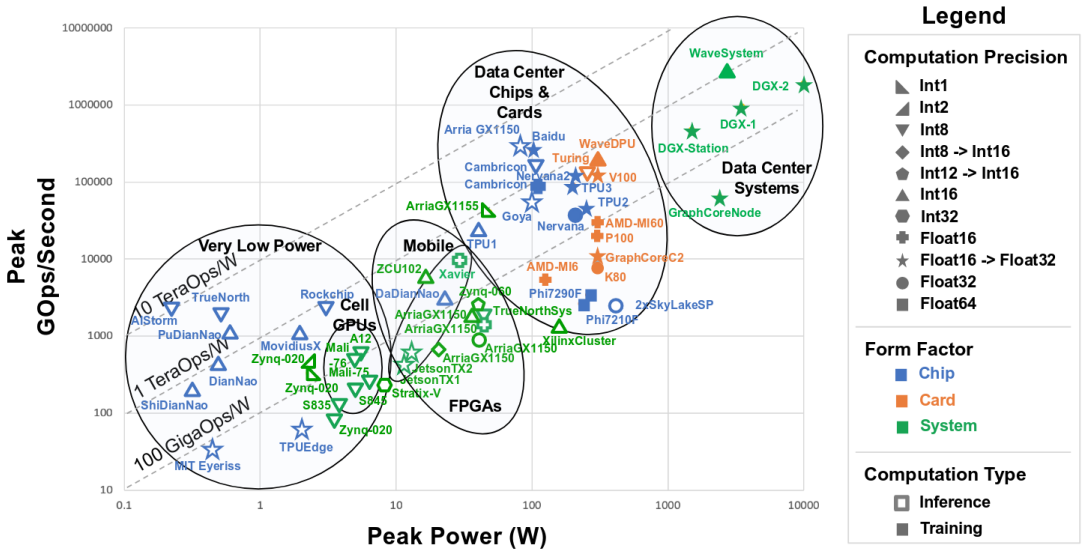


Figure 1.13: Compute throughput vs. power consumption of many types of platforms. Very parallel architectures, such as the V100, tend to achieve better performance-per-Watt than more traditionnal CPUs, like AMD-MI60 or Intel Phi7210F. From (Reuther et al., 2019).

environment, the realm of CPUs, and deployment.

1.2.2 The GPU architecture

The previous section presented several hardware architectures that take advantage of parallel computing to provide low-power and efficient alternatives to monolithic CPU designs. GPUs have combined vector, many-core and dedicated architectures to become a broadly-used solution in the industry for high-performance computing (see fig. 1.14). As the work presented later in this manuscript primarily targets NVIDIA GPUs, we expose their principal design characteristics in this section.

GPUs, as their name indicates, have been developed to compute graphic payloads. They manipulate 3D scenes and render pixels with a complex pipeline of operations that involves managing a model's geometry, textures, and lights. This processing is carried out in parallel by the numerous floating-point units present in the GPU hardware. This considerable computing power has since been leveraged in domains other than scene rendering and display. Image processing, machine learning, bioinformatics, or physics simulations are now heavy users of GPUs for massively parallel workloads.

GPUs work with fundamental execution units called warps in NVIDIA's ter-

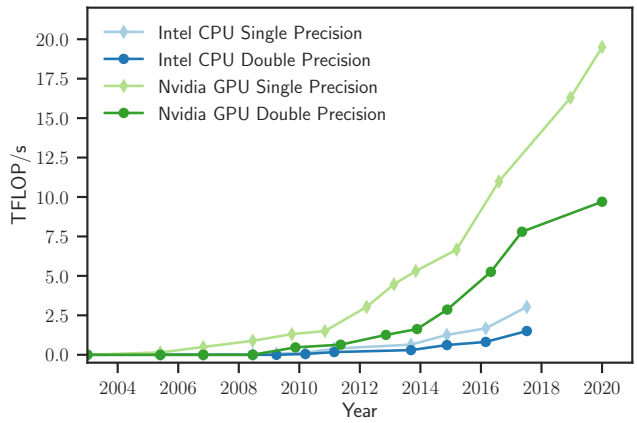


Figure 1.14: In the last few years, the FLOP/s attained by GPUs have largely surpassed those of CPUs.

minology. They are an ensemble of 32 "threads" that process instructions in lock-step. Warps are handled by schedulers that emit an instruction as soon as all conditions required for the execution are met. The warp then disposes of various ALUs (Arithmetic and Logic Units) to perform the operation. Warps have access to registers in the register file, and intra-warp register sharing is possible via "shuffle" instructions. Here, we call the reunion of these ALUs, registers, and schedulers a sub-core, as NVIDIA does not name it specifically.

In the Ampere architecture, each sub-core has a tensor core, a special unit that performs matrix-matrix multiplications. Threads of a warp collaborate to provide input and ouptut registers required by the tensor core. It handles multiple arithmetic precisions: floating point formats (`fp64`, `fp32`, `tf32`, `fp16`, and `b16`) and integers (8, 4 or 1 bit, signed or unsigned). Figure 1.15 explains the different floating point formats. Using tensor cores for matrix-matrix multiplication is significantly faster than relying on traditional ALUs (NVIDIA, 2017).

Figure 1.16 shows the architecture of an Ampere SM (Streaming Multiprocessor). It is made of four sub-cores that share a `L1` cache. This memory location is more than a cache. It also plays the role of a shared memory. Being directly addressable, multiple warps living in the same SM can exchange data via this location. This has the benefit of being faster and less power-hungry than relying on the main memory.

Several SMs are then assembled into one GPU. They use the VRAM (Video RAM) for storing and reading data and share a connection with the CPUs and other peripherals via the PCI-E or NVLINK interfaces. This type of architecture is entirely scalable. Streaming multiprocessors are fundamental building blocks

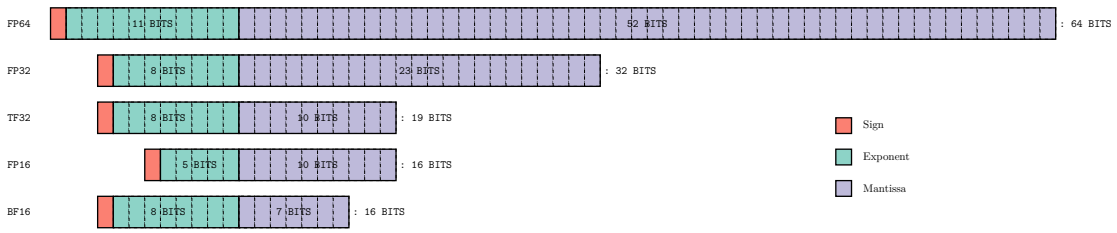


Figure 1.15: Floating-point formats accepted by NVIDIA's tensor cores. FP64, FP32, and FP16 are standardized under IEEE 754. BF16 is a 16-bit truncated version of FP32 and widely used for low-precision arithmetic. TF32 is specific to NVIDIA and have the same range as FP32 and the same precision as FP16.

that can be arranged depending on the constraints. For example, with the Volta architecture, the Titan V card, made for servers, provides 80 SMs, while the Jetson AGX Xavier, an embedded device, has only 8 SMs.

GPUs represent indeed a mix of the examples architectures presented in section 1.2.1. The warp-level parallelism can be considered as a form of SIMD architecture. The numerous streaming multiprocessors with per-core addressable memory are typical of many cores, and tensor cores are an excellent example of specialization for specific computations. It is then necessary to leverage all these characteristics to achieve maximum performance on GPUs.

1.3 Optimizing for hardware performance

In previous sections, we have seen how computer architectures have flourished to offer maximum performance on silicon and what were typical computer vision algorithms. The last essential step is to unify these two topics with the concrete implementation of an algorithm on its execution platform. In this section, we detail the *execute, profile, and optimize* loop. This method aims at reaching maximum performance for a given code on a platform.

1.3.1 From ideas to instructions

The deployment on a particular hardware platform can take different paths. Here, we present solutions that allow the deployment of a given algorithm by the implementation team on the target machine. We label these different ways with categories whose borders are not strict. A language may have characteristics of several groups but mostly belong to one in particular, for example.

Figure 1.16: The architecture of an Ampere Streaming Multiprocessor.

Figure 1.17: The GA100, an NVIDIA GPU based on the Ampere architecture.

Libraries

The easiest solution for deployment is probably to rely on pre-compiled libraries. The APIs (Application Programming Interfaces) they expose must be plugged into pre-existing programs to accelerate a portion of the code. Its un-flexibility counters, however, the ease of use of this solution. Libraries, especially closed-source ones, are opaque from the programmer's point of view. If a functionality is missing or too slow, it is hard to fix the library directly.

Examples of hardware-specific high-performance libraries include Intel's MKL (Math Kernel Library) (Wang et al., 2014), which uses SIMD instructions of Intel CPU to accelerate linear algebra primitives; Arrayfire (Malcolm et al., 2012), an open-source CPU/GPU library that implements linear algebra, image processing, and other utility functions; cuDNN (Chetlur et al., 2014) that has GPU primitives dedicated to neural networks; or AMGX (Naumov et al., 2015), that solves sparse linear systems on GPU, using algebraic multigrid methods.

Language extensions

Language extensions are often a good compromise between performance, flexibility, and proportion of the code modified. OpenMP (Dagum & Menon, 1998), for example, is a framework that consists of C preprocessor macros. They are handled by the compiler and give hints about parts of the code that can use parallel hardware features. This approach allows quick testing on a parallel platform already-existing code (Delisle et al., 2001). OmpSs, developed by the Barcelona Supercomputing Center (Duran et al., 2011), uses the same principle for GPUs and heterogeneous systems, more generally. OmpSs features interoperability with CUDA and MPI. openACC is a similar alternative that primarily targets GPUs with a good trade-off between development effort and performance (Wienke et al., 2012).

Numba is an in-between solution for Python (Lam et al., 2015). It is a library that adds JIT (Just-In-Time) compilation to existing Python programs. The requirement is to annotate the function to compile with `numba.jit`. Numba supports automatic detection of parallel loops and their multi-threaded execution as well as the usage of SIMD instruction and even GPU acceleration on NVIDIA and AMD GPUs (Oden, 2020).

Hardware-focused languages

The specificities of high-performance hardware architectures call for new programming paradigms. The range of languages that produce accelerated code is broad, sometimes limited to a single platform like CUDA or more generic like SYCL (Keryell et al., 2015). This category presents generic languages in the sense

of being applicable for many types of applications, not restricted to signal or image processing, for example.

CUDA and OpenCL are the two main programming languages that focus directly on GPUs and parallel platforms. OpenCL is versatile and targets NVIDIA and AMD GPUs, CPUs, and FPGAs, while CUDA is limited to Nvidia GPUs. This restriction is also advantageous because it is easier to access low-level features of the NVIDIA hardware, such as warp-level instructions or tensor cores. Both languages require the code to be written as if it was executed sequentially. At launch time, the programmer specifies how many threads the function will be run.

SYCL is a programming model developed by the Kronos Group for high-performance computing on various accelerators. Its syntax is based on C++ and relies on templates and lambda functions to support off-loading on the accelerator. SYCL is only the specification, and multiple vendors have provided implementations, Intel with DPC++, codeplay with ComputeCpp. Compared to OpenCL, SYCL implementations usually have shown similar performance with a more generic programming style (Deakin & McIntosh-Smith, 2020).

HDLs (Hardware Description Languages) describe digital circuitry at Register-Transfer-Level (RTL). They serve for designing an architecture to be printed on silicon or synthesized on FPGAs. Being so low level, they model concurrency and parallelism directly and allow to develop an optimal architecture to perform a specific computation. The most notorious languages are VHDL, Verilog, and SystemVerilog.

High-Level Code Generation

Developing with HDLs is long and tedious. That is why HLS (High-Level Synthesis), a type of high-level code generation, has seen its popularity grown over the years. This technique leverages the expressiveness of higher-level languages, such as C, OpenCL (Martelli et al., 2019) or Scala (Bruant et al., 2021) to generate a hardware description. HLS tools are usually a good compromise between development time and performance.

Another example is dataflow languages, that model an algorithm using a directed graph between data and functions. This representation is particularly adapted for tiling or pipelining in image processing. PREESM (Pelcat et al., 2014) follows this principle and generates code for several platforms.

Domain-specific languages (DSLs)

This type of language is not expected to be generic; it can then implement facilities dedicated to particular use cases, favoring ease of expression and performance at

the expense of flexibility.

HALIDE (Ragan-Kelley et al., 2013) is dedicated to image processing. It is based on C++ and creates computation pipelines that support tiling, vectorization, loop-unrolling on several platforms: CPUs (Intel, ARM, RISC-V, ...), GPUs (NVIDIA, AMD, Apple), and even FPGAs.

HipACC (Membarth et al., 2016) is another DSL that uses specific C++ syntax for source-to-source compilation towards different backends: CUDA, C/C++, or Vivado C++ (for HLS), for example. The language has dedicated constructions for image processing and supports boundary conditions, strided image accesses, pyramidal processing, and higher-level operations such as linear filters for convolution.

InKS (Ejjaaouani et al., 2020) is more of a programming model than a single language. It aims at separating the concerns between algorithm development and fine-grained optimizations. For that, several languages have been developped. InKS_{pia} serves at high-level algorithm expression while InKS_{pso} defines lower-level operations.

1.3.2 Understanding the execution

After running the software on the target platform, it is crucial to understand its execution. This phase is called the profiling of a program. It vastly depends on the type of the target. In this section, we propose to examine ways of profiling a program on CPU and GPU.

On CPUs running Linux, a widely used solution is gprof. It depends on GCC to generate instrumentation code (with the `-pg` option). It relies on program counter sampling through CPU interruptions to generate a distribution of where the program spends its time.

Hardware manufacturers usually provide a way to sample performance counters. These special registers measure various things, from memory accesses (cache hits/misses) to instructions per cycle. Different software exists to retrieve such information: nvprof (Nvidia), MAP (Arm), VTune (Intel). For FPGA implementations, the developer designs these counters explicitly or analyzes signal waves.

More specifically, on Nvidia GPUs, three different tools are currently used. The first is Nsight Systems and performs computer-wide profiling. It displays an execution timeline of CPU and GPU functions. This way, it is easy to see when synchronizations happen and if concurrency between data transfers and computation is efficient. Then, depending on the application, a deeper analysis is conducted with Nsight Graphics for computer graphics and 3D scenes or Nsight Compute for compute workloads. Nsight Compute gives insights about GPU utilization, memory accesses, and hardware utilization of the kernels, as seen on fig. 1.18.

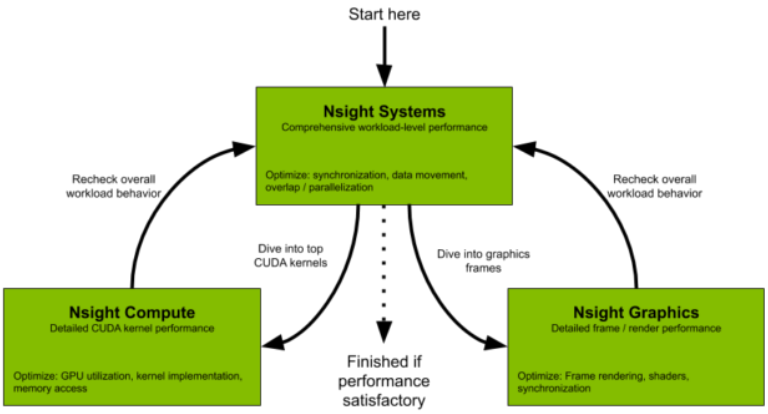


Figure 1.18: NVIDIA tools and workflow for GPU profiling.

1.3.3 Optimize

Choosing the right portion

Efficiently optimizing a code requires a detailed understanding of its execution, which was covered precedently. Often, the algorithm to speed up is complex and made of several parts. Suppose that it takes time T to run the algorithm. The portion of code we seek to optimize is noted p, and we can decompose the total execution time as in eq. (1.12). If this portion achieves a speed-up s, then the optimized program run in T_{opt} (eq. (1.13)). The overall speed-up, computed on the entire program, is given in eq. (1.14).

$$T = (1-p)T + pT \tag{1.12}$$

$$T_{\text{opt}} := (1-p)T + \frac{p}{s}T \tag{1.13}$$

$$S := \frac{T}{T_{\text{opt}}} = \frac{T}{(1-p)T + \frac{p}{s}T} = \frac{s}{p+s-ps} \tag{1.14}$$

These equations are the essence of Amdahl's law (Amdahl, 2013) that formulates an upper bound on achievable speedup with multi-core processors. In our more generic case, if we optimize a fraction $p = 20\%$ with a large $s = 10$, we expect a speed-up of $S \approx 1.21$. While a weaker $s = 1.5$ on a larger fraction $p = 80\%$ leads to $S \approx 1.36$.

This numerical application shows that choosing the right portion to optimize is key to practical speed-up without wasting development time on a smaller part of the code.

Using the roofline model

Once the section of code we seek to optimize is defined, we need to understand what levers we have to make the code faster. Mathematically, let us model the execution time

$$T = l + \frac{w}{b} \tag{1.15}$$

with l the system's latency, w the work required by our program, and b the system's bandwidth in units of work per second. Here, the program is supposed to run in a bandwidth-limited regime with $l \ll \frac{w}{b}$. This assumption makes sense for parallel and data-intensive applications where w is very high.

We can refine the bandwidth model by distinguishing the compute and memory throughput. If the running program does c arithmetic operations and moves m bytes of data on a machine that can process C operations per second with memory bandwidth M, we have

$$T = max(\frac{c}{C}, \frac{m}{M}) \tag{1.16}$$

$$\frac{1}{T} = min(\frac{C}{c}, \frac{M}{m}) \tag{1.17}$$

$$\frac{c}{T} = min(C, M\frac{c}{m}) \tag{1.18}$$

$$\text{FLOP/s} = min(C, M \cdot \text{AI}) \tag{1.19}$$

where AI is the arithmetic intensity, as defined in section 1.1.1.

On fig. 1.19, we plot the attained FLOP/s as a function of AI. The model predicts that measures of performance should lie on the boundaries. In reality, the machine memory and compute bandwidths are an upper bound. Actual implementations perform worse than the model.

The model teaches what kind of optimization is efficient on the implementation. For example, the square measure on fig. 1.19 is bandwidth limited. Increasing its AI will allow it to translate towards a less limited region. This can be achieved by reducing the number of memory operations, for example, with better caches or shared memory utilization. The diamond dot is compute-limited and attains the maximum performance of the hardware. It should be considered sufficiently optimized. The only hope would be to reduce the total number of operations.

Finally, the triangle sample is neither bandwidth nor compute limited. We need to dive deeper into the profiling to understand what is its bottleneck. This simple roofline model only acknowledges arithmetic performance and memory bandwidth but may be refined with new constraints: hierarchical memory bandwidths or per-type-of-instruction throughput (Ding & Williams, 2019; Yang et al., 2020) to help with the analysis.

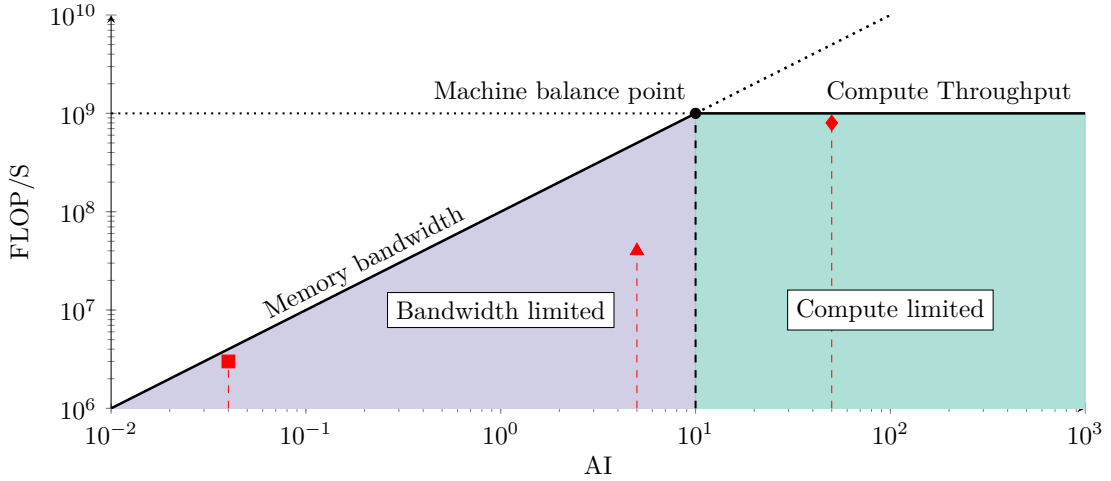


Figure 1.19: A fictional roofline model. The log-log plot indicates throughput vs. arithmetic intensity. Three example measures are given: the square is memory limited due to its low AI. Triangle is far from the theoretical maximum throughput given its AI. There are other factors than memory and compute bandwidth that limits its execution. Diamond uses the hardware almost at its best.

1.4 Conclusion

The journey from the definition of an algorithm to its implementation on industrial platforms is winding and requires precise knowledge along the way. This requirement is enhanced by new hardware designs that have been developed to overcome the pace of innovation slowdown for traditional CPU architectures. Most of the algorithm development phase is still done on these usual platforms due to their ease of use. For best performance and high power efficiency, however, deployment on specialized parallel architectures is a must.

Because of the widening gap in the separation of concern in industrial development, with an algorithm team that designs on CPU and an implementation team that masters efficient architectures, the produced software tends to be sub-optimal. We propose to add an intermediate phase in development that serves as an adaptation between the two. Thanks to the combined knowledge of the software and the hardware, it seems possible to find algorithmic trade-offs to leverage available hardware as efficiently as possible.

To understand a solution's development context, we have presented two algorithms: image convolution and optical flow estimation. With them, we introduced notions such as arithmetic intensity, time, and step complexity. They characterize the algorithm and help in understanding how it performs on actual hardware. Then, we presented why implementations of these algorithms on usual CPUs tend

to become less attractive. A diversity of architecture for high power efficiency has emerged, and we presented some of them, focusing on GPUs that we use for our work. We finally discussed the various ways of targetting a platform of execution, measuring a software implementation's performance, and enhancing its runtime. In the following chapters, we capitalize on this context evaluation to show how the software adaptation phase improves the final performance of the implementation.

Chapter 2

GPU acceleration of image convolutions

This chapter presents the work we did around the convolution operation. It appeared in two contexts: radio-astronomy image reconstruction and a more theoretical setting, where pure performance was sought.

Regarding the development scheme of fig. 1.2, the radio-astronomy experiments are located in phases 2 and 3 of the diagram. The algorithm is set, it is a gradient descent for image deconvolution, but we want to know the required precision to ensure its convergence. Because operating on lower precisions is faster on GPU, we evaluate the trade-off between speed gains and precision degradation on this gradient descent in particular.

Then, we explore a new algorithm for image convolution on tensor cores, a dedicated hardware unit in GPUs for matrix multiplication. This is phase 3 work: the operation is fixed, and we try to attain maximum performance on the execution platform. We evaluate this novel algorithm and compare it with other state-of-the-art implementations regarding the speed and accuracy of the results.

2.1 A Study on Convolution using Half-Precision Floating-Point Numbers on GPU for Radio Astronomy Deconvolution, *SiPS 2018, Published*

This first article stems from the context of the SKA (Square Kilometer Array). This new radio telescope comprises a network of antennas and is expected to provide unprecedented opportunities for space observations. Figure 2.1 shows an artist's view of the type of dishes used for SKA. An enormous amount of data are collected and requires real-time and on-site processing, as raw observations are too

large to be stored. In addition to these constraints, the telescope's location is split across the Australian and South African desert, so power efficiency is a must for the data processing pipeline. These conditions make GPUs a platform of choice to handle the flow of data.

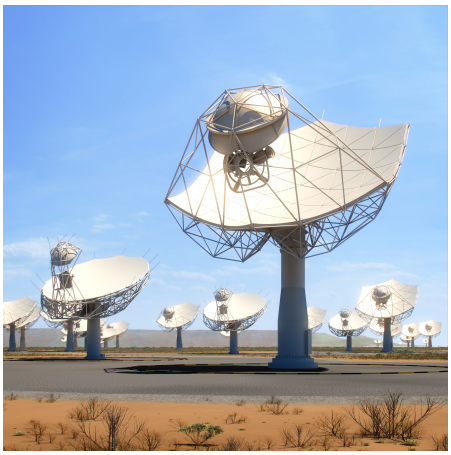


Figure 2.1: Once completed, the SKA will use thousands of such 15m dishes. Credits: skatelescope.org.

We want to identify possible bottlenecks of a typical image processing pipeline on GPU. To do so, we choose a simple yet representative application used in radio-astronomy, deconvolution. This kind of post-processing is generally needed when doing a physical observation of an object. In our case, the object $\boldsymbol{f}$ is an image of the sky. The observation instrument, a telescope, may be modeled as a linear operator, $\boldsymbol{H}$, as a first approximation. We also consider additive measurement noise $\boldsymbol{n}$ so that we observe $\boldsymbol{g} = \boldsymbol{H}\boldsymbol{f} + \boldsymbol{n}$, like illustrated on fig. 2.2. The final goal is to find $\boldsymbol{f}$, knowing $\boldsymbol{g}$ and $\boldsymbol{H}$.

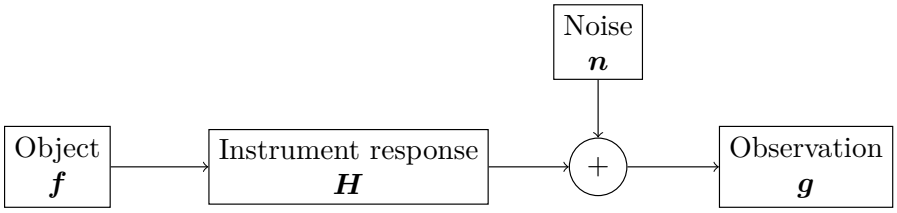


Figure 2.2: Modelisation of an aquisition in the inverse problem framework.

An iterative solver is usually used for finding $\boldsymbol{f}^*$, the re-constructed sky. This processing loop is what constitutes most of the deconvolution's computational needs. Our phase 2 work begins by understanding the iterative solver and find possible optimizations permitted by execution on GPU. With initial profiling, it

appears that the iterative solver heavily uses convolutions. Our first action is then to benchmark several implementations from different libraries.

From this initial point of comparison, we try to leverage reduced-precision floating-point units, such as `fp16` (see fig. 1.15) and see its influence on the algorithm's convergence. This data format offers good speedups on GPUs, but is it sufficient to re-construct images with this algorithm?

A Study on Convolution Using Half-Precision Floating-Point Numbers on GPU for Radio Astronomy Deconvolution

Mickaël Seznec[1], Nicolas Gac[1], André Ferrari[2], François Orieux[1].

[1] Laboratoire des Signaux et Systèmes (L2S), CentraleSupélec, CNRS, Univ Paris sud, Université Paris Saclay, Gif-sur-Yvette, France
[2] Lab. J.-L. Lagrange, Université de Nice Sophia Antipolis, CNRS, Observatoire de la Côte d'Azur, Parc Valrose, F-06108 Nice cedex 02, France

***Abstract*— The use of IEEE 754-2008 half-precision floating-point numbers is an emerging trend in Graphical Processing Units' architecture. Being such a compact way of representing data, its use may speed up programs by reducing the memory bandwidth usage and allowing hardware designers to fit more computing units within the same die space. In this paper, we highlight the acceleration offered by the use of half floating-point numbers over different implementations of the same operation, a 2D convolution. We show that even though it may lead up to a significant speed-up, the degradation brought by this new format is not always negligible. Then, we choose a deconvolution problem inspired by the SKA radio-telescope processing pipeline to show how half floats behave in a more complex application.**

***Index Terms*— deconvolution, radio astronomy, half-precision floating-point, GPU, parallel computing**

I. Introduction

Before appearing in the IEEE standard in 2008 [1] as binary16, half-precision floating-point arithmetic (FP16) has been a topic of interest for computer graphics community since the early 2000s. In parallel, embedded high-performance computing [2] has also investigated its use as an alternative to fixed-point arithmetic in order to design more energy-efficient hardware accelerators. In the same way, deep learning has made a renewed interest to approximate computing [3], [4] especially since GPUs provide half float computation [5]. Indeed, NVIDIA GPUs are offering half float storage since 2015 with CUDA 7.5, half float Multiplier-ACcumulator (MAC) since 2016 with the Pascal architecture [6] and tensor cores, designed for convolutional neural network training, since 2017 with the Volta architecture [7]. These tensor cores offer a Fused-Multiply-Add (FMA) with a mixed precision: a half float multiplication of the FP16 operands followed but an accumulation in the FP32 format.

The half-precision floating point format occupies only 16 bits (1 bit of sign, 5 bits of exponent and 10 bits of mantissa) as illustrated on fig 1 whereas single precision occupies 32 bits (8 bits of exponent and 23 bits of mantissa). Compared to 16-bit integers, it offers an increased dynamic range and compared to 32-bit reals, it divides by 2 the memory storage and bandwidth, of course at the cost of a reduced precision and range. Moreover, the theoretical peak performance (Tflops) on NVIDIA GPU architectures can be significantly increased thanks to half-precision. For instance, the computation power of the Tesla V100 is 15.7 Tflops for FP32 MACs, 31.4 Tflops for FP16 MACs and 125 Tflops for tensors cores.

This work was supported by MAGELLAN (ANR-14-CE23-0004-01)

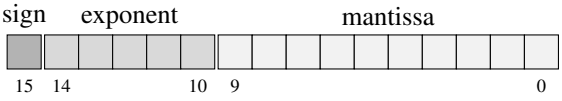


Fig. 1: IEEE 754 binary16 format

Acceleration and energy-efficiency brought by FP16 computation have to be put in the balance with the potential loss of precision. Stability of algorithms using FP16 format is an open problem [8]. Intuitively, one may think that applications where the raw data output of the instrument is integer values, with a dozen bits of accuracy, would not be too much penalized by this compressed-number representation. Like what has been observed for tomography reconstruction [9]. The goal of this study is to observe its use for another inverse problem, the deconvolution for radio astronomy. The optimization algorithm studied (gradient descent) is mainly based on the 2D convolution operator whose acceleration on GPU has been widely investigated for single-precision [10], [11], [12] but as far as we know not yet for half precision. The motivation of this paper is to benefit from the potential acceleration of the 2D convolution with FP16 on GPUs in the perspective of the SKA data processing challenge.

The remainder of this article is organized as follows. Section II describes the deconvolution problem solved using a simple gradient descent. Section III makes a benchmark of 2D convolution on GPU in terms of acceleration and precision. Section IV studies its application for image reconstruction in radio astronomy. Section V presents a discussion and an analysis of the experimental results.

II. Deconvolution

Deconvolution is a classical inverse problem [13] and arises when the observation model is a convolution

$$\boldsymbol{g} = \boldsymbol{H}\boldsymbol{f} + \boldsymbol{n} \tag{1}$$

where $\boldsymbol{g} \in \mathbb{R}^N$ is the data set, $\boldsymbol{f} \in \mathbb{R}^M$ the unknown, $\boldsymbol{n} \in \mathbb{R}^N$ unknown noise and $\boldsymbol{H} \in \mathbb{R}^{N \times M}$ the linear observation model or the convolution operator. If the convolution is circulant, then $N = M$, the matrix $\boldsymbol{H}$ is square and

diagonalizable in Fourier space like $\boldsymbol{H} = \boldsymbol{F}^\dagger \boldsymbol{\Lambda} \boldsymbol{F}$ where $\boldsymbol{F}$ is the linear Fourier transform and $\boldsymbol{\Lambda}$ a diagonal matrix. If the convolution is not circulant the matrix $\boldsymbol{H}$ is not necessary square but remains Tœplitz: all lines of $\boldsymbol{H}$ are shifted version of the first line. In both cases, the matrix $\boldsymbol{H}$ usually leads to ill-conditioned problems with instability and noise amplification.

A standard approach for the reconstruction relies on the regularized least-square where the solution $\hat{\boldsymbol{f}}$ is defined as the minimizer of a data adequacy term and a penalization term

$$\boldsymbol{J} = \|\boldsymbol{g} - \boldsymbol{H}\boldsymbol{f}\|^2 + \lambda \|\boldsymbol{f}\|^2 \tag{2}$$

$$\hat{\boldsymbol{f}} = \arg\min_{\boldsymbol{f}} \boldsymbol{J}(\boldsymbol{f}) \tag{3}$$

The penalization term $\|\boldsymbol{f}\|^2$ on the energy of the solution allows compensating the pathological behavior of the data adequacy and, depending on the balance term λ, leads to a well-posed problem with good conditioning.

The explicit minimizer is known

$$\hat{\boldsymbol{f}} = \left(\boldsymbol{H}^t\boldsymbol{H} + \lambda \boldsymbol{I}\right)^{-1} \boldsymbol{H}^t \boldsymbol{g} \tag{4}$$

and is called the Wiener filter when $\boldsymbol{H}$ is diagonalizable in Fourier space. Otherwise, if the dimension of $\boldsymbol{f}$ is large, the size of the Hessian matrix $\boldsymbol{H}^t\boldsymbol{H}$ forbids the matrix inversion and the solution $\hat{\boldsymbol{f}}$ must be computed with an iterative linear solver [14]. A common one is the gradient descent or conjugate gradient descent described algorithm 1. We consider two versions: one with a fixed step α and one with the optimal one (that corresponds to the maximum descent in the current direction $\boldsymbol{r}$) that needs a little extra computation.

Algorithm 1 The gradient descent algorithm

Require: $\boldsymbol{H}$, λ, $\boldsymbol{g}$, ϵ, N, c

Set $\boldsymbol{b} = \boldsymbol{H}^t\boldsymbol{g}$ and $\boldsymbol{Q} = \boldsymbol{H}^t\boldsymbol{H} + \lambda\boldsymbol{I}$

Set $\boldsymbol{f}^{(0)}$ and $n \leftarrow 0$

repeat

  $\boldsymbol{k} \leftarrow \boldsymbol{H}^t\boldsymbol{H}\boldsymbol{f}^{(n)} - \boldsymbol{b} + \lambda\|\boldsymbol{f}^{(n)}\|^2$

  $\alpha \leftarrow \boldsymbol{k}^t\boldsymbol{k} / \boldsymbol{k}^t\boldsymbol{Q}\boldsymbol{k}$ ▷ or $\alpha \leftarrow c$ with $c \leq \frac{2}{\|\boldsymbol{Q}\|}$

  $\boldsymbol{f}^{(n+1)} \leftarrow \boldsymbol{f}^{(n)} - \alpha\boldsymbol{k}$

  $n \leftarrow n + 1$

until Some criterion is met ▷ e.g.: $n \geq N$

return $\boldsymbol{f}^{(n)}$

III. Convolution Benchmark

The algorithm shown in the previous section relies heavily on the convolution operator: two are needed to find k, and two supplementary ones for the optimal α. Computing a convolution is time-consuming and is often the bottleneck of such methods. There are many ways of implementing this operation on GPU [10]. In this section, we focus on the usage of half floating-point numbers for those methods.

Four implementations are compared in this benchmark: cuBLAS, cuDNN, cuFFT, naive and PRCF. The first three are part of libraries written by Nvidia. cuBLAS is an implementation of the BLAS API [15]. cuDNN (CUDA Deep Neural Network) is a low-level API for deep learning primitives used by other frameworks such as TensorFlow, Caffe2 or PyTorch [16]. cuDNN itself relies on different methods to perform a convolution, depending on many factors: the size of the convolution kernel, whether the images are batched [17]... cuFFT is a GPU implementation of the Fast Fourier Transform method to compute a discrete Fourier transform.

In addition to the implementations found in these libraries, we tested our own algorithms. "naive" launch one GPU thread per image pixel. It then loops over the convolution kernel to perform the multiply-add accumulation. A mixed strategy has also been tested: data are stored in half precision and computation are done using floats. In the kernel code, the GPU threads convert data to floats, do the computation in float and then convert the result back to half. Finally, we used "PRCF" (Parallel Register-only Convolution Filter), that was first presented by Perrot et al [11]. We re-implemented their method but instead on relying on a fixed code generator, we took advantage of C++ templates.

For this benchmark, we use a zero-padding method to handle border issues. Convolutions are done out-of-place. We first transfer the data to the GPU, time 20 convolutions to average the results, stop the timer and transfer the data back to the CPU to check the accuracy of the resulting image. The convolution kernel used is Gaussian and the image is a standard 512×512 pixels cameraman picture. The GPU used is a Nvidia Titan V [7].

In figure 2, the cuFFT curve is almost flat. In fact, the sum of the kernel's width and the image's width is padded to the next power of 2 for performance reasons. In this case, it's always 1024, hence these constant results. cuBLAS and cuDNN are way slower than our custom methods. In fact, cuDNN relies on a matrix multiplication method, just like cuBLAS. They are however useful in neural network contexts as they scale well when many kernels are to be convolved with the same image.

A gap in computation time appears in both "naive" and PRCF implementations for a kernel size of 35 or more. This is due to those implementations being loop unrolled for smaller kernels. However, the compilation time explodes as the size of kernel increase: after 35 we chose to tell the compiler not to optimize them. Finally, because we store the kernel in the GPU texture cache, those implementations are also limited by its size. Once it is too big to fit in, we cannot use them directly.

In terms of performance, once the kernel becomes big (between 35 and 50, depending on implementations and optimizations) it is faster to use a Fourier transform to compute the convolution. We will use this result when choosing an implementation in part IV.

When using half-precision floats, acceleration depends a lot on the chosen algorithm. In CUBLAS, it can reach a x4 speedup (see figure 3). This is due to the library using NVidia Tensor Cores to perform matrix multiplications[7]. The results for the remaining implementations are a bit

disappointing. For the naive algorithm, however, performance increases with a bigger kernel. In fact, the speedup is mainly explained by fewer memory transfers, that bound the algorithm when the kernel becomes large. Regarding PRCF, the poor performance might be due to worse compiler optimizations. Finally, in cuFFT, it is harder to give a justification as we do not have access to the code. Our explanation is that because memory issues do not coalesce, the bandwidth is not saturated, hence no real improvement when using FP16.

In figure 4, the Mean Relative Error (MRE) between the convolution computed on GPU and one done on CPU is displayed. MRE is computed as:

$$MRE = \frac{1}{N} \sum \begin{cases} \left| \frac{x[i] - y[i]}{x[i]} \right|, \text{if } x[i] \neq 0 \\ 0, \text{otherwise} \end{cases} \quad (5)$$

Where x and y are the images to compare and N the total number of pixels in an image.

Please note that we compared our own reference implementation with *convolve2d* from the Python package *SciPy*. The results are clear: when using half floats, the loss of precision is much higher. The error also increases with the size of the kernel. With a width of 115, cuFFT has a 1% error, naive and PRCF, a 10% error. This is due to the multiple imprecisions while accumulating the intermediate results. The naive mixed strategy (half storage and float computation) gives nearly the same acceleration than naive half but provides a lower error (10^{-4}) invariant to kernel size.

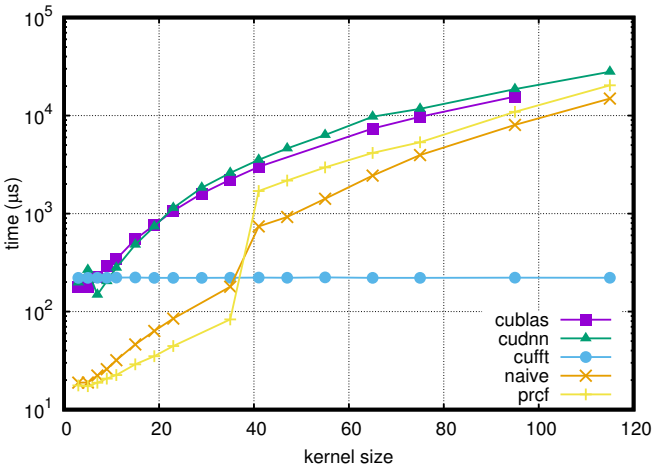


Fig. 2: Execution time in single precision

IV. Application to Image Reconstruction in Radio Astronomy

The future Square Kilometre Array (SKA) will provide radio interferometric data with unprecedented detail. To achieve the nominal performances of the instrument, image reconstruction algorithms are challenged to scale well with TeraByte image sizes never seen before. In the perspective of this challenge, the simulations which follow focus on image deconvolution for radio astronomy.

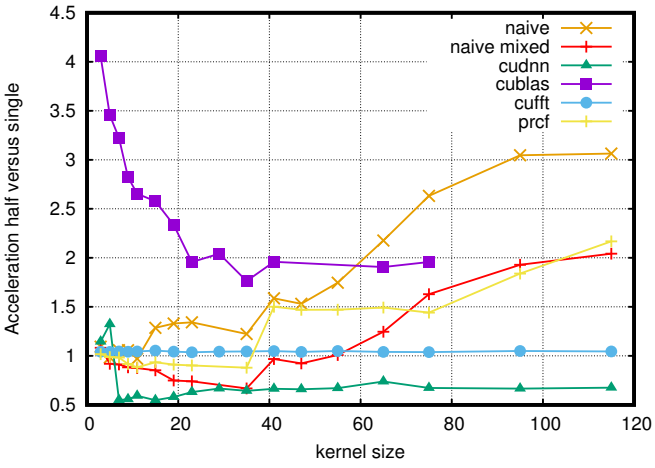


Fig. 3: Acceleration ratio in half vs single

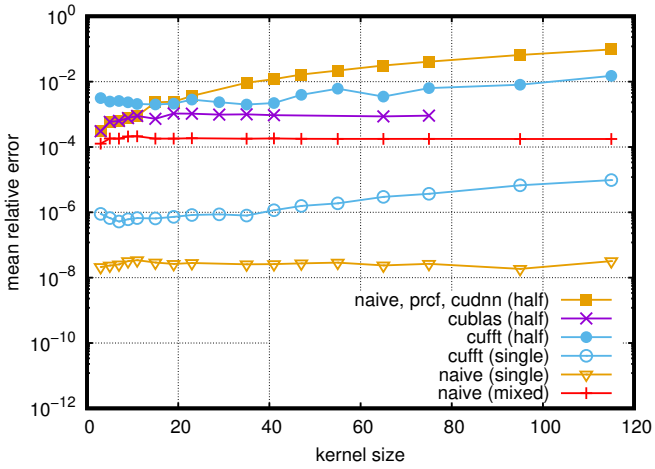


Fig. 4: Error compared to a reference implementation

We used a simulated PSF (Point Spread Function) for the SKA Phase 1 mid-frequency array. The array, which will include 197 dishes, will be built in South Africa from its precursor Meerkat. The PSF was obtained using the HI-inator package based on the MeqTrees software [18] (figure 5a). To ensure a high dynamic range, the simulated sky is composite of point sources and a faint halo modeled by a homogeneous Gaussian field (figure 5b). The ratio between the amplitude of the sources and the maximum value of the halo was set to 10^{-3}. The signal to noise ratio on the observed "dirty" image is set to 37dB.

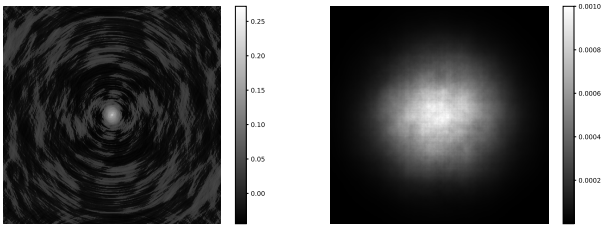
The goal is to reconstruct the image of the sky given a noisy and distorted observation. To accomplish that, we base our approach on the minimization of the quadratically penalized criterion (2) using a gradient descent algorithm as described in section II. Please note that the purpose of these simulations is not to illustrate the performances of the "state of the art" reconstruction algorithms but rather emphasizes advantages and shortcomings of using half-precision

floating-point numbers. All convolutions are done using cuFFT. On figure 6, you can observe multiple reconstructed images using different strategies and precisions. Criterion values across iterations are visible on figure 7. The balance term λ has been set to 0.01 as it provided sensible results.

The FP32 optimal-step curve represents the criterion value $\boldsymbol{J}$ across iterations of the algorithm described in section II (figure 7). As you can see, it quickly decreases and becomes almost flat. The same behavior is observed with the "float fixed-step" curve. In this method, the step α is constant. We chose it by looking at the optimal step values found in the first method and choosing the minimum one. The "mixed" curve behaves the same way. For this implementation, data is stored as halves but computations are made using floats.

The half-precision counterpart curves' behavior is slightly more complex. The optimal step method does not make the criterion decrease for every iteration, hence the noisy values. We can also notice a difference depending on the SNR (Signal-to-Noise Ratio): with a fixed step and a high (37dB) SNR, the criterion seems to decrease but only during the first 250 iterations. With more noise (16dB SNR) "half - fixed step" has the same behavior as the optimal step.

To address this issue, we try a different method: rather than blindingly using $\boldsymbol{f}^{(n+1)} \leftarrow \boldsymbol{f}^{(n)} - \alpha \boldsymbol{g}$, we use a backtracking algorithm. The criterion for the next iteration candidate is computed: if it is higher than the previous one, we step back, set $\alpha \leftarrow \frac{\alpha}{2}$ and try the new value. We proceed until the criterion decreases. Note that sometimes the computed gradient is so inaccurate that it is impossible to make the criterion decrease along its direction. When that happens (after a fixed number of retries), the procedure is ended. We then use the final image of this method as the starting point of an FP32 optimal step method. This is referred to as "half - backtracking than float -optimal step" in figure 7.

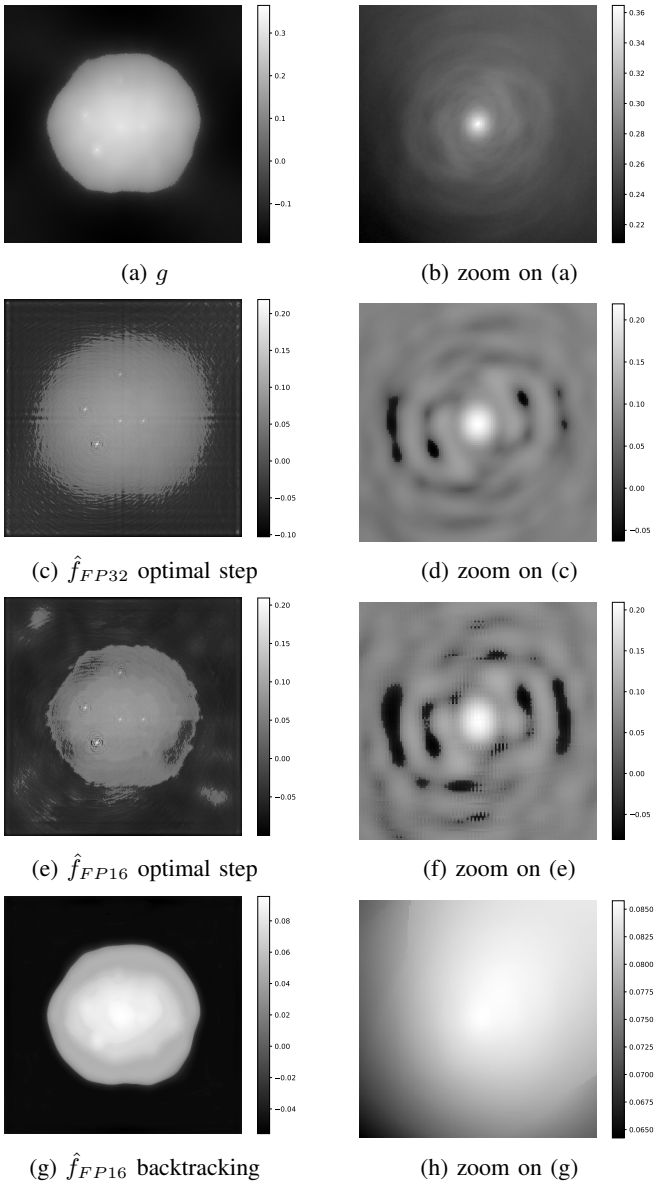


(a) PSF (cubic root) (b) Sky (clipped to 0.001)

Fig. 5: The dataset used

V. Discussion & Analysis

The first thing to point out is that it is difficult to rely on computations done using FP16 numbers. As seen in part II, when doing a convolution, the error increases with the kernel size. In part III, with a 2048x2048 kernel, it is not precise enough to make the criterion decrease at each step. On figure 8, the difference is striking across computations done with FP32 and FP16. The *MSE* between these two images is 9.46, with some points having a relative difference over 1000%.



(a) g (b) zoom on (a)

(c) $\hat{f}_{FP32}$ optimal step (d) zoom on (c)

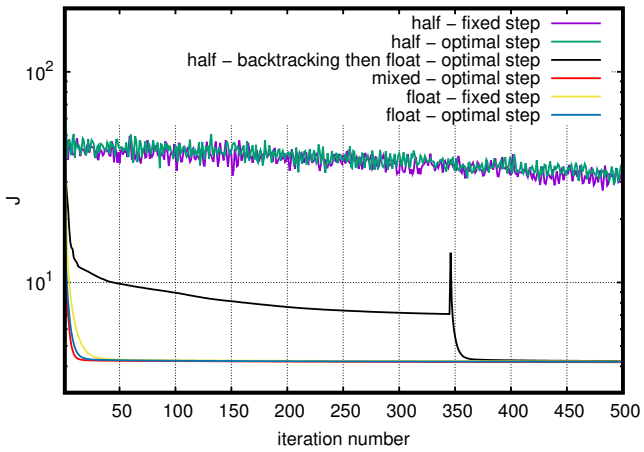
(e) $\hat{f}_{FP16}$ optimal step (f) zoom on (e)

(g) $\hat{f}_{FP16}$ backtracking (h) zoom on (g)

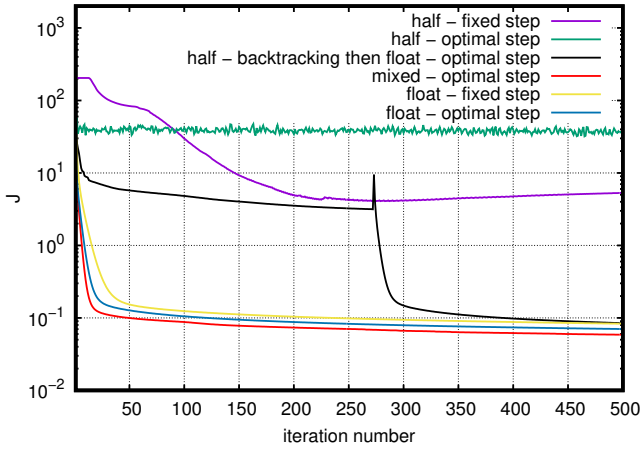
Fig. 6: Reconstructions of f with different precisions (the cubic root is displayed for better contrast)

Deconvolution relying on half floats seems also much more sensitive to noise levels. By comparing figures 7a and 7b, we notice different behaviors in the half-float implementations. The fixed-step version seems noisy with a 16dB SNR but not with a 37dB SNR. There are even differences in the noisy-shaped curves' behavior: they appear to slowly converge on a noisier dataset (16dB SNR). This may be explained by some form of dithering.

In any case, you must put extra care when using half floats as their range is very limited. This issue arises when using cuFFT uses the Fourier domain do compute convolutions. As cuFFT performs non-normalized transforms, half float numbers are easily overflowed. Infinite values will appear in the DFT and lead to wrong results. If you try to first divide your image values by the number of elements, you



(a) Image generated with a 16dB SNR



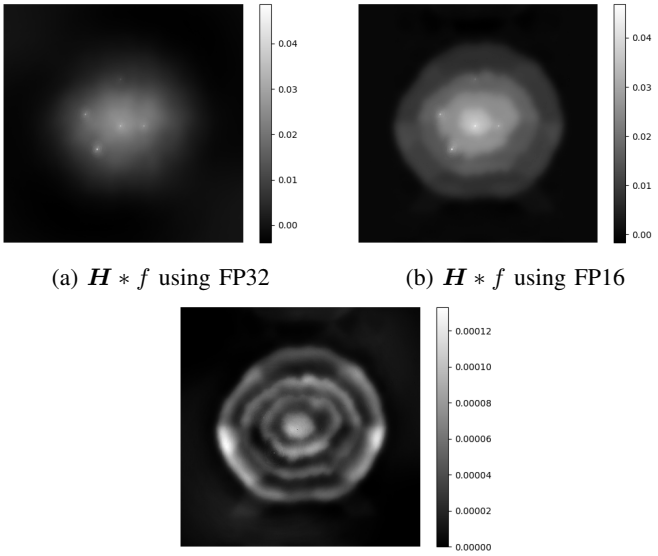
(b) Image generated with a 37dB SNR

Fig. 7: Criterion value across iterations

will underflow and set most values to zero (depending on the size of your data). A solution is to pre-divide by the square root of the number of elements, do the cuFFT, then re-divide by the square root of the number of elements.

Even with this extra care, it was not possible to rely on the convolution in the descent algorithm shown in III. The criterion value is indeed imprecise. This can clearly be seen in figure 7 when stepping from half to single precision in the "half then float" method. Even with the same image, the criterion significantly differs depending on the precision used for its computation. Anyway, using the best image computed with FP16 as an initializer for the FP32 method is slightly better than using a zero-filled image. It is, however, equivalent to an image found after only a few iterations in single precision.

It is unclear why the images produced with the optimal step method using half floats visually give rather good results. Across the iterations, FP16 images do appear to be better even though the criterion does not decreases (even we computed in FP32). The problem might be in the definition of "visually better". Multiple images hold the same criterion value but some may "look" closer to the reconstruction. We



(a) $\boldsymbol{H} * f$ using FP32 (b) $\boldsymbol{H} * f$ using FP16

(c) Squared difference between FP16 and FP32 results

Fig. 8: Convolution errors with our dataset

are currently investigating this issue.

More generally, a dataset involving a smaller kernel may mitigate many problems. When possible, it seems appropriate to use half floats only for storage and convert them on-the-fly as single floats to benefit from lighter data transfer and reasonable accuracy. We can observe on figure 7 that the "mixed" curve has similar performance as the float-only implementation. This kind of strategy is in fact used by Nvidia in their Tensor Cores[7]. In conclusion, the switch from FP32 to FP16 should be done carefully.

VI. Conclusion

In this paper, we have observed the non-negligible loss of precision for 2D convolution using half-precision arithmetic on GPUs. We have pointed out that for limited convolution kernel sizes, a good compromise between acceleration and calculation error is to use a storage in half and a computation in single.

Then, we incorporated half precision arithmetic within a complex application: optimization for image reconstruction in radio astronomy with a kernel convolution of the same size as the 2D images. We tried several methods to choose the step's size in the gradient descent and achieved good visual results. Even though a good convergence does not seem to be achieved using solely half precision, relying on it only for storage but performing computations as floats makes the algorithm converge.

References

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, August 2008.

[2] L. Lacassagne, D. Etiemble, and S. A. O. Kablia. 16-bit floating point instructions for embedded multimedia applications. In *Seventh International Workshop on Computer Architecture for Machine Perception (CAMP'05)*, pages 198–203, July 2005.

[3] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision. *arXiv:1502.02551 [cs, stat]*, February 2015. arXiv: 1502.02551.

[4] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv:1412.7024 [cs]*, December 2014. arXiv: 1412.7024.

[5] N. M. Ho and W. F. Wong. Exploiting half precision arithmetic in Nvidia GPUs. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, September 2017.

[6] Nvidia. GP100 Pascal Whitepaper, https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[7] Nvidia. Volta V100 whitepaper, http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[8] P. Luszczek, J. Kurzak, I. Yamazaki, and J. Dongarra. Towards numerical benchmark for half-precision floating point arithmetic. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–5, September 2017.

[9] Clemens Maaß, Matthias Baer, and Marc Kachelrieß. CT image reconstruction with half precision floating-point values. *Medical Physics*, 38(S1):S95–S105, July 2011.

[10] O. Fialka and M. Cadik. FFT and Convolution Performance in Image Filtering on GPU. In *Tenth International Conference on Information Visualisation (IV'06)*, pages 609–614, July 2006.

[11] Gilles Perrot, Stéphane Domas, and Raphaël Couturier. An optimized GPU-based 2d convolution implementation: AN OPTIMIZED GPU-BASED 2d CONVOLUTION IMPLEMENTATION. *Concurrency and Computation: Practice and Experience*, 28(16):4291–4304, November 2016.

[12] Pavan Yalamanchili, Umar Arshad, Zakiuddin Mohammed, Pradeep Garigipati, Peter Entschev, Brian Kloppenborg, James Malcolm, and John Melonakos. *ArrayFire - A high performance software library for parallel computing with an easy-to-use API*. AccelerEyes, Atlanta, 2015.

[13] Jérôme Idier and Laure Blanc-Féraud. Bayesian Approach to Inverse Problems. pages 141–167. January 2010.

[14] Jonathan Richard Shewchuk. *An introduction to the conjugate gradient method without the agonizing pain*. Carnegie-Mellon University. Department of Computer Science, 1994.

[15] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990.

[16] Deep Learning Frameworks, https://developer.nvidia.com/deep-learning-frameworks. *NVIDIA Developer*, April 2016.

[17] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *arXiv:1410.0759 [cs]*, October 2014. arXiv: 1410.0759.

[18] Jan E. Noordam and Oleg M. Smirnov. The MeqTrees software system and its use for third-generation calibration of radio interferometers. *Astronomy & Astrophysics*, 524:A61, December 2010. arXiv: 1101.1745.

2.2 The Im2Tensor Algorithm for Efficient 2D Convolutions on GPU Tensor Cores, *under review*

In this article, we broaden the study of image convolutions on GPU to the use of tensor cores. These hardware units are capable of performing matrix multiplications, as depicted on fig. 2.3. This feature is then taken into account in our phase 3 optimization method for maximum performance.

We design a dedicated algorithm for tensor cores. While other implementations are already available for convolutions on these units, they are all specialized for particular use-cases, for example, with convolutions between multiple images and many small kernels (Anderson et al., 2017; Chetlur et al., 2014). Our solution is more general and usable in the single-image, single-kernel case.

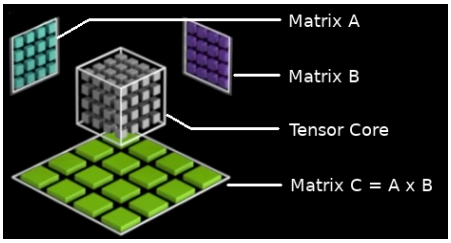


Figure 2.3: A tensor core performs a matrix-matrix multiplication. Adapted from Nvidia.

The paper analyzes first describes the proposed *im2tensor* algorithm and its dual variation. Several optimizations are then studied, using shared memory, atomic operations, and tweaking the work division between GPU cores for maximum performance. This new algorithm is then compared to the state-of-the-art, both in terms of speed and accuracy.

THE IM2TENSOR ALGORITHM FOR EFFICIENT 2D CONVOLUTIONS ON GPU TENSOR CORES*

MICKAËL SEZNEC †‡, NICOLAS GAC †, FRANÇOIS ORIEUX †, AND ALVIN SASHALA NAIK ‡

Abstract. NVIDIA has recently added tensor cores for efficient matrix multiplication in its Graphics Processing Units (GPUs). New DNNs (Deep Neural Networks) or linear algebra programs may leverage them, with typical speedups ranging from 2.5× to 10×. This computing power is, however, limited to specific workloads. For example, the *im2col* algorithm, by Chetlur *et al.*, is broadly used by DNNs but is efficient only when computing convolutions in batches. This article follows the current effort to broaden the use of tensor cores by introducing the *im2tensor* algorithm: a method for efficient single-image, single-kernel convolution. It leverages tensor cores by relying on a matrix-tensor multiplication followed by a sum on diagonals. On small (1024×1024) and larger (4096×4096) image dimensions, our implementation's speed is on par with an optimized GPU convolution in the direct space for small kernels (10-pixel wide) and up to 2× faster for large kernels (30 ∼ 50-pixel wide). In IEEE FP16 precision, the proposed program is 5× more accurate than CUFFT and up to 100× more accurate than direct space convolution due to the extended precision registers used inside tensor cores. Those results have been verified on an embedded GPU (Jetson Xavier) and on a Titan V, a more power-requiring GPU.

Key words. Image Convolution, Hardware Acceleration, GPU Optimisation, Image Processing Systems, GPU Tensor Cores

AMS subject classifications. 65Y05, 65F05, 65G50

1. Introduction.

Tensor cores are a new kind of hardware accelerator made available in NVIDIA GPUs' (Graphics Processing Unit) recent architectures . These units are dedicated to matrix-matrix multiplications, following the computer architecture trend towards specialized accelerators [13].

Tensor cores' stated purpose is to maintain GPUs' ever-growing influence in *compute* workloads, such as DNNs (Deep Neural Networks) or, more generally, BLAS (Basic Linear Algebra Set) operations [6]. Since their introduction in 2017 with the *Volta* architecture [21], they have gained flexibility thanks to the broader range of allowed input types and dimensions. They are now a versatile tool that most GPU users can benefit from when using NVIDIA-provided libraries. Indeed, the tensor core hardware acceleration is already leveraged by CUBLAS, CUTLASS, or CUDNN [8].

While NVIDIA implementations cover many current use cases, the shift introduced by tensor cores towards cheap matrix multiplication operations may prove to have a broad influence on GPU algorithm design. In practice, a current trend is to re-express algorithms in terms of tensor core operations: for parallel primitives [19, 9, 12], image processing via a DSL (Domain-Specific Language) [27] or CT reconstruction [20], for example.

This article shows how to redesign the 2D convolution to take advantage of tensor cores. The 2D convolution is the backbone of many image processing methods for computer vision and appears in a wide range of situations: edge detection, template matching, Gaussian blurring, or feature maps generation for CNNs (Convolutional Neural Networks).

*Submitted to the editors April 6, 2021

†Paris-Saclay University, CNRS, CentraleSupelec, L2S, Gif-sur-Yvette, 91192 France (firstname.lastname@l2s.centralesupelec.fr).

‡Thales Research and Technology, Palaiseau, 91120 France (firstname.lastname@thalesgroup.com).

The variety of situations in which 2D convolutions are used means that each particular context may benefit from a dedicated implementation. In the case of CNNs, there are many small kernels to apply to many images. This setting is handled by *im2col* in the CUDNN library and leverages tensor cores. This algorithm is inadequate in more traditional computer vision workloads, like large spatial gradients or Gaussian blurs. This is the setting where our implementation is most useful.

Our contribution is then as follows:

- The *im2tensor* algorithm for 2D convolutions is described as a sequence of matrix multiplications and summations on the diagonals. In this form, the convolution can leverage tensor cores' power, and the framework can handle different image border policies. The article also explains a *dual* variant of the algorithm.
- The CUDA implementation for Nvidia Titan V and Jetson Xavier devices is exposed, along with the different challenges raised by tensor cores. That ranges from a sensible choice for the underlying matrix dimensions for the tensor cores to the use of shared memory for maximized data utilization.
- This novel algorithm is compared with state-of-the-art methods such as CUDNN, CUFFT, and ArrayFire [33], with results for speed and accuracy. The proposed approach is fastest on a large range of kernel sizes, while being one of the most accurate methods.

Section 2 presents related work on convolution algorithms and an overview of GPU programming and tensor cores. Section 3 explains in detail the *im2tensor* algorithm as well as its dual variant. Section 4 goes through the details of the implementation. It reviews the different strategies used to minimize the runtime of the algorithm. Section 5 provides a comparison between our method and several state-of-the-art implementations. The evaluation is done in two different contexts: embedded (30W) and desktop (500W). Results are given in terms of speed and accuracy with respect to a reference implementation. Finally, section 6 concludes this paper and offers directions to follow for further work.

2. Background.

2.1. 2D Convolutions. Convolutions are a fundamental tool of signal processing. When applied to images, it serves for template-matching methods [7], edge detection [31], or noise reduction [28]. Convolution is such a ubiquitous operation that a lot of work has been devoted to speed up its execution on modern computers:

Separable convolutions. If the kernel K can be written as the outer product of two vectors $K = \boldsymbol{k}_1 \boldsymbol{k}_2^T$, the convolution can be performed in two steps: $R = (I * \boldsymbol{k}_1) * \boldsymbol{k}_2$. This technique reduces the overall memory pressure but is restricted to particular kernels.

Convolutions in the Fourier space. A convolution can be computed with an element-wise multiplication of the Fourier transforms of the image and the kernel. This product should then undergo an inverse Fourier transform. This technique is asymptotically faster than convolutions in the direct space but may be slower for large images and small kernels [26].

Winograd convolutions. This category groups several methods that build optimal algorithms in terms of arithmetic complexity. In [17], Lavin *et al.* first presented a GPU implementation said to be Winograd-based [32]. Like the Fourier method, it requires the input image and kernel to be transformed, pointwise multiplied and then be inverse-transformed. It has been shown to perform well in DNN for small convolution kernels but is also sensitive to numerical instability [4].

Overlap and Add. This algorithm follows the divide-and-conquer strategy: first, divide the input image into smaller images. Then, compute the convolution of all smaller images with the original kernel. Then recombine the full image and sum where the results overlap [2].

GEMM-based techniques. GEMM strategies are motivated by heavily optimized libraries for matrix multiplication (openBLAS, cuBLAS). The transformation from convolution to matrix multiplication stems from two steps: first, flatten the kernel into a vector in a row-major fashion. Second, design a matrix from the image's coefficients such that when multiplying the flattened kernel with this constructed matrix, the vector-matrix product effectively computes the convolution [8]. This method shows all benefits when batching multiple kernels: by stacking the kernels to form a matrix, the now matrix-matrix multiplication computes multiple convolutions at once. Anderson *et al.* [3] extend this idea to different layouts.

Out of the presented methods, only GEMM-based convolutions benefits from the additional power brought by tensor cores. However, they are only useful when computing batches of convolutions, with the same kernels on many images. This scenario is less likely to appear in a traditional computer vision algorithm. A typical processing would use a small number of kernels (spatial derivative, Gaussian blur) on few images.

In the rest of this article, we detail a new algorithm that uses tensor cores efficiently for single-kernel and single-image convolutions.

2.1.1. Notations. In this article, a lowercase a denotes a coefficient, the bold lowercase $\boldsymbol{a}$ is a vector, the uppercase A is a matrix, and the bold uppercase $\boldsymbol{A}$ is a three-dimensional tensor. $A_{i,j}$ denotes the coefficient in the i-th row, j-th column of A. The colon notation selects all elements in a dimension: $A_{i,:}$ is the entire i-th row of A.

Let I be an image of size (h_I, w_I) and K be the kernel of size (h_K, w_K). Let R, of shape (h_R, w_R), be the convolution of I by K, defined by:

$$\forall i \in [\![0, h_I - h_K]\!], \forall j \in [\![0, w_I - w_K]\!], \tag{2.1}$$

$$R_{i,j} = \sum_{y=0}^{h_K-1} \sum_{x=0}^{w_K-1} K_{y,x} I_{i+y,j+y} \tag{2.2}$$

Formally, this definition is a cross-correlation. For the sake of simplicity, it is anyway called a convolution throughout this article. The *real* convolution can be computed by cross-correlating the image with the reversed kernel. With our definition, the result's dimensions are $(h_R, w_R) = (h_I - h_K + 1, w_I - w_K + 1)$. To adhere to numpy and Matlab conventions, this is the so-called *valid* convolution. A *full* convolution yields a $(h_I + h_K - 1, w_I + w_K - 1)$ output while *same* produces a (h_I, w_I) result. The latter two methods require conditions on the borders. They can be computed by doing a *valid* convolution on a pre-padded image. Figure 1 introduces the above notations.

2.2. GPU Programming. GPUs were initially designed for efficient production and display of images on computer screens. They first achieved this goal by means of hardware-fixed functions such as rasterization and pixel shading. With the ever-growing interest in such a powerful processor, GPUs became more flexible and open to general computation. In 2007, Nvidia released the CUDA language that made GPUs handy as a compute platform.

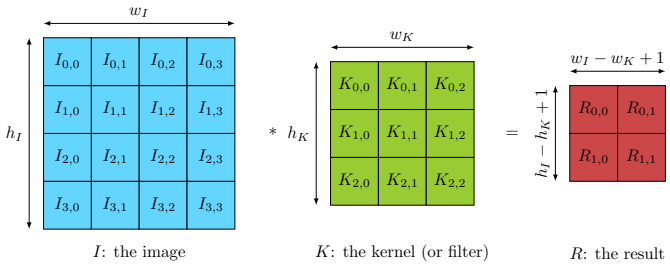


Fig. 1: The *valid* convolution between I and K.

From the software perspective, CUDA bases its programming model on a SIMT (Single Instruction, Multiple Threads) paradigm, a variant of SIMD (Single Instruction, Multiple Data). The programmer writes a single program and specifies how many threads should run it. Threads are partitioned into Thread Blocks (TB) of a customizable size where threads can be synchronized using barrier instructions and share data efficiently through shared memory. This memory location is used in our implementation to share partial matrix multiplication results.

Many challenges must be faced for a program to run efficiently on GPU:

- The program must be parallel enough to maximize the occupation of the GPU.
- It should perform sequential (coalesced) memory accesses and take advantage of the cache hierarchy and the shared memory to boost performance.
- Control flow path divergence should be avoided within the same warp even though this constraint is less of an issue on recent architectures [21].

See [16, 25, 15] for more information about GPU and CUDA programming.

2.3. Tensor Cores. Tensor cores are recent additional hardware built into the *Volta*, *Turing*, and *Ampere* GPU architectures [21, 22, 23]. These special units compute a matrix multiplication and accumulation: $D = AB + C$ as shown in Figure 2. While tensor cores operate on 4×4 matrices at the hardware level, the ISA (Instruction Set Architecture) of NVIDIA GPUs provides instructions for larger operand sizes. This is made possible by combining block matrices operations.

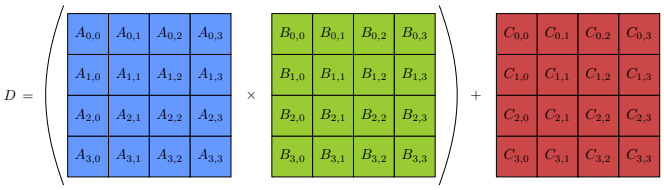


Fig. 2: The matrix multiplication-and-accumulation operation made by a tensor core.

The operands A, B, C, and D may be stored with several numerical precisions. The supported formats depend on the generation of the GPU. Volta GPUs brought the first generation of tensor cores and only supported *fp16* precision (IEEE 754 *binary16*). Turing and Ampere added support for other input types while keeping backward compatibility for already supported precisions. The output type is usually

the same as the input matrices, except for *fp16* inputs, where the user chooses between an *fp16* or *fp32* output.

Several authors have explored the arithmetic accuracy of such reduced or mixed-precision operations [14, 18, 5]. Several experiments are conducted later in this article to assess those effects on our convolution algorithm.

3. The im2tensor algorithms. This section introduces our novel algorithm to compute a *valid* convolution between an image I and a kernel K, as defined in subsection 2.1.1. It uses a 3-dimensional tensor $\boldsymbol{S}$ of size $(h_S, w_S, d_S) = (h_K, w_I, h_I - h_K + 1)$ defined by:

$$\forall i \in [\![0, h_K - 1]\!], \forall j \in [\![0, w_I - 1]\!], \forall k \in [\![0, h_I - h_K]\!], \quad \boldsymbol{S}_{i,j,k} = I_{i+k,j} \tag{3.1}$$

By multiplying K^T with $\boldsymbol{S}$ and summing along the resulting tensor's diagonals, the result is effectively a convolution (Figure 3).

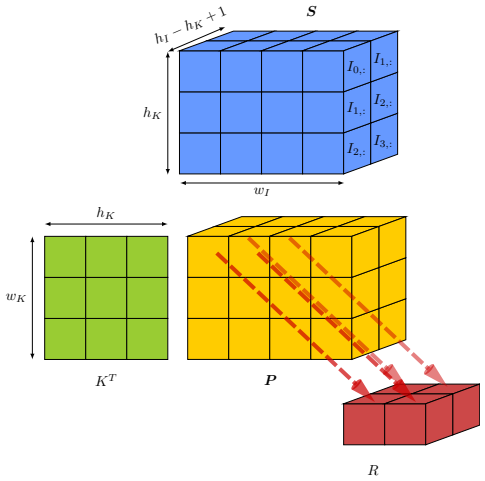


Fig. 3: Convolution as a sum on diagonals of a matrix-tensor product.

The rest of this section explains how this operation works, introduces a dual version of this algorithm, and explains how to handle borders to achieve a *same* convolution.

3.1. Convolution through matrix products.

3.1.1. Row selections of I. From (3.1), it follows that the k-th depth-wise slice of $\boldsymbol{S}$, noted $\boldsymbol{S}_{:,:,k}$, is a row selection of the input image. Figure 4 shows how to construct $\boldsymbol{S}_{:,:,0}$.

Notice that two successive slices of $\boldsymbol{S}$, $\boldsymbol{S}_{:,:,k}$ and $\boldsymbol{S}_{:,:,k+1}$ both use the rows $[\![k+1, k+h_K-1]\!]$ of I. These shared references should be leveraged to avoid duplicates when storing $\boldsymbol{S}$ in a computer's memory.

3.1.2. Sums on diagonals. Let $\boldsymbol{P}$ be the result of the matrix-tensor product $K^T\boldsymbol{S}$. $\boldsymbol{P}$ is a tensor of size $(h_P, w_P, d_P) = (w_K, w_I, h_I - h_K + 1)$. We now seek to compute the sums of the diagonals of a depth-wise slice $\boldsymbol{P}_{:,:,k}$.

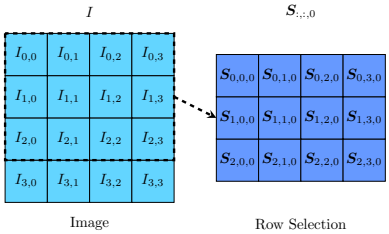


Fig. 4: Selecting the first h_K rows of I to form $\boldsymbol{S}_{:,:,0}$.

$\boldsymbol{P}_{:,:,k}$ is a matrix of size (h_P, w_P). The l-th diagonal of $\boldsymbol{P}_{:,:,k}$ is refered as $\boldsymbol{d}^{(k,l)}$. It is defined by $\forall m$, $\boldsymbol{d}^{(k,l)}_m = \boldsymbol{P}_{m,m+l,k}$.

Only *complete* diagonals of $\boldsymbol{P}_{:,:,k}$, i.e., diagonals with as many elements as $\boldsymbol{d}^{(k,0)}$ are considered. Since $\boldsymbol{P}_{:,:,k}$ has a (w_K, w_I) shape, it has $|w_K - w_I| + 1$ *complete* diagonals.

Figure 5 shows the multiplication of K^T with $\boldsymbol{S}_{:,:,0}$ to get $\boldsymbol{P}_{:,:,0}$. Summing the elements of the two *complete* diagonals of $\boldsymbol{P}_{:,:,0}$, $\boldsymbol{d}^{(0,0)}$ and $\boldsymbol{d}^{(0,1)}$, respectively yields $R_{0,0}$ and $R_{0,1}$.

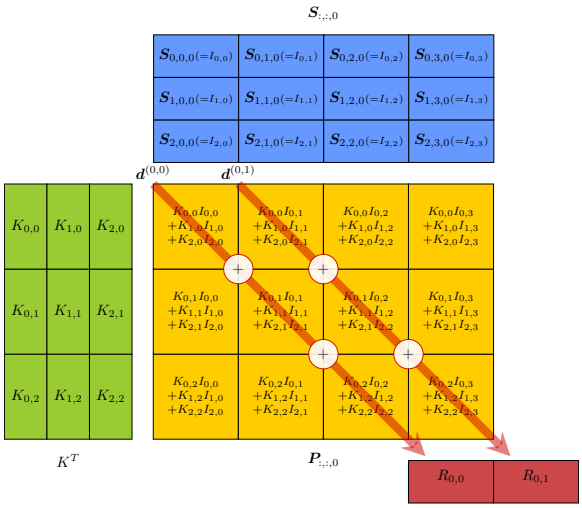


Fig. 5: Detail of the $K^T\boldsymbol{S}_{:,:,0}$ product and sums on diagonals.

Let us reiterate this process with $\boldsymbol{S}_{:,:,1}$, a selection of the last h_K rows of I. Having computed $\boldsymbol{P}_{:,:,1} = K^T\boldsymbol{S}_{:,:,1}$, another sum on the diagonals now gives $R_{1,:}$.

It should now be clear how the matrix-tensor product $\boldsymbol{P} = K^T\boldsymbol{S}$ leads to rows of R by summing on the diagonals of depth-wise slices of $\boldsymbol{P}$.

We shall now formalize the sums over the *complete* positive diagonals. Tr_+ is defined for a matrix A of shape (m, n) such that $m \leq n$:

$$\mathrm{Tr}_+ \colon \mathbb{R}^{m\times n} \to \mathbb{R}^{n-m+1} \tag{3.2}$$

$$A \mapsto \boldsymbol{x}, \boldsymbol{x}_k = \sum_i A_{i,i+k},\ \forall k \in [\![0, n-m]\!] \tag{3.3}$$

Likewise, the bold operator $\mathbf{Tr}_+$ is defined for 3D tensors and stacks the results of Tr_+ on each slice of its input. The im2tensor algorithm can finally be summarized as,

$$R = \mathbf{Tr}_+(K^T \boldsymbol{S}) \tag{3.4}$$

This equation is visualized in Figure 3.

3.2. A Dual Version. While row selection is the backbone of the im2tensor algorithm just presented, another algorithm based on column selection exists. We call it *dual* im2tensor and explain it in this section.

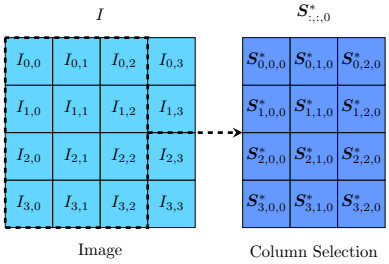


Fig. 6: Selecting the first w_K columns of I to form $\boldsymbol{S}^*_{:,:,0}$.

Similarly to (3.1), let $\boldsymbol{S}^*$ be the tensor of size $(h^*_S, w^*_S, d^*_S) = (h_I, w_K, w_I - w_K + 1)$ such that:

$$\forall i \in [\![0, h_I - 1]\!], \forall j \in [\![0, w_K - 1]\!], \forall k \in [\![0, w_I - w_K]\!], \quad \boldsymbol{S}^*_{i,j,k} = I_{i,j+k} \tag{3.5}$$

Instead of having $\boldsymbol{S}_{:,:,k}$ slices be row selections of I, $\boldsymbol{S}^*_{:,:,k}$ slices are column selections of I (see Figure 6). The matrix-tensor operands are commuted: $\boldsymbol{P}^* = \boldsymbol{S}^* K^T$. (see Figure 7).

The afterward sum must now be done on the lower diagonals of a slice $\boldsymbol{P}^*_{:,:,k}$. The Tr_- operator is defined to be the negative diagonal counterpart of Tr_+. The same logic applies to $\mathbf{Tr}_-$ and $\mathbf{Tr}_+$.

Just like (3.4), the dual im2tensor algorithm is summarized as follows:

$$R = \mathbf{Tr}_-(\boldsymbol{S}^* K^T) \tag{3.6}$$

3.3. Handling borders. So far, we have presented the *im2tensor* algorithm for *valid* convolutions. *Same* and *full* convolutions require a preprocessing step. The image I should be padded accordingly to the desired border conditions.

Figure 8 shows a zero-padding example for a *same* convolution, for a K of size $(3, 3)$.

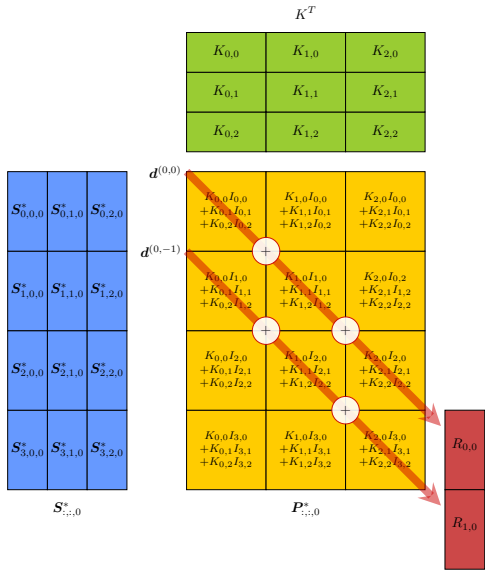


Fig. 7: Detail of the im2tensor dual algorithm. The matrix-matrix product $\boldsymbol{S}^*_{:,:,0}K^T = \boldsymbol{P}^*_{:,:,0}$ leads to one column of R.

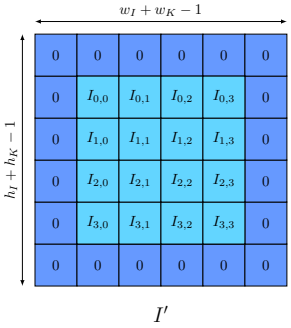


Fig. 8: I' is the zero-padded version of I for a *same* convolution.

The im2tensor algorithm can now be used on I', the padded version of I: $R' = \mathbf{Tr}_+(K^T \boldsymbol{S}')$. Note that no actual storage for I' is needed in computer memory. The algorithm could deduce the values of I' on-the-fly.

4. Efficient GPU implementation.

4.1. Overview. In the previous section, we explained two ways to compute a 2D convolution, both relying on a matrix-tensor product, followed by a sum reduction along diagonals. The following section, in turn, describes details of an implementation of the algorithm. It reviews the characteristics needed for efficient parallel execution on GPUs.

A pseudo-code of our implementation is given in Algorithm 1. It describes the row-selection version of *im2tensor*. In this procedure, we compute a *valid* convolution

```
1  Function Im2Tensor
       input  : An image I, a kernel K
       output: R the 2D valid convolution
2      K^T ← transpose(K)
3      begin CUDA Kernel
4          while any P_block remains do
5              b_idx ← getPBlockIdx (TB_idx)
6              K_block ← readLines (K^T, b_idx)
7              S_block ← readAndBuildSubTensor (I, b_idx)
8              P_block ← matTensorMultiply (K_block, S_block)
9          end
10     end
11     begin CUDA Kernel
12         while any diagonal remains do
13             d^(k), d^(l) ← getDiagIds (Thread_idx)
14             s ← sumDiagonal (P, d^(k), d^(l))
15             storePixel (R, s)
16         end
17     end
```

Algorithm 1: Pseudo-code description of im2tensor.

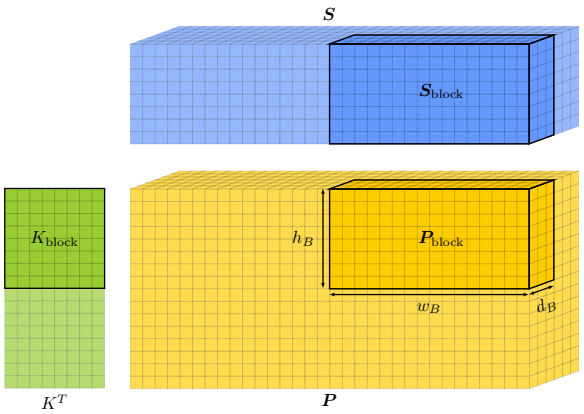


Fig. 9: A $\boldsymbol{P}_{\text{block}}$ is handled by a thread block. To compute its values, the TB must read the associated K_{block} and $\boldsymbol{S}_{\text{block}}$.

and start by transposing the kernel.

The core of the algorithm is composed of two main operations: computing the matrix-tensor product $\boldsymbol{P} = K^T \boldsymbol{S}$, and executing the $\mathbf{Tr}_+$ operator, i.e., a sum over the diagonals of $\boldsymbol{P}$.

Regarding the matrix-tensor product, the strategy is to partition $\boldsymbol{P}$ into blocks. Each individual $\boldsymbol{P}_{\text{block}}$ is computed by a CUDA thread block (TB). Each TB is referred by a unique identifier: TB_{idx} (see Line 4 of Algorithm 1). There can be more $\boldsymbol{P}_{\text{block}}$ than TBs. Then, when a TB finishes its $\boldsymbol{P}_{\text{block}}$, it moves onto another $\boldsymbol{P}_{\text{block}}$ to work on (Line 3).

To compute a $\boldsymbol{P}_{\text{block}}$, a TB must access a selection of rows of K^T, which we call K_{block}, and a sub-tensor of $\boldsymbol{S}$ called $\boldsymbol{S}_{\text{block}}$. Note that a $\boldsymbol{S}_{\text{block}}$ can be formed by reading a selection of columns of I. The $\boldsymbol{P}_{\text{block}}$ has a (h_B, w_B, h_B) shape. The blocked matrix-tensor product can be seen in Figure 9.

The last part of the algorithm is a reduction on the diagonals. On Line 11, each CUDA thread is assigned to a diagonal according to its thread id. $d^{(k)}$ is the depth index and refers to the k-th slice of $\boldsymbol{P}$. $(d^{(k)}, d^{(l)})$ refers to l-th diagonal of $\boldsymbol{P}_{:,:,k}$, as defined in subsection 3.1.2. The CUDA thread iterates over the diagonal, sum the coefficients and stores the result to the GPU's main memory.

4.2. Levers for performance.

4.2.1. Tensor cores considerations. Our algorithm was designed with tensor cores in mind, as presented in subsection 2.3. They are used when computing a $\boldsymbol{P}_{\text{block}}$ (Algorithm 1, Line 7). The matrix-tensor product is indeed a collection of matrix-matrix multiplications handled by tensor cores.

Tensor Core operations are restricted to specific matrix dimensions. NVIDIA provides a reference of the acceptable shapes $(m_{\text{tc}}, k_{\text{tc}})$ and $(k_{\text{tc}}, n_{\text{tc}})$ for the operands A and B, respectively.

This restricted shape imposes padding on the inputs. The choice for the shape's dimensions impacts the overall performance. For example, the choice of $(m_{\text{tc}}, n_{\text{tc}}, k_{\text{tc}}) = (32, 8, 16)$ with a small kernel $(h_K, w_K) = (5, 5)$ requires the extension of K^T with zeros so its height meets 32. Consequently, a large portion of the algorithm's operations will be to compute the zeros of $\boldsymbol{P}$.

Let us analytically derive the overhead introduced by the tensor core padding. To this end, the following notation is introduced: $\lceil m \rceil_{(n)}$. It refers to the multiple of n directly higher than m:

$$\forall n \in \mathbb{N}^*, \forall m \in \mathbb{N},\ \lceil m \rceil_{(n)} \mapsto n \lceil \frac{m}{n} \rceil \tag{4.1}$$

The hat version of matrices and tensors are zero-padded to the nearest multiple of m_{tc}, k_{tc} or n_{tc}:

$$\hat{w}_K = \lceil w_K \rceil_{(m_{\text{tc}})},\ \hat{h}_K = \lceil h_K \rceil_{(k_{\text{tc}})},\ \hat{w}_I = \lceil w_I \rceil_{(n_{\text{tc}})} \tag{4.2}$$

$$\hat{K}^T = \begin{bmatrix} K^T & 0 \\ 0 & 0 \end{bmatrix} \in \mathbb{R}^{\hat{w}_K \times \hat{h}_K} \tag{4.3}$$

$$\forall k \in [\![0,\ d_S - 1]\!],\ \hat{\boldsymbol{S}}_{:,:,k} = \begin{bmatrix} \boldsymbol{S}_{:,:,k} & 0 \\ 0 & 0 \end{bmatrix} \in \mathbb{R}^{\hat{h}_K \times \hat{w}_I} \tag{4.4}$$

The result $\hat{\boldsymbol{P}} = \hat{K}^T \hat{\boldsymbol{S}}$ has a shape $(\hat{w}_K, \hat{w}_I, h_I - h_K + 1)$.

4.2.2. Computational Complexity. Our method uses a simple matrix multiplication algorithm, without Strassen-like methods' sophistication. Its complexity is:

$$\mathcal{O}\left(\hat{w}_K \hat{w}_I \hat{h}_K (h_I - h_K + 1)\right) \tag{4.5}$$

$$\text{i.e., } \mathcal{O}\left(\lceil w_K \rceil_{(m_{\text{tc}})} \lceil w_I \rceil_{(n_{\text{tc}})} \lceil h_K \rceil_{(k_{\text{tc}})} (h_I - h_K + 1)\right) \tag{4.6}$$

Three terms are directly impacted by the padding introduced by tensor cores' input shapes. In our setting, we consider that the image is significantly larger than the kernel: $w_I \gg w_K$, $w_I \gg h_K$. With this assumption, we deduce that k_{tc} and m_{tc} should be kept small while the choice of n_{tc} is less sensitive.

4.2.3. Arithmetic Intensity. The arithmetic intensity of a computer program that transfers Q bytes of data to execute W operations is defined as:

$$\text{ai} = \frac{W}{Q} \tag{4.7}$$

This measure, central to the roofline model analysis[30], must be high enough to claim using the GPU efficiently[10]. It is usually measured at execution time through performance counters. We, however, propose an *a priori* estimation of the arithmetic intensity to guide our design choices.

We are interested in the computation of a P_{block}, of size (h_B, w_B, d_B), as shown in Figure 9. Let us see how these three dimensions should be chosen to maximize the arithmetic intensity.

As for data transfers, this operation fetches data from $\hat{K}^T$: a $(h_B, \hat{h}_K)$ sub-image; and from $\boldsymbol{S}$: a $(\hat{h}_K, w_B, d_B)$ sub-tensor. Due to duplicate rows in $\boldsymbol{S}_{\text{block}}$, this sub-tensor only requires reading a $(\hat{h}_K + d_B - 1, w_B)$ sub-image from I. Memory writes are not counted in this analysis.

Regarding operations, the matrix-tensor products for $\boldsymbol{P}_{\text{block}}$ involve $h_B \hat{h}_K w_B d_B$ multiplication additions. Then, we have:

$$\text{ai}(h_B, w_B, d_B) = \frac{h_B \hat{h}_K w_B d_B}{h_B \hat{h}_K + (\hat{h}_K + d_B - 1) w_B} \tag{4.8}$$

$$\nabla \log(\text{ai}) = \begin{bmatrix} \dfrac{1}{h_B} - \dfrac{\hat{h}_K}{\hat{h}_K + h_B + (\hat{h}_K + d_B - 1) w_B} \\ \dfrac{1}{w_B} - \dfrac{\hat{h}_K + d_B - 1}{\hat{h}_K + h_B + (\hat{h}_K + d_B - 1) w_B} \\ \dfrac{1}{d_B} - \dfrac{w_B}{\hat{h}_K + h_B + (\hat{h}_K + d_B - 1) w_B} \end{bmatrix} \tag{4.9}$$

An analysis of $\nabla \log(\text{ai})$ shows that it is preferable to increase as a priority d_B then h_B and keep w_B low. One must still consider that increasing d_B and h_B means reading more rows of I, which is usually slower than reading longer rows (by increasing w_B) due to the memory layout of the image.

In our experiments, with $(m_{\text{tc}}, n_{\text{tc}}, k_{\text{tc}}) = (8, 32, 16)$, we set (h_B, w_B, d_B) to $(8, 32, 32)$ as it proved provide the best results.

4.2.4. Fusing the operators. Subsection 4.2.3 showed how to choose adequate dimensions for $\boldsymbol{P}_{\text{block}}$ to get a sufficient arithmetic intensity. However, the number of writes in memory to store the intermediate tensor $\boldsymbol{P}$ was not considered.

To mitigate the overutilization of the memory bandwidth, a technique called kernel fusion can be quite efficient [11]. Here, we propose to fuse the computation of $\boldsymbol{P}_{\text{block}}$ and the sums on the diagonals of $\boldsymbol{P}$.

We can take advantage of the fact that a $\boldsymbol{P}_{\text{block}}$ is computed by a TB to store it on the shared memory first, then sum along its diagonals, and finally, write those sums back to main memory. By doing so, the storage space requirement and main memory bandwidth usage drop from $h_B w_B d_B$ to $(h_B + w_B - 1) d_B$.

The proposed algorithm is then modified to include this $\boldsymbol{P}_{\text{block}}$ partial reduction. The sum on diagonals is now split into two passes: the first one within the shared memory for a $\boldsymbol{P}_{\text{block}}$, then recombination of the partial sums from the different $\boldsymbol{P}_{\text{blocks}}$. The pseudo-code in Algorithm 2 summarizes this idea.

```
1  Function Im2TensorFused
       input  : An image I, a kernel K
       output: R the 2D valid convolution
2      K^T ← transpose(K)
3      begin CUDA Kernel
4          while any P_block remains do
5              b_idx ← getPBlockIdx (TB_idx)
6              K_block ← readLines (K^T, b_idx)
7              S_block ← readAndBuildSubTensor (I, b_idx)
8              P_block ← matTensorMultiply (K_block, S_block)
9              d_partial ← sumTensorDiagonals (P_block)
10             store (P_partial, d_partial)
11         end
12     end
13     begin CUDA Kernel
14         while any diagonal remains do
15             d^(k), d^(l) ← getDiagIds (Thread_idx)
16             s ← sumDiagonal (P_partial, d^(k), d^(l))
17             storePixel (R, s)
18         end
19     end
```

Algorithm 2: The fused im2tensor variant.

Finding an efficient data structure to hold the partial sums of a tensor is not straightforward. The perhaps simplest idea would be to store partial sums of a diagonal directly on the pixel of R it contributes to. This requires atomic sums for all thread blocks to work concurrently. In the following, this strategy is named *atomic*. On the one hand, it removes the need for an intermediate buffer. On the other hand, it makes the algorithm more sequential, hence, slower.

For better performance, we developed a method that does not rely on atomic operations. Let us present it in a simple case. A is a (h_A, w_A) matrix on which Tr_+ should be applied. A is partitioned by the submatrices $\{M^{(i,j)}\}$ of size (h_M, w_M) as shown in Figure 10.

$$A = \begin{bmatrix} M^{(0,0)} & \dots & M^{(0,w_A/w_M)} \\ \vdots & \ddots & \vdots \\ M^{(h_A/h_M,0)} & \dots & M^{(h_A/h_M,w_A/w_M)} \end{bmatrix} \tag{4.10}$$

Let us explore a method for computing sums on the $M^{(i,j)}$ concurrently. This would translate into each GPU TB being affected to a $M^{(i,j)}$ block in our implementation. To do so, a 2D buffer C is used to store partial sums computed in each submatrix. Each column of C is reserved for a diagonal of A. Each $M^{(i,j)}$ writes

its partial results on a row of C. In Figure 10, $M^{(0,1)}$ computes partial sums of A-diagonals $d^{(3)}$ to $d^{(9)}$. The results are written to $C_{0,3:9}$.

Reserving one row of C per B-block would waste a lot of space. We adopt two strategies to reduce the memory footprint needed for C. First, blocks on the same block-antidiagonal (i.e. $\forall n$, $\{M^{(i,j)}$, s.t. $i+j=n\}$) do not have any diagonal of A in common to sum. This way, they can safely use the same row C, as the columns they use don't overlap. In Figure 11, $M^{(1,1)}$ and $M^{(0,2)}$ use the third row of C.

Second, let's examine a block-antidiagonal's memory footprint on C: it is a segment of length $h_A - 1 + w_M \min(\frac{h_A}{h_M}, \frac{w_A}{w_M})$. Moreover, the gap in the diagonal indices treated by two consecutive block-antidiagonals is w_M. Thus, when two antidiagonals are sufficiently apart from each other, they may safely use the same row of C. With antidiagonals n and m (with $m > n$) this happens when:

$$(m-n)w_M > h_A - 1 + w_M \min\left(\frac{h_A}{h_M}, \frac{w_A}{w_M}\right) \tag{4.11}$$

$$m - n \geq \lceil \frac{h_A - 1}{w_M} \rceil + \min\left(\frac{h_A}{h_M}, \frac{w_A}{w_M}\right) \tag{4.12}$$

In Figure 11, this is shown with $M^{(0,0)}$ and $M^{(1,2)}$, from antidiagonals 0 and 3, using the same row of C.

With these two techniques, the required number of rows for C is reduced from $(\frac{h_A}{h_M}\frac{w_A}{w_M})$ to $\left(\lceil \frac{h_A - 1}{w_M} \rceil + \min(\frac{h_A}{h_M}, \frac{w_A}{w_M})\right)$.

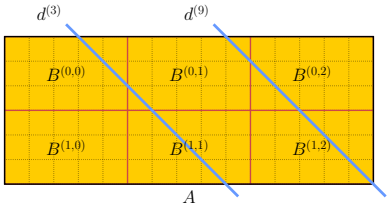


Fig. 10: The matrix A is divided into submatrices $\{M^{(i,j)}\}$. $M^{(0,1)}$ computes partial sums for $d^{(3)}$ to $d^{(9)}$.

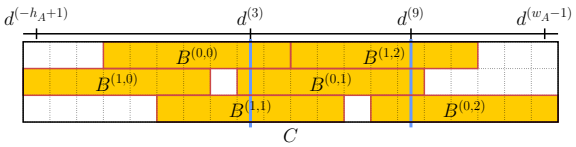


Fig. 11: Each $M^{(i,j)}$ writes its results to a segment of C. To get the full sum for $d^{(3)}$, one should add the partial sums from $M^{(0,0)}$, $M^{(0,1)}$, and $M^{(1,1)}$.

Compared to the initial algorithm, the fused version limits the memory footprint by avoiding the creation of $\boldsymbol{P}$. It also limits the main memory bandwidth through partial summations in the $\boldsymbol{P}_{\text{block}}$.

In a setting where the image is 1024×1024, the kernel 32×32 and $w_B = h_B = 32$, the fused algorithm reduces by $\sim 15\times$ its memory bandwidth.

5. Results.

5.1. Introduction.

5.1.1. Experimental setup. This section presents the results of several experiments. They were run in two environments, as detailed in Table 1. We are interested in two facets of a convolution's performance: the speed and the accuracy of its results.

	Machine #1: desktop	Machine #2: embedded (Jetson AGX Xavier)
OS	Ubuntu 16.04	Ubuntu 18.04
Linux Kernel	4.15.0	4.9.140
CUDA	11.0	10.2
NVIDIA Driver	450	JetPack 4.4
CPU	Intel i7-3820	8-core ARM 64bits
GPU	Titan V (arch. 7.0)	Xavier (arch. 7.2)
TDP	∼500W	∼30W

Table 1: Environments of the experiments.

For speed tests, we run the implementations on the *cameraman* image (see Figure 12), resized to (1024 × 1024) or (4096 × 4096) pixels. This setting has been chosen to mimic an industrial context where an image is to be preprocessed by a large Gaussian kernel.

We summarize the results by taking the median execution time over 20 runs. Our benchmark program measures performance with the help of *cudaEvents*. We do not account for memory management and data transfers in the reported timings. The assumption is that the image and kernel already live on the GPU in a real-world processing pipeline. Only the duration of processing the convolution is relevant.

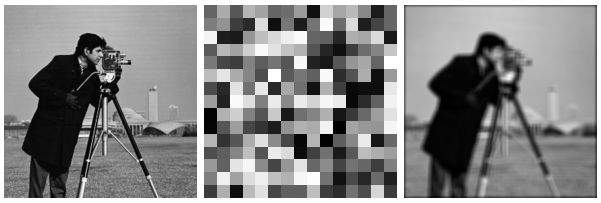


Fig. 12: Left, the original image. Middle, a 15×15 random kernel. Right, the *same* convolution.

For accuracy, we rely on the median absolute percentage error (median APE). For a pixel p_i, generated by the algorithm under test and compared to r_i, the pixel from the reference implementation, the APE is defined as:

$$\text{APE}(p_i) = \begin{cases} |\frac{p_i - r_i}{r_i}|, & \text{if } r_i \neq 0 \\ 0, & \text{if } r_i = 0 \end{cases} \tag{5.1}$$

We aggregate the results on 39 images from the *Miscellaneous* USC-SIPI dataset [29]. Kernels' coefficients are randomly chosen in the interval $[0, 1)$. We use scipy's *correlate2d* with float64 numbers to generate reference results. All images are also stored on disk in float64 precision using the FITS format.

When the precision used by an algorithm is less than float64, we first convert the image and kernel to the lower precision. All computations are done in the requested precision. Then, the result is promoted back to fp64 for storage and comparison.

We also consider mixed-precision: in fp16fp32, the algorithm accesses data in fp16, does the computation in fp32, and store the final result in fp16 (which will later be promoted to fp64 for file storage). For tensor core implementations, the situation is slightly more subtle: with fp16 input, tensor cores always use fp32 internal registers for intermediate results (see Figure 13). In what we call fp16 implementation, an fp16 output is requested from the tensor core. For fp16fp32, we use fp32 results from the tensor core.

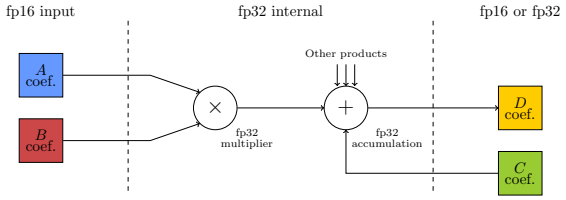


Fig. 13: Internal decomposition of a tensor core operation.
From: https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/.

The exclusive mode is requested for the GPU under test, which means that no other programs will interfere with its execution. Moreover, the GPU clocks are set to fixed values to mitigate dynamic frequency scaling effects.

5.1.2. Implementations Under Test. In this benchmark, we try to cover a broad range of algorithms. We, however, restricted ourselves to implementations that are not optimized for a specific kernel size. This makes the comparison fairer. Those algorithms require a dedicated program for every kernel shape. They may reach 2-5× speedups compared to other algorithms but need a longer compilation time and make the final binary heavy for a general-purpose library.

We compared two families of in-house implementations ("naive", *im2tensor*) with first-party NVIDIA libraries (CUFFT, CUDNN, NPP) and a third-party library (ArrayFire). Let us give a brief description of each algorithm:

im2tensor. This is the algorithm explained in this article. The *+ shmem* versions use the shared memory for efficient reuse of $\boldsymbol{S}_{block}$ and K_{block}. The *+ fused* versions use the optimization explained in subsection 4.2.4. Finally, the *via CUBLAS* version builds the $\boldsymbol{S}$ tensor explicitly and uses CUBLAS to perform matrix multiplications between K^T and the slices of $\boldsymbol{S}$.

"Naive". This is the classical approach for convolutions on GPU: each GPU thread is assigned to computing a resulting pixel. Therefore, each thread loops over the kernel and image pixels to multiply and sum. In the *+ shmem* version, threads in the same thread block use the shared memory to reuse image pixels. Most of the code is inspired by CUDA samples [24].

CUFFT. This algorithm performs convolutions in the Fourier domain. The time to do the Fourier transform of the kernel is not counted, as it could easily be precomputed and stored in a real-world application. What counts for this implementation is the Fourier transform of the image, the pointwise multiplication in the Fourier domain, and the inverse Fourier transform.

CUDNN. This library, used for deep neural networks, features the *im2col* algorithm [8]. We used "cudnnConvolutionForward" with the "CUDNN_CONVOLUTION_FWD_PREFER_FASTEST" setting on version 7.6.5.

NPP. NPP are NVIDIA Performance Primitives. They contain many utility functions for signal and image processing.

ArrayFire. This general-purpose GPU-accelerated library features convolution implementations based on Fourier transform (ArrayFire Freq.) or similar to naive + shmem (ArrayFire Spatial) [33].

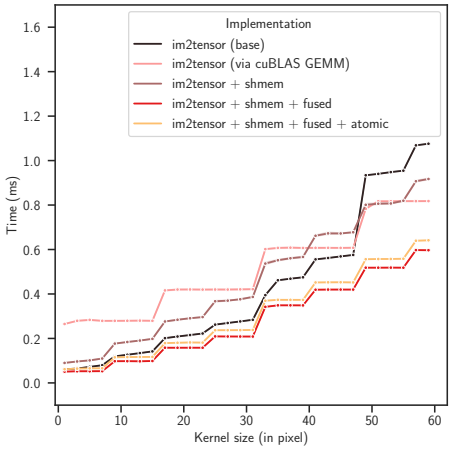


Fig. 14: Effects of im2tensor (*fp16)* optimizations on Titan V on 1024 × 1024 images.

5.2. Performance. Figure 14 shows the results for different implementations of the im2tensor algorithm. The CUBLAS version creates the whole $\boldsymbol{S}$ tensor before using the GEMM CUBLAS implementation for multiple matrix-matrix multiplications, as explained in subsection 5.1.2. Because the time to create $\boldsymbol{S}$ is not counted, it only serves as a reference for the other implementations. Aside from that, even if the CUBLAS library is highly optimized for GEMMs, it is still penalized, with respect to other implementations, by the large data movement that results from fetching $\boldsymbol{S}$ entirely.

The base im2tensor algorithm already achieves satisfying results. Nevertheless, we applied the optimizations discussed previously. The *shmem* version performs slightly worse, we suppose it is mainly caused by bank conflicts in the shared memory and to the L1 cache being already as efficient as using shared memory.

By adding the *fused* optimization, though, the initial algorithm is outperformed. This confirms the efficiency of reusing data as much as possible once they have been moved to the thread block. At last, the additional *atomic* optimization slows down the runtime slightly, but does not rely on any intermediate buffer.

In Figure 15, the *im2tensor* algorithm is compared with other *fp16* implementations. CUDNN proves to be inadequate in the single-kernel and single-image setting. Naive implementations perform well for relatively small kernels. The usage of shared memory is beneficial for larger kernels. Note that both programs use the vectorized *half2* data type to maximize compute-throughput. The Fourier implementation,

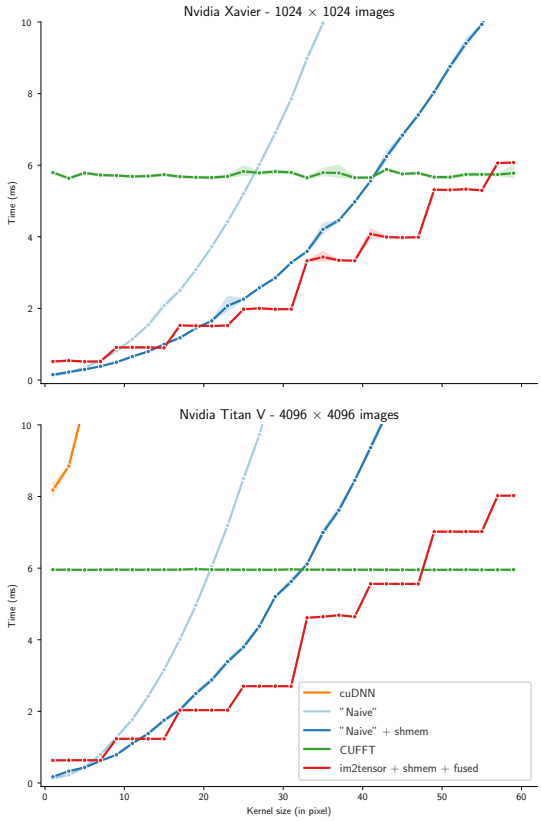


Fig. 15: Comparison of *fp16* algorithms in two contexts: (1024 × 1024) images on Jetson Xavier and (4096 × 4096) images on Titan V. Bands represent the 95% confidence interval.

CUFFT, is almost constant with respect to the kernel size. The overhead for small kernels is prohibitive but becomes less of a problem with large kernels (> 40 pixels).

Our algorithm behaves quite well for all kernel sizes. On small kernels, it is on par with the best implementations. The overhead due to padding for tensor cores does not allow it to be the fastest. As the kernel grows in size, the im2tensor execution time curve is less steep than "naive" implementations. Thus, it is the fastest for kernels between 15 and ~ 50 pixels in size.

For very large kernels, CUFFT remains the fastest. This seems sensible, as the algorithmic complexity is asymptotically better for convolutions in the Fourier space rather than in the direct space.

Table 2 provides timings for various implementations and several precisions. It shows once again that im2tensor is the fastest method for most kernel sizes. Our algorithm also performs well in the mixed fp16fp32 case. In fp32 and fp64, note that *ArrayFire (Spatial)* cannot handle large kernels. When that happens, the result is marked "—".

Implementation	Precision	Kernel Size				
		3	15	25	35	55
"Naive"	fp16	**0.21**	3.11	8.37	16.17	41.27
"Naive" + shmem		0.31	1.64	3.71	7.43	16.48
CUFFT		5.95	5.96	6.08	5.95	**5.96**
im2tensor + shmem + fused		0.63	**1.23**	**2.70**	**4.64**	7.02
im2tensor + shmem + fused + atomic		0.82	1.50	3.04	4.86	7.24
"Naive"	fp16fp32	**0.39**	6.62	17.91	34.72	84.96
"Naive" + shmem		0.47	3.06	7.21	13.55	47.06
im2tensor + shmem + fused		1.07	**2.30**	**4.70**	**8.37**	**12.55**
im2tensor + shmem + fused + atomic		1.44	2.86	5.73	9.51	14.04
"Naive"	fp32	0.41	6.16	16.71	32.31	168.22
"Naive" + shmem		0.51	**2.47**	**6.38**	14.78	38.12
ArrayFire (Freq.)		10.79	10.79	10.78	10.79	10.79
ArrayFire (Spatial)		0.40	2.68	—	—	—
CUFFT		9.65	9.64	9.64	**9.64**	**9.64**
NPP		**0.25**	2.74	7.18	14.01	34.34
"Naive"	fp64	0.52	6.46	17.53	59.79	214.71
"Naive" + shmem		0.69	3.21	**10.82**	**14.89**	45.86
ArrayFire (Freq.)		21.51	21.51	21.51	21.52	**21.52**
ArrayFire (Spatial)		0.56	**2.84**	—	—	—
CUFFT		24.59	24.59	24.58	24.59	24.59
NPP		**0.47**	5.71	14.74	28.46	70.60

Table 2: Median execution time (in ms) on 4096×4096 images vs. size of kernel. Best time per category is highlighted.

5.3. Accuracy. The previous section highlighted the difference in speed across floating-point formats. There is, however, a tradeoff between speed and accuracy when it comes to float operations. Figure 16 compares some implementations' accuracy. The results are averaged over the whole USC-SIPI (Misc) database, with 95% confidence bands. The naive implementation grows from 0.02% to about 3% for kernels from 3 to 60. It means that for a large kernel, you can expect a 3% inaccuracy for each pixel.

This inaccuracy might be too large in some contexts [26]. Fortunately, other implementations perform better. CUFFT is almost constant at 0.1%. im2tensor algorithms in fp16 reach a constant 0.02% inaccuracy, whatever the kernel size. Finally, the performance of the "naive" algorithm meets those of im2tensor when it is executed in fp32fp16 mixed precision.

The remarkable performance of im2tensor is explained by the use of tensor cores. As Figure 13 showed, even in fp16 precision, the intermediate results of the tensor core are computed with fp32 numbers [14, 1]. Given that the accuracy of im2tensor (fp16) is about the same as the mixed (fp16fp32) versions of "naive" and im2tensor, we can deduce that the accuracy is further limited by the storage type rather than the type used for intermediate computations.

For reference, accuracy results are included for higher precision in Table 3. In fp32, the trend is the same, with the "naive" growing with the kernel size. Most results stay within a $10^{-5}, 10^{-6}$% accuracy.

In fp64, the "naive" implementation is bit-accurate, hence the 0 precision. This is due to the reference scipy version making the same sequence of operations to compute the convolution. Other implementations are very close, about 10^{-14}%. Since the reference implementation also cannot be perfectly precise, it is hard to conclude for an implementation to be more accurate than another in fp64.

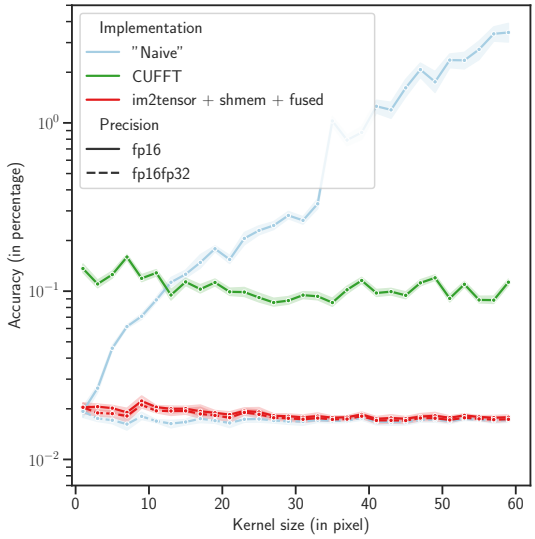


Fig. 16: Accuracy of algorithms in *fp16* or *fp16fp32*. Lower is better.

Implementation	Precision	Kernel Size				
		3	15	25	35	55
"Naive"	fp16	2.73e-02	1.21e-01	2.08e-01	1.04	2.91
CUFFT		1.06e-01	1.12e-01	8.75e-02	8.37e-02	8.57e-02
im2tensor + shmem + fused		2.09e-02	2.03e-02	1.83e-02	1.77e-02	1.78e-02
"Naive"	fp16fp32	1.76e-02	1.69e-02	1.69e-02	1.69e-02	1.70e-02
im2tensor + shmem + fused		1.90e-02	1.97e-02	1.78e-02	1.72e-02	1.72e-02
"Naive"	fp32	3.54e-06	1.48e-05	2.42e-05	3.39e-05	5.30e-05
ArrayFire (Freq.)		2.55e-05	2.89e-05	2.51e-05	2.70e-05	2.76e-05
ArrayFire (Spacial)		3.61e-06	1.48e-05	—	—	—
CUFFT		1.98e-05	1.82e-05	1.99e-05	1.93e-05	1.80e-05
NPP		3.63e-06	1.48e-05	2.42e-05	3.39e-05	5.30e-05
"Naive"	fp64	0	0	0	0	0
ArrayFire (Freq.)		1.93e-14	3.40e-14	4.96e-14	6.59e-14	1.02e-13
ArrayFire (Spacial)		1.11e-14	3.83e-14	—	—	—
CUFFT		2.11e-14	3.38e-14	4.91e-14	6.59e-14	1.02e-13
NPP		1.11e-14	3.83e-14	6.29e-14	8.76e-14	1.37e-13

Table 3: Median accuracy (in percentage) of convolutions vs. size of kernel.

6. Conclusion. In this article, we have proposed a new algorithm for 2D convolutions, *im2tensor*, that uses GPU tensor cores. These are NVIDIA units dedicated to matrix multiplications that increase the compute throughput of new GPUs.

We conducted an analysis of the algorithm in terms of algorithmic complexity and arithmetic intensity. This helped us make the best choice of parameters for the implementation of our algorithm that is based on matrix-tensor multiplication.

To prove the relevance of this new method, we have benchmarked several well-known implementations on GPUs. For completeness, we compared in-house implementations with first and third-party libraries. The effects of floating-point precision on the accuracy of the computation were also reviewed.

We evaluated those methods on two different setups: embedded ($\sim$ 30W) with an NVIDIA Jetson Xavier and desktop ($\sim$ 500W) with an NVIDIA Titan V. Based on our experiments, it appears that our optimized method for computing convolution via matrix-tensor multiplication with tensor cores is competitive for a large range of kernel sizes.

For small kernels ($\leq$ 20-pixel wide), it is on par with shared memory "naive" implementations and 10$\times$ faster than Fourier transforms. For large kernels ($\sim$ 50 pixels), our method is as fast as Fourier transforms and 2$\times$ faster than shared-memory "naive".

Regarding accuracy, compared to other fp16-only methods, our algorithm is 5$\times$ more precise than Fourier transforms and 100$\times$ as good as "naive" implementation for large kernels. This gain is directly provided by tensor cores, as they use extended-precision intermediate registers.

REFERENCES

[1] A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, M. Gates, T. Grützmacher, N. J. Higham, S. Li, N. Lindquist, Y. Liu, J. Loe, P. Luszczek, P. Nayak, S. Pranesh, S. Rajamanickam, T. Ribizel, B. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. M. Tsai, I. Yamazaki, and U. M. Yang, *A Survey of Numerical Methods Utilizing Mixed Precision Arithmetic*, arXiv:2007.06674 [cs, math], (2020), https://arxiv.org/abs/2007.06674.

[2] K. Adámek, S. Dimoudi, M. Giles, and W. Armour, *GPU Fast Convolution via the Overlap-and-Save Method in Shared Memory*, arXiv:1910.01972 [cs], (2020), https://arxiv.org/abs/1910.01972.

[3] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, *Low-memory GEMM-based convolution algorithms for deep neural networks*, arXiv:1709.03395 [cs], (2017), https://arxiv.org/abs/1709.03395.

[4] B. Barabasz, A. Anderson, and D. Gregg, *Improving The Accuracy of Winograd Convolution for Deep Neural Networks*, (2018), p. 18.

[5] P. M. Basso, F. F. dos Santos, and P. Rech, *Impact of Tensor Cores and Mixed Precision on the Reliability of Matrix Multiplication in GPUs*, IEEE Transactions on Nuclear Science, 67 (2020), pp. 1560–1565, https://doi.org/10.1109/TNS.2020.2977583.

[6] S. G. Bhaskaracharya, J. Demouth, and V. Grover, *Automatic Kernel Generation for Volta Tensor Cores*, arXiv:2006.12645 [cs], (2020), https://arxiv.org/abs/2006.12645.

[7] R. Brunelli, *Template Matching Techniques in Computer Vision: Theory and Practice*, John Wiley & Sons, Apr. 2009.

[8] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, *cuDNN: Efficient Primitives for Deep Learning*, arXiv:1410.0759 [cs], (2014), https://arxiv.org/abs/1410.0759.

[9] A. Dakkak, C. Li, I. Gelado, J. Xiong, and W.-m. Hwu, *Accelerating Reduction and Scan Using Tensor Core Units*, Proceedings of the ACM International Conference on Supercomputing, (2019), pp. 46–57, https://doi.org/10.1145/3330345.3331057, https://arxiv.org/abs/1811.09736.

[10] N. Ding and S. Williams, *An Instruction Roofline Model for GPUs*, in 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), Nov. 2019, pp. 7–18, https://doi.org/10.1109/PMBS49563.2019.00007.

[11] J. Filipovic and S. Benkner, *OpenCL Kernel Fusion for GPU, Xeon Phi and CPU*, in 2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Oct. 2015, pp. 98–105, https://doi.org/10.1109/SBAC-PAD.2015.29.

[12] J. S. Firoz, A. Li, J. Li, and K. Barker, *On the Feasibility of Using Reduced-Precision Tensor Core Operations for Graph Analytics*, in 2020 IEEE High Performance Extreme Computing Conference (HPEC), Sept. 2020, pp. 1–7, https://doi.org/10.1109/HPEC43674.2020.9286152.

[13] A. González, *Trends in Processor Architecture*, in Harnessing Performance Variability in Embedded and High-Performance Many/Multi-Core Platforms: A Cross-Layer Approach, W. Fornaciari and D. Soudris, eds., Springer International Publishing, Cham, 2019, pp. 23–42, https://doi.org/10.1007/978-3-319-91962-12.

[14] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, *Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers*, in SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, Nov. 2018, pp. 603–613, https://doi.org/10.1109/SC.2018.00050.

[15] M. Khairy, A. G. Wassal, and M. Zahran, *A survey of architectural approaches for improving GPGPU performance, programmability and heterogeneity*, Journal of Parallel and Distributed Computing, 127 (2019), pp. 65–88, https://doi.org/10.1016/j.jpdc.2018.11.012.

[16] D. B. Kirk and W. H. Wen-Mei, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan kaufmann, 2016.

[17] A. Lavin and S. Gray, *Fast Algorithms for Convolutional Neural Networks*, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 4013–4021.

[18] D. Mukunoki, K. Ozaki, T. Ogita, and T. Imamura, *DGEMM Using Tensor Cores, and Its Accurate and Reproducible Versions*, in High Performance Computing, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, eds., vol. 12151, Springer International Publishing, Cham, 2020, pp. 230–248, https://doi.org/10.1007/978-3-030-50743-512.

[19] C. A. Navarro, R. Carrasco, R. J. Barrientos, J. A. Riquelme, and R. Vega, *GPU Tensor Cores for fast Arithmetic Reductions*, arXiv:2001.05585 [cs], (2020), https://arxiv.org/abs/2001.05585.

[20] M. Nourazar and B. Goossens, *Accelerating iterative CT reconstruction algorithms using Tensor Cores*, Journal of Real-Time Image Processing, (2021), https://doi.org/10.1007/s11554-020-01069-5.

[21] NVIDIA, *V100 GPU Architecture: The world's most advanced datacenter GPU*, tech. report, Tech. Rep., NVIDIA, 2017.

[22] NVIDIA, *NVIDIA Turing GPU Architecture: Graphics Reinvented*, tech. report, Tech. Rep., NVIDIA, 2018.

[23] NVIDIA, *NVIDIA A100 Tensor Core GPU Architecture: Unprecedented Acceleration at Every Scale*, tech. report, Tech. Rep., NVIDIA, 2020.

[24] V. Podlozhnyuk, *CUDA Samples Documentation: convolutionSeparable*, tech. report, Tech. Rep., NVIDIA, 2007.

[25] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, 2010.

[26] M. Seznec, N. Gac, A. Ferrari, and F. Orieux, *A Study on Convolution using Half-Precision Floating-Point Numbers on GPU for Radio Astronomy Deconvolution*, in 2018 IEEE International Workshop on Signal Processing Systems (SiPS), Cape Town, Oct. 2018, IEEE, pp. 170–175, https://doi.org/10.1109/SiPS.2018.8598342.

[27] S. Sioutas, S. Stuijk, T. Basten, L. Somers, and H. Corporaal, *Programming tensor cores from an image processing DSL*, in Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems, SCOPES '20, New York, NY, USA, May 2020, Association for Computing Machinery, pp. 36–41, https://doi.org/10.1145/3378678.3391880.

[28] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing, 1997.

[29] A. G. Weber, *The USC-SIPI image database version 5*, USC-SIPI Report, 315 (1997).

[30] S. Williams, A. Waterman, and D. Patterson, *Roofline: An insightful visual performance model for multicore architectures*, Communications of the ACM, 52 (2009), pp. 65–76, https://doi.org/10.1145/1498765.1498785.

[31] H. Winnemöller, J. E. Kyprianidis, and S. C. Olsen, *XDoG: An eXtended difference-of-Gaussians compendium including advanced image stylization*, Computers & Graphics, 36 (2012), pp. 740–753, https://doi.org/10.1016/j.cag.2012.03.004.

[32] S. Winograd, *Arithmetic Complexity of Computations*, SIAM, Jan. 1980.

[33] P. Yalamanchili, U. Arshad, Z. Mohammed, P. Garigipati, P. Entschev, B. Kloppenborg, J. Malcolm, and J. Melonakos, *ArrayFire - A High Performance Software Library for Parallel Computing with an Easy-to-Use API*, AccelerEyes, Atlanta, 2015.

2.3 Conclusion

This chapter has performed two optimization strategies for efficient image convolutions on GPUs. The first, based on a radio-astronomy reconstruction, corresponds to our proposed development strategy's combined algorithm/implementation stage. It evaluates the possibility of performing the gradient descent algorithm on low precision floats. It studies several algorithmic strategies for taking into account the loss of precision with back-tracking, optimal or fixed step for the gradient descent. Finally, this algorithmic and optimization exploration concludes that using mixed precision (`fp16` for storage, `fp32` for computing) is a good trade-off. Acceleration comes from the reduced memory bandwidth, and the accuracy degradation is sufficiently low, so the gradient descent converges.

In a second time, the *im2tensor* article shows work for the implementation phase. The algorithmic context is fixed, and only the convolution operation is under optimization. Regarding accuracy, we used tensor cores in `fp16` precision. The inaccuracy added by the reduced format is nuanced by the fact that intermediate registers used by the tensor are in `fp32`. In the end, our algorithm, *im2tensor* is faster than other algorithms in the spatial domain for large kernels. Implementations based on Fourier transforms are still better for even larger kernels (> 50-pixel wide). Still, for a sweet spot at around 30-pixel wide, our algorithm is two times faster than other methods.

In the end, this work on image convolution in several contexts shows various ways to exploit the specificities of GPUs. For radio-astronomy, it showed multiple ways of taking into account accuracy loss in the gradient descent step choice. For tensor cores, performance gains are achieved with a new hardware feature, originally designed for DNN acceleration. These results show the importance of phases 2 and 3 of the implementation method for maximum performance on GPU.

Chapter 3

Implementation strategy for variational optical flow estimation

This chapter relates the method we followed for the implemenation of an optical flow estimation algorithm on GPU. The context stems for Thales that develops data processing solutions to be embedded in real-world situations. The deployment of such algorithms must fit tight weight, size, and power constraints. In these situations, embedded GPUs like the Jetson Xavier, on fig. 3.1, are often a tool of choice thanks to their high performance per watt ratio. In our case, the goal is to run an optical flow estimation algorithm, CLG (Bruhn et al., 2005), on this specific hardware.

The initial algorithm being fixed, we start our analysis at the end of the algorithm phase of fig. 1.2. The second stage consists of adapting the CLG algorithm to GPUs by chosing a solver that have fast hardware execution and good algorithmic properties, i.e. converges quickly. We also push the work to the third phase,

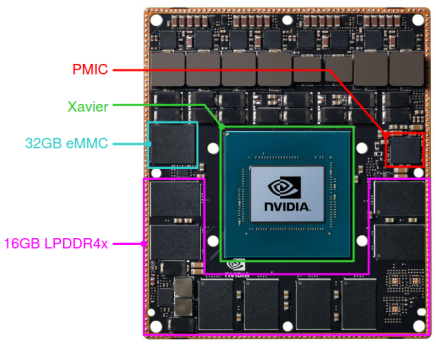


Figure 3.1: The Xavier chip on its compute module. Credits: Nvidia.

where we find kernel-level optimizations on GPU.

3.1 Real-Time Optical Flow Processing on Embedded GPU: an Hardware-Aware Algorithm to Implementation Strategy, *under review*

Similar to the radio-astronomy reconstruction, the optical flow estimation consists of an iterative solver that generates successive approximations of the optical flow. This time, we analyze the performance of different linear solvers on GPU. The time to solution of a solver is dictated by two factors, the convergence rate per iteration and the time per iteration. The first is usually well-studied in the literature and invariant with respect to the hardware. On the contrary, the second differs on each execution platform

Our work begins by comparing widely available solvers. With third-party implementations, we evaluate the convergence rate per iteration and discard irrelevant algorithms on CPUs first. Then, we implement a selection of solvers on GPU and choose the best one. We also check the influence of hyper-parameters on time to convergence. This estimation allows users of the optical flow to tune the algorithm based on real-time constraints.

Once the solver is chosen, it is time for lower-level optimizations. In section 3.1, we use memory re-utilization to avoid buffer copies, iteration fusion of the linear solver to increase the arithmetic intensity, and kernel batching to limit the overhead of CUDA functions launches.

Journal of Real-Time Image Processing manuscript No.
(will be inserted by the editor)

Real-Time Optical Flow Processing on Embedded GPU: an Hardware-Aware Algorithm to Implementation Strategy

Mickaël Seznec · Nicolas Gac · François Orieux · Alvin Sashala Naik

Received: date / Accepted: date

Abstract Determining the optical flow of a video is a compute-intensive task essential for computer vision. For achieving this processing in real-time, the whole algorithm deployment chain must be thought of for efficiency first. The development is usually divided into two parts: first, designing an algorithm that meets precision constraints, then, implementing and optimizing its execution on the targeted platform. We argue that unifying those operations enhances performance on the embedded processor.

This paper is based on an industrial use case of computer vision. The objective is to determine dense optical flow in real-time on an embedded GPU platform: the Nvidia AGX Xavier. The CLG (Combined Local-Global) optical flow method, initially chosen, is analyzed to understand the convergence speed of its underlying optimization problem. The *Jacobi* solver is selected for implementation because of its parallel nature. The whole multi-level processing is then ported to the GPU, using several specific optimization strategies. In particular, we analyze the impact of fusing the solver's iterations with the roofline model.

As a result, with a 30W power budget, our implementation runs at 60FPS, on 640×512 images, with a four-level processing. This example should hopefully provide feedback on the issues that arise when trying to port a method to a parallel platform and serves for further implementations of computer vision algorithms on specialized hardware.

Mickaël Seznec · Alvin Sashala Naik
Thales Research and Technology. Palaiseau, France

Mickaël Seznec · Nicolas Gac · François Orieux
Laboratoire des Signaux et Systèmes, Université Paris-Saclay, CNRS, CentraleSupélec. Gif-Sur-Yvette, France

Keywords Algorithm design, Optical Flow, GPU Optimization, Linear Solvers, Image Processing

1 Introduction

Computer vision has become an essential aspect of widely adopted electronic devices in various fields: medicine [7], unmanned flight [13], or autonomous driving [5], for instance. The constant progress of these applications is driven by more sophisticated algorithms and more efficient hardware architectures. As both of these fields continue to progress, the difficulty of finding an optimal match between the two increases.

On the one hand, the algorithm design space of image processing methods is broad. New techniques are constantly developed that often depend on hyper-

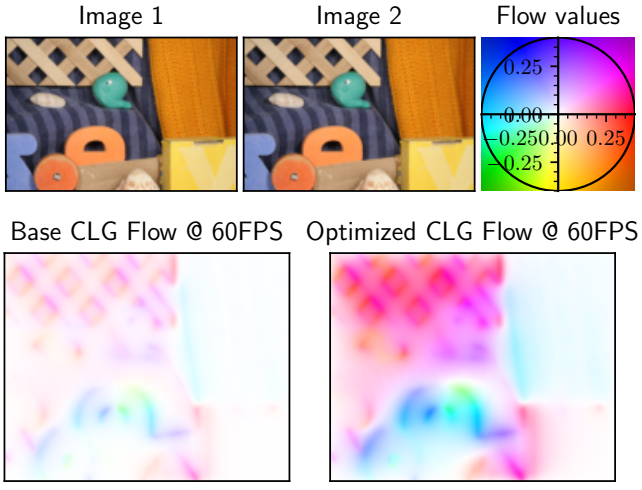


Fig. 1: For the same framerate on Jetson Xavier, our GPU-optimized multi-scale CLG Optical Flow converges further than the initial implementation.

parameters to control a trade-off between the speed and accuracy of the results. On the other hand, modern hardware architectures such as GPUs (Graphics Processing Units), FPGAs (Field-Programmable Gate Array), or SIMD (Single-Instruction Multiple-Data) processors have successfully improved the execution of vision algorithms. The increasing complexity in both of these domains calls for expertise that keeps being more and more specific. It is then challenging to combine these two skills to find an optimal match between the algorithm and the target.

In this article, we focus on the optical flow problem. The goal is, given two successive frames of a video, to find a per-pixel displacement vector. First numerical methods to solve it have been found by Horn and Schunk in the 1980s [10] and numerous refinements have been developed since [8,3]. For our analysis, we select the CLG (Combined Local-Global) method [4] as it is the basis of one of our industrial applications.

Our analysis then serves two goals. First, finding the impact of the solver choice and the values of hyper-parameters on the speed and accuracy of the CLG method. This initial study gives rise to an initial implementation on the NVIDIA Jetson AGX Xavier, an embedded GPU SOC (System On Chip). The next goal is finding efficient optimization procedures for this algorithm to achieve maximum performance. Overall, the study aims at finding algorithm-implementation synergies through the perspective of optical flow processing.

The main novelties brought by this article are listed below.

- It extends previous work [17] on the influence on speed and accuracy of the hyper-parameters of the CLG optical flow. Notably, the spectral radiuses of splitting solvers are provided, and new performance results on the Xavier GPU are presented.
- It introduces a complete implementation of the algorithm on the Jetson AGX Xavier, optimized in-depth with diverse techniques: buffer re-utilization, solver iteration fusion, and kernel launches batching.
- It analyses the impact of the multi-scale scheme on the performance of our implementation.

The rest of this article is structured as follows: section 2 outlines related work on optical flow processing for real-time systems and optimization strategies for parallel systems. Section 3 introduces mathematical notations for optical flow and analyzes solvers and hyper-parameters on the convergence speed. Section 4 deals with the implementation optimizations on GPU and focuses on arithmetic intensity to explain achieved performance. Section 5 concludes this paper and gives direction for further work.

2 Related Work

Optical flow has received a lot of attention since pioneering numerical methods introduced by Horn and Schunk [10] and Lucas & Kanade [12]. From there, many refinements have been incorporated on top of these frameworks. Review papers [2,19] explore comprehensively the different strategies used for computing optical flow.

In this article, we focus on a differential method, a family that was introduced by Horn & Schunk. It consists of minimizing a penalization function usually composed of two types of terms: model attach and regularization. On top of the original penalization function found in [10], Farnebäck *et al.* replace the linear interpolation with a quadratic one for better accuracy [8]. Brox *et al.* add a gradient conservation term [3] while Zach *et al.* use a L1-norm penalization instead of a quadratic one [22] to obtain better-defined object boundaries. The selected algorithm for our study is the CLG (Combined Local-Global) method, as defined by Bruhn *et al.* [4]. This method adds a neighboring condition to the model attach term, similar to the one found in [12]. This unifying model is less sensitive to noise, as the local information is averaged over multiple pixels. Furthermore, the method does not require many more operations than the traditional Horn & Schunk approach.

Efficient implementation has always been key to an attractive optical flow method. For CLG, a CPU implementation has been described in [11] and Moussu [14] detailed its GPU counterpart. With respect to this previous work, our article details how to choose the right solver and hyper-parameters of CLG for fast convergence. It is completed by GPU optimizations, especially for the Jacobi solver.

There is plenty of literature about GPU optimization for linear algebra. Kernel fusion is a frequent technique, manually applied to a sparse CG (Conjugate Gradient) solver in [1], or BCG (Biconjugate-CG) in [20]. Filipovic *et al.* propose a source-to-source compiler to perform fusion at the compilation stage [9]. Regarding the Jacobi solver specifically, Aslam *et al.* have benchmarked many computations and synchronization techniques. We differ from this work by not relying on sparse matrices to implement the Jacobi solver but by implementing the operators defined by those matrices directly. In [15], Nguyen *et al.* compare several GPU solvers for fastest convergence and comes to similar conclusions as ours: more iterations on simpler solvers are more efficient on GPUs.

To guide our optimization strategy, we rely on the roofline model, as introduced by Williams in [21]. It is

a general and a powerful tool to find bottlenecks in an application, that has already been used for GPUs [6].

3 Method-level approach

In this section, we examine the CLG algorithm from a mathematical perspective. This first analysis serves our optimization method by highlighting the degrees of freedom allowed by our application. After giving some mathematical context, we then explore the implications of the solver choice and the tuning of hyper-parameters.

3.1 Modeling optical flow

A category of optical flow algorithms provides a result by finding the solution to an optimization problem. In this section, we introduce the mathematical notations associated with this optimization problem.

The variables are named using the following convention: the lowercase $a \in \mathbb{R}$ is a coefficient, the bold lowercase $\boldsymbol{a} \in \mathbb{R}^n$ is a vector, the uppercase $A \in \mathbb{R}^{n\times m}$ is a matrix. The over-line symbol $\bar{a} \in \mathbb{R}^{I_h \times I_w}$ represents the field a over a two-dimensional image. Likewise, $\bar{\boldsymbol{a}}$ is a vector field, $\bar{A}$ is a matrix field. Finally, the double bar notation introduces *flattened* fields: $\bar{\bar{a}}$ is a two-dimensional field represented as a vector with a row-major convention.

In a sequence of images at time t, the optical flow at (x, y) is noted $\boldsymbol{w}_{x,y,t} = (u_{x,y,t}, v_{x,y,t}, 1)^T$. It has three components: $u_{x,y,t}$ and $v_{x,y,t}$ are the displacement in the x and y axis, respectively, with the time displacement equals 1. We also introduce $\boldsymbol{w}^* = (u_{x,y,t}, v_{x,y,t})^T$, for ease of notation.

Finally, $f_{x,y,t}$ represents the pixel intensity of the frame at time t, at coordinates x, y. Images are gray-scale, so $f_{x,y,t}$ is a scalar. Later in the article, and for the sake of brevity, we may omit the x, y, t indices, so $\boldsymbol{w}_{x,y,t}$ becomes $\boldsymbol{w}$, for example.

With the variables now set, we present an energy definition that serves as a framework for many variational methods

$$E(\bar{\boldsymbol{w}}) = \int_\Omega D(\boldsymbol{w}, \bar{f}) + R(\boldsymbol{w}) \,\mathrm{d}x\,\mathrm{d}y \tag{1}$$

where D is the data-fitting term while R plays the role of regularization, and Ω represents the 2D image domain.

For example, in [10], Horn & Schunck set D and R to

$$D_{\mathrm{HS}}(\boldsymbol{w}, \bar{f}) := \boldsymbol{w}^T J_0 \boldsymbol{w} \tag{2}$$

and

$$R_{\mathrm{HS}}(\boldsymbol{w}) := \alpha\Big(\|\nabla_{x,y} u_{x,y}\|^2 + \|\nabla_{x,y} v_{x,y}\|^2\Big), \tag{3}$$

where $\nabla_{x,y}$ is a two-dimensional spatial gradient and $\alpha \in \mathbb{R}^+$ is the trade-off between data fitting and regularization penalization. $\bar{J}_0$ is a matrix field and corresponds to a quadratic penalization of the image intensity conservation, eq. (4), with a linear approximation, eq. (5) of the image's values

$$\begin{aligned}
&\|\bar{f}(x+u, y+v, t+1) - \bar{f}(x, y, t)\|^2 && (4)\\
&\quad \approx \|\bar{f}(x, y, t) + \nabla \bar{f}(x, y, t)^T \boldsymbol{w} - \bar{f}(x, y, t)\|^2 && (5)\\
&\quad = \|\nabla \bar{f}(x, y, t)^T \boldsymbol{w}\|^2 && (6)\\
&\quad = \boldsymbol{w}^T \nabla \bar{f}(x, y, t) \nabla \bar{f}(x, y, t)^T \boldsymbol{w} && (7)\\
&\quad = \boldsymbol{w}^T J_0 \boldsymbol{w}. && (8)
\end{aligned}$$

With this definition, the data-fitting term only incorporates pixel-wise intensity conservation. In [4], Bruhn *et al.* leverage the energy penalization found in [12] to average the intensity conservation over the pixel's neighborhood.

$$D_{\mathrm{Bruhn}}(\boldsymbol{w}, \bar{f}) := \boldsymbol{w}^T J_\rho \boldsymbol{w} \tag{9}$$

with

$$J_\rho = (K_\rho \circledast \bar{J}_0)(x, y, t) \quad \text{and} \quad \rho \in \mathbb{R}^+. \tag{10}$$

Here, K_ρ is a 2D Gaussian kernel with a standard deviation ρ, and $\circledast$ is a per-channel 2D-convolution operator applied to the matrix field $\bar{J}_0$. It means that the solution $\boldsymbol{w}$ should solve its intensity conservation equation and its neighbors'.

By replacing D by eq. (10) and R by eq. (3) in eq. (1), we have the CLG (Combined Local-Global) model, as defined in [4]

$$\begin{aligned}
E_{\mathrm{CLG}}(\bar{\boldsymbol{w}}) := \int_\Omega &\boldsymbol{w}^T J_\rho \boldsymbol{w}\\
&+ \alpha(\|\nabla_{x,y} u\|^2 + \|\nabla_{x,y} v\|^2)\,\mathrm{d}x\,\mathrm{d}y.
\end{aligned} \tag{11}$$

The convex optimization problem is now entirely defined. It is usually solved with an iterative gradient descent technique: each step yields a new approximate solution by displacing the current solution towards the opposite direction of the gradient. Two methods exist to compute the gradient of $E(\bar{\boldsymbol{w}})$: the first one considers $\bar{f}$ and $\bar{\boldsymbol{w}}$ to be continuous functions and employs the Euler-Lagrange equations. The second one discretizes

$\bar{f}$ and $\bar{\boldsymbol{w}}$ over the two-dimensional pixel grid first. This version is detailed in this article, with

$$E_{CLG}(\bar{\bar{\boldsymbol{w}}}^*) = \bar{\bar{\boldsymbol{w}}}^T H \bar{\bar{\boldsymbol{w}}} + \alpha\Big(\|D_x S_u \bar{\bar{\boldsymbol{w}}}^*\|^2 + \|D_y S_u \bar{\bar{\boldsymbol{w}}}^*\|^2 + \|D_x S_v \bar{\bar{\boldsymbol{w}}}^*\|^2 + \|D_y S_v \bar{\bar{\boldsymbol{w}}}^*\|^2\Big). \quad (12)$$

Equation (12) introduces $\bar{\bar{\boldsymbol{w}}}$, a $3 \times I_h \times I_w$ vector, such that $\bar{\bar{\boldsymbol{w}}}^T = \begin{bmatrix}\bar{u}^T & \bar{v}^T & \bar{\bar{1}}^T\end{bmatrix}$, similarly, $\bar{\bar{\boldsymbol{w}}}^{*T} = \begin{bmatrix}\bar{u}^T & \bar{v}^T\end{bmatrix}$. S_u and S_v are diagonal matrices that respectively select $\bar{u}$ and $\bar{v}$ parts of $\bar{\bar{\boldsymbol{w}}}$. D_x and D_y are discrete partial derivative operators along the x and y axes.

H is composed of diagonal matrices

$$H = \begin{bmatrix} \operatorname{diag} \bar{\bar{j}}_{\rho,0,0} & \operatorname{diag} \bar{\bar{j}}_{\rho,0,1} & \operatorname{diag} \bar{\bar{j}}_{\rho,0,2} \\ \operatorname{diag} \bar{\bar{j}}_{\rho,1,0} & \operatorname{diag} \bar{\bar{j}}_{\rho,1,1} & \operatorname{diag} \bar{\bar{j}}_{\rho,1,2} \\ \operatorname{diag} \bar{\bar{j}}_{\rho,2,0} & \operatorname{diag} \bar{\bar{j}}_{\rho,2,1} & \operatorname{diag} \bar{\bar{j}}_{\rho,2,2} \end{bmatrix}, \quad (13)$$

with

$$J_\rho = \begin{bmatrix} j_{\rho,0,0} & j_{\rho,0,1} & j_{\rho,0,2} \\ j_{\rho,1,0} & j_{\rho,1,1} & j_{\rho,1,2} \\ j_{\rho,2,0} & j_{\rho,2,1} & j_{\rho,2,2} \end{bmatrix}. \quad (14)$$

Let us now compute the derivative of eq. (12) with respect to $\bar{\bar{w}}^*$

$$\nabla_{\bar{\bar{\boldsymbol{w}}}^*} E_{\mathrm{CLG}}(\bar{\bar{\boldsymbol{w}}}^*) = 2 S_{u,v} H \bar{\bar{\boldsymbol{w}}} + 2\alpha\Big(S_u^T (D_x^T D_x + D_y^T D_y) S_u \bar{\bar{\boldsymbol{w}}}^* + S_v^T (D_x^T D_x + D_y^T D_y) S_v \bar{\bar{\boldsymbol{w}}}^*\Big) \quad (15)$$

$$S_{u,v} H \bar{\bar{\boldsymbol{w}}} = \begin{bmatrix} \operatorname{diag} \bar{\bar{j}}_{\rho,0,0} & \operatorname{diag} \bar{\bar{j}}_{\rho,0,1} \\ \operatorname{diag} \bar{\bar{j}}_{\rho,1,0} & \operatorname{diag} \bar{\bar{j}}_{\rho,1,1} \end{bmatrix} \bar{\bar{\boldsymbol{w}}}^* + \begin{bmatrix} \bar{\bar{j}}_{\rho,0,2} \\ \bar{\bar{j}}_{\rho,1,2} \end{bmatrix}. \quad (16)$$

The selection matrix $S_{u,v}$ is necessary as H is applied to the vector $\bar{\bar{\boldsymbol{w}}}$ that contains ones in addition to u's and v's.

Equation (15) should now be set to zero to find a minimizer of E_{CLG}. By doing so, we obtain an equation of the generic form

$$A\boldsymbol{x} = \boldsymbol{b} \quad (17)$$

where

$$A = \begin{bmatrix} \operatorname{diag} \bar{\bar{j}}_{\rho,0,0} & \operatorname{diag} \bar{\bar{j}}_{\rho,0,1} \\ \operatorname{diag} \bar{\bar{j}}_{\rho,1,0} & \operatorname{diag} \bar{\bar{j}}_{\rho,1,1} \end{bmatrix} - \alpha \begin{bmatrix} L & 0 \\ 0 & L \end{bmatrix} \quad (18)$$

with $L = D_x^T D_x + D_y^T D_y$, $\boldsymbol{x} = \bar{\bar{\boldsymbol{w}}}^*$, (19)

and $\boldsymbol{b} = -\begin{bmatrix} \bar{\bar{j}}_{\rho,0,2} \\ \bar{\bar{j}}_{\rho,1,2} \end{bmatrix}$ (20)

3.2 Solver Overview

The linear system of equations $A\boldsymbol{x} = \boldsymbol{b}$ can be solved in various ways. However, the characteristics of the optical flow setting restrict the choice of possible solvers. In a typical environment, with an HD image stream of dimensions 1280×480, there are over $2 \cdot 10^9$ coefficients in the matrix A. As is, an embedded system would never be able to store the whole matrix. Hopefully, the matrix is sparse, with more than 99.999% of its coefficients being zeros. It is then crucial to find a solver that takes advantage of this sparsity to make the computation possible on embedded devices.

The two following sections present two principal families of solvers for the optical flow. First, matrix splitting methods have been chosen in seminal work on flow estimation [10] and remain widely used to solve these linear systems [11]. Second, Krylov methods are often used for numerical simulations and benefit from a well-supplied scientific corpus [16].

3.2.1 Matrix Splitting

The matrix splitting methods partition the matrix A into two: $A = B + C$. Using this equality in $A\boldsymbol{x} = \boldsymbol{b}$ yields

$$B\boldsymbol{x} = \boldsymbol{b} - C\boldsymbol{x}. \quad (21)$$

Assuming B is invertible, an iterative scheme is constructed

$$\boldsymbol{x}^{k+1} = B^{-1}(\boldsymbol{b} - C\boldsymbol{x}^k) \quad (22)$$

$$\boldsymbol{x}^{k+1} = (I - B^{-1}A)\boldsymbol{x}^k + B^{-1}\boldsymbol{b}. \quad (23)$$

The choice of B leads to different methods. For example, choosing B to hold the diagonal of A: $B_{\mathrm{J}} := D_A$ is the Jacobi solver, while $B_{\mathrm{GS}} := D_A + L_A$ is the Gauss-Seidel method (with L_A, the lower triangular part of A).

In the case of optical flow problems, we can craft custom B matrices based on the structure of A. These variants contain the four non-empty diagonals of A

$$B_{\mathrm{J}}^{(\mathrm{diags})} := \begin{bmatrix} \operatorname{diag} \bar{\bar{j}}_{\rho,0,0} - \alpha D_L & \operatorname{diag} \bar{\bar{j}}_{\rho,0,1} \\ \operatorname{diag} \bar{\bar{j}}_{\rho,1,0} & \operatorname{diag} \bar{\bar{j}}_{\rho,1,1} - \alpha D_L \end{bmatrix} \quad (24)$$

$$B_{\mathrm{GS}}^{(\mathrm{diags})} := \begin{bmatrix} \operatorname{diag} \bar{\bar{j}}_{\rho,0,0} - \alpha L_L & \operatorname{diag} \bar{\bar{j}}_{\rho,0,1} \\ \operatorname{diag} \bar{\bar{j}}_{\rho,1,0} & \operatorname{diag} \bar{\bar{j}}_{\rho,1,1} - \alpha L_L \end{bmatrix}. \quad (25)$$

This construction of B matrices is called the pointwise-coupled method in [11], as these matrices update $u_{x,y}$ and $v_{x,y}$ simultaneously. Later in this article, we call these versions "preconditioned" by analogy with the Krylov methods.

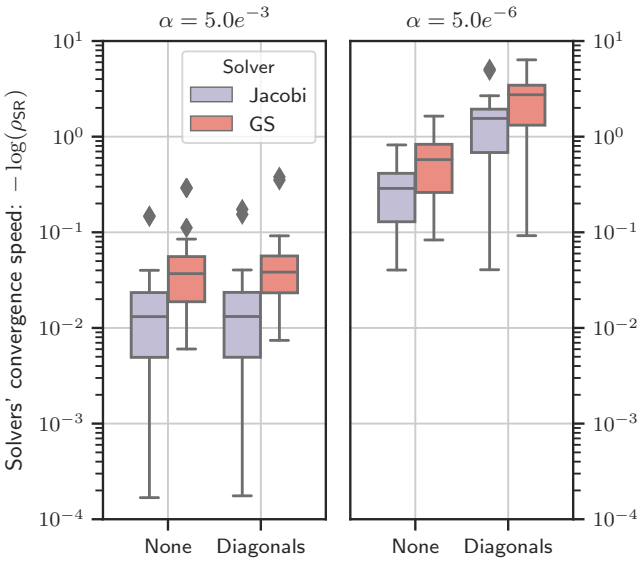


Fig. 2: Comparison of the theoretical convergence speed of the Jacobi and Gauss-Seidel (GS) methods. Standard solvers are refered by *None* and precontioned, or pointwise-coupled, variations with *Diagonals*.

The spectral radius ρ_{SR} of $I - B^{-1}A$ must be studied to show how the specially designed matrices compare to the traditional ones. At each iteration of the solver, the error's norm $\|\boldsymbol{e}^k\| = \|\boldsymbol{b} - A\boldsymbol{x}^k\|$ is multiplied by a factor ρ_{SR}. The end goal is then to find a matrix B such that the corresponding ρ_{SR} is as close to zero as possible.

Figure 2 presents results for the Jacobi and Gauss-Seidel solvers with their derived pointwise-coupled methods. The optical flow is analyzed under two parameter settings, with $\alpha = 5e^{-3}$ or $\alpha = 5e^{-6}$. The plot shows $-\log(\rho_{\text{SR}})$ for easier comparison between solvers. Note that finding ρ_{SR} is a computationally heavy task, so images have been cropped to obtain such results.

We can draw three conclusions from fig. 2. First, a low alpha dramatically increases the convergence speed for all types of solvers by factors of $20 \sim 100$. Second, with the same flow parameters, Gauss-Seidel is about three times faster than Jacobi. Last, the pointwise-coupled method is useful only in a low alpha setting where a $5\times$ speedup is achieved.

3.2.2 Krylov's methods

Krylov solvers all emerge from the same premise: at each iteration, increase the possible solutions' space's dimension. Such spaces, called Krylov spaces, are defined by

$$\mathcal{K}_n(A, \boldsymbol{b}) = \text{span}\{\boldsymbol{b}, A\boldsymbol{b}, A^2\boldsymbol{b}, \dots, A^{n-1}\boldsymbol{b}\},\ n \in \mathbb{N}^*. \quad (26)$$

The choice of the solution in these subspaces leads to different methods: Conjugate Gradient (CG), MINimal RESidual (MINRES), or Generalized Minimal RESidual (GMRES), for example.

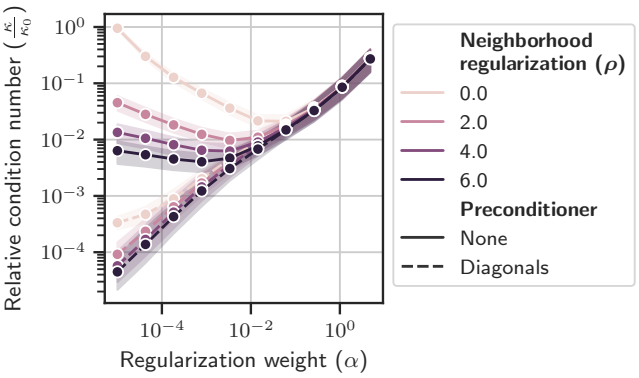


Fig. 3: Normalised condition number of the problem versus parameters' value.

The speed of Krylov's methods depends on the matrix condition number κ of the matrix A. This characteristic quantifies how much our model's result changes with a small perturbation in the input data. A low condition number reflects a robust modelization of our problem. It also hints that Krylov solvers should converge rapidly [18].

Sometimes, the system can be enhanced by the use of a preconditioner M. With M being an invertible matrix, the equation eq. (17) becomes

$$M^{-1}A\boldsymbol{x} = M^{-1}\boldsymbol{b}. \quad (27)$$

Solving the system in eq. (27) is equivalent to solve eq. (17) with a change of variables $A' = M^{-1}A$ and $b' = M^{-1}\boldsymbol{b}$. This system should be faster to solve if $\kappa(A') < \kappa(A)$.

Similarly to the pointwise-coupled matrices defined for splitting solvers, a natural preconditioner for the optical flow is

$$M = \begin{bmatrix} \text{diag}\,\bar{\bar{j}}_{\rho,0,0} - \alpha D_L & \text{diag}\,\bar{\bar{j}}_{\rho,0,1} \\ \text{diag}\,\bar{\bar{j}}_{\rho,1,0} & \text{diag}\,\bar{\bar{j}}_{\rho,1,1} - \alpha D_L \end{bmatrix}. \quad (28)$$

Figure 3 summarizes the value of κ for several model parametrizations: with ρ, the local radius parameter, ranging from 0 to 6 and α, the global regularization weight, from $1e^{-5}$ to 10. The values shown are the ratio κ by κ_0 with κ_0 the value of κ taken with $\alpha = 0$ and $\rho = 0$. Just like in section 3.2.1, images have been cropped to compute κ.

We can now conduct a similar analysis for fig. 3 as we did for fig. 2. First, without preconditioning, κ follows a V-shape with respect to α. However, with a preconditioner M defined as in eq. (28), κ always increases with α. This difference is important, as, for low values of α, the preconditioner decreases κ by orders of magnitudes. With higher values of α, though, the effect

of M is barely noticeable. Finally, we can assert that increasing ρ is significant with low α and no preconditioning.

Figure 3 confirms the results of fig. 2: solvers are the fastest when preconditioned and with low α values. The effect of α can be analyzed by looking back at eq. (11): with α close to zero, most of the penalization comes from $\boldsymbol{w}^T J_\rho \boldsymbol{w}$. This term is directly sensitive to the value ρ. Moreover, the preconditioner M "targets" this term. It is then not a surprise to see how effective it is with low α.

With high values of α, the term $\|\nabla_{x,y} u\|^2 + \|\nabla_{x,y} v\|^2$ dominates. Then, the influences of ρ and M are negligible. Conversely, κ increases because the solution is solely determined by having a zero derivative so that any constant field would be a potential solution.

3.3 A coarse solver benchmark

While sections 3.2.1 and 3.2.2 presented theoretical results for solver convergence on small images, the actual performance is yet to be measured. In this section, we are interested in two indicators: convergence vs. iterations and convergence vs. time.

Since convergence vs. iterations is platform-independent, we can rely on it as an initial filter for limiting the number of solvers to test on the target hardware.

Then comes an implementation on target for the actual solvers' performance. In fact, a performant solver under the convergence vs. iterations measure may become less attractive if the time to perform one iteration is too slow on the targeted hardware.

3.3.1 Convergence vs. iterations

For fig. 4, we chose two sets of parameters to compare the convergence of the solvers mentioned above. We tried several Krylov solvers from the *sparse* module of Scipy but only reported Conjugate-Gradient (CG) as it was the most relevant. We developed two splitting methods: Jacobi and Red-Black Gauss-Seidel (Red-Black GS). The more traditional Gauss-Seidel solver has been discarded from the benchmark. It requires all pixels to be treated sequentially and thus is not appropriate for a parallel implementation. Red-Black GS is a variation on Gauss-Seidel that updates half of the pixels simultaneously [16].

The results differ greatly depending on α. On fig. 4, when α is low, preconditioned method converges quickly (up to 10^{-9} in 100 iterations). The CG method is the fastest, but splitting methods are not far behind. On the contrary, when α is higher, all solvers converge slowly

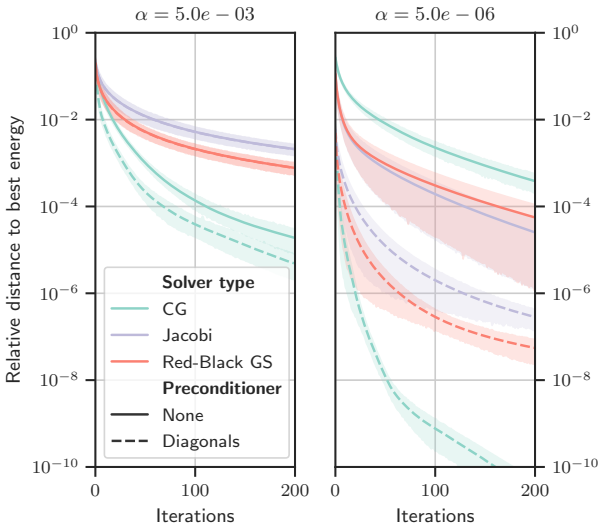


Fig. 4: Convergence vs. iterations with $\rho = 2.5$. On the lest $\alpha = 5e^{-3}$, on the right $\alpha = 5e^{-6}$

($\sim 10^{-5}$ in 100 iterations), and splitting methods are still slower.

Consistently with the results found in section 3.2, the effects of the preconditioner are less visible with higher α. On the left graph of fig. 4, the preconditioned helps the convergence of CG a little, but not as much as when $\alpha = 5e^{-6}$. Regarding splitting solvers, the results with or without preconditioning overlap.

3.3.2 Convergence vs. time

This subsection presents solvers' results as implemented on the embedded target GPU: the Jetson AGX Xavier. The time spent on all implementations was roughly equal. Splitting solvers' implementation is relatively straightforward: all (Jacobi) or half (Red-Black GS) of the pixels are updated in parallel, in a "embarrassingly parallel" fashion.

For the Conjugate-Gradient method, one difficulty is to compute a vector's norm. This operation is not so well adapted to GPUs. Then, we leveraged the *CUB* library (CUDA UnBound) for state-of-the-art GPU reduction performance. We, moreover, took extra care to keep all intermediate results on GPU to avoid expensive latency in CPU-GPU communication.

Figure 5 shows convergence timings for different solvers on GPU until they reach a runtime of 200ms. Globally, the curves follow the same trend as fig. 4 and the order of the curves is respected. Splitting solvers are, however, catching up with CG's performance.

With a low alpha (left-hand side), Jacobi and Red-Black GS are faster than CG in the very first iterations and stay close to CG's performance for a longer time.

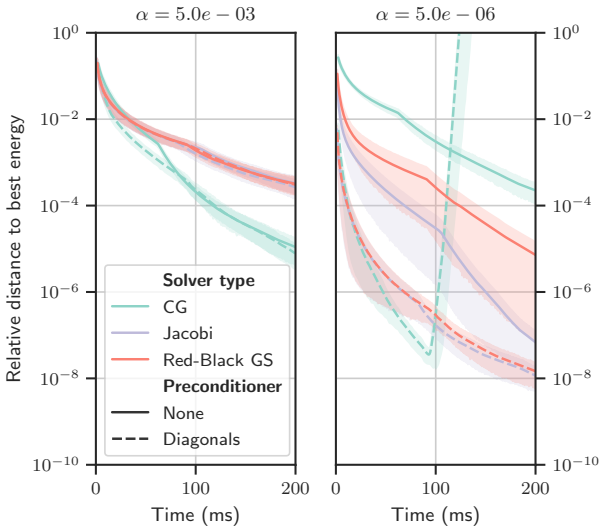


Fig. 5: Convergence vs. time on a Jetson AGX Xavier. Parameters are the same as on fig. 4.

With a high alpha (right-hand side), all preconditioned methods are on par.

An important finding of the benchmark is that the Conjugate-Gradient method is sensitive to numerical precision. On the right-hand side graph of fig. 5, the method diverges after about 100ms of compute. While arithmetic is done in FP32 (IEEE 754 *binary32*) precision, we observed identical behavior in FP64 [17]. This phenomenon also happened with $\alpha = 5.0e^{-3}$, after a vast number of iterations, though. We attribute this divergence to the sensitivity of the Conjugate-Gradient method to its search direction.

For further optimization, Jacobi was chosen as it is the simplest to implement, with adequate performance and good numerical stability. As described later in the article, it is also possible to fuse iterations of Jacobi.

4 Implementation-level approach

This section extends the analysis done in section 3. As a starting point, the solver is now considered fixed. That choice is possible thanks to the initial benchmark on the actual target.

An initial implementation of the CLG method on GPU is done, including the underlying solver and the multi-scale strategy, as detailed in [11]. First, we find sources of optimizations for the solver or elsewhere in the method. Then, we analyze the effects of multi-scale processing by measuring the performance of working on a particular level and the computational cost of changing scale.

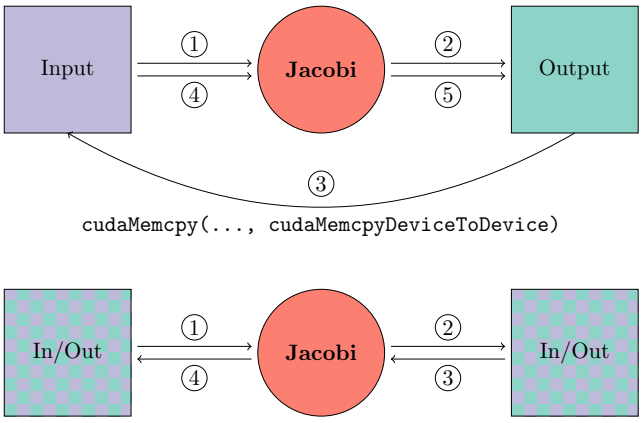


Fig. 6: Two iterations of Jacobi without (top) and with (bottom) buffer reuse.

4.1 Framework and optimizations

When optimizing the code, it is essential to follow a consistent strategy. One must profile the application first to find its main bottlenecks, then try to solve these hotspots, and always check that the application provides the same results. On the Jetson AGX Xavier platform, Nsight Systems and Nsight Compute are two NVIDIA-provided tools that profile executions of programs.

The first one analyzes the whole system and provides CPU and GPU execution traces. This information highlight which kernels would benefit the most from optimization.

The second one dives deeper into a single kernel execution. It provides multiple metrics such as GPU cores occupancy, bandwidth, or a roofline model plot. These lower-level indicators facilitate the discovery of bottlenecks within the kernel.

4.1.1 Optimizations' overview

In this sub-section, we detail the different optimizations that we added to our CLG GPU implementation. They are presented in their order of importance: after each optimization, a new hotspot is selected until speed gains become marginal.

Buffer Reuse: this optimization acts on the Jacobi solver. At the k-th iteration, the program needs one location in memory for the input $x^{(k)}$ and one for the output $x^{(k+1)}$. An initial approach is to fix the memory position of inputs and outputs. This strategy then rely on a copy of the previous output to the current iteration's input: $x^{(k)} \leftarrow x^{(k-1)}$. The memory operation can be avoided by changing the input and output locations at each solver iteration, in a back-and-forth fashion. Figure 6 illustrates this technique.

Jacobi Fusion: the Jacobi solver consumes a lot of memory bandwidth: for each pixel, it fetches a neighborhood of values to compute the Laplacian in addition to coefficients from $\boldsymbol{b}$. All this data is processed with few operations: the solver is bandwidth limited. Our solution is to combine the computation of several iterations within a single kernel launch. This optimization is probably the most important one so that section 4.1.2 extends its analysis.

Batched convolutions: the multi-scale processing of CLG relies on up and down-sampling the image to solve the problem at different scales. This change of resolution uses a Gaussian kernel convolution on images to preserve the down-sampling for high-frequency artifacts. Rather than launching a CUDA kernel for each convolution, we prefer to launch a single kernel that performs convolutions on many images at once. This means that the kernel launch overhead is limited and that the Gaussian filter weights are loaded once and reused for all images.

4.1.2 Fusing iterations of Jacobi

As mentioned previously, the main issue of the Jacobi solver on GPU is its high demand for memory resources. As is, the implementation saturates the VRAM bandwidth, and GPU compute units are starving. To quantify the phenomenon, let us introduce the arithmetic intensity of a program defined by the ratio between the number of floating-point operations (FLOPs) performed by a computed unit over the number of bytes moved to do these operations

$$\mathrm{AI} = \frac{\mathrm{FLOPs}}{\mathrm{Bytes\ loaded}}. \tag{29}$$

A low AI is symptomatic of over-used memory bandwidth. Conversely, if AI is too high, the program requests so many FLOPs that the compute units cannot process them fast enough. Further analysis of the role of AI on a program's execution may be found in [21].

In our initial case, the CUDA kernel is programmed to do a single Jacobi iteration. This approach is straightforward but has several limitations: it requires one kernel launch per iteration so that the call overhead might become an issue. Moreover, each iteration output is written back to main memory, but this is not strictly needed. Combining several iterations within the same kernel would allow direct reuse of intermediate iterations in addition to load coefficients of $\boldsymbol{b}$ only once.

Bottom fig. 7 exposes a fusion of two iterations of Jacobi within a single kernel launch. Static parameters are loaded once and serve for both iterations. The output of the first Jacobi iteration is immediately reused

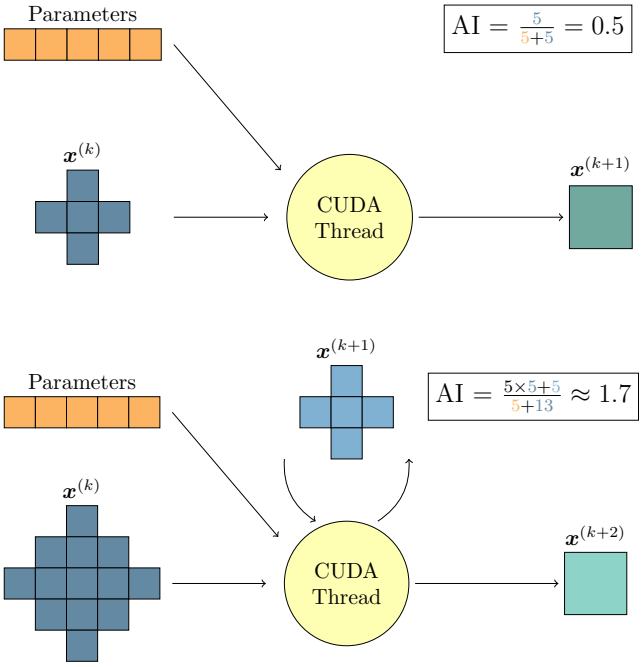


Fig. 7: Top: an iteration of Jacobi for a single output. Bottom: fusion of two Jacobi iterations. Arithmetic intensity is given for reference only, assuming one operation per $\boldsymbol{x}^{(k)}$.

for the second one. The two-iteration scheme requires loading a larger neighborhood of $\boldsymbol{x}$ values to satisfy all further dependencies.

Another important aspect of this implementation is shared memory. In the CUDA model, GPU threads are partitioned into Thread Blocks (TB). Threads of a common TB are executed on a single processing unit, the streaming multiprocessor, and have access to shared memory. This location is used to share the coefficient of $\boldsymbol{x}$ between GPU threads, leveraging the pixels' neighborhoods' spatial redundancies.

For now, let us set the TB size to 32×32. Initially, each thread of the TB load one coefficient of $\boldsymbol{x}^{(k)}$ from the main memory to the shared memory. Then, threads compute a first Jacobi iteration and wait for the TB to have finished thanks to the synchronization primitive `__syncthreads`. With the $\boldsymbol{x}^{(k+1)}$ coefficients being computed, the TB computes the subsequent Jacobi iteration.

Let us now find the approximate value of AI based on an implementation that fuses j iterations. At each new iteration, the size of the computed area decreases because of spatial dependencies. At the i-th iteration, $i \leq j$, the footprint's size is $(32 - 2i) \times (32 - 2i)$. We can now express AI as a function of j, the number of

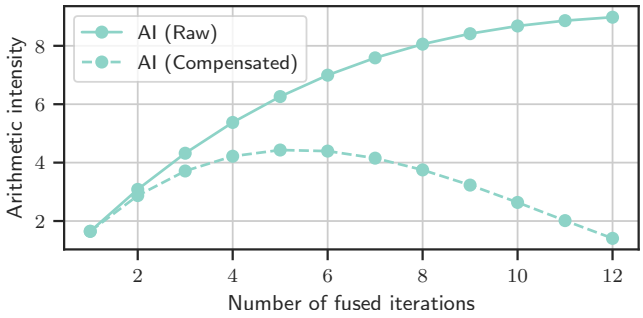


Fig. 8: Arithmetic intensity w.r.t. the number of fused iterations.

fused iteration

$$\mathrm{AI}(j) = \frac{\alpha \sum_{i=1}^{j} (32-2i)^2}{\beta(32\times 32)} \tag{30}$$

α is the number of FLOPs needed per pixel and per iteration and β is the number of bytes to load per pixel.

While the AI expressed in eq. (30) increases with j and then seems to benefit the implementation, it is important to understand that the total FLOPs required by the algorithm are not constant with j. A single non-fused Jacobi needs

$$W_{\text{no fusion}} = \alpha HW. \tag{31}$$

operations. With H and W the dimensions on the processed image. In comparison, in a j-fused implementation, each TB computes a patch of $(32-2j)^2$ pixels. To compute the entire image, we have

$$W_{\text{j-fusion}} = \lceil \frac{H}{32-2j} \rceil \lceil \frac{W}{32-2j} \rceil \alpha \sum_{i=1}^{j} (32-2i)^2. \tag{32}$$

Some operations are redundant with the fused iterations technique to handle patch borders and avoid inter-TB communication.

The ratio between $W_{\text{j-fusion}}$ and $j \cdot W_{\text{no fusion}}$ expresses the overhead of operations due to the fusion of operations

$$\frac{W_{\text{j-fusion}}}{jW_{\text{no fusion}}} \approx \frac{1}{j(32-2j)^2} \sum_{i=1}^{j} (32-2i)^2 \tag{33}$$

Figure 8 shows the AI for different choices of j, the number of fused iteration. The solid curve represents the AI computed by the formula in eq. (30). The dashed curve is arithmetic intensity divided by the *compute* overhead, as expressed in eq. (33).

The *raw* AI is an increasing function of j: by looking at this metric only, it would make sense to choose j as large a possible to reduce the memory pressure. Conversely, the refined metric, *compensated* AI, indicates that because higher values of j induce too much redundant work, it is better to choose j close to 5.

This study of the Jacobi iteration fusion has exhibited the pros and cons of using many fused iterations. While done in a theoretical setting, it should help to analyze GPU execution performance.

4.2 Results

To measure the effects of the various optimizations presented in section 4.1.1, we have taken measurements on two GPU cards. The first one, an NVIDIA Titan V, is used in PCs and computing servers. We use it as the baseline of our development process. The second one, a Jetson AGX Xavier, is the actual target of our industrial application. After initial implementation and verification on Titan V, we deploy on Xavier, and we check if the optimization has the expected effect.

In our method, the optimizations' order is guided by results on the Jetson Xavier. For example, fig. 9 shows us that once the buffer reuse optimization is implemented, the time spent in memory transfers is still relatively high on Titan V but not on Xavier. Since we ultimately focus on this embedded target, we will not dwell on further optimization for memory transfers.

All the details about the hardware used for our experiments are available on table 1.

	Machine #1: desktop	Machine #2: embedded (Jetson AGX Xavier)
OS	Ubuntu 16.04	Ubuntu 18.04
Linux Kernel	4.15.0	4.9.140
CUDA	11.0	10.2
NVIDIA Driver	450	JetPack 4.4
CPU	Intel i7-3820	8-core ARM 64bits
GPU	Titan V (arch. 7.0)	Xavier (arch. 7.2)
TDP	∼500W	∼30W

Table 1: Environments of the experiments.

Buffer Reuse: On the initial runtime bar of fig. 9, we can see that a good part of the computation time spent on GPU is dedicated to memory transfers. The effects of the buffer reuse optimization are pretty different depending on the platform.

On Jetson Xavier, we can see that the time spent in memory operations goes from about 3ms to 0.5ms. The remaining memory time is spent uploading the input images and downloading the output flow. A further optimization could lead to marginal gains by using Unified Memory. This makes buffer transfers with zero-copy, because the GPU and the CPU share the same memory.

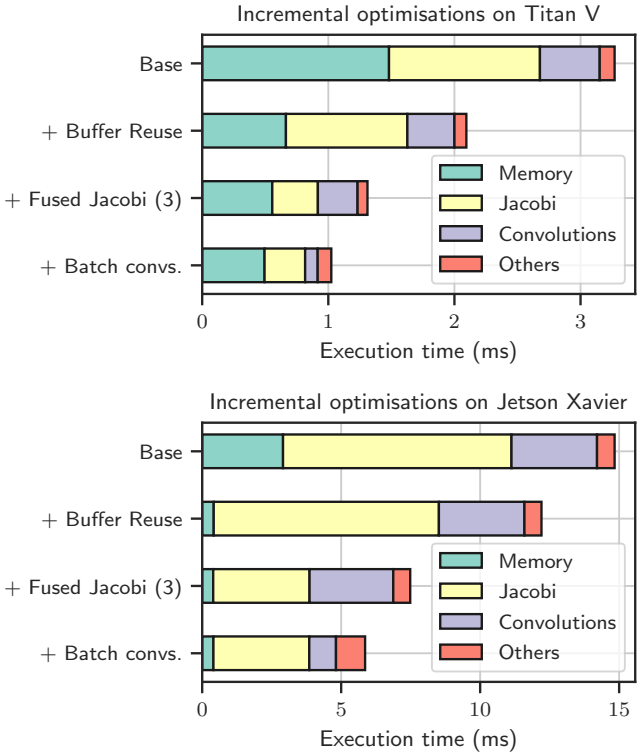


Fig. 9: Effects of cumulative optimizations on Titan V (top) and Jetson Xavier (bottom).

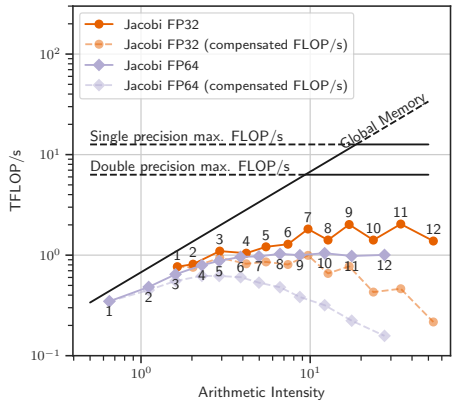


Fig. 10: A roofline model analysis of Jacobi iteration fusion on Titan V.

On Titan V, we can see that those memory operations still require a lot of time (∼33% of the computation time). This is explained by the fact that the CPU and GPU memory are disjoint, so it takes more time (proportionally to the power of the machine) to transfer the inputs and outputs.

Jacobi Fusion: tests shown on figs. 10 and 11 evaluates the performance of different number of fused Jacobi iterations, as explained in section 4.1.2. Figure 10 plots the achieved TFLOP/s (Tera Floating-Point Operations per second) with respect to the measured arithmetic intensity. This figure first shows that, for Titan V, the FP64 machine balance is reached for an AI of 10, while 20 is needed for FP32 operations. This value exposes the minimum number of operations per byte to compute to benefit from maximal hardware performance.

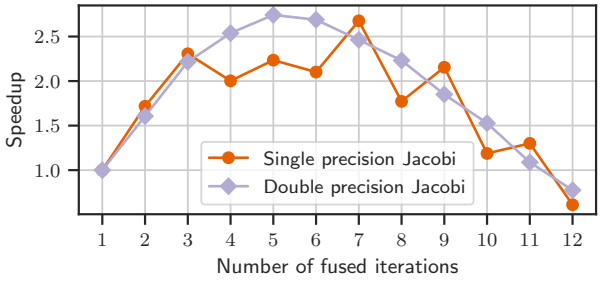


Fig. 11: Speedup vs. number of fused iterations on Titan V.

Without any fusion, it is clear that both FP32 and FP64 implementations are bandwidth-limited. As expected, the arithmetic intensity of increases with the number of fused iterations. With an ideal execution, the points should appear close to the roofline. Here, after few fusions, it is clear that the progression stalls. While the FLOP/s continue to increase, this growth is not sufficient to stay close to the roofline. We explain this behavior by the low number of active threads within a TB. At each new fused iteration, the size of the region of interest of a TB decreases, then, more of its threads are idle.

Comparing raw performance on fig. 10 is complex. The *FLOP/s* metric, given by Nvidia Nsight Compute, directly measures the activity of computing units. As explained in section 4.1.2, some computations are redundant from the method point-of-view. To correct the *FLOP/s* metric, we divide it by the work overhead defined in eq. (33). This compensated curve draws a different conclusion than the initial one. For example, in FP32, the *raw FLOP/s* is highest for nine fused iterations. In the compensated model, the best fusion is lower: around three iterations.

This difference highlight a drawback of the analysis based on the roofline model only. When the total number of operations changes from one implementation to another, the acheived *FLOP/s* is not comparable. In our case, fig. 11 is more straightforward: it shows the execution time gain for different numbers of fused iterations.

The maximum performance is achieved with seven fused iterations in FP32 and five in FP64. These results tend to confirm the analysis of the *compensated* roofline model made on fig. 10.

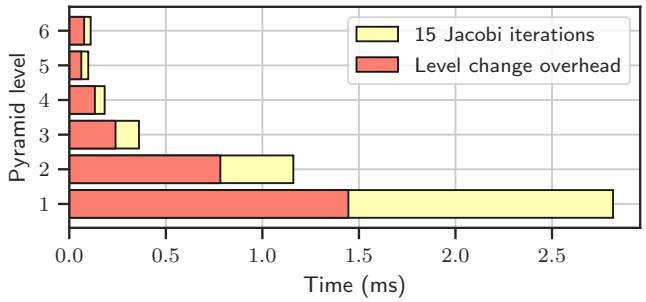


Fig. 12: Time to reach a new pyramid level and perform 15 Jacobi iterations on that level. Infered from timing measurements.

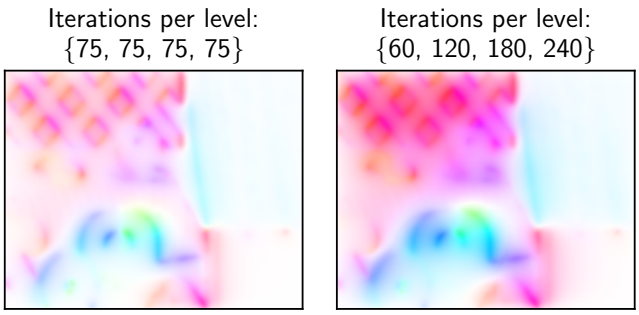


Fig. 13: Results on 640 × 512 images at 60 FPS. Doing more iterations at higher levels converges faster.

We now choose a value of three fused iterations in FP32 for two reasons: it achieves good speedup both in FP32 and FP64 precisions, and it is more convenient to have a total number of iterations that is a multiple of three, than seven, for example. On fig. 9, the time spend for Jacobi iterations is almost divided by three on Titan V and about by two on Jetson Xavier. The additional speedup, especially on Titan V, is explained by reducing kernel launch overhead.

Batched convolutions: after optimizing the Jacobi solver, fig. 9 shows that on Jetson Xavier, almost half of the runtime is spent doing convolutions. As explained in section 4.1, convolutions have been re-expressed to be run in a single CUDA function. Instead of launching a kernel per convolution, the batch computation reduces the overhead and lets the convolution filter's coefficients in the CUDA thread registers. This makes the runtime of convolutions decrease by a factor of two on Xavier.

4.3 Execution model and method configuration

As detailed in [4], multi-level processing aims at finding optical flows at different scales of the problem. The technique is helpful for finding large displacement and for iterating quickly on higher levels due to the reduced problem size. Consequently, we measure the actual performance of our GPU implementation on the different sub-sampled problems. Those results should guide decisions back at the algorithm level. With a 60 FPS real-time constraint, we estimate the number of possible iterations during that time. This information makes a more educated choice of the hyper-parameters possible by knowing how precisely the model converges within the limited time frame.

Figure 12 presents results for the multi-scale CLG optical flow. At the first level, the flow resolution is the same as the image's one. For each subsequent level, the problem is down-sampled by a factor of two. The graph reports the time needed for reaching a higher scale as well as fifteen Jacobi iterations on this level.

Level changes and Jacobi iterations have an ideal speedup of 4× at each higher level because of the lower number of pixels to process. In reality, gains are not optimal: even if levels 1-2-3 see consequent runtime reduction, higher scales stagnate. This is the consequence of the GPU not being in a throughput-limited regime. There is not enough parallelism with low-resolution images to use all cores; the GPU latency then limits the runtime. In light of these results, it seems interesting to benefit from the speedups at levels 3-4 and restrict the first levels use to a minimum.

On fig. 13, the optical flow for two choices of parameters is displayed. The left-hand flow was obtained by doing 75 iterations at each level. The right-hand flow sets 60 iterations at the finest level and 120, 180, and 240 iterations at the higher ones. While both configurations run at the same speed, 60 FPS on the Jetson Xavier with 640 × 512 images, the configuration using more iterations on the higher levels seems smoother. It has converged more on less textured regions and seems better for practical use.

5 Conclusion

This article has shown the interest in combining analyses at the algorithm and implementation levels to obtain the best performance.

Initially, we pre-selected candidate GPU solvers for a subsequent GPU optimization. This first analysis also provided an understanding of the hyper-parameters on the convergence speed. Then, the multi-scale CLG algorithm was ported on the embedded Jetson AGX Xavier GPU. Several optimizations have enhanced the algorithm's run time: re-utilization of intermediate Jacobi buffers, solver iteration fusion and batching of convolution. Overall, these techniques decreased the runtime of the algorithm by more than 2×.

The multi-scale behavior of the method has also been studied. Results have shown that higher levels are processed faster but that the speedup plateaus for images smaller than 80×64. This result allowed us to choose the right parameters for the best possible convergence within a limited time frame.

In the end, our GPU implementation of the CLG optical flow method runs at 60 frames per second on 640×512 images with a 30W power budget. Moreover, we were able to tweak the hyper-parameters and multi-scale behavior to converge quickly.

References

1. Aliaga, J.I., Pérez, J., Quintana-Ortí, E.S.: Systematic Fusion of CUDA Kernels for Iterative Sparse Linear System Solvers. In: J.L. Träff, S. Hunold, F. Versaci (eds.) Euro-Par 2015: Parallel Processing, Lecture Notes in Computer Science, pp. 675–686. Springer, Berlin, Heidelberg (2015). DOI: 10.1007/978-3-662-48096-0_52
2. Baker, S., Scharstein, D., Lewis, J.P., Roth, S., Black, M.J., Szeliski, R.: A Database and Evaluation Methodology for Optical Flow. International Journal of Computer Vision **92**(1), 1–31 (2011). DOI: 10.1007/s11263-010-0390-2
3. Brox, T., Bruhn, A., Papenberg, N., Weickert, J.: High Accuracy Optical Flow Estimation Based on a Theory for Warping. In: T. Kanade, J. Kittler, J.M. Kleinberg, F. Mattern, J.C. Mitchell, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M.Y. Vardi, G. Weikum, T. Pajdla, J. Matas (eds.) Computer Vision - ECCV 2004, vol. 3024, pp. 25–36. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). DOI: 10.1007/978-3-540-24673-2_3
4. Bruhn, A., Weickert, J., Schnörr, C.: Lucas/Kanade Meets Horn/Schunck: Combining Local and Global Optic Flow Methods. International Journal of Computer Vision **61**(3), 1–21 (2005). DOI: 10.1023/B:VISI.0000045324.43199.43
5. Capito, L., Ozguner, U., Redmill, K.: Optical Flow based Visual Potential Field for Autonomous Driving. In: 2020 IEEE Intelligent Vehicles Symposium (IV), pp. 885–891 (2020). DOI: 10.1109/IV47402.2020.9304777
6. Ding, N., Williams, S.: An Instruction Roofline Model for GPUs. In: 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp. 7–18 (2019). DOI: 10.1109/PMBS49563.2019.00007
7. Dougherty, L., Asmuth, J.C., Gefter, W.B.: Alignment of CT Lung Volumes with an Optical Flow Method. Academic Radiology **10**(3), 249–254 (2003). DOI: 10.1016/S1076-6332(03)80098-3
8. Farnebäck, G.: Two-Frame Motion Estimation Based on Polynomial Expansion. In: J. Bigun, T. Gustavsson (eds.) Image Analysis, Lecture Notes in Computer Science, pp. 363–370. Springer, Berlin, Heidelberg (2003). DOI: 10.1007/3-540-45103-X_50
9. Filipovič, J., Madzin, M., Fousek, J., Matyska, L.: Optimizing CUDA Code By Kernel Fusion—Application on BLAS. The Journal of Supercomputing **71**(10), 3934–3957 (2015). DOI: 10.1007/s11227-015-1483-z
10. Horn, B.K.P., Schunck, B.G.: Determining optical flow. Artificial Intelligence **17**(1), 185–203 (1981). DOI: 10.1016/0004-3702(81)90024-2
11. Jara-Wilde, J., Cerda, M., Delpiano, J., Härtel, S.: An Implementation of Combined Local-Global Optical Flow. Image Processing On Line **5**, 139–158 (2015). DOI: 10.5201/ipol.2015.44
12. Lucas, B.D., Kanade, T.: An Iterative Image Registration Technique with an Application to Stereo Vision. In: Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2, IJCAI'81, pp. 674–679. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1981)
13. McGuire, K., de Croon, G., De Wagter, C., Tuyls, K., Kappen, H.: Efficient Optical Flow and Stereo Vision for Velocity Estimation and Obstacle Avoidance on an Autonomous Pocket Drone. IEEE Robotics and Automation Letters **2**(2), 1070–1076 (2017). DOI: 10.1109/LRA.2017.2658940
14. Moussu, C.: GPU based real-time optical Flow computation. Tech. rep., Imperial College London (2010)
15. Nguyen, M.T., Castonguay, P., Laurendeau, E.: GPU parallelization of multigrid RANS solver for three-dimensional aerodynamic simulations on multiblock grids. The Journal of Supercomputing **75**(5), 2562–2583 (2019). DOI: 10.1007/s11227-018-2653-6
16. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM (2003)
17. Seznec, M., Gac, N., Orieux, F., Naik, A.S.: An Efficiency-Driven Approach For Real-Time Optical Flow Processing On Parallel Hardware. In: 2020 IEEE International Conference on Image Processing (ICIP), pp. 3055–3059 (2020). DOI: 10.1109/ICIP40778.2020.9191164
18. Shewchuk, J.R.: An Introduction to the Conjugate Gradient Method without the Agonizing Pain. Carnegie-Mellon University. Department of Computer Science (1994)
19. Sun, D., Roth, S., Black, M.J.: A Quantitative Analysis of Current Practices in Optical Flow Estimation and the Principles Behind Them. International Journal of Computer Vision **106**(2), 115–137 (2014). DOI: 10.1007/s11263-013-0644-x
20. Tabik, S., Ortega, G., Garzón, E.M.: Performance evaluation of kernel fusion BLAS routines on the GPU: Iterative solvers as case study. The Journal of Supercomputing **70**(2), 577–587 (2014). DOI: 10.1007/s11227-014-1102-4
21. Williams, S.W.: Auto-tuning performance on multicore computers. Ph.D. thesis, University of California at Berkeley, USA (2008)
22. Zach, C., Pock, T., Bischof, H.: A Duality Based Approach for Realtime TV-L 1 Optical Flow. In: F.A. Hamprecht, C. Schnörr, B. Jähne (eds.) Pattern Recognition, vol. 4713, pp. 214–223. Springer Berlin Heidelberg, Berlin, Heidelberg (2007). DOI: 10.1007/978-3-540-74936-3_22

3.2 Conclusion

The work on the second and third stages of the deployment strategy for the optical flow has made possible a real-time implementation on a embedded GPU. First, the algorithm/hardware phase consisted in a benchmark of several solvers and the influence of the hyper-parameters on the convergence speed. This initial comparison allowed us to select an efficient method on GPU.

Then, the GPU optimization work has tested several strategies such as kernel batching, iteration fusion and buffer re-utilization to speed up the implementation by a factor of two.

Finally, the optical flow estimation algorithm runs on an embedded GPU, the Nvidia Xavier, at 60 frames per second. This processing is real-time and embeddable in power-constrained situations. We have also established a run-time model to analyze the effect of the multi-scale processing. With it, it is easier to choose a suitable configuration in terms of iterations per level.

Chapter 4

Towards real-time optical flow with DNNs

Artificial Neural Networks (ANNs) have played a major role in the Machine Learning (ML) domain, an important part of Artificial Intelligence (AI) (Lecun, 1985). For image processing tasks, Convolutional Neural Network (CNNs) is the most popular ANN model. CNNs draw their inspiration from biological neurons, and the association of convolution filters and activation functions, also called a layer, emulates brain cells' behavior (Kietzmann et al., 2019).

The term deep neural network (DNN) finds its origin in modern networks that stack many layers to overpass traditional algorithms' results, including for optical flow estimation (Ilg et al., 2017; OMahony et al., 2020). The performance of CNNs has, for example, augmented the resilience and autonomy of drones, and these networks are now a critical factor in adapting a system to rapid changes in the environment and operational needs (Doll & Schiller, 2019; Ferrari, 2019; Parly, 2019). Embedding CNN processing close to sensors is thus a definitive advantage in conflict zones and urban areas where telecommunication jamming is predominant. The offloading of AI analysis to cloud-based systems is indeed impossible, as all communications are blocked.

In these conditions, embedded systems are crucial assets but limited by SWaP (Size, Weight, and Power) constraints. On the other side, DNNs require massive computing power and have a large memory footprint. Their integration is then a challenging task. In this chapter, the research work we present is part of the CALYPSO expertise platform, currently being developed at Thales Research & Technology. CALYPSO englobes algorithm development for artificial intelligence (AI), software stacks for optimized code generation, and neuromorphic hardware expertise for more efficient AI processing at the edge. Its objective is to master the complete toolchain from algorithm development (with the industrial use-case in mind) through an optimized software stack dedicated to embedded systems down to hardware-specific code generation for energy and power-efficient execution on

the targeted embedded processors. In other terms, CALYPSO's expertise targets the algorithm/implementation and performance loop phases that we presented in fig. 1.2 to exploit the full capabilities offered by embedded hardware platforms.

In our work, we use this deployment strategy to implement PWC-Net (Pyramid, Warping, and Cost Volume Network) (Sun et al., 2018b) , an optical flow estimation DNN, on the embedded Jetson AGX Xavier. After several optimizations, such as the use of shared memory for a correlation operation and using `fp16` arithmetic, we attain near real-time on the energy-efficient platform. Then, at the algorithm/implementation level, we design an architecture derived from PWC-Net that leverages MobileNetV2 (Sandler et al., 2018), which was created for low-end hardware platforms.

The chapter starts with section 4.1 that gives an overview of current work in the field of deep convolutional networks. First, our analysis targets architectures designed for the estimation of optical flow. Then, it tackles strategies used for the development and deployment of neural networks on embedded targets. Then, implementation of PWC-Net on the Jetson AGX Xavier is analyzed in section 4.2 and shows that it achieves high performance on this embedded platform. Section 4.3 presents *MobileFlow*, a new network architecture designed for more efficient processing. Our work's novelty lies in this lightweight architecture that combines PWC-Net and MobileNetV2, leverages transfer learning, and re-designs flow estimation layers. Section 4.4 concludes this chapter and discuss possible perspectives.

4.1 State of the art

This review begins with an introduction to CNNs with Alexnet (Krizhevsky et al., 2012), which classifies images according to the object present in the picture. Then, the survey is extended to optical flow estimation networks, such as PWC-Net (Sun et al., 2018b), and their specificities in terms of architecture and training procedures.

These DNNs are usually computationally heavy. Consequently, techniques at the implementation and design levels have been developed to allow execution on constrained platforms. Section 4.1.2 explains, for example, the advantages of running a network with reduced precision and also why depth-separable convolutions present in MobileNet are an asset for efficient processing.

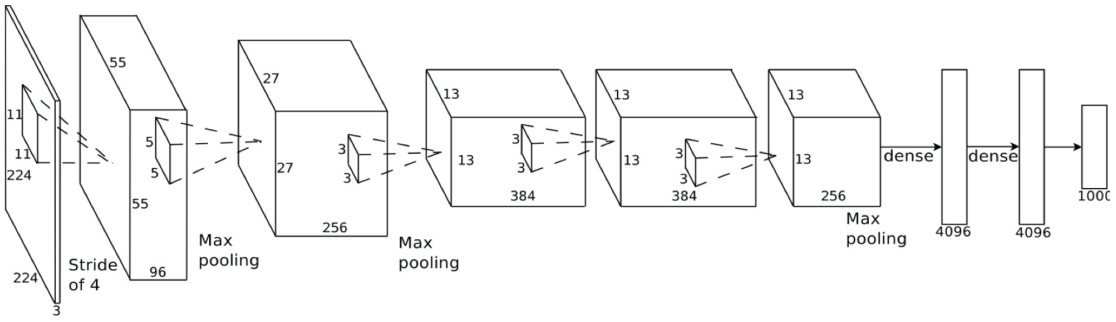


Figure 4.1: The Alexnet CNN architecture that won the ImageNet classification challenge. The first five layers are a grouping of convolution, activation and max pooling. This generates features for the input image that are discriminated by two dense layers to assign probabilites to a thousand classes. Credits: Adam Geitgey.

4.1.1 Optical flow estimation via DNNs

Convolutional Neural Networks for classification

Convolutional neural networks have reached considerable popularity since the demonstration by Krizhevsky et al., 2012 that they outperform traditional image classification methods on the ImageNet dataset (Deng et al., 2009). The core idea of CNNs is to apply successive convolution filters and activation functions on an input image. This series of operations generate three-dimensional tensors, called feature maps. Then, a linear classifier combines this computed coefficient to assign probabilities to recognized objects in the image. Figure 4.1 illustrates the architecture used by Alexnet. The first operation is a convolution with a 11×11 kernel and a stride of four, which generates a tensor with a lower resolution than the image. At the end of the network, *dense* layers are matrix multiplications, and the final output is a vector of size 1000 that corresponds to the probabilities of the recognizable classes.

This type of network is often trained in a supervised setting. For image classification, that means having a collection of images and their corresponding label, like a dog or a car. The training procedure consists in presenting a picture to the network and collect its output probabilities. Then, a loss function determines the error between the CNN's result and the correct classification. Ultimately, the goal is to minimize this loss function over the entire training dataset. For that, one can differentiate the loss and perform gradient descent. With the backpropagation (Rumelhart et al., 1986), the learnable weights of the network are then adapted to provide correct results.

Towards optical flow estimation with CNNs

As is, these image classification networks can already be used for optical flow estimation. The high-level features they learn may serve as the data term of traditional optimization approaches. For example, the general Equation (1.9), presented in section 1.1.2, can use CNN features' conservation for D. This method has the advantage of leveraging higher-level features of the image, with semantic information, rather than just pixels' values (Bai et al., 2016).

The integration of CNNs can go one step further with direct neural optical flow estimation Dosovitskiy et al., 2015. This pioneer architecture, Flownet, as presented in fig. 4.2, is entirely based on CNNs. The network outputs per-pixel results, just like Long et al., 2015 did for image segmentation. In details, two architectures were originally proposed by Dosovitskiy et al., `FlownetS` and `FlownetC`. They differ in how they handle images at the beginning of the network. The first concatenates the image pair to form one single tensor fed into a fully convolutional network. The second handles both images separately initially and computes correlations between features computed for the two. This network uses a multi-scale processing, and the tensors' spatial dimensions are first reduced. Some features are stored in memory to be used later during the "refinement" step that generates optical flow prediction. This time, the first prediction is at the smallest resolution level, and larger flows are iteratively generated, see fig. 4.3.

An optical flow-specific layer, the feature correlation

The correlation operation serves at finding features that are identical, or close to each-other, in the two input images. Even if results of (Dosovitskiy et al., 2015) showed that FlowNetSimple was better, correlations have been successfully used in other networks (Ilg et al., 2017; Sun et al., 2018b). Given two tensors $\boldsymbol{F}_1$ and $\boldsymbol{F}_2$ of identical size (H, W, C) the correlation between $\boldsymbol{F}_1$ at coordinates (y_1, x_1), $\boldsymbol{F}_2$ at coordinates (y_2, x_2) is defined as

$$c(y_1, x_1, y_2, x_2) = \sum_{(x,y) \in [-k,k]^2} \langle \boldsymbol{F}_1(y_1 + y, x_1 + x),\ \boldsymbol{F}_2(y_2 + y, x_2 + x) \rangle. \tag{4.1}$$

This expression uses vector products of slices of $\boldsymbol{F}_1$ and $\boldsymbol{F}_2$. The outer sum over a neighborhood defined by k serves for averaging and regularization. The correlation c is a measure of how close features from $\boldsymbol{F}_1$ and $\boldsymbol{F}_2$ are from each other. In terms of optical flow, if $c(y_1, x_1, y_2, x_2)$ is high, it means that pixel (y_1, x_1) has probably "moved" from coordinates (y_1, x_1) to (y_2, x_2) between the two tensors.

The global displacement features tensor $\boldsymbol{C}$, of size (H, W, H', W') is a collection of coordinates correlation. Given (y, x), the slice $\boldsymbol{C}(y, x)$ holds correlations of $\boldsymbol{F}_1(y, x)$ with points in $\boldsymbol{F}_2$, in a neighborhood of size (H', W') around (y, x),

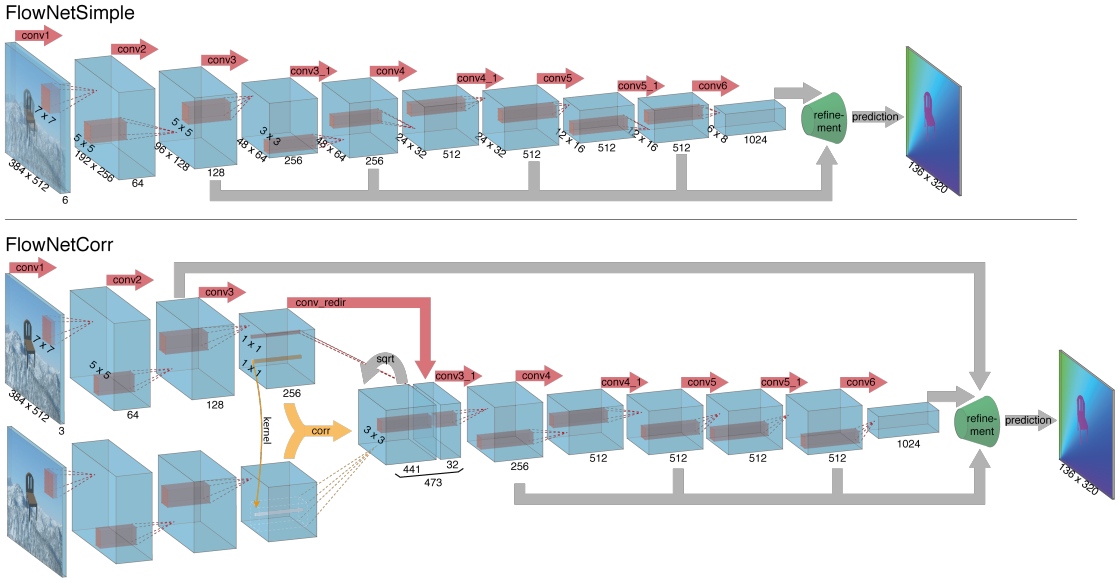


Figure 4.2: The two DNN architectures for optical flow introduced by Dosovitskiy et al., 2015. Top, `FlownetS`, which stacks two images to form the input, then generates features via convolutions, and finally aggregates results of different layers to generate the flow. Bottom, `FlownetC`, replaces the start of the network with two siamese (which share the same weights) CNNs that operate on the two images independently. Features of both images are then combined with a correlation operation.

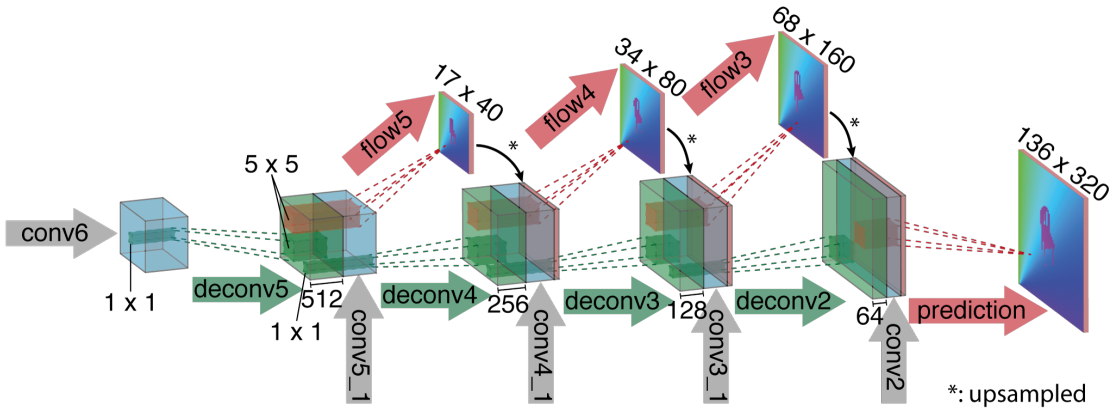


Figure 4.3: Zoom on the refinement module presented in fig. 4.2. Its uses previous feature tensors to generate increasingly larger flow estimations. From (Dosovitskiy et al., 2015).

such that

$$\boldsymbol{C}(y, x, y', x') = c(y, x, y + y' - \lfloor \frac{H'}{2} \rfloor, x + x' - \lfloor \frac{W'}{2} \rfloor). \tag{4.2}$$

For convenience with other operations in the network, the last two dimensions of the 4D tensor $\boldsymbol{C}$ are often flattened to create a tensor $\boldsymbol{C'}$ of size (H, W, $H' \times W'$).

Training procedures

During training, FlowNet uses a specially crafted synthetic dataset (Dosovitskiy et al., 2015). The images represent 3D models of chairs "flying" over a static background. Because image pairs are computer-generated, it is possible to know the ground-truth optical flow. With more than 20,000 optical flow examples, this dataset is beneficial for the initial training of the network. A common second step consists of using another dataset, such as KITTI or Sintel, to fine-tune the network to specific examples. The underlying assumption is that the network learns how to generate optical flow with many examples of FlyingChairs. Only a few data are required to specialize the flow estimation for specific situations.

The total number of examples seen during training can be chosen according to different schedules, defined in (Ilg et al., 2017). For example, the *long* schedule uses 1.2M iterations of eight examples per mini-batch. It also defines the learning rate, starting at 1×10^{-4} and halving it after 400k, 600k, 800k and 1M iterations.

The size of the dataset can also be artificially increased thanks to augmentation procedures. Flipping an image pair along the vertical axis, for example, produces a new sample, but the ground-truth flow must be modified to account for this transformation. More generally, a linear image transformation, combining rotation, translation, shear, and zoom, can be used (Pinard, 2017, January 27/2021). The choice of the input image's dimensions is also sensitive, and depending on the cropping strategy, the training may be improved (Bar-Haim & Wolf, 2020).

Other DNN architectures for optical flow

FlowNet2 (Ilg et al., 2017) is the direct descendant of FlowNet. It employs FlowNetS and FlowNetC as sub-modules to find small and large displacements between the two images, achieves better results, but is longer to train and process images.

The SPyNet (Spatial Pyramid Network), presented in (Ranjan & Black, 2017), uses a multi-scale architecture that resembles the classic optical flow scheme, as shown previously on fig. 1.8. The input image pair is successively downscaled to form a pyramid of images. A CNN uses current images and flow estimation of the previous level to warp the image and find residual flow for each pyramid level. The

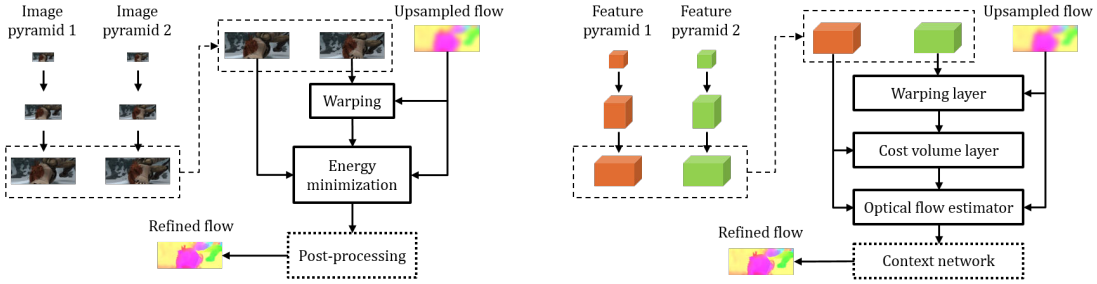


Figure 4.4: Left, the usual multi-scale processing for optical flow. Replacing the `Energy Minimization` step with a neural network would resut in SPyNet's approach (Ranjan & Black, 2017). Right, PWC-Net operates on CNN features directly and uses the correlation operator (`Cost Volume Layer`) defined in Flownet. From (Sun et al., 2018b).

fact that this architecture incorporates optical flow-specific design choices results in a 96% reduction in the number of weights needed by the network, compared to FlowNet, and attains similar results.

PWC-Net, which we use later in our work, can be thought of as an extension of SPyNet. Instead of operating on a pyramid of "raw" images, it uses a pyramid of learned features, like presented on fig. 4.4 (Sun et al., 2018b). It allows the network to learn higher-level features of the image and outperforms SPyNet and FlownetS. The same authors presented in (Sun et al., 2018a) a better way of training optical flow networks. They change the data augmentation scheme and the learning rate schedule to increase PWC-Net's accuracy by 11% and FlownetC's by 56%.

The PWC-Net network has since been re-used in (Bar-Haim & Wolf, 2020) to introduce new techniques at the learning stage to increase even more the performance of the network. In (P. Liu et al., 2019), self-supervised learning of optical flow is studied with PWC-Net. At the training stage, no "ground-truth" flow is present, but the network seeks to minimize a photometric loss between the two images after a warp by the resulting flow. PWC-Net also serves in (Zhao et al., 2020) for joint estimation of optical flow and occlusion mask. Concurrently, the LiteFlowNet architectures (Hui & Loy, 2020; Hui et al., 2018; Hui et al., 2020) use a similar approach as PWC-Net but add a regularization module on each level of the pyramid.

A comparison in terms of performance of several architectures and techniques of deep-learning estimation of optical flow has been made by (Hur & Roth, 2020). It gathers results on the Sintel *final* dataset and shows that FlowNetS and FlowNetC attain, respectively, an EPE of 7.2 and 7.9. PWC-Net reduces the error down to 5.0 while LiteFlownet achieves 5.4. These results show the progress made by optical flow CNN architectures at being more precise while being relatively lightweight.

4.1.2 Real-time strategies for DNNs

Deep neural networks tend to be computationally heavy and demand a large amount of memory space for execution. Their implementation on constrained and embedded platforms is complex, and real-time processing requires some techniques to lighten the network (Han et al., 2016). In this section, we review two different approaches that can be combined. First, starting from an existing trained network and modify the way it executes to increase efficiency. Then, designing a DNN from scratch by using layers dedicated to low-end platforms.

Optimizing an existing network

Weight pruning removes useless connections in a network. For example, weights with low magnitudes are discarded, and their operations no longer need to be computed. The near-zero values of these weights show that the associated connections do not participate significantly in the result of the network. After removing these weights, it is essential to re-train the pruned network for better accuracy. This technique is particularly effective on fully connected layers, where more than 90% of the weights can be discarded with almost no impact on accuracy (Han et al., 2015). This reduction has an immediate effect on the memory footprint of the network. Speed-up gains are, however, not automatic, as the platform must take advantage of the now sparse operations (Yao et al., 2019).

Another widely used technique is weight compression. Networks are usually trained with `fp32` arithmetic. Reducing it to `fp16` numbers or even fixed-point numbers with eight or fewer bits of accuracy compresses both the model and the feature maps. It accelerates the execution by reducing the memory bandwidth and leveraging low-precision hardware units (Han et al., 2016).

The use of 16-bit floating-point numbers is usually the first step towards reducing the model's weight. `binary16` is sometimes prefered, because it has the same representation range as `fp32` but offers lower precision (see fig. 1.15). IEEE `fp16` is challenging to handle, especially with near-zero `fp32` numbers that cannot be represented in this format. This problem can be handled by scaling the operations, so the results are in the representable range of `fp16` (Micikevicius et al., 2018).

Using fixed-point arithmetic is the next challenge after floating-point size reduction. Once the network has been trained, a standard method for quantizing the weights is to perform inference with representative dataset examples. Then, collect the activation values at each layer to determine the range to represent with fixed-point numbers. Fixed-point quantification can provide good performance while maintaining accuracy and even outperform floating-point networks (Lin et al., 2016).

Pruning, weight reduction and other platform-specific optimization can be done

by automatic tools such as TensorRT (Vanholder, 2016), TVM (Chen et al., 2018), or N2D2 (Bichler et al., 2017). Starting from a graph representation of the neural network, in formats such as Pytorch, Tensorflow, or ONNX, those toolchains attempt to perform optimizations that lead to efficient deployment on the target platform: GPU, FPGA, or mobile CPU. A GPU-specific optimization is, for example, to merge several network operations into one function. The interest is to keep intermediate values in registers and re-use them as much as possible instead of going back and forth in the main memory.

Designing an efficient network

The other strategy employed for high-performance neural networks is to design them directly with embedded platform deployment in mind. This resulting architecture can also benefits from the optimizations we introduced previously for optimal performance. Efficient designs take place at two levels: micro and macro architectures. The micro-level chooses an efficient operation or module that is repeated throughout the network. The choice of how to arrange and connect the modules constitutes the macro-level.

For Squeezenet, Iandola et al., 2016 modify Alexnet's architecture to remove traditional 3×3 convolution kernels. In fact, to transform a layer map $\boldsymbol{F}_1$ of size (C_1, H, W) to $\boldsymbol{F}_2$, of size (C_2, H, W), requires the learning of

$$W_{\text{base}} = C_1 \times C_2 \times 3 \times 3 \tag{4.3}$$

weights. Reducing the extent of convolutions from 3×3 to 1×1 divides the number of weights to learn by 9. A reduction of the convolution's radius also diminishes the receptive field of the layer. Squeezenet then employs an hybrid strategy to reduce the number of features on which the 3×3 convolution operates. Starting from $\boldsymbol{F}_1$, it applies a 1×1 convolution to generate $\boldsymbol{F}_1^{'}$, of size $(C_1^{'}, H, W)$ with $C_1^{'} < C_1$. It concludes with a 3×3 convolution to go from $\boldsymbol{F}_1^{'}$ to $\boldsymbol{F}_2$. The total number of weights is now

$$W_{\text{opt}} = C_1 \times C_1^{'} \times 1 \times 1 + C_1^{'} \times C_2 \times 3 \times 3. \tag{4.4}$$

By comparison with the initial number of weight needed, the weight ratio is

$$\frac{W_{\text{opt}}}{W_{\text{base}}} = \frac{C_1 C_1^{'} + 9 C_1^{'} C_2}{9 C_1 C_2} = \frac{C_1^{'}}{9 C_2} + \frac{C_1^{'}}{C_1}. \tag{4.5}$$

This fraction shows that when choosing small values for $C_1^{'}$, the number of weights decreases.

This strategy, applied by (Iandola et al., 2016), is the foundation of the micro-architectural `Fire` module. With it, they reduce Alexnet's number of parameters by $50\times$ and achieve similar results.

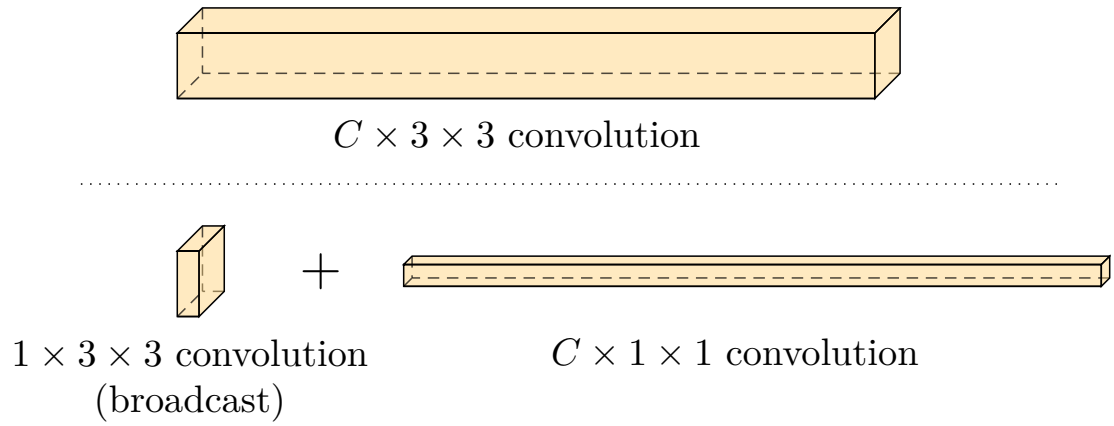


Figure 4.5: Difference between a traditional 3×3 convolution and a separable one. On top, the regular filter's size is $C \times 3 \times 3$. On the bottom, using first a $1 \times 3 \times 3$ filter, applied on the C layers, then performing a $C \times 1 \times 1$ convolution reduces the number of weights needed.

The Mobilenet architectures are based on a quite similar idea with separable convolutions (Howard et al., 2017). Once again, the idea is to limit the use of traditional 3×3 convolutions. Separable convolution first applies a 3×3 filter, constant for all input layers. Then, a 1×1 convolution is applied, as explained on fig. 4.5. At the macro-architecture level, Mobilenets are generated based on a coefficient α that impacts the depth of the layers in the network. With this parameter, it is easy to limit the network size, depending on the run-time platform. Figure 4.6 shows the results of Mobilenets depending on the size of the networks. In our work, we try Mobilenets, with different sizes, as feature extractors.

In a follow-up paper, Sandler et al., 2018 employ the so-called inverted-residual modules. Compared to the original Mobilenet module, they change the layers where the activation function is applied. They also introduce skip-connections between layers with a few channels.

Shufflenet is another efficient architecture that bases its design on group convolutions. This function splits a feature tensor into several parts and applies convolutions to them independently. Group convolution reduces the number of operations to do and the number of weights to learn (Zhang et al., 2018).

All this design effort for lightweight and fast classification neural networks has been re-used for per-pixel output architectures (Briot et al., 2018). For example, Shufflenet has been utilized by Gamal et al., 2018 as the backbone of a semantic segmentation network, or Mobilenet by Ghosh et al., 2019. At the outer-architecture level, a particular focus for pixel-wise-output networks is the image resolution and where to downsample it. Having smaller tensors to handle is com-

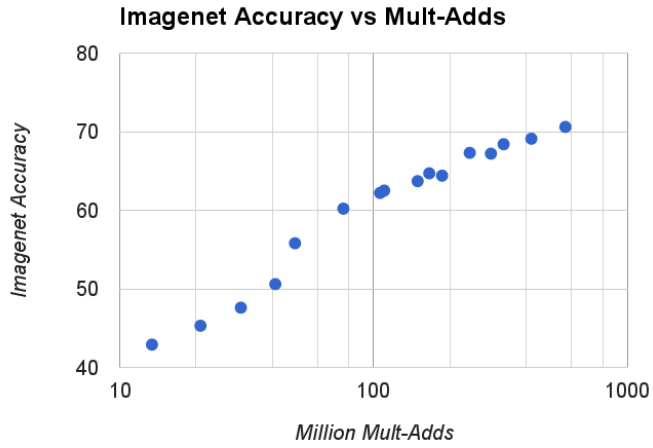


Figure 4.6: MobileNets' accuracy results with different choices of α. Higher is better. Note the log-linear relation between the number of mult-adds and the attained accuracy on Imagenet classification. From (Howard et al., 2017).

putationally efficient but also results in a loss of accuracy. With BiSeNet, Yu et al., 2018 create two paths in the network. One is used for context information and is downscaled early, and the other is focused on details and has a finer resolution but shallower.

More recently, methods have emerged for automatic neural architecture search (NAS). The first solution uses evolutionary algorithms or reinforcement learning techniques that test many networks to find the optimal one. This search method is costly. For example, Zoph et al., 2018 attain state-of-the-art results, but at the cost of searching the architecture for 2000 GPU days. Conversely, differentiable methods need less computational power to find a solution. For that, they define a large, directed acyclic graph. Nodes represent operations, such as a convolution, and edges are the network's connections. Every edge is assigned a weight that represents its contribution to the network. The NAS then consists in finding optimal edges' weight, via a gradient descent (C. Liu et al., 2019; Yan et al., 2020). Hybrid strategies also exist to optimize non-differentiable criteria in reasonable time (Vahdat et al., 2020).

4.2 Deploying PWC-Net on Jetson Xavier

PWC-Net has been chosen as the base network for our study. This architecture is often re-used in the litterature (Bar-Haim & Wolf, 2020; P. Liu et al., 2019; Zhao et al., 2020) and provides accurate results for a relatively small computational and memory footprint. The initial objective of our study is twofold, deploy the network

on an embedded chip, the NVIDIA Jetson Xavier, and find ways of improving its run-time without changing the network's architecture.

The first section details the PWC-Net architecture. Then, we explain the workflow we used for inference on an edge device and show results with an `fp16` inference.

4.2.1 Architecture

PWC-Net, as presented in fig. 4.4, can be sought of as two different parts. A "back-end" that computes multi-scale features of the two input images and a "head" that generates optical flow from those features. The back-end, or feature extractor, is a traditional CNN, as presented on fig. 4.7. Only some feature maps serve as input for subsequent flow generation. Like traditional optical flow approaches, the multi-scale flow is used for a warping step. This time, on features directly, instead of the sub-sampled images.

This operation can be seen as $\mathcal{W}$ on fig. 4.8. The warped features of the second image are then correlated, through $\mathcal{C}$ with those of the first image. Features from this correlation are concatenated with the first image features, the previous flow, and hidden features from the previous level to generate flow. $\mathcal{F}_f$ is a sub-network, a CNN, that generates a new tensor. From this intermediate result, a prediction of the flow is made with $\mathcal{P}$. In addition, and a shallower but larger tensor is generated for the next level by U_h. U_f upscales the flow with a learned "transposed convolution".

In terms of training, PWC-Net is sensitive to its initialization (Sun et al., 2018b). We tried, without success, to reproduce results from the authors in the PyTorch environment. The results we provide in this chapter have been obtained by re-using network weights converted from a Caffe training to its equivalent PyTorch model. They are available on the authors' repository (Sun, 2018, June 13/ 2021).

4.2.2 Deployment

The high popularity of neural networks in the image processing community has seen the emergence of various frameworks for developing and training deep networks. These tools are designed for ease of use and flexibility and adapted for creating new architectures and training networks. For the deployment of those algorithms, it is, however, often necessary to use specialized software, as mentioned in section 4.1.2.

Our work uses PyTorch as the training platform, and we have chosen the TensorRT toolkit to target NVIDIA GPUs. The inter-operability of these two

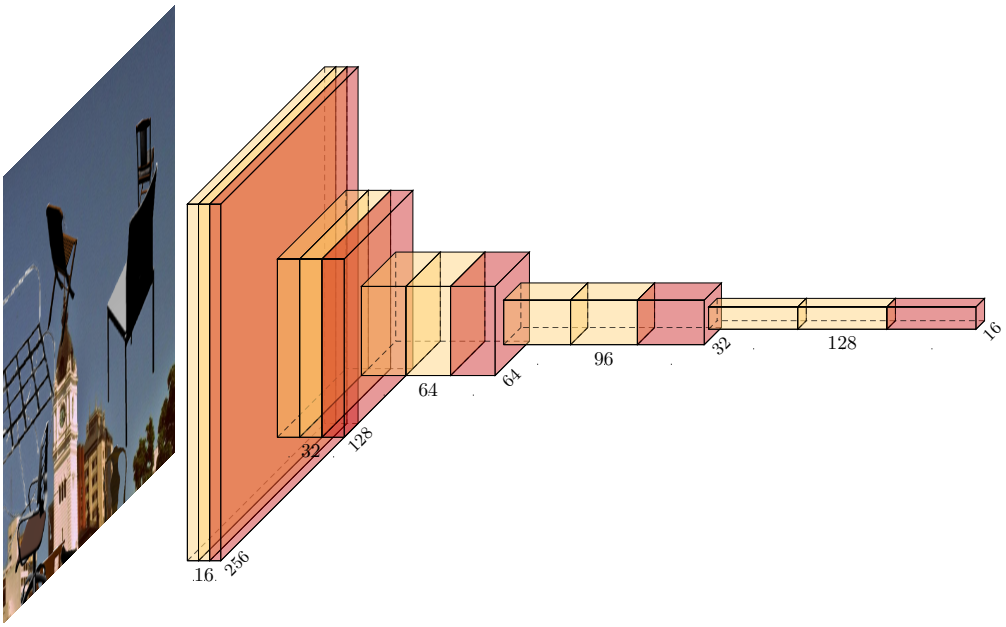


Figure 4.7: PWC-Net's feature extraction. Each block represent a feature tensor. Red blocks are used for optical flow estimation.

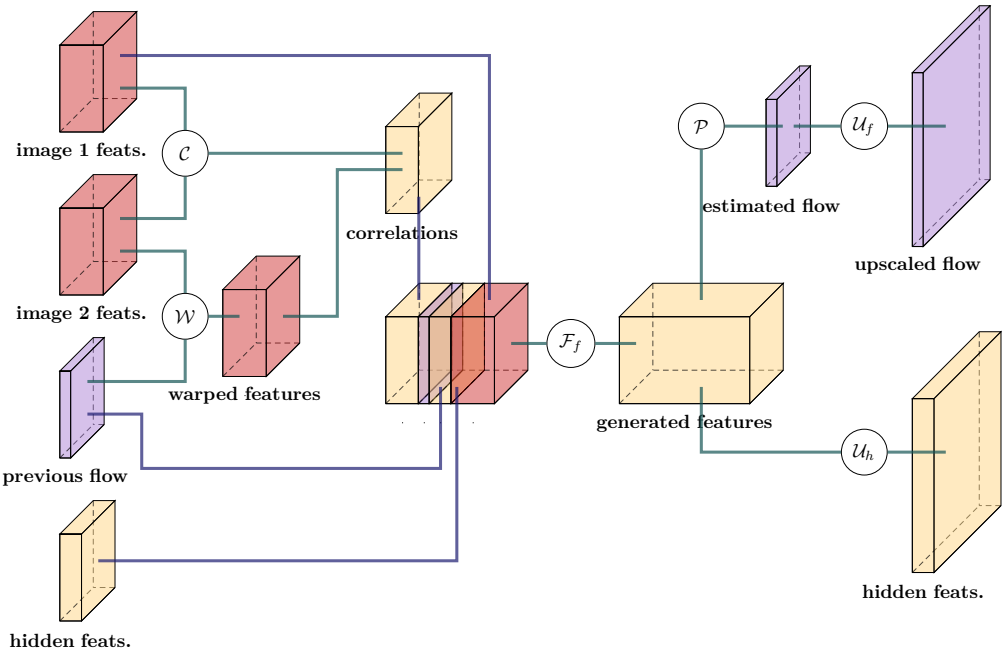


Figure 4.8: Optical flow prediction of PWC-Net at a single scale. The previous flow, features of the two images and other hidden coefficients serve at generating a new tensor. From this tensor, a new flow is estimated and up-scaled.

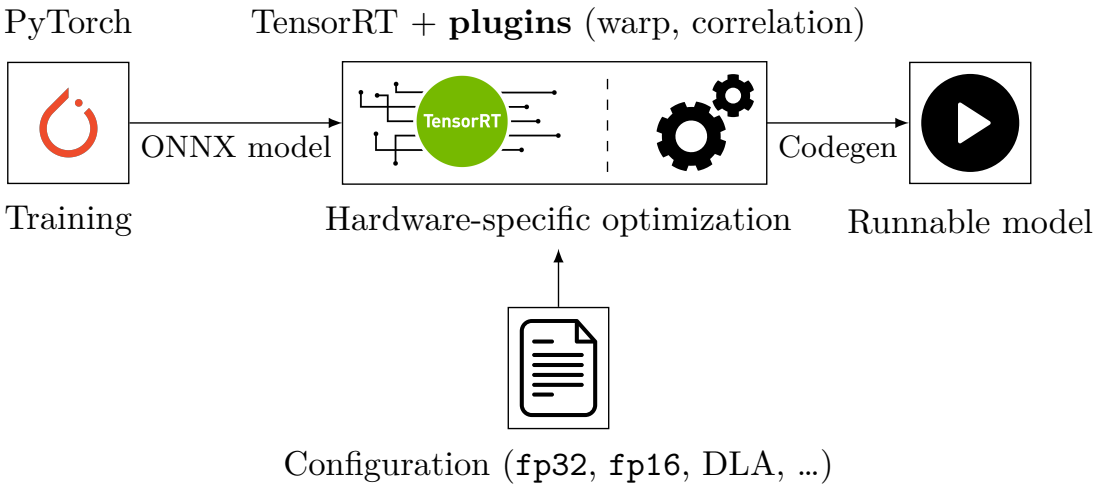


Figure 4.9: The deployment path from a PyTorch model to its execution on Xavier. It is first converted to the ONNX format that TensorRT accepts for optimization. We developped correlation and warp plugins for TensorRT to handle these non-standard operations.

tools is permitted by the intermediate ONNX description of a neural network, as depicted on fig. 4.9.

The network is exported from the PyTorch environment and imported into TensorRT. Then, it goes through a series of optimizations and run-time parameters tuning, depending on the actual hardware. When TensorRT has found its best possible way of executing the network, it saves the configuration into a binary file. This file can later be loaded to run the network.

One common limitation of tools like TensorRT is that they rely on standardized layer operations. The ONNX specification, for example (ONNX, 2017, September 7/2021), defines the standard 2D convolution with variants like stride, dilation, or group convolutions. If an operator is missing from the tool, it is necessary to rely on custom definitions. An ONNX description can reference custom operators, and TensorRT must have a corresponding implementation to execute the network. In our case, the warp $\mathcal{W}$ and feature correlation $\mathcal{C}$ operators were missing from ONNX and TensorRT.

These two operations are indeed not that common in the neural network community. We have implemented both of them as TensorRT plugins to execute PWC-Net on an embedded platform. The warp operation is straightforward and consists of a bi-linear interpolation of pixels' value, displaced by their optical flow.

An implementation of feature correlation, as defined in (Dosovitskiy et al., 2015), can be found in (Mayer, 2017, April 25/2021), but it made for the Caffe

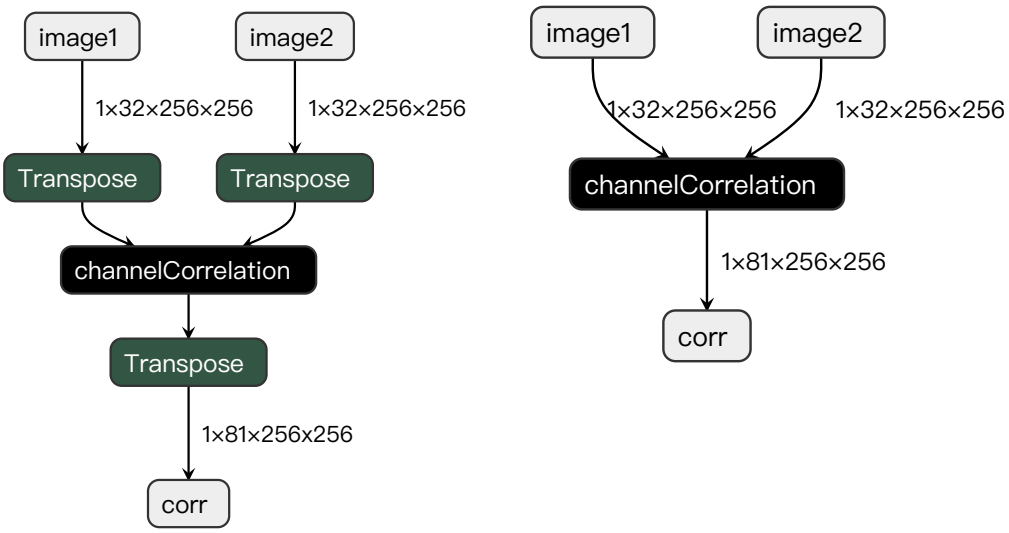


Figure 4.10: Two equivalent ways of performing the correlation operation. Left: the method used in the original implementation, with tensors transposed in the NHWC format. Right: our proposed implementation, that operates directly in the NCHW format.

	NHWC (transpose)	NCHW	NCHW + shared mem.
Timings (ms)	19.7	18.7 (×0.95)	**15.2** (×0.77)

Table 4.1: Median runtime of the correlation layer on Jetson AGX Xavier with (1, 32, 256, 256) tensor inputs.

framework. We started by porting this code to TensoRT as a plugin. The main issue is that Caffe supports network tensors in the NHWC (batch, height, width, channel) memory ordering, and TensorRT rather uses NCHW. Re-using the initial implementation is still possible, at the price of transpositions, as shown in fig. 4.10.

With the missing $\mathcal{W}$ and $\mathcal{C}$ operations implemented, the TensorRT toolkit can deploy PWC-Net. We use the `trtexec` executable to optimize the network on Jetson Xavier and generate a binary model representation. Then, using TensorRT's Python API, we profile the network execution on images from the FlyingChairs dataset.

Precision	Architecture	EPE	FPS
`fp32`	PWC-Net	2.28	20.7
`fp16`		2.28	**36.3 (×1.75)**

Table 4.2: PWC-Net's performance on a Jetson AGX Xavier with 512×384 images.

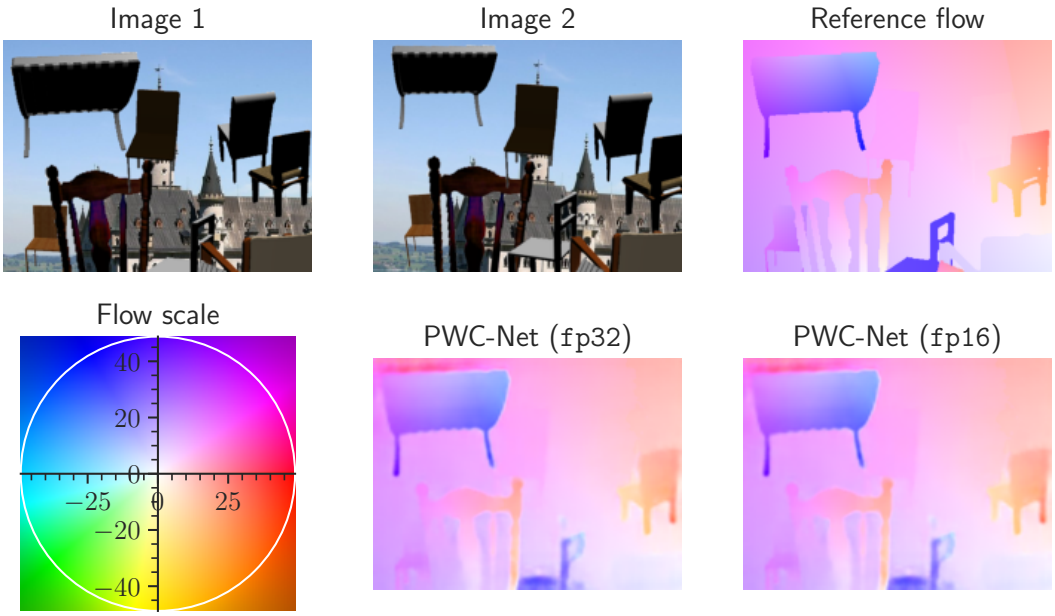


Figure 4.11: PWC-Net's execution on Jetson Xavier. With or without `fp16` enabled when generating the model with TensorRT. Figure created with `flowpy` (M. Seznec, 2021).

4.2.3 Results

The first tests we conducted were on the correlation layer alone. We made a re-implementation that supports natively NCHW tensors. Since this operation involves accessing the pixel's neighbors, it is possible to use the GPU's shared memory to efficiently re-utilize shared values. In table 4.1 we show that with this optimization, we attain a 30% speed-up versus the initial NHWC implementation.

Table 4.2 shows the results of PWC-Net's execution on the Jetson AGX Xavier. Its initial implementation, in `fp32`, attains 20.7 frames per second (FPS) on images from the FlyingChair dataset. The EPE is computed averaged over its validation split. Then, we configured TensorRT to use `fp16` where applicable. The engine chooses from `fp32` or `fp16` to find the best trade-off between acceleration and

keeping a good accuracy (nvidia, 2021). Indeed, the EPE in `fp16` does not change from the `fp32` version, which can be seen on fig. 4.11, but the performance increases by 75% to reach 36.3 FPS.

These results are promising. With more than 30 FPS, the execution is already acceptable on relatively small images (512×384) for a real-time embedded application.

4.3 MobileFlow: an hybrid model based on efficient networks

We now explore architectural changes to push the initial results PWC-Net further. For that, we leverage an existing neural network classifier, MobileNetV2, designed for embedded applications. Using it should both help the learning phase of the network by relying on transfer learning and ultimately increase inference's performance.

4.3.1 Architecture and learning method

The original PWC-Net is trained as a whole, include both feature extraction and flow generation. We propose to replace its back-end, the feature extraction, with a pre-existing architecture: MobileNetV2 (Sandler et al., 2018). This network is designed for image classification, but its initial layers extract image features that can be used to feed PWC-Net's optical flow head. "Plugging" MobileNet into PWC-Net's head is the first step towards the MobileFlow architecture.

The second change deals with the flow generation step. First, the $\mathcal{F}_f$ sub-network, as defined in fig. 4.8, is switched. In the original PWC-Net, it is made of classic convolutions with dense connections. For MobileFlow, we change those convolutions with depth-separable convolutions that were presented in fig. 4.5. The predictor $\mathcal{P}$ follows the same replacement strategy. Finally, $\mathcal{U}_f$ and $\mathcal{U}_h$, that upscale the flow and hidden features, are changed from transposed convolutions to a separable convolution followed by bilinear re-sampling. This choice has the advantage of reducing the number of learnable weights and reduces potential artifacts (Odena et al., 2016).

We re-use MobileNets of several widths, controlled by α. For example, MobileFlow (0.25) refers to the network obtained with a Mobilenet of width 0.25. The Mobilenet back-end is pre-trained on ImageNet, an open-source dataset (Deng et al., 2009), and the weights are frozen for the subsequent flow learning. During the training, only the optical flow estimation layers are learned. We use the *long* training schedule defined in (Ilg et al., 2017).

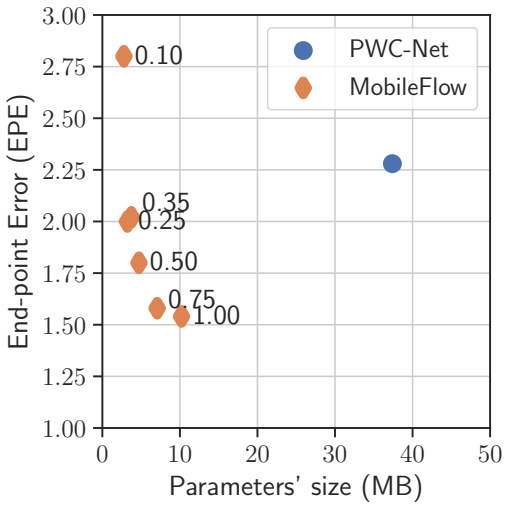


Figure 4.12: End-point errors vs. network parameters' size on the validation split of FlyingChairs. Lower is better. The disc is the PWC-Net reference (Sun et al., 2018b). Diamonds represent MobileFlow networks, labelled with their corresponding Mobilenet widths.

4.3.2 Results

On fig. 4.12, we report the average end-point error (EPE) obtained on the validation data of the FlyingChairs database. In terms of size, it is clear that MobileFlow networks are lighter than PWC-Net. The largest MobileFlow architecture requires learning 10MB of parameters, while PWC-Net uses more than 35MB. The end-point error of our proposed architecture is close to 1.50 for the largest network but degrades with smaller networks. The accuracy of PWC-Net has been estimated with weights from (Sun, 2018, June 13/2021) at 2.26 but is likely overestimated, as the training procedure used for this result was not optimal (Sun et al., 2018a).

These results prove that using a pre-trained architecture as a feature extractor work for optical flow estimation. Even if the database used for pre-training, ImageNet, is not the same as the one involved in the optical flow, FlyingChairs, this transfer learning procedure provides satisfying results. We also report that training this hybrid architecture was less sensitive to weight initialization and local minima, compared to PWC-Net (Sun et al., 2018a). The pre-trained feature extractor seems to, indeed, provide starting points that avoid poor local minima.

Figure 4.13 shows an example of flow estimation on the image 06565 of the FlyingChairs dataset. Compared to PWC-Net, MobileFlows provide similar results visually. The networks accurately estimate the background displacement and large

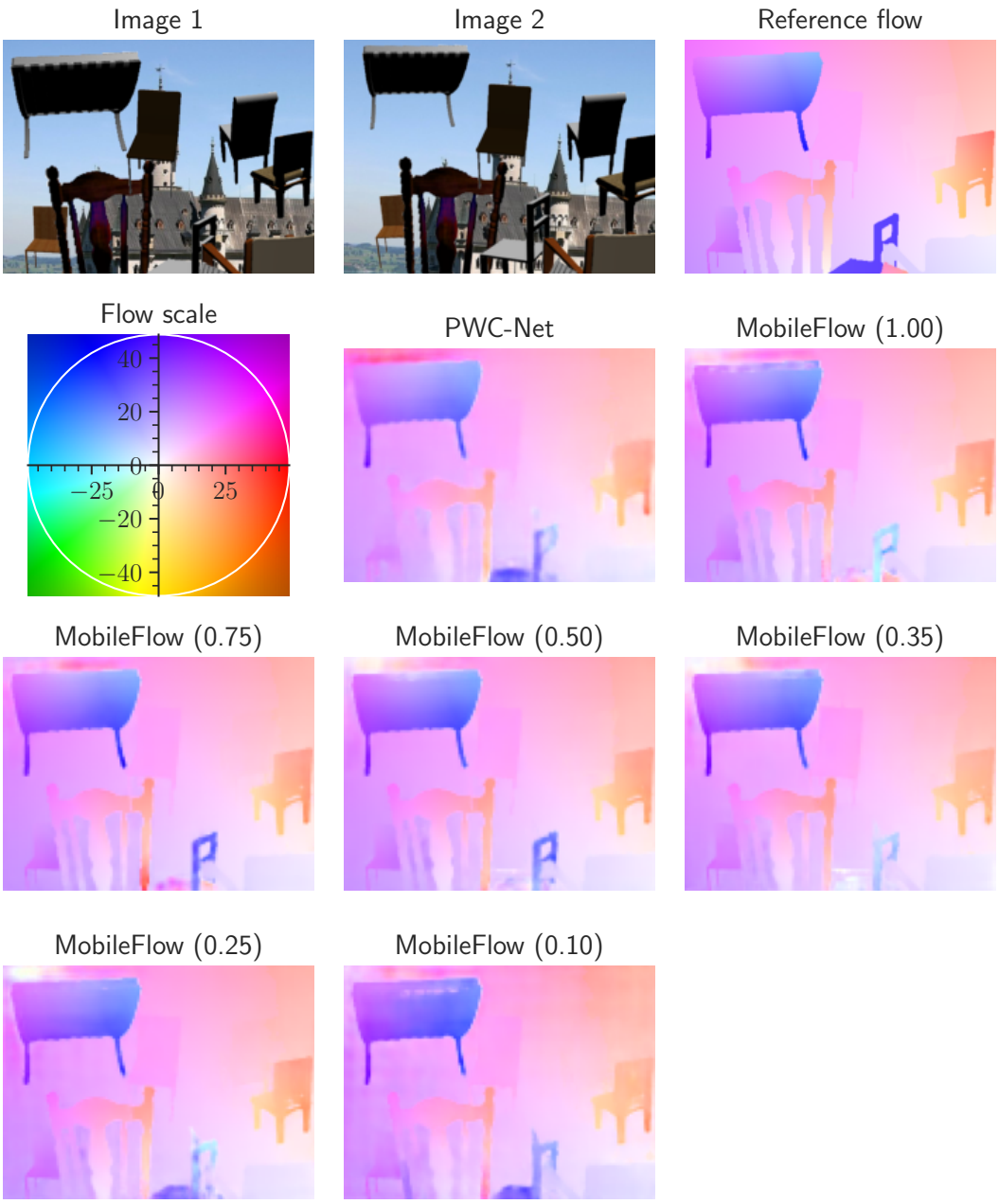


Figure 4.13: Results of PWC-Net and different versions of MobileFlow on a FlyingChairs sample.

Precision	Architecture	Width	EPE	FPS
`fp32`	PWC-Net	-	2.28	20.6
	MobileFlow	0.10	2.80	**22.4**
		0.25	2.00	21.1
		0.35	2.01	20.5
		0.50	1.80	19.2
		0.75	1.58	17.3
		1.00	**1.54**	16.6
`fp16`	PWC-Net	-	2.28	36.1
	MobileFlow	0.10	2.80	**44.1**
		0.25	2.00	41.2
		0.35	2.01	39.9
		0.50	1.80	36.7
		0.75	1.58	32.2
		1.00	**1.54**	31.0

Table 4.3: Runtime of MobileFlow networks and PWC-Net on the Jetson AGX Xavier with 512×384 images.

displacements, with the top-left chair, for example. Smaller objects sometimes generate artifacts, especially with overlapping chairs, on the bottom-right corner, for example.

We then compared the FPS throughput of MobileFlows with PWC-Net, once again, using TensorRT on Jetson Xavier. There is no need for additional plugins as TensorRT already handles operations such as depth-wise convolutions. Table 4.3 details the results in `fp32` and `fp16` of PWC-Net and MobileFlow with different back-end widths.

The largest versions of MobileFlow are slower than PWC-Net, but below a depth of 0.35, they overcome the reference network's performance. Since the accuracy of PWC-Net is likely to be under-estimated, we can conservatively use MobileFlow (0.25) as a fair comparison with PWC. Then, we observe that its performance it marginally better in `fp32` (+2%) but this gain increases with `fp16` arithmetic (+14%). The probable explanation is that TensorRT allows more layers to use `fp16` on MobileFlow than PWC because its output is more stable numerically.

Figure 4.14 details the time spent by the different networks' inference, and was obtained with `trtexec` profiling. The plot groups the results into three categories: convolutions, correlations, and other layers. It appears that the most significant difference between those architectures is the time spend doing correlations. Indeed,

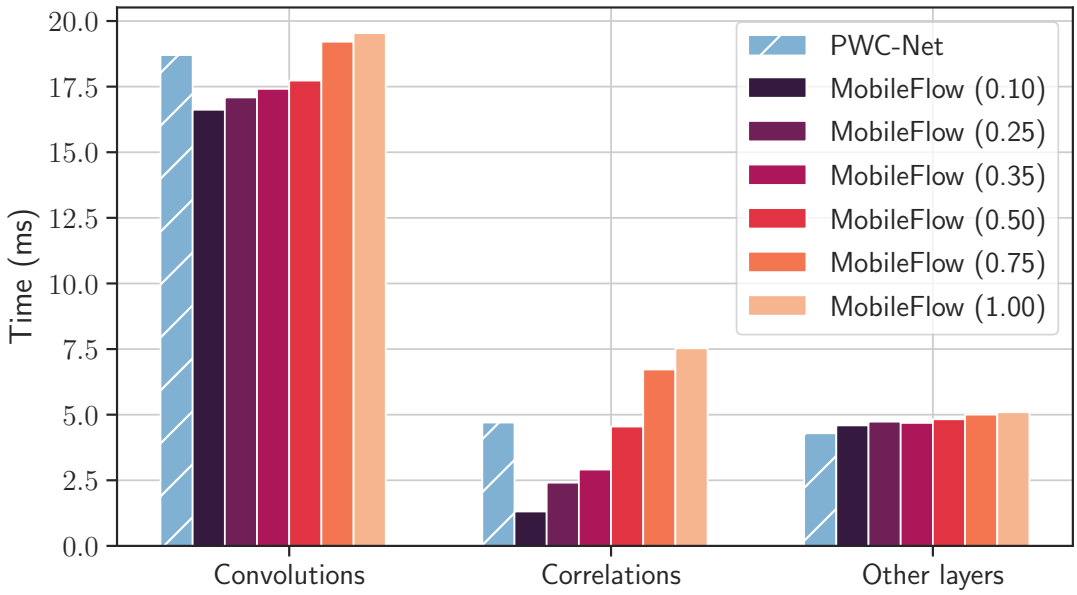


Figure 4.14: Inference runtime breakdown, grouped by layer type, in `fp16` precision.

this processing is affected by the depth of the input tensors, which varies quite a lot across the different versions.

Conversely to what we expected, the time spend doing convolutions is not decreased significantly on MobileFlow models. Across the different versions of MobileFlow, this processing time is reduced from about 19 ms to 16.5 ms. This small difference suggests that the optical flow estimation part of the network occupies most of the GPU time. Also, this result seems to indicate that depth-wise convolutions are not much faster than their traditional counterparts.

4.4 Conclusion

This study on neural networks for optical flow has permitted an embedded implementation of PWC-Net on Jetson AGX Xavier. To the best of our knowledge, this step had never been done before. Developing two TensorRT plugins and exporting the corresponding layers in ONNX before running the network on Jetson Xavier has been necessary. With an initial performance of 20.6 FPS, the inference was then allowed to use `fp16` to reach 36.1 FPS, a 75% increase.

For even better performance, we have searched for new efficient network designs. At the architectural level, we have demonstrated that it is possible to use

a pre-trained "back-end", derived from a classification network, and only learn the optical flow estimation layers. This transfer learning procedure has several advantages.

First, it simplifies the design of the network. By leveraging pre-existing architectures, we can capitalize on state-of-the-start results and focus on flow generation layers. Second, it is easier to find a database to train the back-end. It is indeed challenging to obtain annotated examples for optical flow compared to classification labeling. Moreover, classification networks are easily found with pre-trained versions on popular datasets. Third, a complex image processing pipeline may share the MobileNet back-end with classification, detection, or pixel segmentation "heads". Several tasks can then share a common backbone to save many computations.

After proving that this hybrid network structure is possible, we have also deployed the DNN on an embedded target. In `fp16` precision and compared to PWC, MobileFlow (0.25) shows similar accuracy, but is 14% faster. The network weights are also reduced by 92%, even if the memory footprint of the inference is mostly determined by tensors that must be kept in memory.

Going further, it should be interesting to assess the performance of `int8` quantization of the network. This step might increase the network's performance drastically (Venkata Bhargava Narendra et al., 2021) but has a more significant impact on the accuracy. The use of NVIDIA's DLAs (Deep Learning Accelerators), which are present on the Xavier SoC, might also be beneficial. Developing and integrating custom operations, such as correlation and warp, requires, however, a significant development effort.

Other work may target the design of the network to increase the performance of MobileFlow. More analysis should be conducted to understand the effect of depth-wise convolutions and why they do not bring much acceleration. Also, a new architecture that contracts the feature tensors before performing the correlation could be considered. This "squeezenet-like" (Iandola et al., 2016) could dramatically reduce the time spend in this operator. Going further, neural architecture search techniques may also provide new efficient designs.

Finally, it should be interesting to assess the back-end pre-training's effects on the final accuracy. Is there an advantage at training the back-end for classification on images used in production if no optical flow ground truth is available for them?

Conclusion

This manuscript has detailed the research effort conducted with the objective of a better understanding of the way various algorithms can be modified to fit modern GPU architectures. This work was motivated by the need to run computer vision methods on low-power platforms. To that end, it is crucial to use the provided hardware to its full capability. While traditional deployment strategies use two main phases, software design, and hardware implementation, we proposed an additional in-between stage. It consists of a software-hardware adaptation of an algorithm and finds whether it is possible to run it on the proposed platform. The advantage of this additional step in the deployment process is that it combines expertise from both software and hardware fields to find high-level optimizations.

We began by introducing the optical flow problem, an algorithmic context that served as a basis for most of the work of this thesis. This complex computer vision processing is quite representative of the difficulties of pixel-level tasks. This way, we hope that our results may be transferable to other related tasks, such as semantic segmentation. Then, we drew an overview of the hardware landscape. This complex field is motivated by a significant observation: the mono-CPU architecture can no longer reach the highest performance and has a low performance per Watt ratio. Due to power density and frequency limitations, various parallel accelerators have been brought to the forefront. Each of them has capabilities and restrictions in the type of computations they perform. The GPU architecture was presented in-depth as it is now a common type of hardware for parallel computing and is used industrially. We also discussed a methodology for implementing computer programs on a specific platform. We showed that extending the execute-profile-optimize loop to algorithmic choices improves the journey from the algorithm to the implementation.

With the context set and the development method being in place, we have presented results in the context of radio-astronomy. As a first step, we proposed to optimize a single method, the image convolution, to obtain the best performance on a GPU. After comparing several methods that execute this operation, we have selected a frequency-based implementation with the help of the `cuFFT` library. We have shown that the direct use of the `fp16` precision leads to the divergence of the

iterative reconstruction process. Instead, storing data in `fp16` but operating on them with the extended `fp32` precision is a beneficial trade-off between accuracy and speed. We then explored the use of tensor core units for the image convolution in a more general context. The algorithm we proposed is more efficient than other spatial domain convolutions with large kernels. Our implementation outpaces the best-known algorithms by a factor of two for kernels of size 30 while maintaining a better accuracy because of the extended precision used within tensor cores.

When chapter 2 showed results with changes for single operators within an algorithm, chapter 3 extended the analysis to entire sub-algorithms. For that, we used the Combined Local-Global algorithm for optical flow estimation as the root of our work. This method relies on an iterative scheme to solve a system of linear equations. Many solvers can perform this task, but some are more efficient than others on GPU hardware. Thanks to an initial review, we have chosen the Jacobi solver as it showed good properties in terms of speed and accuracy. With this algorithm-level choice made, we deepened our optimization to the implementation level. Techniques like iteration fusion, kernel batching, or memory re-utilization allowed us to increase the throughput of the computations by a factor of two. Thanks to this combined reflection on the algorithm at the method and implementation levels, we ran the CLG method on a Jetson Xavier at 60 FPS.

In chapter 4, we opened the perspective of algorithm modifications to deep neural networks. After reviewing the state-of-the-art of machine learning for optical flow estimation, we detailed DNN architectures designed for efficient inference on low-power platforms. Our initial contribution consisted in the port of PWC-Net to the Jetson AGX Xavier, thanks to TensorRT. Then, we modified this initial network to replace its feature extraction module with MobileNet, an efficient classifier. This transfer-learning procedure has led to the MobileFlow architecture. It has better accuracy than PWC-Net for similar or better inference speeds.

In the end, our experiments plead for a stronger bond between algorithm design and implementation teams in industrial development. In many contexts, they are two distinct entities with limited interactions. Here, we played the role of the middleman that allows cooperation between the two. This position seems essential for algorithm performance enhancement, as it opens up a new range of possible improvements.

Several areas of improvement emerge from the work explored in this manuscript. Regarding the use of reduced or mixed precision, mentioned in section 2.1, automatic floating-precision accuracy checks exist (Févotte & Lathuilière, 2019). This kind of tool helps to understand the numerical stability of arithmetic expressions and provides insights on where using lower precision is possible. Regarding our *im2Tensor* algorithm, presented in section 2.2, improvements may come from the use of a lower-level API for tensor core usage (the `mma` API). This finer-grained

control could lead to better performance by avoiding conflicting accesses to the GPU's shared memory.

Chapter 3 presented CLG, an optical flow estimation algorithm, and ways of improving its linear algebra system resolution. Solvers such as Jacobi or Gauss-Seidel have been explored, but other types of solving procedures, like multi-grid methods, could be studied, especially their impact on GPU. This kind of technique aims to solve the linear system on lower resolutions, thereby reducing the complexity but also the possible parallelization. Exploring the trade-off between these two factors, especially on GPU, seems essential for a high-performance implementation. Going further, it would make sense to understand the impact of CLG in a more complex image processing pipeline. The usage of optical flow for real-time object detection or image super-resolution requires heavy processing. The parametrization of CLG (number of scales, iterations per scale) impacts its run-time but also the quality of the estimation. Knowing how the flow is used afterward makes possible the exploration of the speed/accuracy balance.

For deep neural networks design, the current trend focuses on automatic neural architecture search. This process permits the development of neural architectures for a specific hardware (Cai et al., 2020). In the case of optical flow estimation, this technique should be applicable for searching for better feature extractors or optical flow estimators, either separately or all together.

Finally, this thesis has targeted a specific software/hardware context, image processing methods on GPU. This particular setting offered insights on what is possible to achieve in terms of optimizations, but it could be extended to other platforms and types of algorithms. As mentioned in chapter 1, FPGAs, TPUs, or manycore architectures are privileged hardware that may lead to other types of optimizations. The type of algorithm is also fundamental. GPUs, for example, favor image and video processing. Conversely, DSPs or FPGAs might be more efficient for unidimensional signal processing. Following this idea, it might be interesting to relax the fixed-hardware constraint in our implementation methodology.

Opening up the choice of the architecture in the deployment loop would probably lead to more efficient solutions at the cost of increasing the optimization search space and, therefore, the overall complexity. This difficulty could be handled by relying on higher-level tools for software deployment, such as the ones presented in section 1.3, and would also be mitigated by the development of hardware-agnostic profiling and analysis tools, using the roofline model, for example.

Scientific contributions

- **International conference:** Seznec, M., Gac, N., Ferrari, A., & Orieux, F. (2018, October). A Study on Convolution using Half-Precision Floating-Point Numbers on GPU for Radio Astronomy Deconvolution, In *2018 IEEE International Workshop on Signal Processing Systems (SiPS)*. 2018 IEEE International Workshop on Signal Processing Systems (SiPS), Cape Town, IEEE. https://doi.org/10.1109/SiPS.2018.8598342 (https://hal.archives-ouvertes.fr/hal-01837982)

- **International conference:** Seznec, M., Gac, N., Orieux, F., & Naik, A. S. (2020b, October). An Efficiency-Driven Approach For Real-Time Optical Flow Processing On Parallel Hardware, In *2020 IEEE International Conference on Image Processing (ICIP)*. 2020 IEEE International Conference on Image Processing (ICIP). https://doi.org/10.1109/ICIP40778.2020.9191164 (https://hal.archives-ouvertes.fr/hal-02604755)

- **Poster:** Seznec, M., Gac, N., Orieux, F., & Naik, A. S. (2020a, May). A new convolutions algorithm to leverage tensor cores. Retrieved August 13, 2021, from https://hal.archives-ouvertes.fr/hal-02605077
  Published: GPU Technology Conference (GTC)
  (https://hal.archives-ouvertes.fr/hal-02605077)

- **Software package:** Seznec, M. (2021). *Flowpy: Tools for working with optical flow* (Version 0.6.0). Retrieved August 13, 2021, from https://gitlab-research.centralesupelec.fr/2018seznecm/flowpy https://pypi.org/project/flowpy/

- **Journal article:** Seznec, M. Gac, N. Orieux, F. & Naik, A. S. The Im2Tensor Algorithm for Efficient 2D Convolutions on GPU Tensor Cores. Journal Article. (Under review by the Journal of Real-Time Image Processing)

- **Journal article:** Seznec, M. Gac, N. Orieux, F. & Naik, A. S. Real-Time Optical Flow Processing on Embedded GPU: an Hardware-Aware Algorithm

to Implementation Strategy. Journal Article. (Under review by the SIAM Journal on Scientific Computing)

Résumé en français

Les algorithmes de traitement numérique actuels nécessitent une puissance de calcul accrue pour obtenir des résultats plus précis et traiter des données plus volumineuses. Dans le même temps, les architectures matérielles se spécialisent, avec des accélérateurs très efficaces pour des tâches spécifiques. Dans ce contexte, le chemin du déploiement de l'algorithme à l'implémentation est de plus en plus complexe. Il est donc crucial de déterminer comment les algorithmes peuvent être modifiés pour tirer parti des capacités du matériel. Dans notre étude, nous intéressons aux unités graphiques (GPU), un type de processeur massivement parallèle. Notre travail consiste à l'adaptation entre l'algorithme et le matériel d'exécution.

Le premier chapitre présente le contexte de notre étude. En premier lieu, nous décrivons les algorithmes étudiés dans ce manuscrit : la convolution d'image et l'estimation de flux optique. Ensuite, nous effectuons un état de l'art des différents processeurs pour le calcul haute performance. Nous montrons que la pérennité de la loi de Moore bénéficie à présent aux architectures matérielles spécialisées et fortement concurrentes. Ainsi, nous justifions notre utilisation des processeurs graphiques pour atteindre un rapport performance sur puissance consommée très élevé. Enfin, nous montrons les challenges que posent ce genre d'architecture. Le déploiement d'application est souvent plus compliqué qu'avec des processeurs séquentiels classiques. Nous faisons donc un tour d'horizons des langages de programmation, outils de développement et méthodes qui permettent atteindre les meilleurs performances possibles.

Dans le chapitre deux, nous modifions un algorithme de convolution d'images pour utiliser les tensor cores. Ces unités matérielles, propres aux GPU de la marque NVIDIA, permettent de calculer des produits matriciels très efficacement. La convolution d'image se calcule habituellement selon deux méthodes. Dans l'espace direct, il s'agit d'une somme de produit entre le filtre et l'image en chaque pixel. Dans l'espace de Fourier, c'est une simple multiplication point-à-point. Mais il faut alors procéder à une transformée de Fourier de l'image, du noyau, ainsi qu'une transformée inverse du résultat. Ces deux méthodes traditionnelles ne faisant pas appel à des multiplications de matrices, nous avons développé un nouvel algorithme qui ré-exprime la convolution pour utiliser les tensors cores.

Cette nouvelle méthode algorithmique fait intervenir la transposée du filtre ainsi qu'un tenseur construit à partir des coefficients de l'image. Après implémentation sur GPU, nous avons montré l'intérêt de la méthode, avec des résultats jusqu'à deux fois plus rapides pour de grands noyaux de convolution. Contrairement à d'autres algorithmes prévus particulièrement pour les réseaux neuronaux convolutifs, nous n'avons pas besoin d'utiliser des noyaux en paquets pour avoir une exécution efficace.

L'objet d'étude du chapitre trois se situe au niveau de la méthode algorithmique. Nous y évaluons des solveurs linéaires pour l'estimation de flux optique afin de trouver le plus adéquat sur GPU. Lorsque l'on compare les solveurs en vitesse de convergence par itérations effectuées, la méthode par gradient conjugués est bien meilleure que les autres de l'étude : Jacobi et Gauss-Seidel. Cependant, après implémentation sur GPU, cette avance est moins marquée, une itération de gradients conjugués étant plus longue qu'une de Jacobi. De plus, les gradients conjugués semblent plus sensibles au bruit numérique et ne permettent une convergence précise sur GPU. Nous avons donc sélectionné la méthode Jacobi et poursuivit son optimisation sur ce processeur spécifique. La fusion d'itérations nous permet notamment de doubler la rapidité de ce solveur. Une fois ce travail accompli, notre implémentation de l'estimation de flux optique fonctionne à plus de 60 images par seconde sur la carte électronique embarquée Jetson Xavier d'NVIDIA, pour une consommation électrique de 30W.

Le quatrième chapitre présente l'utilisation de réseaux neuronaux convolutifs pour l'estimation de flux optique. En effet, les réseaux de neurones profonds ont permis des nombreuses avancées quasiment tous les domaines de la vision par ordinateur. Pour le flux optique, ces méthodes dominent maintenant l'état de l'art. Cependant, l'utilisation de telles méthodes requiert un nombre de calculs conséquent ainsi qu'une grande empreinte mémoire. Il est alors difficile de les utiliser sur du matériel embarqué. Dès lors, nous avons cherché à concevoir un réseau léger qui permettrait un déploiement plus aisé. Pour cela nous sommes parti du réseau PWC-Net qui nous avons fusionné avec MobileNet. Ce dernier réseau, conçu pour la classification d'images, est léger et prévu pour être déployé sur des cibles à puissance limitée. En utilisant la stratégie de l'apprentissage par transfert, nous entrainons notre propre architecture, MobileFlow, composée en partie de PWC-Net et de MobileNet. Ce réseau hybride, comparé à PWC-Net, a un temps d'exécution similaire sur la carte Jetson Xavier, mais requiert sept fois moins de paramètre et a une précision accrue.

En conclusion, notre étude a permis de mieux comprendre les leviers qui permettent une implémentation plus efficace de certaines méthodes logicielles. Notre objet d'analyse se situe à la frontière entre une optimisation au niveau logiciel et matérielle. Trouver les meilleurs formulations algorithmiques en se basant sur les

forces du processeur qui se chargera de l'exécution nous a permis d'obtenir des implémentations très efficaces dans différentes situations. Pour continuer notre étude, on pourrait utiliser notre l'algorithme de convolution par multiplication de matrice sur d'autres plateformes que les GPU NVIDIA. L'étude du flux optique par solveur linéaire pourrait être raffinée pour prendre en compte les solveurs multi-échelles. Enfin, concernant l'utilisation des réseaux de neurones de flux optique, une perspective d'amélioration serait l'utilisation de coefficient à précision réduite (int8) pour faciliter leur utilisation sur plateformes embarquées.

Bibliography

Amdahl, G. M. (2013). Computer Architecture and Amdahl's Law. *Computer*, *46*(12), 38–46. https://doi.org/10.1109/MC.2013.418

Anderson, A., Vasudevan, A., Keane, C., & Gregg, D. (2017, September 8). *Low-memory GEMM-based convolution algorithms for deep neural networks*. Retrieved September 10, 2019, from http://arxiv.org/abs/1709.03395

Bai, M., Luo, W., Kundu, K., & Urtasun, R. (2016). Exploiting Semantic Information and Deep Matching for Optical Flow (B. Leibe, J. Matas, N. Sebe, & M. Welling, Eds.). In B. Leibe, J. Matas, N. Sebe, & M. Welling (Eds.), *Computer Vision ECCV 2016*, Cham, Springer International Publishing. https://doi.org/10.1007/978-3-319-46466-4_10

Baker, S., & Kanade, T. (1999). *Super-Resolution Optical Flow*. Carnegie Mellon University.

Baker, S., Scharstein, D., Lewis, J. P., Roth, S., Black, M. J., & Szeliski, R. (2011). A Database and Evaluation Methodology for Optical Flow. *International Journal of Computer Vision*, *92*(1), 1–31. https://doi.org/10.1007/s11263-010-0390-2

Bar-Haim, A., & Wolf, L. (2020, February 25). *ScopeFlow: Dynamic Scene Scoping for Optical Flow*. Retrieved March 11, 2020, from http://arxiv.org/abs/2002.10770

Beauchemin, S. S., & Barron, J. L. (1995). The computation of optical flow. *ACM Computing Surveys*, *27*(3), 433–466. https://doi.org/10.1145/212094.212141

Bichler, O., Briand, D., Gacoin, V., Bertelone, B., Allenet, T., & Thiele, J. (2017). *N2D2-neural network design & deployment*. https://github.com/CEA-LIST/N2D2

Briot, A., Viswanath, P., & Yogamani, S. (2018). Analysis of Efficient CNN Design Techniques for Semantic Segmentation. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops. Retrieved

August 5, 2021, from https://openaccess.thecvf.com/content_cvpr_2018_workshops/w12/html/Briot_Analysis_of_Efficient_CVPR_2018_paper.html

Brox, T., Bruhn, A., Papenberg, N., & Weickert, J. (2004). High Accuracy Optical Flow Estimation Based on a Theory for Warping (T. Pajdla & J. Matas, Eds.). In T. Pajdla & J. Matas (Eds.), *Computer Vision - ECCV 2004*, Berlin, Heidelberg, Springer. https://doi.org/10.1007/978-3-540-24673-2_3

Bruant, J., Horrein, P.-H., Muller, O., Groléat, T., & Pétrot, F. (2021). Towards Agile Hardware Designs with Chisel: A Network Use-case. *IEEE Design Test*, 1–1. https://doi.org/10.1109/MDAT.2021.3063339

Bruhn, A., Weickert, J., & Schnörr, C. (2005). Lucas/Kanade Meets Horn/Schunck: Combining Local and Global Optic Flow Methods. *International Journal of Computer Vision*, *61*(3), 1–21. https://doi.org/10.1023/B:VISI.0000045324.43199.43

Butler, D. J., Wulff, J., Stanley, G. B., & Black, M. J. (2012, October). A naturalistic open source movie for optical flow evaluation (A. Fitzgibbon et al. (Eds.), Ed.). In A. Fitzgibbon et al. (Eds.) (Ed.), *European conf. on computer vision (ECCV)*, Springer-Verlag.

Cai, H., Gan, C., Wang, T., Zhang, Z., & Han, S. (2020, April 29). *Once-for-All: Train One Network and Specialize it for Efficient Deployment*. Retrieved August 31, 2021, from http://arxiv.org/abs/1908.09791

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., & Krishnamurthy, A. (2018, October 5). *TVM: An Automated End-to-End Optimizing Compiler for Deep Learning*. Retrieved August 4, 2021, from http://arxiv.org/abs/1802.04799

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., & Shelhamer, E. (2014, October 3). *cuDNN: Efficient Primitives for Deep Learning*. Retrieved September 3, 2019, from http://arxiv.org/abs/1410.0759

Dagum, L., & Menon, R. (1998). OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, *5*(1), 46–55. https://doi.org/10.1109/99.660313

Deakin, T., & McIntosh-Smith, S. (2020, April 27). Evaluating the performance of HPC-style SYCL applications, In *Proceedings of the International Workshop on OpenCL*, New York, NY, USA, Association for Computing Machinery. https://doi.org/10.1145/3388333.3388643

de Dinechin, B. D., Ayrignac, R., Beaucamps, P.-E., Couvert, P., Ganne, B., de Massas, P. G., Jacquet, F., Jones, S., Chaisemartin, N. M., Riss, F., & Strudel, T. (2013, September). A clustered manycore processor architec-

ture for embedded and accelerated applications, In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. 2013 IEEE High Performance Extreme Computing Conference (HPEC). https://doi.org/10.1109/HPEC.2013.6670342

Delisle, P., Krajecki, M., Gravel, M., & Gagné, C. (2001, September). Parallel implementation of an ant colony optimization metaheuristic with OpenMP, In *International Conference on Parallel Architectures and Compilation Techniques*, Barcelone, France. Retrieved August 12, 2021, from https://hal.archives-ouvertes.fr/hal-02572435

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009, June). ImageNet: A large-scale hierarchical image database, In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009 IEEE Conference on Computer Vision and Pattern Recognition. https://doi.org/10.1109/CVPR.2009.5206848

Ding, N., & Williams, S. (2019, November). An Instruction Roofline Model for GPUs, In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). https://doi.org/10.1109/PMBS49563.2019.00007

Doll, T., & Schiller, T. (2019, November). *Artificial Intelligence in Land Forces*. Army Concepts and Capabilities Development Centre (ACCDC). https://www.bundeswehr.de/resource/blob/156026/79046a24322feb96b2d8cce168315249/download-positionspapier-englische-version-data.pdf

Dosovitskiy, A., Fischer, P., Ilg, E., Hausser, P., Hazirbas, C., Golkov, V., van der Smagt, P., Cremers, D., & Brox, T. (2015, December). FlowNet: Learning Optical Flow with Convolutional Networks, In *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015 IEEE International Conference on Computer Vision (ICCV), Santiago, IEEE. https://doi.org/10.1109/ICCV.2015.316

Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., & Planas, J. (2011). OmpSs: A PROPOSAL FOR PROGRAMMING HETEROGENEOUS MULTI-CORE ARCHITECTURES. *Parallel Processing Letters*, *21*(02), 173–193. https://doi.org/10.1142/S0129626411000151

Ejjaaouani, K., Aumage, O., Bigot, J., Méhrenberger, M., Murai, H., Nakao, M., & Sato, M. (2020). InKS: A programming model to decouple algorithm from optimization in HPC codes. *The Journal of Supercomputing*, *76*(6), 4666–4681. https://doi.org/10.1007/s11227-019-02950-2

Farnebäck, G. (2003). Two-Frame Motion Estimation Based on Polynomial Expansion (J. Bigun & T. Gustavsson, Eds.). In J. Bigun & T. Gustavsson (Eds.),

Image Analysis, Berlin, Heidelberg, Springer. https://doi.org/10.1007/3-540-45103-X_50

Farshchi, F., Huang, Q., & Yun, H. (2019, February). Integrating NVIDIA Deep Learning Accelerator (NVDLA) with RISC-V SoC on FireSim, In *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. 2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2). https://doi.org/10.1109/EMC249363.2019.00012

Ferrari, V. (2019). Manmachine Teaming: Towards a New Paradigm of Manmachine Collaboration? In *Disruptive Technology and Defence Innovation Ecosystems* (pp. 121–137). John Wiley & Sons, Ltd. https://doi.org/10.1002/9781119644569.ch6
_eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119644569.ch6

Févotte, F., & Lathuilière, B. (2019, February). *Debugging and optimization of HPC programs in mixed precision with the Verrou tool*. Retrieved August 31, 2021, from https://hal.archives-ouvertes.fr/hal-02044101

Flynn, M. J. (1966). Very high-speed computing systems. *Proceedings of the IEEE*, *54*(12), 1901–1909. https://doi.org/10.1109/PROC.1966.5273

Gamal, M., Siam, M., & Abdel-Razek, M. (2018, March 15). *ShuffleSeg: Real-time Semantic Segmentation Network*. Retrieved August 4, 2021, from http://arxiv.org/abs/1803.03816

Geiger, A., Lenz, P., & Urtasun, R. (2012, June). Are we ready for autonomous driving? The KITTI vision benchmark suite, In *2012 IEEE Conference on Computer Vision and Pattern Recognition*. 2012 IEEE Conference on Computer Vision and Pattern Recognition. https://doi.org/10.1109/CVPR.2012.6248074

Ghosh, S., Pal, A., Jaiswal, S., Santosh, K. C., Das, N., & Nasipuri, M. (2019). SegFast-V2: Semantic image segmentation with less parameters in deep learning for autonomous driving. *International Journal of Machine Learning and Cybernetics*, *10*(11), 3145–3154. https://doi.org/10.1007/s13042-019-01005-5

Han, S., Mao, H., & Dally, W. (2016, October 1). Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. ICLR.

Han, S., Pool, J., Tran, J., & Dally, W. J. (2015, October 30). *Learning both Weights and Connections for Efficient Neural Networks*. Retrieved August 3, 2021, from http://arxiv.org/abs/1506.02626

Hennessy, J. L., & Patterson, D. A. (2011, October 7). *Computer Architecture: A Quantitative Approach*. Elsevier.

Horn, B. K. P., & Schunck, B. G. (1981). Determining optical flow. *Artificial Intelligence*, *17*(1), 185–203. https://doi.org/10.1016/0004-3702(81)90024-2

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017, April 16). *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. Retrieved March 9, 2020, from http://arxiv.org/abs/1704.04861

Hui, T.-W., & Loy, C. C. (2020). LiteFlowNet3: Resolving Correspondence Ambiguity for More Accurate Optical Flow Estimation (A. Vedaldi, H. Bischof, T. Brox, & J.-M. Frahm, Eds.). In A. Vedaldi, H. Bischof, T. Brox, & J.-M. Frahm (Eds.), *Computer Vision ECCV 2020*, Cham, Springer International Publishing. https://doi.org/10.1007/978-3-030-58565-5_11

Hui, T.-W., Tang, X., & Change Loy, C. (2018). LiteFlowNet: A Lightweight Convolutional Neural Network for Optical Flow Estimation. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. Retrieved September 4, 2019, from http://openaccess.thecvf.com/content_cvpr_2018/html/Hui_LiteFlowNet_A_Lightweight_CVPR_2018_paper.html

Hui, T.-W., Tang, X., & Loy, C. C. (2020, January 13). *A Lightweight Optical Flow CNN - Revisiting Data Fidelity and Regularization*. Retrieved March 11, 2020, from http://arxiv.org/abs/1903.07414

Hur, J., & Roth, S. (2020). Optical Flow Estimation in the Deep Learning Age. In N. Noceti, A. Sciutti, & F. Rea (Eds.), *Modelling Human Motion: From Human Perception to Robot Design* (pp. 119–140). Cham, Springer International Publishing. https://doi.org/10.1007/978-3-030-46732-6_7

Hwu, W., & Patt, Y. N. (1986). HPSm, a high performance restricted data flow architecture having minimal functionality. *ACM SIGARCH Computer Architecture News*, *14*(2), 297–306. https://doi.org/10.1145/17356.17391

Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., & Keutzer, K. (2016, November 4). *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size*. Retrieved March 9, 2020, from http://arxiv.org/abs/1602.07360

Ilg, E., Mayer, N., Saikia, T., Keuper, M., Dosovitskiy, A., & Brox, T. (2017, July). FlowNet 2.0: Evolution of Optical Flow Estimation with Deep Networks, In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, IEEE. https://doi.org/10.1109/CVPR.2017.179

Keryell, R., Reyes, R., & Howes, L. (2015, May 12). Khronos SYCL for OpenCL: A tutorial, In *Proceedings of the 3rd International Workshop on OpenCL*,

New York, NY, USA, Association for Computing Machinery. https://doi.org/10.1145/2791321.2791345

Kietzmann, T. C., McClure, P., & Kriegeskorte, N. (2019, January 25). *Deep Neural Networks in Computational Neuroscience*. Oxford Research Encyclopedia of Neuroscience. https://doi.org/10.1093/acrefore/9780190264086.013.46

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, *25*, 1097–1105.

Lam, S. K., Pitrou, A., & Seibert, S. (2015, November 15). Numba: A LLVM-based Python JIT compiler, In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, New York, NY, USA, Association for Computing Machinery. https://doi.org/10.1145/2833157.2833162

Le Besnerais, G., & Champagnat, F. (2005). Dense optical flow by iterative local window registration, In *IEEE International Conference on Image Processing 2005*. 2005 International Conference on Image Processing, Genova, Italy, IEEE. https://doi.org/10.1109/ICIP.2005.1529706

Lebrun, M., Colom, M., & Morel, J. M. (2014, October). The noise clinic: A universal blind denoising algorithm, In *2014 IEEE International Conference on Image Processing (ICIP)*. 2014 IEEE International Conference on Image Processing (ICIP). https://doi.org/10.1109/ICIP.2014.7025541

Lecun, Y. (1985). Une procedure d'apprentissage pour reseau a seuil asymmetrique (A learning scheme for asymmetric threshold networks). *Proceedings of Cognitiva 85, Paris, France*, 599–604. Retrieved August 23, 2021, from https://nyuscholars.nyu.edu/en/publications/une-procedure-dapprentissage-pour-reseau-a-seuil-asymmetrique-a-l

Lee, G. G., Chen, Y., Mattavelli, M., & Jang, E. S. (2009). Algorithm/Architecture Co-Exploration of Visual Computing on Emergent Platforms: Overview and Future Prospects. *IEEE Transactions on Circuits and Systems for Video Technology*, *19*(11), 1576–1587. https://doi.org/10.1109/TCSVT.2009.2031376

Leordeanu, M., Zanfir, A., & Sminchisescu, C. (2013). Locally Affine Sparse-to-Dense Matching for Motion and Occlusion Estimation. Proceedings of the IEEE International Conference on Computer Vision. Retrieved July 16, 2021, from https://www.cv-foundation.org/openaccess/content_iccv_2013/html/Leordeanu_Locally_Affine_Sparse-to-Dense_2013_ICCV_paper.html

Lin, D., Talathi, S., & Annapureddy, S. (2016, June 11). Fixed Point Quantization of Deep Convolutional Networks, In *International Conference on Machine*

Learning. International Conference on Machine Learning, PMLR. Retrieved August 4, 2021, from http://proceedings.mlr.press/v48/linb16.html

Liu, C., Chen, L.-C., Schroff, F., Adam, H., Hua, W., Yuille, A. L., & Fei-Fei, L. (2019). Auto-DeepLab: Hierarchical Neural Architecture Search for Semantic Image Segmentation. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. Retrieved August 5, 2021, from https://openaccess.thecvf.com/content_CVPR_2019/html/Liu_Auto-DeepLab_Hierarchical_Neural_Architecture_Search_for_Semantic_Image_Segmentation_CVPR_2019_paper.html

Liu, P., Lyu, M., King, I., & Xu, J. (2019). SelFlow: Self-Supervised Learning of Optical Flow. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. Retrieved August 3, 2021, from https://openaccess.thecvf.com/content_CVPR_2019/html/Liu_SelFlow_Self-Supervised_Learning_of_Optical_Flow_CVPR_2019_paper.html

Lomont, C. (2011). Introduction to intel advanced vector extensions. *Intel white paper*, *23*.

Long, J., Shelhamer, E., & Darrell, T. (2015). Fully Convolutional Networks for Semantic Segmentation. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. Retrieved August 2, 2021, from https://openaccess.thecvf.com/content_cvpr_2015/html/Long_Fully_Convolutional_Networks_2015_CVPR_paper.html

Lucas, B. D., & Kanade, T. (1981). An Iterative Image Registration Technique with an Application to Stereo Vision, In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, San Francisco, CA, USA, Morgan Kaufmann. Vancouver, BC, Canada. Retrieved September 4, 2019, from http://dl.acm.org/citation.cfm?id=1623264.1623280

Malcolm, J., Yalamanchili, P., McClanahan, C., Venugopalakrishnan, V., Patel, K., & Melonakos, J. (2012, May 4). ArrayFire: A GPU acceleration platform, In *Modeling and Simulation for Defense Systems and Applications VII*. Modeling and Simulation for Defense Systems and Applications VII, International Society for Optics and Photonics. https://doi.org/10.1117/12.921122

Maleki, S., Gao, Y., Garzarłn, M. J., Wong, T., & Padua, D. A. (2011, October). An Evaluation of Vectorizing Compilers, In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 2011 International Conference on Parallel Architectures and Compilation Techniques. https://doi.org/10.1109/PACT.2011.68

Martelli, M., Gac, N., Mérigot, A., & Enderli, C. (2019). 3D Tomography Back-Projection Parallelization on Intel FPGAs Using OpenCL. *Journal of Signal Processing Systems*, *91*(7), 731–743. https://doi.org/10.1007/s11265-018-1403-6

Mayer, N. (2021, August 12). *Caffe for FlowNet2*. Retrieved August 19, 2021, from https://github.com/lmb-freiburg/flownet2

Membarth, R., Reiche, O., Hannig, F., Teich, J., Körner, M., & Eckert, W. (2016). HIPAcc: A Domain-Specific Language and Compiler for Image Processing. *IEEE Transactions on Parallel and Distributed Systems*, *27*(1), 210–224. https://doi.org/10.1109/TPDS.2015.2394802

Menze, M., & Geiger, A. (2015, June). Object scene flow for autonomous vehicles, In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). https://doi.org/10.1109/CVPR.2015.7298925

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., & Wu, H. (2018, February 15). *Mixed Precision Training*. Retrieved August 4, 2021, from http://arxiv.org/abs/1710.03740

Moloney, D., Barry, B., Richmond, R., Connor, F., Brick, C., & Donohoe, D. (2014, August). Myriad 2: Eye of the computational vision storm, In *2014 IEEE Hot Chips 26 Symposium (HCS)*. 2014 IEEE Hot Chips 26 Symposium (HCS). https://doi.org/10.1109/HOTCHIPS.2014.7478823

Monchiero, M., Palermo, G., Silvano, C., & Villa, O. (2006, July). Exploration of Distributed Shared Memory Architectures for NoC-based Multiprocessors, In *Modeling and Simulation 2006 International Conference on Embedded Computer Systems: Architectures*. Modeling and Simulation 2006 International Conference on Embedded Computer Systems: Architectures. https://doi.org/10.1109/ICSAMOS.2006.300821

Moore, G. E. (1965). Cramming more components onto integrated circuits, *38*(8), 4.

Naumov, M., Arsaev, M., Castonguay, P., Cohen, J., Demouth, J., Eaton, J., Layton, S., Markovskiy, N., Reguly, I., Sakharnykh, N., Sellappan, V., & Strzodka, R. (2015). AmgX: A Library for GPU Accelerated Algebraic Multigrid and Preconditioned Iterative Methods. *SIAM Journal on Scientific Computing*, *37*(5), S602–S626. https://doi.org/10.1137/140980260

Nussbaumer, H. J. (1981). The Fast Fourier Transform. In H. J. Nussbaumer (Ed.), *Fast Fourier Transform and Convolution Algorithms* (pp. 80–111). Berlin, Heidelberg, Springer. https://doi.org/10.1007/978-3-662-00551-4_4

NVIDIA. (2017). *V100 GPU Architecture: The worlds most advanced datacenter GPU*. Tech. Rep., NVIDIA. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

nvidia. (2021). NVIDIA TensorRT Documentation. Nvidia. Retrieved August 26, 2021, from https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html

Oden, L. (2020, March). Lessons learned from comparing C-CUDA and Python-Numba for GPU-Computing, In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). https://doi.org/10.1109/PDP50117.2020.00041

Odena, A., Dumoulin, V., & Olah, C. (2016). Deconvolution and checkerboard artifacts. *Distill*. https://doi.org/10.23915/distill.00003

OMahony, N., Campbell, S., Carvalho, A., Harapanahalli, S., Hernandez, G. V., Krpalkova, L., Riordan, D., & Walsh, J. (2020). Deep Learning vs. Traditional Computer Vision (K. Arai & S. Kapoor, Eds.). In K. Arai & S. Kapoor (Eds.), *Advances in Computer Vision*, Cham, Springer International Publishing. https://doi.org/10.1007/978-3-030-17795-9_10

ONNX. (2021, August 19). *Use ONNX*. Retrieved August 19, 2021, from https://github.com/onnx/onnx

Parly, F. (2019, September). *Artificial Intelligence in Support of Defence*. Ministère de la Défense. Retrieved August 23, 2021, from https://webcache.googleusercontent.com/search?q=cache:j_SdUrGw-eIJ:https://www.defense.gouv.fr/content/download/573877/9834690/Strat%25C3%25A9gie%2520de%2520l%2527IA-UK_9%25201%25202020.pdf+&cd=1&hl=en&ct=clnk&gl=fr&lr=lang_en%7Clang_fr&client=ubuntu

Pelcat, M., Desnos, K., Heulot, J., Guy, C., Nezan, J.-F., & Aridhi, S. (2014, September). Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming, In *2014 6th European Embedded Design in Education and Research Conference (EDERC)*. 2014 6th European Embedded Design in Education and Research Conference (EDERC), Milano, Italy, IEEE. https://doi.org/10.1109/EDERC.2014.6924354

Petreto, A., Hennequin, A., Koehler, T., Romera, T., Fargeix, Y., Gaillard, B., Bouyer, M., Meunier, Q. L., & Lacassagne, L. (2018, October). Energy and Execution Time Comparison of Optical Flow Algorithms on SIMD and GPU Architectures, In *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. 2018 Conference on Design and Architectures for Signal and Image Processing (DASIP). https://doi.org/10.1109/DASIP.2018.8597004

Pinard, C. (2021, August 24). *FlowNetPytorch*. Retrieved August 24, 2021, from https://github.com/ClementPinard/FlowNetPytorch

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., & Amarasinghe, S. (2013). Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, *48*(6), 519–530. https://doi.org/10.1145/2499370.2462176

Ranjan, A., & Black, M. J. (2017). Optical Flow Estimation Using a Spatial Pyramid Network. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. Retrieved August 3, 2021, from https://openaccess.thecvf.com/content_cvpr_2017/html/Ranjan_Optical_Flow_Estimation_CVPR_2017_paper.html

Reddy, V. G. (2008). Neon technology introduction. *ARM Corporation*, *4*(1).

Reuther, A., Michaleas, P., Jones, M., Gadepally, V., Samsi, S., & Kepner, J. (2019, August 29). *Survey and Benchmarking of Machine Learning Accelerators*. Retrieved September 9, 2019, from http://arxiv.org/abs/1908.11348

Ruhnau, P., Kohlberger, T., Schnörr, C., & Nobach, H. (2005). Variational optical flow estimation for particle image velocimetry. *Experiments in Fluids*, *38*(1), 21–32. https://doi.org/10.1007/s00348-004-0880-5

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. https://doi.org/10.1038/323533a0
Bandiera_abtest: a Cg_type: Nature Research Journals Primary_atype: Research

Rupp, K. (2015). *40 Years of Microprocessor Trend Data | Karl Rupp*. Retrieved July 26, 2021, from https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

Saad, Y. (2003). *Iterative methods for sparse linear systems*. SIAM.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L.-C. (2018). MobileNetV2: Inverted Residuals and Linear Bottlenecks. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. Retrieved August 3, 2021, from https://openaccess.thecvf.com/content_cvpr_2018/html/Sandler_MobileNetV2_Inverted_Residuals_CVPR_2018_paper.html

Seznec, A., Felix, S., Krishnan, V., & Sazeides, Y. (2002, May). Design tradeoffs for the alpha EV8 conditional branch predictor, In *Proceedings 29th Annual International Symposium on Computer Architecture*. Proceedings 29th Annual International Symposium on Computer Architecture. https://doi.org/10.1109/ISCA.2002.1003587

Seznec, M. (2021). *Flowpy: Tools for working with optical flow* (Version 0.6.0). Retrieved August 13, 2021, from https://gitlab-research.centralesupelec.fr/2018seznecm/flowpy

Seznec, M., Gac, N., Ferrari, A., & Orieux, F. (2018, October). A Study on Convolution using Half-Precision Floating-Point Numbers on GPU for Radio Astronomy Deconvolution, In *2018 IEEE International Workshop on Signal Processing Systems (SiPS)*. 2018 IEEE International Workshop on Signal

Processing Systems (SiPS), Cape Town, IEEE. https://doi.org/10.1109/SiPS.2018.8598342

Seznec, M., Gac, N., Orieux, F., & Naik, A. S. (2020a, May). A new convolutions algorithm to leverage tensor cores. Retrieved August 13, 2021, from https://hal.archives-ouvertes.fr/hal-02605077
Published: GPU Technology Conference (GTC)

Seznec, M., Gac, N., Orieux, F., & Naik, A. S. (2020b, October). An Efficiency-Driven Approach For Real-Time Optical Flow Processing On Parallel Hardware, In *2020 IEEE International Conference on Image Processing (ICIP)*. 2020 IEEE International Conference on Image Processing (ICIP). https://doi.org/10.1109/ICIP40778.2020.9191164

Sharon, E., Brandt, A., & Basri, R. (2000, June). Fast multiscale image segmentation, In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2000 (Cat. No.PR00662)*. Proceedings IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2000 (Cat. No.PR00662). https://doi.org/10.1109/CVPR.2000.855801

Smeulders, A. W. M., Chu, D. M., Cucchiara, R., Calderara, S., Dehghan, A., & Shah, M. (2014). Visual Tracking: An Experimental Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *36*(7), 1442–1468. https://doi.org/10.1109/TPAMI.2013.230

Sun, D. (2021, August 26). *NVlabs/PWC-Net*. Retrieved August 26, 2021, from https://github.com/NVlabs/PWC-Net

Sun, D., Yang, X., Liu, M.-Y., & Kautz, J. (2018a, September 14). *Models Matter, So Does Training: An Empirical Study of CNNs for Optical Flow Estimation*. Retrieved May 28, 2020, from http://arxiv.org/abs/1809.05571

Sun, D., Yang, X., Liu, M.-Y., & Kautz, J. (2018b, June). PWC-Net: CNNs for Optical Flow Using Pyramid, Warping, and Cost Volume, In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, IEEE. https://doi.org/10.1109/CVPR.2018.00931

Sutter, H. et al. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs journal*, *30*(3), 202–210.

Vahdat, A., Mallya, A., Liu, M.-Y., & Kautz, J. (2020). UNAS: Differentiable Architecture Search Meets Reinforcement Learning. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. Retrieved August 5, 2021, from https://openaccess.thecvf.com/content_CVPR_2020/html/Vahdat_UNAS_Differentiable_Architecture_Search_Meets_Reinforcement_Learning_CVPR_2020_paper.html

Vanholder, H. (2016). Efficient inference with tensorrt. ed.

Venkata Bhargava Narendra, V., Rangababu, P., & Balabantaray, B. K. (2021). Low-Power U-Net for Semantic Image Segmentation (E. S. Gopi, Ed.). In E. S. Gopi (Ed.), *Machine Learning, Deep Learning and Computational Intelligence for Wireless Communication*, Singapore, Springer. https://doi.org/10.1007/978-981-16-0289-4_35

Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q., & Wang, Y. (2014). Intel Math Kernel Library. In E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, & Y. Wang (Eds.), *High-Performance Computing on the Intelő Xeon Phi: How to Fully Exploit MIC Architectures* (pp. 167–188). Cham, Springer International Publishing. https://doi.org/10.1007/978-3-319-06486-4_7

Waterman, A., Lee, Y., Patterson, D. A., & Asanovic, K. (2011). The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, *116*.

Wei, X., Yu, C. H., Zhang, P., Chen, Y., Wang, Y., Hu, H., Liang, Y., & Cong, J. (2017, June). Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs, In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC). https://doi.org/10.1145/3061639.3062207

Wienke, S., Springer, P., Terboven, C., & an Mey, D. (2012). OpenACC First Experiences with Real-World Applications (C. Kaklamanis, T. Papatheodorou, & P. G. Spirakis, Eds.). In C. Kaklamanis, T. Papatheodorou, & P. G. Spirakis (Eds.), *Euro-Par 2012 Parallel Processing*, Berlin, Heidelberg, Springer. https://doi.org/10.1007/978-3-642-32820-6_85

Wulf, W. A., & McKee, S. A. (1995). Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, *23*(1), 20–24.

Yan, X., Jiang, W., Shi, Y., & Zhuo, C. (2020). MS-NAS: Multi-scale Neural Architecture Search for Medical Image Segmentation (A. L. Martel, P. Abolmaesumi, D. Stoyanov, D. Mateus, M. A. Zuluaga, S. K. Zhou, D. Racoceanu, & L. Joskowicz, Eds.). In A. L. Martel, P. Abolmaesumi, D. Stoyanov, D. Mateus, M. A. Zuluaga, S. K. Zhou, D. Racoceanu, & L. Joskowicz (Eds.), *Medical Image Computing and Computer Assisted Intervention MICCAI 2020*, Cham, Springer International Publishing. https://doi.org/10.1007/978-3-030-59710-8_38

Yang, C., Kurth, T., & Williams, S. (2020). Hierarchical Roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system. *Concurrency and Computation: Practice and Experience*, *32*(20), e5547. https://doi.org/10.1002/cpe.5547 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5547

Yao, Z., Cao, S., Xiao, W., Zhang, C., & Nie, L. (2019). Balanced Sparsity for Efficient DNN Inference on GPU. *Proceedings of the AAAI Conference on Artificial Intelligence*, *33*(01), 5676–5683. https://doi.org/10.1609/aaai.v33i01.33015676

Yu, C., Wang, J., Peng, C., Gao, C., Yu, G., & Sang, N. (2018). BiSeNet: Bilateral Segmentation Network for Real-time Semantic Segmentation. Proceedings of the European Conference on Computer Vision (ECCV). Retrieved August 5, 2021, from https://openaccess.thecvf.com/content_ECCV_2018/html/Changqian_Yu_BiSeNet_Bilateral_Segmentation_ECCV_2018_paper.html

Zach, C., Pock, T., & Bischof, H. (2007). A Duality Based Approach for Realtime TV-L 1 Optical Flow. In F. A. Hamprecht, C. Schnörr, & B. Jähne (Eds.), *Pattern Recognition* (pp. 214–223). Berlin, Heidelberg, Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-74936-3_22

Zhang, X., Zhou, X., Lin, M., & Sun, J. (2018). ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. Retrieved August 3, 2021, from https://openaccess.thecvf.com/content_cvpr_2018/html/Zhang_ShuffleNet_An_Extremely_CVPR_2018_paper.html

Zhao, S., Sheng, Y., Dong, Y., Chang, E. I.-C., & Xu, Y. (2020, April 8). *MaskFlownet: Asymmetric Feature Matching with Learnable Occlusion Mask*. Retrieved May 19, 2020, from http://arxiv.org/abs/2003.10955

Zhou, G., Zhou, J., & Lin, H. (2018, November). Research on NVIDIA Deep Learning Accelerator, In *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*. 2018 12th IEEE International Conference on Anti-Counterfeiting, Security, and Identification (ASID). https://doi.org/10.1109/ICASID.2018.8693202

Zoph, B., Vasudevan, V., Shlens, J., & Le, Q. V. (2018). Learning Transferable Architectures for Scalable Image Recognition. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. Retrieved August 5, 2021, from https://openaccess.thecvf.com/content_cvpr_2018/html/Zoph_Learning_Transferable_Architectures_CVPR_2018_paper.html

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : De l'algorithme à l'implémentation, flot d'optimisations pour le calcul haute performance sur GPU embarqués

Mots clés : adéquation algorithme architecture, implémentation et optimisation, GPU, traitement d'images, systèmes embarqués, réseaux de neurones convolutifs

Résumé : Les algorithmes de traitement numérique actuels nécessitent une puissance de calcul accrue pour obtenir des résultats plus précis et traiter des données plus volumineuses. Dans le même temps, les architectures matérielles se spécialisent, avec des accélérateurs très efficaces pour des tâches spécifiques. Dans ce contexte, le chemin du déploiement de l'algorithme à l'implémentation est de plus en plus complexe. Il est donc crucial de déterminer comment les algorithmes peuvent être modifiés pour tirer parti des capacités du matériel. Dans notre étude, nous nous sommes intéressé aux unités graphiques (GPU), un type de processeur massivement parallèle. Notre travail a consisté à l'adaption entre l'algorithme et le matériel d'exécution. À l'échelle d'un opérateur mathématique, nous avons modifié un algorithme de convolution d'images pour utiliser les tensor cores et montré qu'on peut en doubler les performances pour de grands noyaux de convolution. Au niveau méthode, nous avons évalué des solveurs linéaires pour l'estimation de flux optique afin de trouver le plus adéquat sur GPU. Grâce à ce choix et après de nouvelles optimisations spécifiques, la méthode est deux fois plus rapide que l'implémentation initiale, fonctionnant à 60 images par seconde sur plateforme embarquée (30W). Enfin, nous avons également montré l'intérêt, dans le cadre des réseaux de neurones profonds, de cette méthode de conception d'algorithmes adaptée au matériel. Avec pour exemple l'hybridation entre un réseau conçu pour le flux optique avec une autre architecture préentrainée et conçue pour être efficace sur des cibles à faible puissance de calcul.

Title: From the algorithm to the targets, optimization flow for high performance computing on embedded GPUs

Keywords: Hardware-Aware Algorithm design, Implementation and Optimization, GPU, Image Processing, Embedded Systems, Convolutional Neural Networks

Abstract: Current digital processing algorithms require more computing power to achieve more accurate results and process larger data. In the meantime, hardware architectures are becoming more specialized, with highly efficient accelerators designed for specific tasks. In this context, the path of deployment from the algorithm to the implementation becomes increasingly complex. It is, therefore, crucial to determine how algorithms can be modified to take advantage of new hardware capabilities. Our study focused on graphics processing units (GPUs), a massively parallel processor. Our algorithmic work was done in the context of radio-astronomy or optical flow estimation and consisted of finding the best adaptation of the software to the hardware. At the level of a mathematical operator, we modified the traditional image convolution algorithm to use the matrix units and showed that its performance doubles for large convolution kernels. At a broader method level, we evaluated linear solvers for the combined local-global optical flow to find the most suitable one on GPU. With additional optimizations, such as iteration fusion or memory buffer re-utilization, the method is twice as fast as the initial implementation, running at 60 frames per second on an embedded platform (30 W). Finally, we also pointed out the interest of this hardware-aware algorithm design method in the context of deep neural networks. For that, we showed the hybridization of a convolutional neural network for optical flow estimation with a pre-trained image classification network, MobileNet, that was initially designed for efficient image classification on low-power platforms.

Maison du doctorat de l'Université Paris-Saclay
2e étage aile ouest, École normale supérieure Paris-Saclay
4 avenue des Sciences,
91190 Gif sur Yvette, France