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

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

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

Nicolas Gac, Mickael Seznec, François Orieux, Alvin Sashala Naik

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. Hopefully, this example should provide feedback on the issues that arise when trying to port a method to a parallel platform and serve 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 [8], unmanned flight [15], or autonomous driving [6], 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-

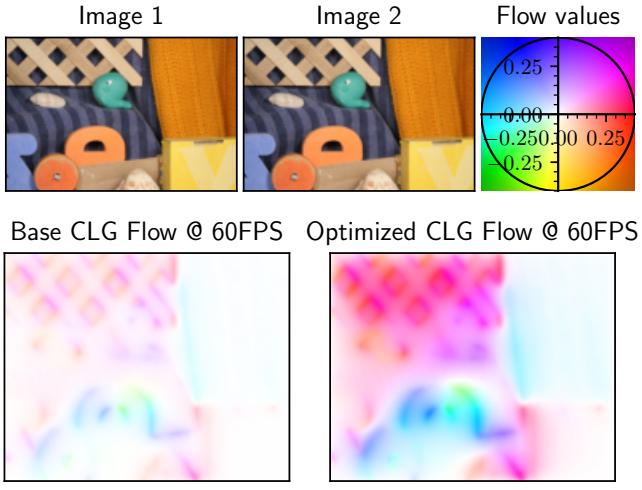

Fig. 1: For the same framerate on Jetson Xavier, our GPU-optimized multi-scale CLG Optical Flow converges further than the initial implementation.

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 [12] and numerous refinements have been developed since [9,3]. For our analysis, we consider the CLG (Combined Local-Global) method [4], because it serves in an industrial application we use.

Our analysis then serves two goals. First, finding the impact of the solver choice and the values of hyper-parameters on the speed 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). Doing optical flow processing on this kind of lightweight device is crucial for power constrained systems, like drones [15]. The other goal is finding efficient optimization procedures for the CLG 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 [20] 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 algorithm implementation 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 [12] and Lucas & Kanade [14]. From there, many refinements have been incorporated on top of these frameworks. Review papers [2,22] explore comprehensively the different strategies used for computing optical flow.

In this article, we focus on a differential method, a family 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 [12], Farnebäck *et al.* replace the linear interpolation with a quadratic one for better accuracy [9]. Brox *et al.* add a gradient conservation term [3] while Zach *et al.* use a L1-norm penalization instead of a quadratic one [26] 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 [14]. This unifying model is less sensitive to noise, as the local information is averaged over multiple pixels. We consider this algorithm fixed in our work, so our results match the original implementation's accuracy. A detailed comparison with other related methods is made in [22].

Efficient implementation has always been key to an attractive optical flow method. For CLG, a CPU implementation has been described in [13] and Moussu [16] 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 [23]. Filipovic *et al.* propose a source-to-source compiler to perform fusion at the compilation stage [10]. 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 directly implementing the operators defined by those matrices. An implemenation on massively parallel processors of this solver is studied in [18] and also analyse the trade-off between memory locality and overhead computa-

tions. In [17], 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 [24]. It analyses a program's run based on its arithmetic intensity and the performance limits of the hardware in terms of memory and compute throughput. It is a general and powerful tool to find bottlenecks in an application that has already been used for GPUs [7].

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 grayscale, 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 [12], Horn & Schunck set D and R to

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

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

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

$$\|\bar{f}(x+u, y+v, t+1) - \bar{f}(x, y, t)\|^2 \tag{4}$$

$$\approx \|\bar{f}(x, y, t) + \nabla\bar{f}(x, y, t)^T \boldsymbol{w} - \bar{f}(x, y, t)\|^2 \tag{5}$$

$$= \|\nabla\bar{f}(x, y, t)^T \boldsymbol{w}\|^2 \tag{6}$$

$$= \boldsymbol{w}^T \nabla\bar{f}(x, y, t) \nabla\bar{f}(x, y, t)^T \boldsymbol{w} \tag{7}$$

$$= \boldsymbol{w}^T J_0 \boldsymbol{w}. \tag{8}$$

With this definition, the data-fitting term only incorporates pixel-wise intensity conservation. In [4], Bruhn *et al.* leverage the energy penalization found in [14] to average the intensity conservation over the pixel's neighborhood.

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

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

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]

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

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 $A\boldsymbol{x} = \boldsymbol{b}$ where

$$A = \begin{bmatrix}\operatorname{diag}\bar{\bar{j}}_{\rho,0,0} & \operatorname{diag}\bar{\bar{j}}_{\rho,0,1} \\ \operatorname{diag}\bar{\bar{j}}_{\rho,1,0} & \operatorname{diag}\bar{\bar{j}}_{\rho,1,1}\end{bmatrix} - \alpha \begin{bmatrix} L & 0 \\ 0 & L\end{bmatrix} \quad (17)$$

with $L = D_x^T D_x + D_y^T D_y$, $\boldsymbol{b}^T = -[\bar{\bar{j}}_{\rho,0,2}, \bar{\bar{j}}_{\rho,1,2}]^T$, and $\boldsymbol{x} = \bar{\bar{\boldsymbol{w}}}^*$.

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 [12] and remain widely used to solve these linear systems [13]. Second, Krylov methods are often used for numerical simulations and benefit from a well-supplied scientific corpus [19].

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 $B\boldsymbol{x} = \boldsymbol{b} - C\boldsymbol{x}$. Assuming B is invertible, an iterative scheme is constructed

$$\boldsymbol{x}^{k+1} = B^{-1}(\boldsymbol{b} - C\boldsymbol{x}^k) \quad (18)$$

$$\boldsymbol{x}^{k+1} = (I - B^{-1}A)\boldsymbol{x}^k + B^{-1}\boldsymbol{b}. \quad (19)$$

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

$$B_{\mathrm{J}}^{(\mathrm{diags})} := \begin{bmatrix}\operatorname{diag}\bar{\bar{j}}_{\rho,0,0} - \alpha D_L & \operatorname{diag}\bar{\bar{j}}_{\rho,0,1} \\ \operatorname{diag}\bar{\bar{j}}_{\rho,1,0} & \operatorname{diag}\bar{\bar{j}}_{\rho,1,1} - \alpha D_L\end{bmatrix} \quad (20)$$

$$B_{\mathrm{GS}}^{(\mathrm{diags})} := \begin{bmatrix}\operatorname{diag}\bar{\bar{j}}_{\rho,0,0} - \alpha L_L & \operatorname{diag}\bar{\bar{j}}_{\rho,0,1} \\ \operatorname{diag}\bar{\bar{j}}_{\rho,1,0} & \operatorname{diag}\bar{\bar{j}}_{\rho,1,1} - \alpha L_L\end{bmatrix}. \quad (21)$$

This construction of B matrices is called the pointwise-coupled method in [13], 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.

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 $\|\boldsymbol{e}^k\|_2 = \|\boldsymbol{b} - A\boldsymbol{x}^k\|_2$ 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 $-\log(\rho_{\mathrm{SR}})$ for easier comparison between solvers. Note

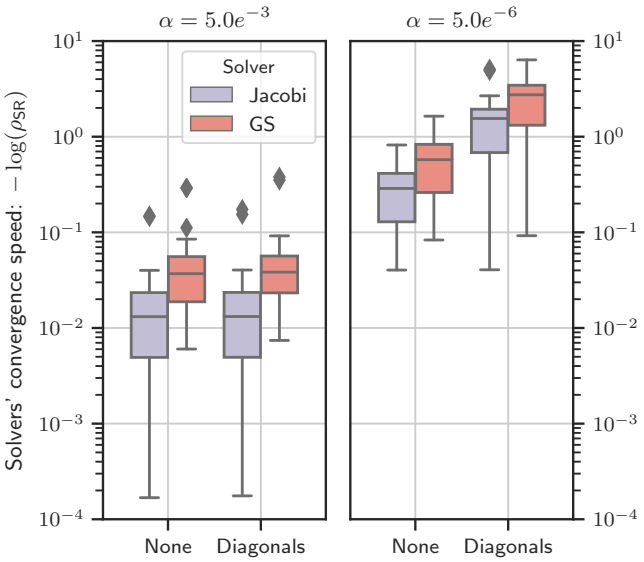

Fig. 2: Comparison of the theoretical convergence speed of the Jacobi and Gauss-Seidel (GS) methods. Standard solvers are refered by *None* and precontioned, or pointwise-coupled, variations with *Diagonals*.

that finding ρ_{SR} is a computationally heavy task, so this numerical analysis is done on cropped images from the Middlebury dataset [2].

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

$$\mathcal{K}_n(A, \boldsymbol{b}) = \text{span}\{\boldsymbol{b}, A\boldsymbol{b}, A^2\boldsymbol{b}, \dots, A^{n-1}\boldsymbol{b}\},\ n \in \mathbb{N}^*. \quad (22)$$

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.

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 [21].

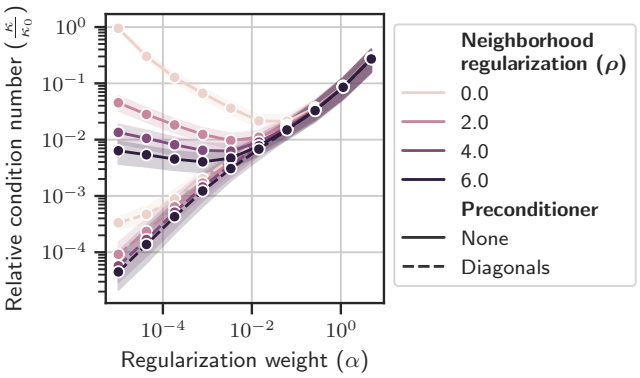

Fig. 3: Normalised condition number of the problem versus parameters' value.

Sometimes, the system can be enhanced by the use of a preconditioner M. With M being an invertible matrix, the linear system to solve becomes

$$M^{-1}A\boldsymbol{x} = M^{-1}\boldsymbol{b}. \quad (23)$$

Solving the system eq. (23) is equivalent to solve $A\boldsymbol{x} = \boldsymbol{b}$ 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

$$M = \begin{bmatrix} \text{diag}\,\bar{\bar{j}}_{\rho,0,0} - \alpha D_L & \text{diag}\,\bar{\bar{j}}_{\rho,0,1} \\ \text{diag}\,\bar{\bar{j}}_{\rho,1,0} & \text{diag}\,\bar{\bar{j}}_{\rho,1,1} - \alpha D_L \end{bmatrix}. \quad (24)$$

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. (24), 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. The data is averaged over 30 images from several databases [2,5,11].

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 [19].

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 ($\sim 10^{-5}$ 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.

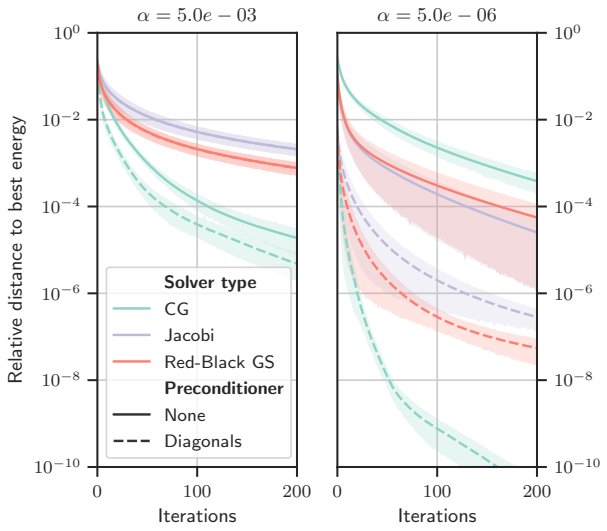

Fig. 4: Convergence vs. iterations with $\rho = 2.5$. On the lest $\alpha = 5e^{-3}$, on the right $\alpha = 5e^{-6}$

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. 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 [20]. This phenomenon also happened with $\alpha = 5.0e^{-3}$, after a

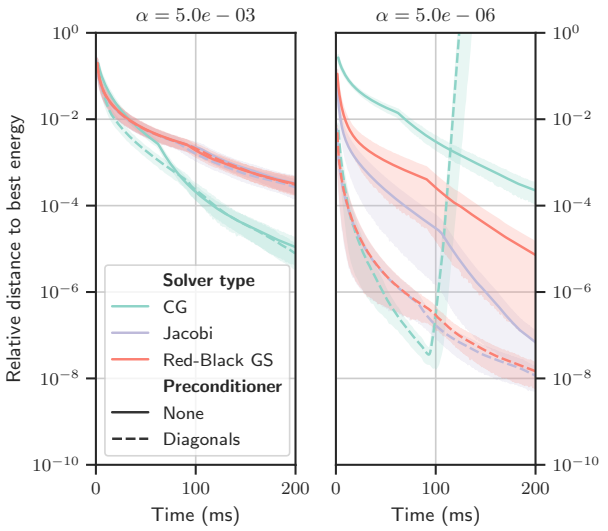

Fig. 5: Convergence vs. time on a Jetson AGX Xavier. Parameters are the same as on fig. 4.

vast number of iterations, though. We attribute this divergence to the sensitivity of the Conjugate-Gradient method to roundoff errors [25].

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 [13]. 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.

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.

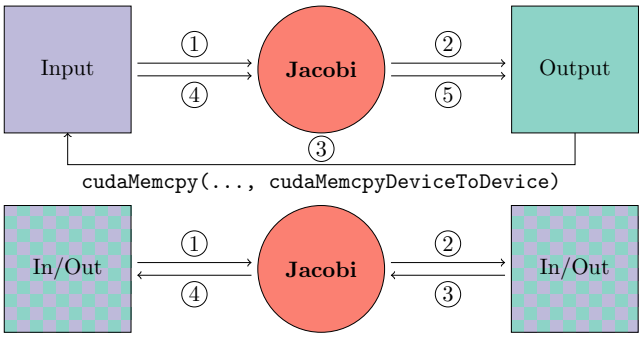

Fig. 6: Two iterations of Jacobi without (top) or with (bottom) buffer reuse. Top version reads from (1), writes to (2), copies output to input (3) and does a new iteration (4) and (5). Bottom version avoids copying by using buffers as both input or output.

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 standard techniques known in the literature, but their application for optical flow is original, as well as their analysis in this context. We present them 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

$$\text{AI} = \frac{\text{FLOPs}}{\text{Bytes loaded}}. \tag{25}$$

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 [24].

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 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,

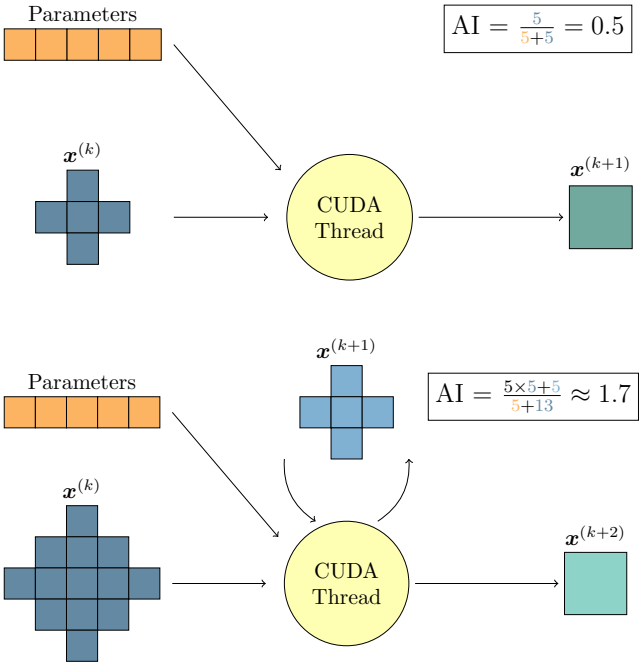

Fig. 7: Top: an iteration of Jacobi for a single output. Bottom: fusion of two Jacobi iterations. Arithmetic intensity is given for reference only, assuming one operation per $\boldsymbol{x}^{(k)}$.

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 fused iteration

$$\text{AI}(j) = \frac{\alpha \sum_{i=1}^{j} (32 - 2i)^2}{\beta(32 \times 32)} \tag{26}$$

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. (26) increases with j and then seems to benefit the implementation, it is important to understand that the total FLOPs required

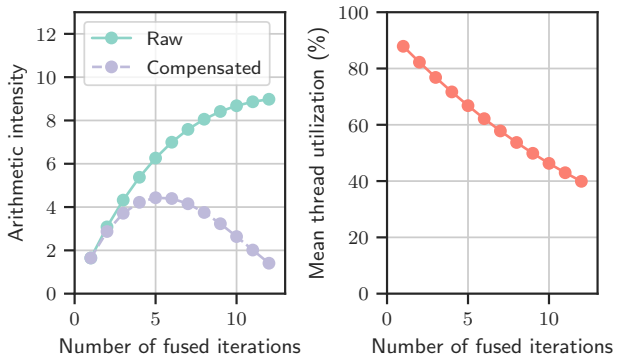

Fig. 8: Arithmetic intensity and mean thread utilization w.r.t. the number of fused iterations.

by the algorithm are not constant with j. A single non-fused Jacobi needs

$$W_{\text{no fusion}} = \alpha HW. \tag{27}$$

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

$$W_{\text{j-fusion}} = \lceil \frac{H}{32-2j} \rceil \lceil \frac{W}{32-2j} \rceil \alpha \sum_{i=1}^{j} (32-2i)^2. \tag{28}$$

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

$$\frac{W_{\text{j-fusion}}}{jW_{\text{no fusion}}} \approx \frac{1}{j(32-2j)^2} \sum_{i=1}^{j} (32-2i)^2 \tag{29}$$

The left-hand side graph of fig. 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. (26). The dashed curve is arithmetic intensity divided by the *compute* overhead, as expressed in eq. (29).

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.

The right-hand side of fig. 8 shows the percentage of active threads during the entire fused iteration. With each supplementary fused iteration, the footprint of computable coefficients shrinks, so fewer threads are operating.

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.

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

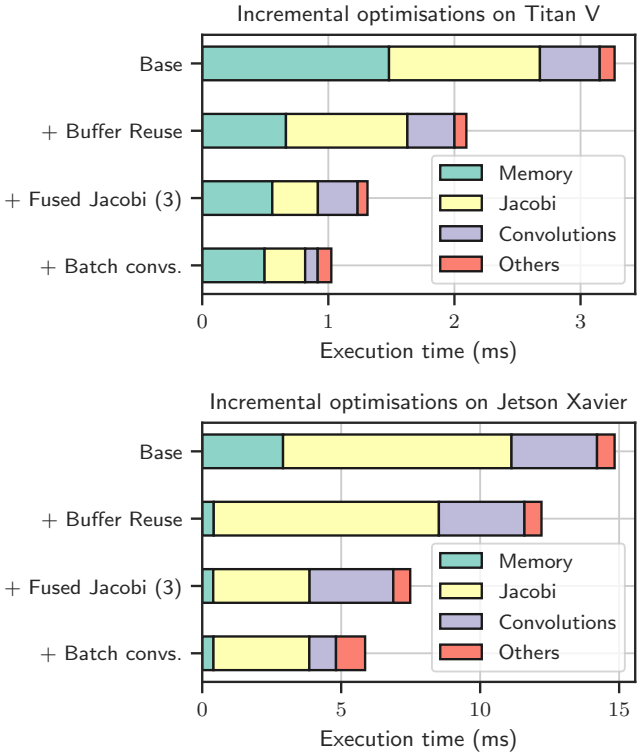

Fig. 9: Effects of cumulative optimizations on Titan V (top) and Jetson Xavier (bottom).

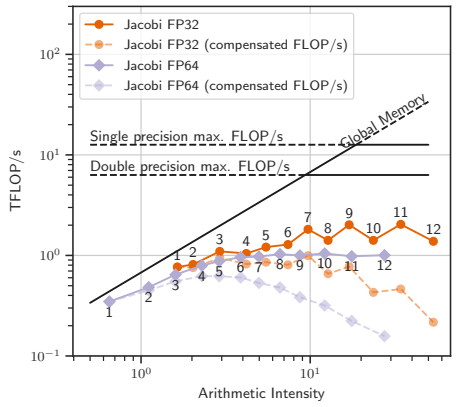

Fig. 10: A roofline model analysis of Jacobi iteration fusion on Titan V.

(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

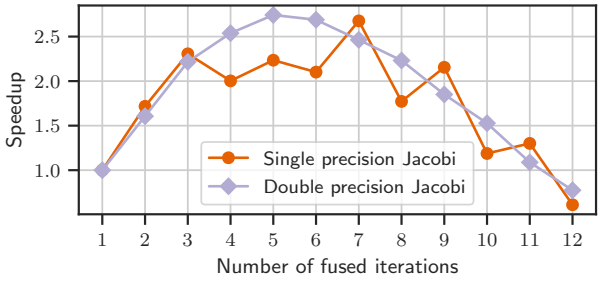

Fig. 11: Speedup vs. number of fused iterations on Titan V.

to compute to benefit from maximal hardware performance.

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, as modeled on fig. 8. 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. (29). 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 achieved *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.

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.

		Ours (Jetson Xavier)		
	Moussu [16]	CPU	GPU (baseline)	GPU (optimized)
FPS	4.2*	0.4	4.5	**15.0**

Table 2: Throughput of 3000 Jacobi iterations. Same conditions as [16]. *scaled results.

To show the benefits of these optimizations, we compare our results with another Jacobi GPU implementation in table 2. This reference [16] was measured on a Geforce 9400 GT, released in 2008. We scaled that result to account for the 14.3× larger memory bandwidth in the Jetson Xavier. We chose to multiply according to this metric as memory requests are the bottleneck of non-optimized Jacobi implementations (see fig. 10). Our original GPU implementation is then on-par with results extrapolated from Moussu. The optimized version, however, benefits from a large speed-up (3.5× vs. [16]) thanks to the buffer reuse and iteration fusion strategies. For reference, we included a CPU version in the comparison. This implementation leverages OpenMP parallelization but under-performs compared to GPU versions.

4.3 Multi-level analysis

As detailed in [4], multi-level processing aims at finding optical flows at different problem scales. The technique helps in finding large displacements and iterates quickly on higher levels due to the reduced problem size. Consequently, we measure the actual performance of our GPU implementation on different image sizes. Those results should guide decisions back at the algorithm level to set the number of iterations per level that fits a given time frame.

Table 3 presents results for the Jacobi solver on various image sizes. Below 320 × 256, there is not enough processing to saturate the GPU, so the time does not vary vastly between different sizes. However, with larger dimensions we see that the processing time grows linearly with the number of pixels in the image. We can now use this knowledge to choose the number of iterations per scale of the problem.

Size	20 × 16	40 × 32	80 × 64	160 × 128
Time (ms)	6.34	6.39	10.4	13.1
Size	320 × 256	640 × 512	1280 × 1024	2560 × 2048
Time (ms)	26.3	83.6	341	1,360

Table 3: Time to perform 1000 Jacobi iterations on Jetson Xavier vs. image size.

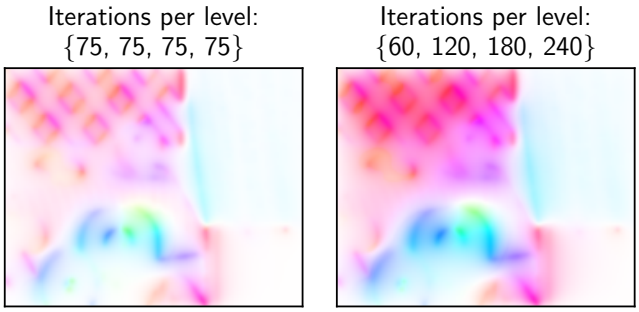

Fig. 12: Results on 640 × 512 images at 60 FPS. Doing more iterations at higher levels converges faster.

On fig. 12, the optical flow for two choices of parameters is displayed. The left-hand flow was obtained by doing 75 iterations on each scale, from the 640 × 512 level, to the upper ones: 320×256, 160×128, and 80×64. The right-hand flow sets 60 iterations at the highest-resolution level and 120, 180, and 240 iterations at the lower 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 yields a smoother flow. 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 convo-

lution. 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. Tuning the number of iterations per level set allowed us to produce a smoother flow in the same time frame. Overall, this implementation opens up the use of CLG optical flow for embedded applications like drones or robotics.

Further work could analyze the performance of multigrid solvers and their optimal configuration on GPUs or apply this optimization method to other optical flow methods.

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. Springer. 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 **92**(1), 1–31. 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. Pajdla, J. Matas (eds.) Computer Vision - ECCV 2004. Springer. 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 **61**(3), 1–21. DOI: 10.1023/B:VISI.0000045324.43199.43
5. Butler, D.J., Wulff, J., Stanley, G.B., Black, M.J.: A naturalistic open source movie for optical flow evaluation. In: A. Fitzgibbon et al. (Eds.) (ed.) European Conf. on Computer Vision (ECCV), Part IV, LNCS 7577, pp. 611–625. Springer-Verlag
6. 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. DOI: 10.1109/IV47402.2020.9304777
7. 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. DOI: 10.1109/PMBS49563.2019.00007
8. Dougherty, L., Asmuth, J.C., Gefter, W.B.: Alignment of CT Lung Volumes with an Optical Flow Method **10**(3), 249–254. DOI: 10.1016/S1076-6332(03)80098-3
9. 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. DOI: 10.1007/3-540-45103-X_50
10. Filipovič, J., Madzin, M., Fousek, J., Matyska, L.: Optimizing CUDA Code By Kernel Fusion—Application on BLAS **71**(10), 3934–3957. DOI: 10.1007/s11227-015-1483-z
11. Geiger, A., Lenz, P., Stiller, C., Urtasun, R.: Vision meets robotics: The KITTI dataset **32**(11), 1231–1237. DOI: 10.1177/0278364913491297
12. Horn, B.K.P., Schunck, B.G.: Determining optical flow **17**(1), 185–203. DOI: 10.1016/0004-3702(81)90024-2
13. Jara-Wilde, J., Cerda, M., Delpiano, J., Härtel, S.: An Implementation of Combined Local-Global Optical Flow **5**, 139–158. DOI: 10.5201/ipol.2015.44
14. 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. Morgan Kaufmann. URL http://dl.acm.org/citation.cfm?id=1623264.1623280
15. 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 **2**(2), 1070–1076. DOI: 10.1109/LRA.2017.2658940
16. Moussu, C.: GPU based real-time optical Flow computation. p. 108. Imperial College London
17. Nguyen, M.T., Castonguay, P., Laurendeau, E.: GPU parallelization of multigrid RANS solver for three-dimensional aerodynamic simulations on multiblock grids **75**(5), 2562–2583. DOI: 10.1007/s11227-018-2653-6
18. Podestá, E., do Nascimento, B.M., Castro, M.: Energy Efficient Stencil Computations on the Low-Power Many-core MPPA-256 Processor. In: M. Aldinucci, L. Padovani, M. Torquati (eds.) Euro-Par 2018: Parallel Processing. Springer. DOI: 10.1007/978-3-319-96983-1_46
19. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM
20. 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. DOI: 10.1109/ICIP40778.2020.9191164
21. Shewchuk, J.R.: An Introduction to the Conjugate Gradient Method without the Agonizing Pain. Carnegie-Mellon University. Department of Computer Science
22. Sun, D., Roth, S., Black, M.J.: A Quantitative Analysis of Current Practices in Optical Flow Estimation and the Principles Behind Them **106**(2), 115–137. DOI: 10.1007/s11263-013-0644-x
23. Tabik, S., Ortega, G., Garzón, E.M.: Performance evaluation of kernel fusion BLAS routines on the GPU: Iterative solvers as case study **70**(2), 577–587. DOI: 10.1007/s11227-014-1102-4
24. Williams, S.W.: Auto-Tuning Performance on Multicore Computers. EECS Department University of California
25. Woźniakowski, H.: Roundoff-error analysis of a new class of conjugate-gradient algorithms **29**, 507–529. DOI: 10.1016/0024-3795(80)90259-1
26. 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. DOI: 10.1007/978-3-540-74936-3_22