Noname manuscript No.
(will be inserted by the editor)

Optical flow algorithms optimized for speed, energy and accuracy on embedded GPUs

Thomas Romera · Andrea Petreto · Florian Lemaitre · Manuel Bouyer · Quentin Meunier · Lionel Lacassagne[1] · Daniel Etiemble[3]

Received: date / Accepted: date

Abstract Embedded Computer Vision is a hot field of research that requires *trade-offs* in order to balance execution time, power consumption and *accuracy*. In that field, dense optical flow estimation is a major tool used in many applications. Many algorithms have been designed, focusing on accuracy, very few works address trade-offs and implementation on embedded hardware.

This paper tackles these trade-offs for embedded GPU through the example of the well-known TV-L[1] algorithm. Thanks to High Level Transforms – operator fusion and pipeline – and taking into account the iterative aspect of these algorithms, we achieve a speedup of ×3.7 versus OpenCV. Moreover, we show that a 16-bit half precision implementation has a higher *accuracy* than the 32-bit precision one for the same frame processing time on NVIDIA Jetson boards.

Furthermore, this work can be generalized to any kind of iterative stencil-based algorithms.

Keywords Computer Vision · Embedded GPU · Iterative Stencils · Optical Flow · Energy Consumption · Half-precision Computation

1 Introduction

Embedded Computer Vision is getting more and more widespread with applications in many fields. The algorithms used today are complex, time demanding and power consuming, while they face real-time and electrical consumption constraints. One of the main challenges in Computer Vision algorithms is to optimize their implementation in order to reduce both execution time and power consumption.

A common problem in Computer Vision is optical flow estimation, defined as the apparent motion of the pixels between two image frames. First introduced by Horn and Schunck in 1981 [18], it is used in many applications such as video denoising [31,38], motion deblurring [2], action recognition [9,14,25], 3D scene reconstruction [27], super-resolution [24], particle image velocimetry [10], video enhancement [31], object detection [3], object tracking [21], robotics navigation [11] and 3D scene reconstruction [27].

The Middlebury database [4] is the reference database for optical flow algorithms and publications and includes nearly 200 of them. The main comparisons criteria in this base are qualitative, e.g. the average endpoint error or the average angular error. In fact, most articles provide very little information about the execution time, as optical flow quality is regarded as the main – if not the only – metric. Yet, most of these algorithms are iterative and can take a long time to converge. Machine learning algorithms make up the second largest share of algorithms type and can also require a lot of time for inference. Therefore, if this database is a major contribution to this field of research, it lacks information about the performance and the multi-constraint trade-offs in terms of flow quality, execution time and power consumption for these algorithms.

Optical flow algorithms have different characteristics in terms of computational power requirements and optical flow output quality. While some of them have to be run offline, others can be used in a real-time context, i.e. process a frame of a given size in less than 40 ms.

Thomas Romera · Andrea Petreto · Florian Lemaitre · Manuel Bouyer · Quentin Meunier · Lionel Lacassagne
LIP6, Sorbonne University, CNRS
E-mail: {first name}.{last name}@lip6.fr

Thomas Romera · Andrea Petreto
LERITY - ALCEN, Cergy-Pontoise, FRANCE

Daniel Etiemble
LISN, University Paris Saclay

For the latter, high performance is necessary but having a good trade-off between computation time, optical flow accuracy and energy consumption is also crucial, especially in an embedded context. In this perspective, finding and using high performance and power efficient architectures is a real need.

Embedded GPUs are natural candidates for Computer Vision algorithms as they offer high performance computing capabilities and relatively low-power consumption – typically between 10 and 30 Watts. However, only a few implementations of optical flow algorithms use them as they require efficient programming to take advantage of their computational resources, which are much more limited than on full-size, non-embedded GPUs. Furthermore, these implementations focus more on the mathematical formulations rather than on efficient implementation details.

This article focuses on the TV-L^1 optical flow estimation [40] since it is a well-known and robust algorithm, able to deal with occlusions. The structure of this algorithm lends itself well to algorithmic transformations, allowing for fast and low energy consuming implementations on embedded devices. In this work, we study new optimized implementations of this algorithm on embedded GPUs, and compare them with State-of-the-Art CPU and GPU implementations. Our contributions are three-fold:

- we give an evaluation of High Level Transforms (operator fusion and pipeline) for iterative operators like optical flow algorithms,
- we provide a quantitative and qualitative approach for multi-constraint trade-offs: speed and precision vs power consumption and accuracy,
- we perform a comparison of 16-bit and 32-bit precision implementations from an accuracy point of view.

Several optimized CPU implementations of TV-L^1 have been developed [40,36], as well as an OpenMP version [28] and a SIMD version [30]. An FPGA implementations has also been developed [6] and optimized in terms of memory allocation and power consumption [15]. Yet, GPU implementations are much less widespread. [36] and [5] indicate timing results for straightforward GPU implementations but no in-depth study was performed and no implementation details were mentioned. This is also the case for the implementations in [12]: timing is indicated but few implementation details are provided. Finally, the TV-L^1 algorithm is also available in the OpenCV library [7], for both CPUs and GPUs.

NVIDIA Jetson boards (AGX Xavier, TX2 and Nano) have been selected to evaluate the GPU implementations. It is a family of embedded boards widely used in Computer Vision and autonomous vehicles.

We have two claims. First we claim that our TV-L^1 implementations *outperforms* the State-of-the-Art by a factor of $\times 3.7$, thanks to High Level Transforms. Second, we claim that for the same amount of per-frame computation time, a 16-bit half precision implementation has a *higher accuracy* than a 32-bit one.

Section 2 demonstrates the main challenges associated with efficient optical flow estimation on embedded architectures. Section 3 presents the optimizations and the design choices of the TV-L^1 implementations on embedded GPUs. Section 4 presents several benchmarks and their results regarding processing speed, energy consumption, floating-point precision and optical flow accuracy. Finally, section 5 concludes.

2 Background and GPU basics

2.1 Related Work

Dense optical flow estimation algorithms have been a focus for optimizations on embedded CPU platforms. Papers [30,31] present efficient CPU SIMD implementations of the Horn-Schunck and the TV-L^1 algorithms on the NVIDIA Jetson embedded systems. Paper [34] presents a first optimization of TV-L^1 on embedded GPUs. The efficiency of GPUs has also been shown for other iterative optical flow algorithms like the Lucas-Kanade method [26] using both OpenMP [17] and OpenACC [16] API. However, most of the optical flow estimation studies on GPUs focus only on the optical flow accuracy and precision [4,1,5,23]. Computing speed is sometimes provided but not studied in depth and energy consumption is usually not measured at all.

Recently, many new methods based on deep learning [20,39,19] have shown impressive results in terms of optical flow quality. However, these methods are not good candidates for embedded systems since they are slower and require large GPUs to run and thus have a large power requirement (typically several hundred watts). Even the fast FlowNet 2.0 [20] method takes 7 ms to process 1024×436 pixel images on an NVIDIA GTX 1080 GPU. We tested FlowNet 2.0 on the embedded Jetson AGX platform. On the MPI-Sintel dataset it takes 660 ms per frame with its regular pre-trained model and 343 ms using its fastest pre-trained model. A speedup of $\times 8.6$ is needed to achieve real-time processing at 25 image frames per second for 1024×436 pixels images. This resolution is not even an HD format. In comparison, the TV-L^1 OpenCV implementation requires only 10ms per frame at this resolution [34, 7], without providing a fully optimized implementation. TV-L^1 is therefore a much better candidate for embedded applications than deep-learning methods.

TV-L^1 is a well-known and still widely used algorithm which provides an interesting tread-off between speed and quality compared to the other State-of-the-Art methods [4]. It is also used as the baseline for other more complex methods [12,37,36].

This article first focuses on speedups brought by High Level Transforms [22], which are well suited for TV-L^1, and then deals with optical flow quality, accuracy and convergence speed with respect to computation precision. As far as we know, the impact on both speed and accuracy when using half precision floating point values [13,32] for storage and computation instead of single precision has not yet been studied.

2.2 GPU hardware specificities

In this section, we describe some of the GPU's specificities exploited for our optimized implementations.

Compared to CPUs, GPUs are composed of simpler computation cores called CUDA cores but in much greater number. The goal is to maximize throughput over minimizing latency. The computation cores are aggregated within multiple *streaming-multiprocessors* (SM). All the cores of one sub-core are synchronized and execute the same instruction at the same time.

GPUs have several memory levels. Accessible to all threads, the global memory is implemented on a separate DRAM chip. The L2 cache is a small on-chip SRAM shared between all the Streaming Processors (SMs). The L1 cache is a fast on-chip SRAM shared between all the sub-cores of the same SM.

There is usually much less cache memory per core on a GPU than on a CPU. For example, on the Jetson AGX, there is a total of 1024 KB of L1 data cache and 512 KB of L2 cache on the GPU for 512 F_{32} cores. The 8-core ARM v8.2 64-bit CPU has a total of 512 KB of L1 data cache, 8192 KB of L2 cache and 4096 KB of L3 cache for 8 cores. Memory-intensive transformations such as the one performed on CPU in [31] are therefore much more complex to implement on GPU.

2.3 GPU CUDA Programming model

In NVIDIA's CUDA programming model, computation is parallelized into multiple threads. Each thread executes, on different data, the same series of instructions described in a CUDA GPU function called a *kernel*. Threads are grouped into multiple *thread blocks* which form a *grid*. Thread blocks and grids can be either one, two or three-dimensional. All threads of the same thread block are executed on the same SM. They all share the same L1 cache. Part of the L1 cache can be explicitly managed and used as *Shared memory*. Shared memory is specifically allocated for each thread block so that all of its threads share the same data. They cannot access the Shared memory from other blocks, however. Inside a thread block, threads are scheduled in groups of 32 synchronized threads called *warps*. One warp is executed on the same sub-core. Threads are not synchronized beyond warp level except by using specific barrier instructions. Another SIMD parallelism is also available on GPU at an instruction level. The 32-bit *__half2* vector type can be used to exploit sub-word parallelism: Two F_{16} numbers are processed simultaneously.

Fine-tuning multiple parameters is critical to achieve the best performances on GPU. It is necessary to find a good balance between thread blocks and grid size dimensions as well as carefully using the different levels of GPU memories. Appropriate use of the Shared memory is important.

3 TV-L^1 algorithm implementations and optimizations

This section presents the mathematical scheme of the TV-L^1 optical flow estimation, and then details the designed implementations and optimizations on both embedded CPU and GPU.

3.1 TV-L^1 Optical Flow Estimation

The optical flow constraint equation states that for a given image intensity $I(x, y, t)$ function of time t and spatial coordinates x and y:

$$\nabla I \cdot \mathbf{u} + \frac{\partial I}{\partial t} = 0. \tag{1}$$

where $\mathbf{u} = (u_1, u_2) = \left(\frac{dx}{dt}, \frac{dy}{dt}\right)$ is the vector field of all the pixel trajectories of the image at one instant t.

This linear equation has two unknown variables u_1 and u_2 and thus cannot be solved without the introduction of additional constraints. This indeterminacy problem is know as the aperture problem in optical flow estimation methods.

The method proposed by [40] introduces an auxiliary variable $\mathbf{v} = (v_1, v_2)$ and a fixed-point iterative scheme over the optical flow $\mathbf{u}$ and a dual vector field $\mathbf{P} = (\mathbf{p}_1, \mathbf{p}_2) = ((p_{11}, p_{12}), (p_{21}, p_{22}))$.

Algorithm 1 TV-L^1 Numerical Scheme

1: Compute ∇I_1 ▷ centered gradient
2: **for** $w = 0$ to N_{warps} **do**
3: Compute $I_1(\mathbf{x}+\mathbf{u}^0))$ and $\nabla I_1(\mathbf{x}+\mathbf{u}^0)$ ▷ bicubic warp
4: **for** $k = 0$ to N_{iters} **do**
5: $\mathbf{v} \leftarrow \mathbf{u} + \text{TH}(\mathbf{u}, \mathbf{u}^0)$ ▷ thresholding
6: Compute $\text{Div}(\mathbf{P})$ ▷ divergence
7: $\mathbf{u} \leftarrow \mathbf{v} + \theta\,\text{Div}(\mathbf{P})$ ▷ update optical flow
8: Compute $\nabla\mathbf{u}$ ▷ forward gradient
9: $\mathbf{P} \leftarrow \frac{\mathbf{P}+\tau/\theta\nabla(\mathbf{u})}{1+\tau/\theta|\nabla(\mathbf{u})|}$ ▷ estimate dual variable

The gradient ∇I_1 is computed using central differences ; the warpings $I_1(\mathbf{x}+\mathbf{u}^0))$ and $\nabla I_1(\mathbf{x}+\mathbf{u}^0)$ are computed using bicubic interpolation ; the thresholding operation used to update the auxiliary variable $\mathbf{v}$ is defined in equations 2 and 3 ; the divergence $\mathrm{Div}(\mathbf{P})$ is computed uing backward differences ; the gradient $\nabla\mathbf{u}$ is computed using forward differences.

$$\rho(\mathbf{u}) = \nabla I_1(\mathbf{x}+\mathbf{u}^0) \cdot (\mathbf{u}-\mathbf{u}^0) + I_1(\mathbf{x}+\mathbf{u}^0) - I_0(\mathbf{x}) \qquad (2)$$

$$\mathrm{TH}(\mathbf{u},\mathbf{u}^0) = \begin{cases} \lambda\theta\nabla I_1(\mathbf{x}+\mathbf{u}^0) & \text{if } \rho(\mathbf{u}) < -\lambda\theta|\nabla I_1(\mathbf{x}+\mathbf{u}^0)|^2, \\ -\lambda\theta\nabla I_1(\mathbf{x}+\mathbf{u}^0) & \text{if } \rho(\mathbf{u}) > \lambda\theta|\nabla I_1(\mathbf{x}+\mathbf{u}^0)|^2, \\ \rho(\mathbf{u})\frac{\nabla I_1(\mathbf{x}+\mathbf{u}^0)}{|\nabla I_1(\mathbf{x}+\mathbf{u}^0)|^2} & \text{otherwise.} \end{cases} \qquad (3)$$

Based on the TV-L^1 numerical scheme, 5 steps are needed to update the optical flow:

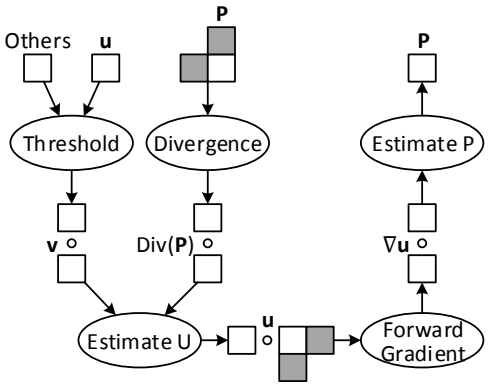


Fig. 1: Iterative steps of the TV-L^1 scheme before operator fusion. This consumer/producer representation highlights the data dependencies of each subsection of the iterative scheme. Operators are centered on the white square and the spatial dependencies shown in gray. ∘ symbolizes function composition i.e. the output of a step is the input of the next.

- a thresholding step (Threshold),
- a step computing the divergences of $\mathbf{P}$ (Divergence),
- a step estimating $\mathbf{u}$ (Estimate U),
- a step for the forward gradient of $\mathbf{u}$ (For. Gradient),
- a step estimating $\mathbf{P}$ (Estimate P).

This is shown in figure 1 along with the horizontal and vertical dependencies needed to compute one central pixel for each step. These horizontal and vertical dependencies are introduced by the backward and forward differences used to compute the divergences and the forward gradients respectively. A complete TV-L^1 iteration is the composition of all these 5 steps.

This iterative scheme estimates the optical flow for small displacements. For bigger displacements, downscaled versions of the input image are used. The flow is computed at the smallest resolution and refined up to the original resolution.

The general TV-L^1 scheme also includes an outer iterative scheme, which compensates the partially computed optical flow on the input images to get more accurate results using bicubic image warping. Given its small impact on optical flow quality but its large impact on computation time [29], the number of these outer iterations, called warps, was set to 1 in the rest of the article.

The general scheme is embedded in a Gaussian pyramid to downscale the input images until the displacements are small enough. The method is first run on the most downscaled level and the estimated motion propagated to the next scale and refined until the original resolution is reached.

3.2 CPU Implementation

This section summarizes the main optimization steps of the CPU SIMD implementation presented in [30,31].

First, it is possible to perform operator fusion and only use a first step to update the vector field $\mathbf{u}$ and a second step to update the auxiliary dual vector field $\mathbf{P}$. Intermediate results are used immediately [22] without

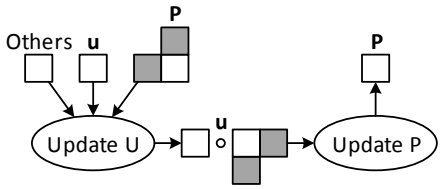


Fig. 2: Inner iterative steps of the TV-L^1 method after operator fusion.

memory transfers. However, due to the data dependencies, further operator fusion is more complex. This is illustrated in figures 3 and 4. Figure 3 shows the two full stencils used to update $\mathbf{u}$ and $\mathbf{P}$. They are computed from the two half stencils shown previously in figure 2 and are the result of mathematical composition. The ensuing data dependencies are identical for the computation of $\mathbf{u}$ and $\mathbf{P}$. As the number of TV-L^1 iterations increases, the input data dependencies required to compute one pixel also increases for both stencils. This growing spatial dependency is shown in figure 4 and illustrates the increased data access and computation needed to compute a single pixel.

Instead of updating the entire vector fields $\mathbf{u}$ and $\mathbf{P}$, we establish an iteration pipeline and update the lines as soon as it is possible. Doing so increases memory locality and performances.

3.3 GPU Implementation

In order to obtain the best performances from our GPU hardware, we must ensure to limit superfluous compu-

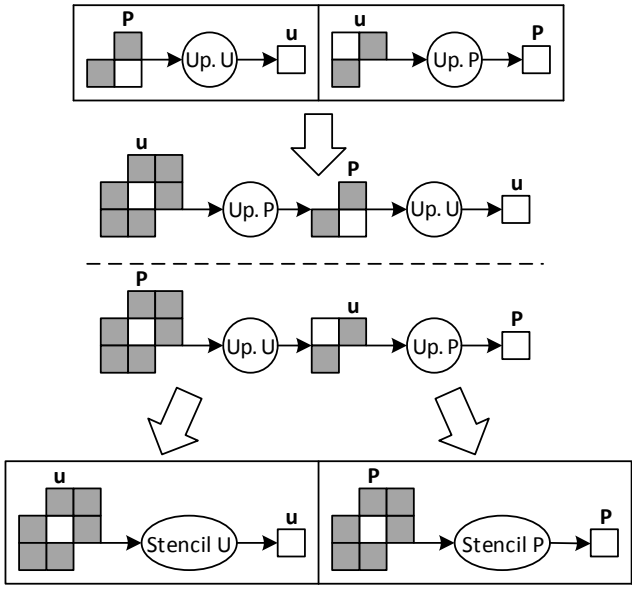


Fig. 3: Full stencil derivation for one iteration of **u** and **P** from the two initial half-stencils.

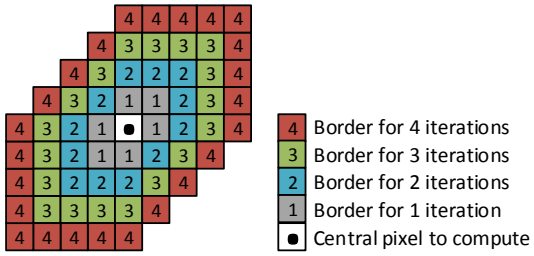


Fig. 4: Full stencil size growth depending on the number of iterations of TV-L^1.

tation already performed and maximize data reuse in fast memory. If many data accesses are needed, they must make use of the fast shared memory or registers. As such, our optimized GPU implementations make use of the CPU pipeline optimization scheme presented previously. We present four levels of optimization schemes for the GPU implementations: Global, Shared_fusion, Global_pipeline and Shared_pipeline. Several additional schemes were developed and tested but were found to be slower and less efficient than the Global version. They will not be mentioned in this paper. This was notably the case for implementations making use of *shuffle* instructions in order for threads in the same warp of 32 threads to share data. The limited ensuing block size and added warp border management degrade overall performance. These schemes are declined in F_{32} and F_{16} floating point numbers and use CUDA vector type *__float2* and *__half2*. Finally, the grid and thread block dimensions were tuned for each version and for each tested platform. The Global and Shared_fusion versions corresponds to the implementations presented in [34].

The Global scheme implements operator fusion and only uses the Global memory. The two half-stencils are executed one after the other by to consecutive kernels. Memory accesses are reduced, data locality and arithmetic intensity improved. The overhead due to GPU kernel launch is also reduced since there are only two sequential kernels to run per TV-L^1 iteration: one to update **u** and one to update **P**. For border accesses outside of the image, the index is clamped and the border value reused. It is the most straightforward parallel scheme as one thread is tasked to compute one half-stencil for one pixel of the image. This scheme serves as a baseline for our optimizations.

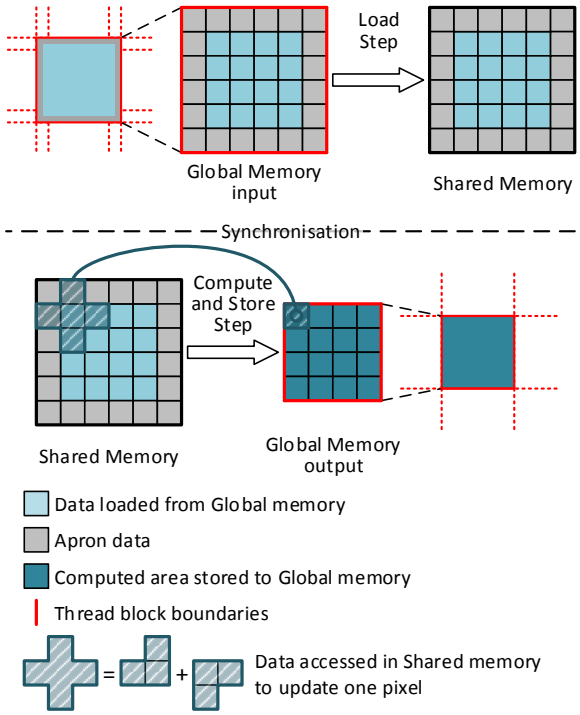


Fig. 5: Shared_fusion thread block memory access pattern for 1 iteration. All the data needed to compute one iteration is loaded in Shared memory. The computation step is then performed using only Shared memory. Finally, the results within the red area are written into Global memory.

The Shared_fusion scheme further reduces the number of launched CUDA kernels to one per iteration by merging the two kernels of the Global version. Each thread requires the union of the neighboring pixels of both half-stencils described in the stencil illustration of figure 3. The number of memory accesses to the Global memory is decreased by using the Shared memory. The thread block memory access pattern is described in figure 5 and includes a load step from the Global memory into

the Shared memory. Since the Shared memory is shared by all the threads of the same thread block, the data loaded by adjacent threads can be reused. This transfer is synchronized with a barrier. Some threads are used only to load border data which are processed by neighboring threads. These threads return after performing the load step and do not perform the compute step. Like in the Global scheme, the index outside the image area are clamped and the nearest border values used. This level of optimization usually corresponds to what is found in other State-of-the-Art implementations [35, 33]. [35] performs further optimizations and fuses several iterations of the numerical scheme.

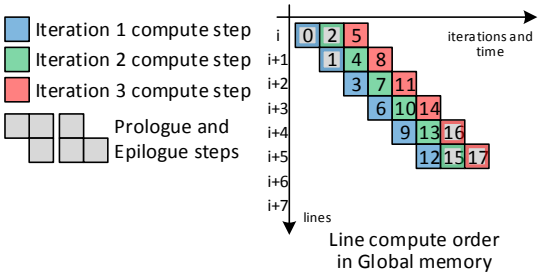


Fig. 6: Global_pipeline thread block computer order for the steady-state of a 3-iteration pipeline on 6 consecutive lines. The thread block computes a full iteration on each entire line.

The Global_pipeline scheme implements the iteration pipeline described in section 3.2 using only the Global memory. The pipeline follows the vertical dependencies of TV-L[1]: a whole line is entirely processed and used as soon as possible to compute the next iterations steps. The CUDA grid contains several blocks, each working in parallel on a sub-strip of the input images. Each thread block is in charge of fully processing one strip. If the number of threads is smaller than the line width, lines are computed in multiple steps by the same thread block. The Global_pipeline compute order pattern is shown in figure 6. The prologue and epilogue are executed in border areas of the arrays that are not clamped: those areas are allocated. The horizontal accesses outside of the image width are clamped like in the Global scheme. This scheme aims at reusing new computed data as soon as it is available in order to maximize data locality in the cache. This scheme makes use of nearby data still present in cache to update a line. An increased number of pipelined iterations increases the data dependencies for the computation of a line however. As such, top and bottom border extensions are added depending of the number of pipelined iterations.

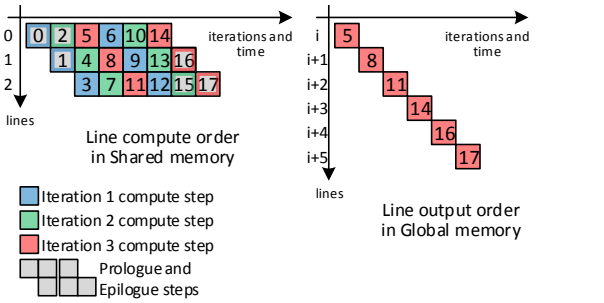


Fig. 7: Shared_pipeline thread block compute order for the steady-state of 3 pipelined iterations on 6 consecutive lines. This pattern is run multiple times with an offset for lines wider than the thread block width.

The Shared_pipeline scheme also implements the iteration pipeline but uses the Shared memory to share data already computed by one thread to the other threads of the same block. Modular accesses are used to reduce the number of lines needed for the computation steps. The horizontal accesses outside of the arrays width are still clamped. Similarly, several of these blocks work in parallel to process smaller sub-strips of the input images. Like the previous Global_pipeline scheme, this implementations aims at maximizing data locality. The data reuse is performed here in the fast Shared memory.

The size of the Shared Memory is limited to 48 KB per blocks and is too small to contain all the data needed to process lines wider than 128, 256 or 512 pixels depending on the pipeline depth and the computation format precision. The amount of Shared memory (in bytes) needed to process 1D blocks of w threads computing a pipeline of P_i iterations is:

$$Q_{\text{Shared Memory}} = Q_{\text{Floating}} \times N_a \times w \times (P_i + 1), \qquad (4)$$

where Q_{Floating} is the size in bytes of the floating point numbers used in the kernel (8 bytes for versions using *__float2* and 4 bytes for *__half2* versions), and with $N_a = 6$ the number of arrays required for computation.

For F_{32}, up to 7 iterations can be computed using blocks of 128 threads and up to 3 by using blocks of 256 threads. For F_{16}, thanks to their smaller footprint, the limit is 15 iterations for blocks of 128 threads, 7 for blocks of 256 and 3 for blocks of 512. The thread block must therefore first processes all the iterations on one part of the line before processing the rest of the line. An entire part of the sub-strip is processed before computing the next parts. This computer order pattern is further illustrated in figure 7.

4 Results

Table 1 shows the required memory operations and floating-point operations (FLOP) needed to compute 1 warp and 10 iterations for all our implemented

Version	Floating Point Operations per pix	MEMs accesses per pix	Arithmetic Intensity	Time (ns per pix)	TP (GFLOPS)	BW (GB/s)
Global_F32	640	181	0.44	8.17	78	165
Global_F16	508	181	0.70	4.36	117	155
Shared_fusion_F32	640	251	0.32	8.13	79	230
Shared_fusion_F16	488	261	0.47	4.31	113	226
Global_pipeline_F32	640	181	0.44	6.65	96	203
Global_pipeline_F16	508	181	0.70	4.02	126	168
Shared_pipeline_F32	660	219	0.38	5.10	129	**320**
Shared_pipeline_F16	553	219	0.63	**2.86**	**193**	285

Table 1: Memory accesses and floating point operations per pixel of our TV-L1 implementations on the Jetson AGX. The associated execution time, memory bandwidth and computational throughput were computed for 2048×2048 images and monoscale, 1 warp and 10 iterations.

monoscale GPU implementations for 2048×2048 images.

To evaluate the gains brought by our optimizations and the use of F_{16} numbers, various benchmarks were performed and are presented in this section. The tested versions of the TV-L^1 implementation are described hereafter. Version names F⟨XX⟩ refer to the floating point computation format, F_{32} or F_{16}:

- Neon_F⟨XX⟩: optimized Neon SIMD CPU version,
- Global_F⟨XX⟩: GPU optimized implementation with operator fusion using only the Global memory,
- Shared_fusion_F⟨XX⟩: GPU optimized implementation with operator fusion and kernel fusion using the Shared memory,
- Global_pipeline_F⟨XX⟩: GPU optimized implementation with operator fusion and iteration pipeline using only the Global memory,
- Shared_pipeline_F⟨XX⟩: GPU optimized implementation with operator fusion, kernel fusion and using the Shared memory.

Those versions are compared to find the best implementation on the 3 NVIDIA Jetson platforms, the Jetson AGX Xavier, TX2 and Nano. They are compared in terms of processing time, energy consumption and optical flow quality.

4.1 Processing Time

First, the different versions are evaluated in terms of processing time, normalized to find the average time taken to compute one pixel of the input images. For this series of benchmarks, qualitative comparisons are not taken into account. Only the computation speed of the different TV-L^1 iterations optimizations is measured. Therefore, a monoscale version of TV-L^1 was tested with 1 warp and 10 iterations per warp with empty images ranging from 128×128 up to 2048×2048 pixels. The pipelined versions iterate twice a pipeline of depth 5 iterations in order to get a total of 10 iterations.

Figure 8 shows a detailed comparison between our GPU implementations on the Jetson AGX. Overall, the

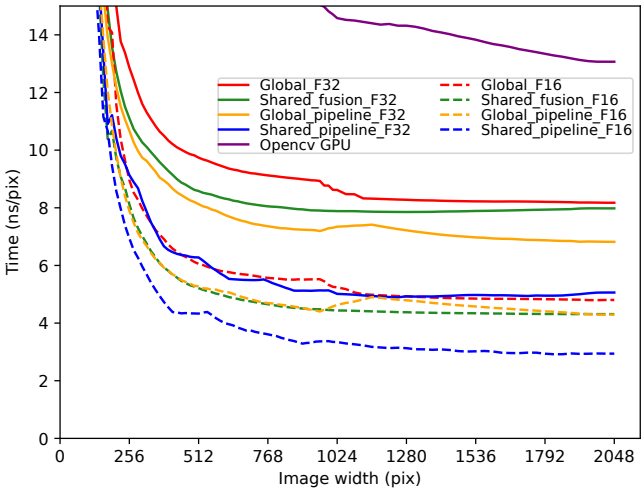


Fig. 8: Execution time (ns/pix) of TV-L^1 implementations on AGX.

fastest implementation is the Shared_pipeline_F16. All the optimizations enable us to reach 2.9 ns/pix for resolutions up to 2048×2048 pixels. The Shared_pipeline_F32 is also the fastest F_{32} version with a speed of 5.1 ns/pix for 2048×2048 pixel frames. Thanks to the pipeline, data locality is increased and the number of memory transfers with the Global memory decreases. Furthermore, using the Shared memory offers lower latency memory (similar to the L1 cache) and an increased memory bandwidth (128 bytes per clock and per SM) compared to the Global memory.

Using only the Shared memory like in the Shared_fusion versions is however not sufficient to achieve the greatest speedup. The amount of data reused in the Shared memory in the non-pipelined versions is very limited. The Global_pipeline_F32 version does not use the Shared memory but is able to reuse more data in the Global memory thanks to the L1 and L2 data cache. It is faster than the Shared_fusion_F32 version. This phenomenon is much less present when using F_{16} numbers. The Shared_fusion_F16 version is much closer and slightly

faster for images greater than 1024×1024 pixels than the Global_pipeline_F16 version.

Those results are further detailed in table 1 and shows that our fastest Shared_pipeline_F16 implementation is the one with the highest computation throughput and the second highest memory bandwidth.

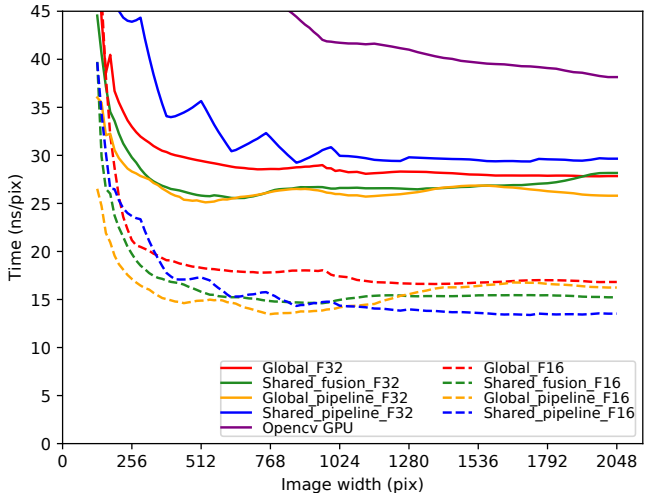


Fig. 9: Execution time (ns/pix) of TV-L^1 implementations on TX2.

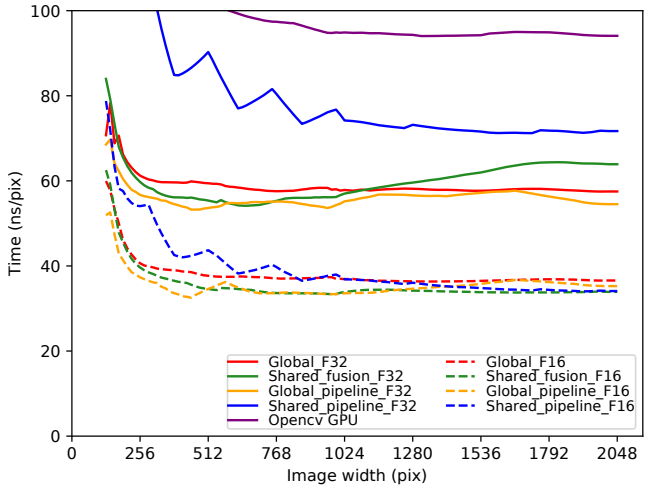


Fig. 10: Execution time (ns/pix) of TV-L^1 implementations on Nano.

Figures 9 and 10 show the same comparison on the Jetson TX2 and Nano. The main difference here is the slower performance of the Shared_pipeline_F32 version. It is the slowest version for these two platforms whereas it is the fastest F_{32} version on the Jetson AGX. This is due to different hardware limitations in the TX2 and Nano GPU architectures.

Firstly, on the Jetson AGX, 2 Shared_pipeline_F32 thread blocks can be concurrently active on the same SM. This number is limited by the Shared memory available on the GPU SMs. One thread block requires 36 KB of Shared memory for the Shared_pipeline_F32 version. On the Jetson TX2 and Nano, only 1 thread block can be launched per SM since they only have 64 KB of available Shared memory per SM. For the Shared_pipeline_F16 version, 5 blocks of 128 threads can be launched per SM on the AGX but only 3 per SM on the TX2 and Nano.

Secondly, there are half as many 32-bit registers per SM on the TX2 and Nano than on the AGX (32768 for the TX2 and Nano and 65536 for the AGX). This limits the maximum number of threads that can be launched in the pipelined versions. We require 1472 registers per warp for the Global_pipeline_F32 and 1536 for the Global_pipeline_F16 version on TX2 and Nano. Thread blocks of a maximum size of 640 threads are able to be allocated on both architectures. On the AGX, we are able to push the maximum block size up to 1024 threads, the maximum CUDA thread block width in one dimension.

Finally, more SM are available on the AGX (8 SMs) than on the TX2 (2 SMs) and the Nano (1 SM).

Overall, more blocks of a greater size can be launched on the AGX than on the other boards.

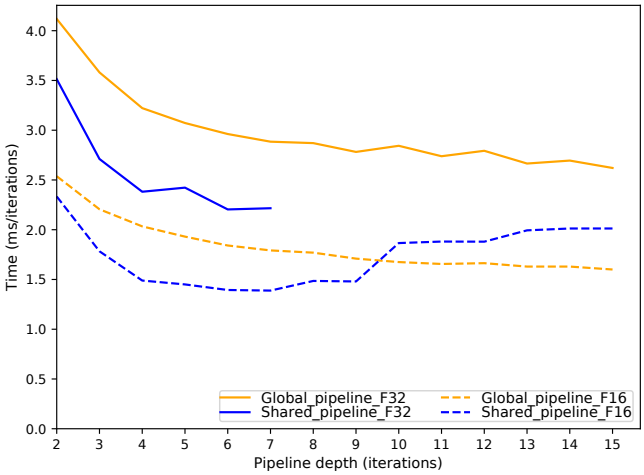


Fig. 11: Time per iterations (ms/iters) according to pipeline depth (iterations) for the pipelined versions (with and without Shared memory) on the Jetson AGX.

In addition, it is also necessary to determine the optimal pipeline depth for the pipelined schemes. Figure 11 shows the time needed to compute one iteration of TV-L^1 depending of the pipeline depth for the global_pipeline and shared_pipeline implementations using F_{32} and F_{16} numbers on 2048×2048 pixels images. For the Global_pipeline scheme, the pipeline depth is limited only by the amount of Global memory on the board. For the Shared_pipeline scheme, we found that the optimal pipeline depth is 6 for the F_{32} version and 7 for the

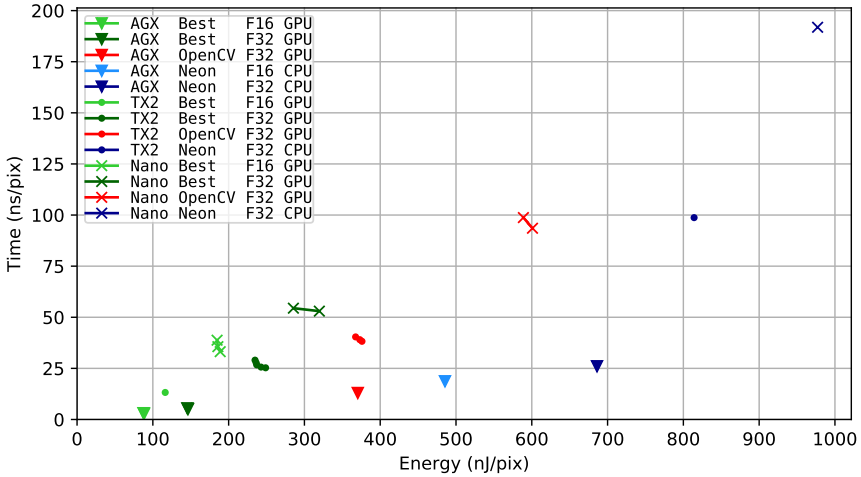


Fig. 12: Time (ns/pix) and energy (nJ/pix) operating points for several TV-L^1 implementations on each Jetson boards. Several clock frequencies are tested for each versions for a 2048 × 2048 pixel image.

F_{16} version. As previously said, the pipeline depth is strongly limited by the amount of Shared memory. For convenience, we use a pipeline of depth 5 iterations for our benchmarks since it is very close to the best value while allowing us to test iterations multiples of 5.

4.2 Energy Consumption

The implementations are then evaluated from an energy point of view. The same monoscale configuration of 1 warp and 10 iterations per warps is chosen. Again, a pipeline depth of 5 iterations is set and run twice for the pipelined versions. Images with a resolution of 2048 × 2048 pixels are used to ensure a sufficient load on the GPU.

Figure 12 presents the results in the (time per pixel, energy per pixel) space for the TV-L^1 implementations on GPU along with the optimized Neon CPU versions and the GPU OpenCV version. Multiple points on a curve correspond to different clock frequencies. Only the maximum frequency is shown for each Neon CPU versions. The AGX achieves the best performance both in terms of runtime and energy consumption, followed by the TX2 and the Nano. The fastest GPU implementation is Shared_pipeline_F16 and it is 7.4× faster and 6.4× more energy efficient than the optimized version using Neon_F16 of the ARM CPU on the AGX. It is also 5.2× faster and 4.9× more energy efficient than the OpenCV F_{32} GPU version. The best F_{32} GPU implementation is the Shared_pipeline_F32 version and is 5.7× faster and 5.3× more energy efficient compared to the CPU Neon_F16 version on CPU. Again, compared to the F_{32} GPU version, it is 2.9× faster and 2.8× more energy efficient.

The TX2 is the second best board in terms of processing time and power consumption. The best implementation on this board is also the Shared_pipeline_F16 version. Compared to the AGX, it is 4.6× slower and 1.4× less power efficient.

Newer and more powerful hardware, while more power demanding allows for our optimization to run more efficiently. Compared to the two smaller and older boards, the Jetson AGX has a reduced total energy consumption and enables the optimized Shared_pipeline_F16 to run the fastest.

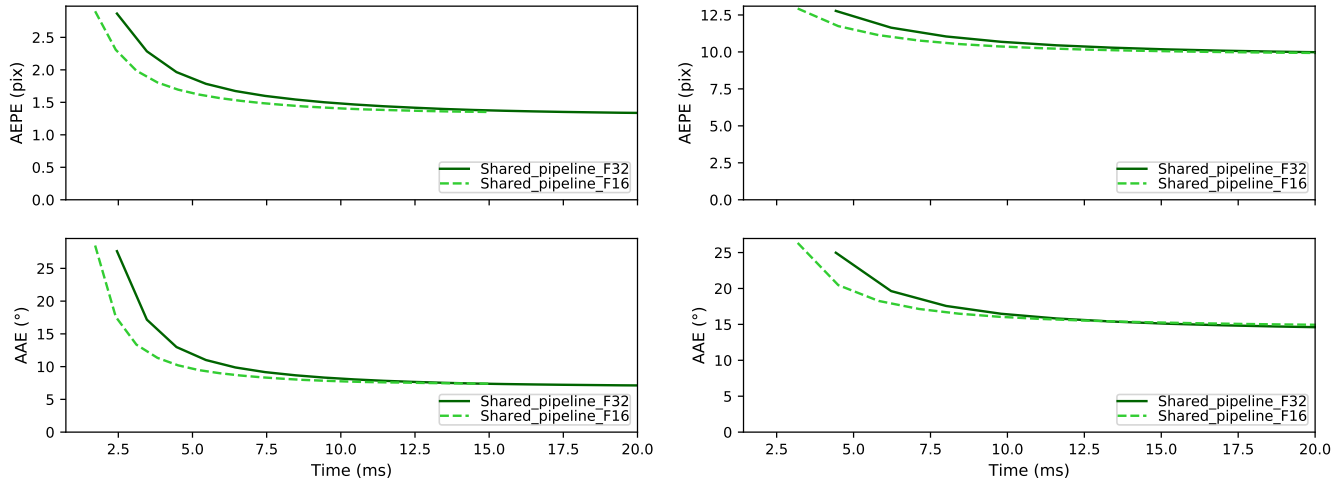
4.3 Optical Flow Quality

For this series of benchmarks, the fastest F_{32} and F_{16} pipelined implementations are tested on the public Middlebury optical flow datasets collection [4] and the MPI Sintel flow dataset [8]. We used the Jetson AGX board to run our tests. The TV-L^1 configuration used for the benchmarks is a 3-scale pyramid for the Middlebury datasets and a 5-scale pyramid for the MPI-Sintel datasets. Each pyramid level runs the same number of iterations. A 5-iteration pipeline is used to vary the number of iterations up to 100 to generate a set of time and accuracy points. We compare the computed optical flow with the provided ground truth in terms of average endpoint error (AEPE) and in terms of average angular error (AAE).

4.3.1 Middlebury dataset

The Middlebury dataset consists in 8 pairs of grayscale images of various resolutions (3 sequences of resolution 584 × 388 pixels, 4 sequences of resolution 640 × 480 pixels, and 1 sequence of resolution 420 × 380 pixels).

Figure 13b shows the impact of F_{16} precision on the computed flow accuracy. First, all the versions converge toward the same flow quality given enough time (between 1.3 and 1.4 pixels of average endpoint error and 7.2 and 7.9 degrees of average angular error). Second, for an equivalent computation time, the F_{16} versions have a lower average error than the F_{32} versions. Since



(a) Middlebury dataset.

(b) MPI-Sintel datasets.

Fig. 13: Average Endpoint Error (AEPE) and Average Angular Error (AAE) versus Execution time (ms) of our best TV-L^1 implementations on AGX.

	2.5 ms				5 ms				10 ms			
	AEPE		AAE		AEPE		AAE		AEPE		AAE	
Dataset	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}
Dimetrodon	0.91	**0.55**	19.60	**10.74**	0.29	**0.25**	5.47	**4.52**	0.20	**0.19**	3.43	**3.36**
Grove2	1.52	**0.74**	24.54	**10.28**	0.40	**0.26**	5.74	**3.78**	**0.22**	0.24	**3.33**	3.45
Grove3	2.47	**1.71**	27.15	**15.99**	1.34	**1.10**	11.71	**9.73**	1.01	**0.98**	8.88	**8.74**
Hydrangea	1.69	**0.93**	17.88	**7.78**	0.37	**0.33**	3.52	**3.35**	**0.30**	0.32	**2.92**	3.13
RubberWhale	0.42	**0.34**	13.39	**10.84**	0.28	**0.27**	9.05	**8.82**	**0.24**	0.25	**7.74**	7.87
Urban2	7.44	**6.77**	40.04	**28.32**	6.38	**5.92**	22.36	**17.23**	5.59	**5.30**	14.12	**12.08**
Urban3	6.34	**5.58**	50.08	**35.48**	5.03	**4.33**	27.70	**20.65**	3.95	**3.53**	17.89	**15.62**
Venus	1.86	**1.42**	21.51	**15.88**	0.89	**0.72**	10.77	**9.68**	0.52	**0.52**	**8.05**	8.12
Average all datasets	2.83	**2.25**	26.77	**16.91**	1.87	**1.65**	12.04	**9.72**	1.50	**1.42**	8.30	**7.80**

(a) Middlebury sequences.

	2.5 ms				5 ms				10 ms			
	AEPE		AAE		AEPE		AAE		AEPE		AAE	
Dataset	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}
alley_2	4.48	**3.07**	25.65	**18.37**	1.44	**0.94**	9.00	**6.27**	0.79	**0.79**	**5.16**	5.16
sleeping_1	1.48	**1.05**	19.09	**13.63**	0.54	**0.38**	7.07	**4.95**	0.28	**0.27**	3.66	**3.52**
cave_4	15.39	**12.73**	51.80	**40.25**	9.57	**7.92**	25.77	**19.79**	7.07	**6.84**	17.39	**16.95**
Average all datasets	14.99	**13.91**	37.38	**31.85**	12.33	**11.47**	22.74	**19.37**	10.65	**10.35**	16.38	**15.99**

(b) MPI-Sintel sequences.

Table 2: Average Endpoint Error (AEPE) and Average Angular Error (AAE) compared to ground-truth of our TV-L^1 implementations on AGX at 2.5, 5 and 10 ms of execution time.

the F_{16} versions are faster than the F_{32} versions, more iterations can be computed in the same time span.

Table 2a details both endpoint error and angular error at the fixed time intervals of 2.5 ms, 5 ms and 10 ms of execution time. For nearly all the datasets, the F_{16} is always better for both error metrics at each of the three intervals. Before the convergence point that occurs roughly at around 10 ms of execution time, we can observe an average decrease of 20% concerning the end-point error and a decrease of 37% for the angular error at 2.5 ms of execution time. At 5 ms, the F_{16} version has an end-point error 12% lower and an angular error 19% lower on average. For some sequences such as "Grove 2" or "Hydrangea", the error is even lower and reaches 50% for the 2.5 ms mark and 35% for the 5 ms mark.

Overall, the use of F_{16} on the Middlebury dataset is beneficial as it performs more TV-L^1 iterations in the same time span as the F_{32} implementation. The lower precision of F_{16} numbers is compensated and the computed flow is as much as or more precise than when using F_{32} numbers.

Implementation	5 ms		10 ms		20 ms		40 ms	
	Max resolution	W	Max resolution	W	Max resolution	W	Max resolution	W
OpenCV GPU (F_{32})	320×320	23.8	616×616	24.9	1000×1000	26.2	1484×1484	27.8
Best GPU F_{32}	644×644	23.2	962×962	25.7	1416×1416	28.1	2050×2050	29.7
Best GPU F_{16}	**896×896**	24.9	**1314×1314**	28.0	**1984×1984**	31.6	**2848×2848**	34.2

Table 3: Maximum frame resolution that can be computed in 5, 10, 20 and 40 ms depending on the TV-L^1 implementation on the AGX. The TV-L^1 configuration is a 3-scale, 1 warp per scale and 10 iterations per warp.

4.3.2 MPI-Sintel dataset

This dataset is composed of 23 sequences with associated ground truth along with 12 testing sequences. These sequences contain between 20 and 50 images of resolution 1024×436 pixels. The computed flow is compared with provided ground-truth flow for each consecutive pair of images for each sequences.

Figure 13a shows the impact of F_{16} precision on the computed flow accuracy. Like for the Middlebury database, the F_{16} versions have a lower average error compared to the F_{32} versions for both the end-point error and the angular error. This is less pronounced in figure 13a since this dataset contains about 3 times the number of sequences and more than 10 times the number of frames per sequences.

As such, table 2a focuses on the 3 sequences with the lowest average errors at the fixed time intervals of 2.5 ms, 5 ms and 10 ms of execution time. For these 3 sequences, the F_{16} versions have approximately a 30% lower end-point and angular error compared to their F_{32} counterparts. This is the case for times up to 5 ms. After the convergence point at around 10 ms, both F_{16} and F_{32} versions present similar end-point and angular errors. On average, the F_{16} version present a 7% lower end-point error and a 15% lower angular error for both 2.5 ms and 5 ms of execution time. Some sequences feature greater and more complex movements and increase both average error metrics for the entire MPI-Sintel datasets. The F_{16} version still remains the most accurate one compared to the F_{32} version for 2.5 ms and 5 ms of execution time.

4.4 Overall optimizations synthesis

Table 3 shows the maximum resolution and its associated required power that can be computed in 5, 10, 20 and 40 ms for the OpenCV TV-L^1 reference version and our best F_{32} and F_{16} implementations on the Jetson AGX Xavier. Our optimizations enable the use of bigger images frames between 7.8× larger for the smallest duration (5 ms) and 3.7× larger for the largest duration (40 ms) while limiting the increase in power to 1.23×. As shown before, F_{16} numbers are very efficient. Their use reduces the optical flow errors compared to ground-truth and increases the frames resolution compared to F_{32} versions.

5 Conclusion

Designing a fast, low power and accurate implementation of TV-L^1 algorithm for embedded GPUs is challenging. It needs many optimizations and trade-offs.

This article has shown that the proposed optimizations outperform the OpenCV reference by a factor of ×3.7. But the most important and *counter-intuitive* experimental result is that half-precision F_{16} computations, despite a smaller precision than F_{32} computations, and thank to a higher number of iterations, for the same amount of time, provide more accurate results.

Finally, we are confident that this work can be adapted to any iterative optical flow algorithms and more generally, to many stencil-based algorithms.

Acknowledgement

This work has been partially funded by the *Direction Générale de l'Armement* (DGA), French Ministry of Armed Forces.

References

1. Adarve, J.D., Mahony, R.: A filter formulation for computing real time optical flow. IEEE Journal of Robotics and Automation Letters (RA-L) **1**(2), 1192–1199 (2016)
2. Anger, J., Meinhardt-Llopis, E.: Implementation of local Fourier burst accumulation for video deblurring. Image Processing On Line **7**, 56–64 (2017)
3. Aslani, S., Mahdavi-Nasab, H.: Optical flow based moving object detection and tracking for traffic surveillance. International Journal of Electrical, Computer, Energetic, Electronic and Communication Engineering **7**(9), 1252–1256 (2013)
4. Baker, S., Scharstein, D., Lewis, J., Roth, S., Black, M.J., Szeliski, R.: A database and evaluation methodology for optical flow. International Journal of Computer Vision (IJCV) **92**(1), 1–31 (2011)
5. Bao, L., Jin, H., Kim, B., Yang, Q.: A comparison of TV-L1 optical flow solvers on GPU. GTC Posters **6** (2014)
6. Beretta, I., Rana, V., Akin, A., Nacci, A.A., Sciuto, D., Atienza, D.: Parallelizing the Chambolle algorithm for performance-optimized mapping on FPGA devices. ACM Transactions on Embedded Computing Systems (TECS) **15**(3), 1–27 (2016)
7. Bradski, G.: The OpenCV Library. Dr. Dobb's Journal of Software Tools (2000)
8. Butler, D.J., Wulff, J., Stanley, G.B., Black, M.J.: A naturalistic open source movie for optical flow evaluation. In: European conference on computer vision, pp. 611–625. Springer (2012)

9. Carreira, J., Zisserman, A.: Quo vadis, action recognition? a new model and the kinetics dataset. In: Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 6299–6308 (2017)
10. Champagnat, F., Plyer, A., Le Besnerais, G., Leclaire, B., Davoust, S., Le Sant, Y.: Fast and accurate PIV computation using highly parallel iterative correlation maximization. Experiments in fluids **50**(4), 1169–1182 (2011)
11. Chao, H., Gu, Y., Napolitano, M.: A survey of optical flow techniques for robotics navigation applications. Journal of Intelligent & Robotic Systems **73**(1), 361–372 (2014)
12. d'Angelo, E., Paratte, J., Puy, G., Vandergheynst, P.: Fast TV-L1 optical flow for interactivity. In: Proceedings of the 18th IEEE International Conference on Image Processing (ICIP), pp. 1885–1888 (2011)
13. Etiemble, D., Bouaziz, S., Lacassagne, L.: Customizing 16-bit floating point instructions on a NIOS II processor for FPGA image and media processing. In: 3rd Workshop on Embedded Systems for Real-Time Multimedia, 2005., pp. 61–66. IEEE (2005)
14. Feichtenhofer, C., Pinz, A., Wildes, R.P.: Spatiotemporal multiplier networks for video action recognition. In: Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 4768–4777 (2017)
15. Garcia, P., Bhowmik, D., Stewart, R., Michaelson, G., Wallace, A.: Optimized memory allocation and power minimization for FPGA-based image processing. Journal of Imaging **5**(1), 7 (2019)
16. Haggui, O., Tadonki, C., Sayadi, F., Ouni, B.: Efficient GPU implementation of Lucas-Kanade through OpenACC. In: VISIGRAPP (5: VISAPP), pp. 768–775 (2019)
17. Haggui, O., Tadonki, C., Sayadi, F., Ouni, B.: Memory efficient deployment of an optical flow algorithm on GPU using OpenMP. In: International Conference on Image Analysis and Processing, pp. 477–487. Springer (2019)
18. Horn, B.K., Schunck, B.G.: Determining optical flow. Journal of Artificial Intelligence (AIJ) **17**(1-3), 185–203 (1981)
19. Hui, T.W., Loy, C.C.: LiteFlowNet3: Resolving correspondence ambiguity for more accurate optical flow estimation. In: Proceedings of the 2020 European Conference on Computer Vision (ECCV), pp. 169–184 (2020)
20. Ilg, E., Mayer, N., Saikia, T., Keuper, M., Dosovitskiy, A., Brox, T.: Flownet 2.0: Evolution of optical flow estimation with deep networks. In: Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2462–2470 (2017)
21. Kale, K., Pawar, S., Dhulekar, P.: Moving object tracking using optical flow and motion vector estimation. In: 2015 4th international conference on reliability, infocom technologies and optimization (ICRITO)(trends and future directions), pp. 1–6. IEEE (2015)
22. Lacassagne, L., Etiemble, D., Hassan Zahraee, A., Dominguez, A., Vezolle, P.: High level transforms for SIMD and low-level computer vision algorithms. In: Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing (WPMVP), pp. 49–56 (2014)
23. Lazcano, V., Rivera, F.: GPU based Horn-Schunck method to estimate optical flow and occlusion. In: International Conference on Theory and Applications of Models of Computation, pp. 424–437. Springer (2019)
24. Lin, F., Fookes, C., Chandran, V., Sridharan, S.: Investigation into optical flow super-resolution for surveillance applications. In: WDIC 2005: APRS Workshop on Digital Image Computing: Workshop Proceedings, pp. 73–78. University of QLD (2005)
25. Lin, J., Gan, C., Han, S.: Tsm: Temporal shift module for efficient video understanding. In: Proceedings of the 2019 IEEE/CVF International Conference on Computer Vision (ICCV), pp. 7083–7093 (2019)
26. Lucas, B.D., Kanade, T., et al.: An iterative image registration technique with an application to stereo vision. In: Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI) (1981)
27. Newcombe, R.A., Davison, A.J.: Live dense reconstruction with a single moving camera. In: Proceedings of the 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1498–1505 (2010)
28. Pérez, J.S., Meinhardt-Llopis, E., Facciolo, G.: TV-L1 optical flow estimation. Image Processing On Line (IPOL) **3**, 137–150 (2013)
29. Petreto, A.: Débruitage vidéo temps réel pour systèmes embarqués. Ph.D. thesis, Sorbonne université (2020)
30. Petreto, A., Hennequin, A., Koehler, T., Romera, T., Fargeix, Y., Gaillard, B., Bouyer, M., Meunier, Q.L., Lacassagne, L.: Energy and execution time comparison of optical flow algorithms on SIMD and GPU architectures. In: Proceedings of the 2018 Conference on Design and Architectures for Signal and Image Processing (DASIP), pp. 25–30 (2018)
31. Petreto, A., Romera, T., Lemaitre, F., Masliah, I., Gaillard, B., Bouyer, M., Meunier, Q.L., Lacassagne, L.: A new real-time embedded video denoising algorithm. In: Proceedings of the 2019 Conference on Design and Architectures for Signal and Image Processing (DASIP), pp. 47–52 (2019)
32. Piskorski, S., Lacassagne, L., Bouaziz, S., Etiemble, D.: Customizing CPU instructions for embedded vision systems. In: Computer Architecture, Machine Perception and Sensors (CAMPS), pp. 59–64. IEEE (2006)
33. Plyer, A., Le Besnerais, G., Champagnat, F.: Massively parallel Lucas Kanade optical flow for real-time video processing applications. Journal of Real-Time Image Processing (JRTIP) **11**(4), 713–730 (2016)
34. Romera, T., Petreto, A., Lemaitre, F., Bouyer, M., Meunier, Q., Lacassagne, L.: Implementations impact on iterative image processing for embedded GPU. In: 2021 29th European Signal Processing Conference (EUSIPCO), pp. 736–740. IEEE (2021)
35. 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 of Real-Time Image Processing **19**(2), 317–329 (2022)
36. Wedel, A., Pock, T., Zach, C., Bischof, H., Cremers, D.: An improved algorithm for TV-L1 optical flow. In: Statistical and geometrical approaches to visual motion analysis, pp. 23–45. Springer (2009)
37. Werlberger, M., Trobin, W., Pock, T., Wedel, A., Cremers, D., Bischof, H.: Anisotropic Huber-L1 optical flow. In: BMVC, vol. 1, p. 3 (2009)
38. Xue, T., Chen, B., Wu, J., Wei, D., Freeman, W.T.: Video enhancement with task-oriented flow. International Journal of Computer Vision (IJCV) **127**(8), 1106–1125 (2019)
39. Yang, G., Ramanan, D.: Volumetric correspondence networks for optical flow. NeurIPS **5**, 12 (2019)
40. Zach, C., Pock, T., Bischof, H.: A duality based approach for realtime TV-L1 optical flow. In: Proceedings of the 29th DAGM Conference on Pattern Recognition (DAGM GCPR), pp. 214–223 (2007)