Real-time and approximate iterative optical flow implementation on low-power embedded CPUs

Maxime Millet*‡, Adrien Cassagne*, Nicolas Rambaux†, Lionel Lacassagne*
*LIP6, Sorbonne University, CNRS, Paris, France
‡Lerity-ALCEN, Cergy, France
†IMCCE, Observatoire de Paris PSL, Sorbonne University, CNRS, Paris, France

***Abstract*—Optical flow estimation is used in many embedded computer vision applications, and it is known to be computationally intensive. In the literature, many methods exist to estimate optical flow. Thus, the challenge is to find a method that matches the applicative constraints. In an embedded system, a trade-off between power consumption and execution time has to be made to meet both energy and framerate constraints. This work proposes methods to implement an approximate HORN & SCHUNCK optical flow estimation that meets embedded CPUs constraints. This is achieved thanks to architectural optimizations, software optimizations and algorithm tuning. For instance, on the NVIDIA Jetson Nano, and for HD video sequences, the achieved frame latency is 12 ms for 5 Watts. To the best of our knowledge, this is the fastest optical flow implementation on embedded CPUs.**

***Index Terms*—computer vision, optical flow, SIMD, approximate computing, low power, tradeofs, embedded systems.**

I. INTRODUCTION

Real-time optical flow, which is the apparent motion estimation between two consecutive frames in a video sequence, has many applications in embedded systems but with different constraints. Usually, applications require an accurate estimate for quality or safety reasons, such as video denoising or control of autonomous cars and UAV. Both constraints, real-time and accuracy implies high power consumption.

However, sometimes a rough estimate may be sufficient. For instance, it is not necessary to compute the most accurate estimate to identify the moving regions or to detect the fastest ones. In these cases, trades-off between accuracy and speed may be useful to save computational time and power consumption.

Meteor detection is an example of application where optical flow can be approximated. This work is a part of the Meteorix nano-satellite project [1]. In this project, the payload is a camera and a low-power SoC. Thus, one of the challenge is to design a real-time application to detect space debris and extraterrestrial matter that enter the Earth's atmosphere.

The application has to run in real-time, defined as the ability to process at least 25 FPS (frames per second), for two reasons. Firstly, bandwidth with the ground is very limited, it is not possible to send all frames on Earth and process them on it. Secondly, it is not possible to delay the processing due to a low storage space and a small battery capacity. Moreover, only 7 Watts are available for the processing.

Work partially funded by DIM RFSI, DIM ACAV and ESEP.

The Meteorix project includes an implementation of the HORN & SCHUNCK method to approximate the optical flow. This processing takes most of the execution time (95%) and the whole application does not match real-time constraints.

The contribution of this work is to reduce the computational time as well as the power consumption of the optical flow implementation. Three ways are explored and evaluated:

- a combination of architectural and algorithmic optimizations,
- a scheduling method to maximize data reuse within the CPU caches to get a cache-aware implementation,
- a fine tuning of the optical flow algorithm for the targeted application.

The paper is organized as follow. Sec. II discusses the related works. Sec. III presents the HORN & SCHUNCK method and the implementation choices. Sec. IV describes the architectural optimizations and Sec. V explains the software optimizations. Sec. VI evaluates the results of the proposed optimization methods. Finally, Sec. VII concludes.

II. RELATED WORKS

Few space missions for meteor detection are developed at the time of the writing. To the best of our knowledge, only one has been completed. This mission consisted in a camera pointing towards Earth and a PC aboard the International Space Station. However, the processing chain did not work [2] and the video sequences where therefore recorded and sent to Earth to be analyzed manually to find meteors.

In the literature, a CubeSat project for meteor detection is in development [3]. This project shares the same constraints as us. The first results show that optical flow is also the critical step of the processing chain, requiring 2000 ms on CPU or 380 ms on FPGA while the total execution time is 2450 ms on CPU and 810 ms on FPGA. By accelerating some other parts, the authors expect to reach 0.5 FPS which is still far from real-time ($\geq$ 25 FPS).

Recent works present real-time implementation of HORN & SCHUNCK and other more recent optical flow algorithms on GPU [4] and on FPGA [5], but not on CPU. Indeed, these are more often used as a comparison reference for other architectures and the optimizations details are not described [6].

DOI 10.1109/ASAP57973.2023.00032

III. Horn & Schunck Algorithm

Horn & Schunck [7] is one a the first methods to estimate dense optical flow, giving a motion vector $\vec{w} = (u, v)$ per pixel. It is a good candidate for low-power embedded system and its quality is enough for the studied application which requires a dense estimation.

The initial step is the computation of the first derivatives I_x, I_y and I_t, between consecutive frames, followed by two iterative steps:

1) Average speed estimation $(\bar{u}, \bar{v})$ with a 3×3 kernel:

$$A = \frac{1}{12}\begin{bmatrix}1 & 2 & 1\\ 2 & 0 & 2\\ 1 & 2 & 1\end{bmatrix}, \bar{u}^i = A * u^{i-1}, \bar{v}^i = A * v^{i-1}, \quad (1)$$

2) Updating the flow of each pixel:

$$\begin{aligned} u^{i+1} &= \bar{u}^i - I_x \times \frac{I_x\bar{u}^i + I_y\bar{v}^i + I_t}{\alpha^2 + {I_x}^2 + {I_y}^2}, \\ v^{i+1} &= \bar{v}^i - I_y \times \frac{I_x\bar{u}^i + I_y\bar{v}^i + I_t}{\alpha^2 + {I_x}^2 + {I_y}^2}. \end{aligned} \quad (2)$$

with i the i-th iteration, α a smoothing regularization parameter and $*$ the convolution operator.

This iterative nature allows some trade-offs between quality of the estimation and the execution time. Moreover, its regular computational scheme makes it well suited for optimizations and parallelization.

A. Coarse-to-fine Estimation

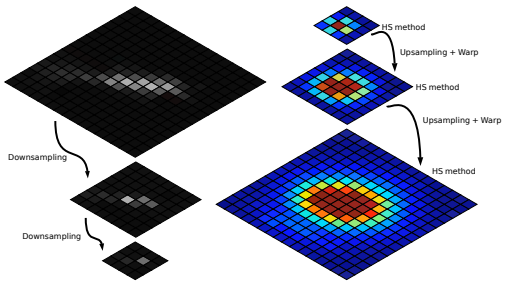

Fig. 1. Coarse-to-fine strategy of Optical Flow (OF) starts by downsampling consecutive images, then OF is estimated from coarsest to finest scales.

In the previous section, only motions smaller than 1 pixel can be well estimated (we call this method *mono-scale*). A *coarse-to-fine* estimation has to be used to go further, where an estimate is computed on each scale, propagated to the next fine one and then refined (see Fig. 1). The number of scales depends on the desired motion range. In this work, the number of scales is fixed to 3 and each scale is half of the previous one. This configuration allows to estimate motions up to ± 7 pixels in both direction. This is adapted to detect meteors entering in Earth's atmosphere (for 25 FPS cameras).

IV. Architectural Optimizations

A. Instructions and Threads Parallelism

Modern architectures have Single Instruction Multiple Data (SIMD) Instruction Set Architecture (ISA) extensions allowing the processing at the same time of several data packed within a register. The number of elements depends on the data type and the length of the register. In this work, we consider NEON extension as it is well spread in embedded SoCs and 32-bit floating-point numbers. NEON offers 128-bit registers that can contain four 32-bit floating-point numbers (= 4×32-bit).

Today, compilers are able to vectorize some code patterns but not all. For instance, on GCC v12.2.0, the stencil in Eq. (1) is not vectorized. Writing vectorized code is a tedious task requiring knowledge about the target architectures. However, one of the limitation of writing SIMD code is the portability. Indeed, generally, SIMD codes are written with intrinsic functions that are specific to one ISA. In this work, a minimal wrapper for intrinsic functions has been implemented (with only the necessary instructions) to guarantee code portability.

Moreover, to take advantage of multi-core CPUs, the image is divided in horizontal blocks that are distributed among the cores with OpenMP. Threads are synchronized at each scale.

B. Avoiding Division and Reciprocal Estimation

On all CPU architectures, the division has a low throughput and and a high latency. But it can be replaced by a reciprocal estimation (in half precision). The two divisions in Eq. (2) are replaced by one reciprocal used twice.

V. Software Algorithmic Optimizations

High Level Transforms [8] aka algorithmic optimizations have a huge impact of stencil performance: A) to reduce stencil complexity and amount of memory accesses – B) to enhance cache performance by pipelining several stencil iterations.

A. Code Optimizations to Reduce Stencil Complexity

1) Scalarization and Register Rotation: As there is an overlay of the stencil from one iteration to the other one, each pixel is used for 3 successive computations. Memorising them into registers reduces the number of loads from 25 to 9.

2) Loop Unrolling: An unrolling order equal to the stencil size (here 3) along a line (the *inner* loop) leads to an *optimal* loop without extra load and removes all register-to-register copies due to register rotation.

3) Column-Wise Reduction: The 2D average stencil can be rewrited into a separable form of two 1D stencils: $A = [1 \quad 2 \quad 1]^T * [1 \quad 2 \quad 1] - 4c$ where c is the centered value. Thus the computation of the vertical stencil is *reduced* (later named *red*) into one value (memorized in a register) and used three times, reducing both stencil complexity and loads. Unrolling and reduction can be combined like `ilu3red` (see Sec. VI).

B. Pipelining Stencil Iterations

Processing the image may be done in many ways (embarrassingly parallel problem) especially since it is an iterative stencil. The "common" way (named *direct*) consists in applying the stencil to the whole image before applying a second iteration of the stencil (Fig. 2, left). The pipelined way consists in applying a second iteration of the stencil as soon as there is enough processed data (Fig. 2, right). This results is a cache-aware strategy that maximizes the persistence of data in caches and minimizes cache misses.

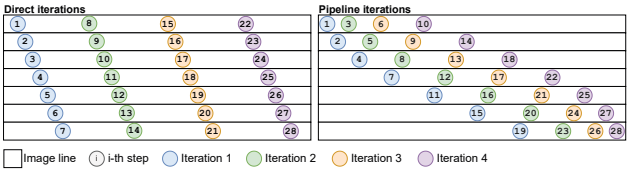

Fig. 2. Direct scheme vs pipelined scheme for a 7-line image and for 4 iterations.

VI. Benchmarks Results and Analysis

A. Testbed and Experiment Conditions

Three boards (see Tab. I) are used to benchmark the implementations. The Nvidia Jetson Nano (later named *Jetson*) is a good candidate as this board has already been sent into space on some nano-satellite projects [9]. The Raspberry Pi 4 (later named *RPi4*) has a big user community and also used in meteor detection projects [10]. The Apple M1 SoC (later named *M1*) is a heterogeneous CPU composed of 4 efficiency cores (E-cores) and 4 performance cores (P-cores). In this benchmark, only the E-cores are of interest, as the P-cores are too energy consuming.

Jetson runs on Linux (v4.9 kernel), RPi4 runs on Linux (v5.4 kernel) and M1 runs on macOS Monterey (v12.4 kernel). For each board, the GCC v12.2.0 compiler has been used with the following optimization flags: `-march=native -funroll-loops -fstrict-aliasing -O3`.

B. Impact of Software Algorithmic Optimizations

The impact of algorithmic optimizations is showed in Fig. 3. All versions are multi-threaded, SIMDized and the board frequency was set to the maximum. The *basic* version is a reference version without optimization.

The x-axis is selected to focus on the cache overflow. The left side shows that unrolling and reduction decrease the execution time, but do not shift the cache overflow. On the contrary, the right side shows that pipeline iterations, applied to the best version, shifts the cache overflow. A pyramid of 3 levels for image size equivalent to HD (in pixel count) gives $n = 960$, $n = 480$ and $n = 240$. The speedup between the *basic* version and the best pipelined version for the complete pyramid is $\times 1.8$ on Jetson (1.8, 1.7 and 1.5 for each level), $\times 3.1$ on RPi4 (3.0, 3.9, 2.7) and $\times 1.8$ on M1 (1.8, 2.0, 2.2).

C. Power Consumption Analysis

Fig. 4 shows the measured power consumption of each board for various frequencies and number of threads for the most accurate tested configuration. The reported power is the average power over the whole run.

TABLE I
Tested boards characteristics.

			Proc.	Freq.	Cache (KB)		RAM	
Name	Year	Cores	(nm)	(GHz)	L1	L2	Size	Bandwidth
Jetson	2019	4 × ARM Cortex A57	20	1.49	32	2048	4 GB	7.3 GB/s
RPi4	2019	4 × ARM Cortex A72	28	1.50	32	1024	8 GB	3.4 GB/s
M1	2020	4 × Icestorm (E-core)	5	2.10	64	4096	8 GB	32.4 GB/s

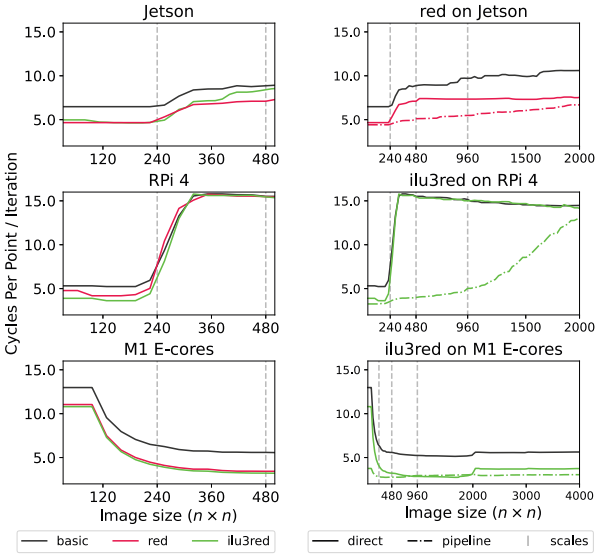

Fig. 3. Impact of algorithmic optimizations (left) and pipeline (right). Horn & Schunck mono-scale implementation. The dashed vertical lines represent the 3 pyramid scales for HD equivalent images.

For Jetson and RPi4, a device between the power supply and the board measures at 5 KHz the voltage and current, giving the power. Thus, the reported power is the power of the entire board.

For M1, the reported power is the power of the CPU only (the Apple `powermetrics` tool is used). As a result, all boards are always under the project constraint of 7 Watts.

D. Algorithm and Hardware Tuning

Instead of having the same number of iterations for all scales, a usual tuning for a better convergence (and flow estimation) is to have more iterations on coarse levels and few on grain levels. For instance, for a 3-level pyramid, we can have (16, 8, 4) iterations. To go further, since the finest level can only estimate small motions, we propose to have no iteration on the finest level. Thus, the previous example becomes (16, 8, 0). This tuning allows to save at least half of the computation time in exchange for a coarser final estimate. However, the meteors detection rate remains unchanged.

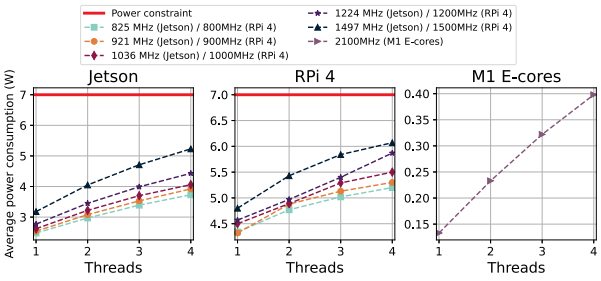

Fig. 4. Impact of frequency and thread number on power consumption. Horn & Schunck coarse-to-fine implementation with HD images (1280px × 720px) and (32, 16, 0) iteration scheme.

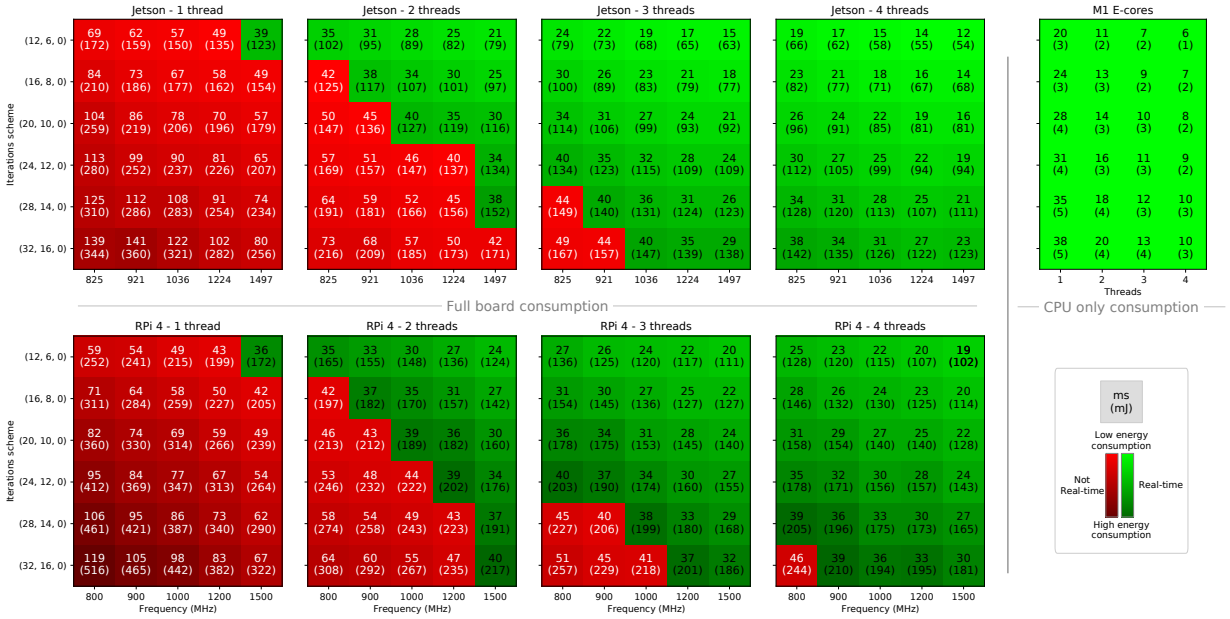

Fig. 5. Computation time (in ms) and energy consumption (in mJ) for HD images (1280 × 720 pixels). HORN & SCHUNCK coarse-to-fine implementation.

In a previous work [11], all the proposed combinations have been validated on the only available dataset [12]. All the proposed configurations have a similar detection rate around 95% (± 2%).

In Sec. VI-B, the exploration helped us select the fastest mono-scale combination of optimizations depending on the architecture. In this section, only the fastest implementation is considered. For each board the goal is to achieve a real-time coarse-to-fine computation on them. Fig. 5 summarizes the time and energy consumed to process a HD image depending on several configurations in term of frequency, number of threads and iteration scheme.

On Jetson and RPi4, most of the tested configurations are real-time. The latter with its more recent cores is faster in single-thread. However, the trend is reversed in multi-threads and Jetson is always faster with at least 3 threads. This is explained by a higher memory bandwidth on Jetson. Indeed, as shown in Tab. I, the memory bandwidth of RPi4 is 3.4 GB/s while it is 7.3 GB/s on Jetson, more than double. When the number of threads increases, the pressure on the global memory increases. Thus, the Jetson design is better suited to optical flow estimation than the RPi4.

Fig. 5 shows that the higher the frequency or the number of threads, the lower the energy consumption. The best hardware tuning is the one using 4 threads running at the highest frequency. For the biggest iteration scheme (32 16 0), the optical flow latency is 23 ms on Jetson, 30 ms on RPi4 and 10 ms on M1.

On M1, real-time is reached in all cases. This demonstrates the efficiency of this SoC. This makes the latest Apple iPads interesting for real-time and low-power computer vision applications. However, since it is not possible to put a tablet on board a nano-satellite, the focus for our project is on Jetson and RPi4.

VII. CONCLUSION

In this work, we evaluated the impact of algorithmic optimizations on iterative stencils like those of optical flow and showed that a fine tuning of these algorithms for the target application leads to real-time implementation while enforcing power constraint. As far as we know, this is the most efficient optical flow implementation on embedded CPUs.

REFERENCES

[1] N. Rambaux *et al.*, "Meteorix: A cubesat mission dedicated to the detection of meteors and space debris," in *ESA NEO and Debris Detection Conference*, 2019.

[2] T. Arai *et al.*, "On-going status of meteor project onboard the international space station," in *LPSC*, 2018.

[3] J. Petri *et al.*, "Hardware accelerated onboard image processing for space-based meteor observation - concept and implementation of spacemedal," 2022.

[4] M. Seznec *et al.*, "Real-time optical flow processing on embedded GPU: an hardware-aware algorithm to implementation strategy," *Elsevier JRTIP*, vol. 19, no. 2, pp. 317–329, 2022.

[5] I. Bournias *et al.*, "FPGA acceleration of the horn and schunck hierarchical algorithm," in *ISCAS*. IEEE, 2021.

[6] A. Petreto *et al.*, "Energy and execution time comparison of optical flow algorithms on SIMD and GPU architectures," in *IEEE DASIP*, 2018.

[7] B. Horn and B. Schunck, "Determining optical flow," *Artificial Intelligence*, vol. 17, pp. 185–203, 1981.

[8] L. Lacassagne *et al.*, "High level transforms for SIMD and low-level computer vision algorithms," in *WPMVP*. ACM, 2014.

[9] "Lockheed martin and university of southern california build smart cubesats, la jument," https://news.lockheedmartin.com/news-releases?item=128962, accessed: 2023-02-05.

[10] J. Vaubaillon *et al.*, "The "mobile observation of meteor" (MoMET) device," *Experimental Astronomy*, vol. 54, no. 1, pp. 1–22, 2022.

[11] M. Millet *et al.*, "Meteorix - a new processing chain for real-time detection and tracking of meteors from space," *IMO WGN*, vol. 49, no. 6, pp. 1–5, 2022.

[12] T. Arai *et al.*, "International space station-based meteor observation project: Initial results," in *LPSC*, 2017.