Development of efficient image processing applications

Thomas Kœhler

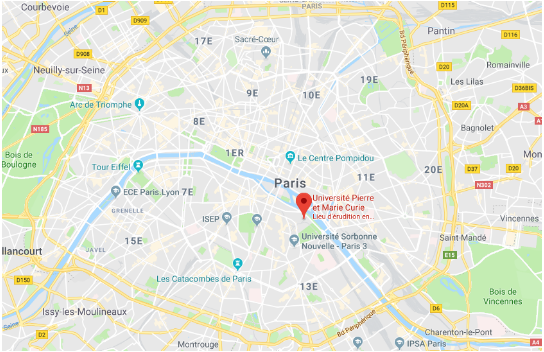
supervised by Michel Steuwer and Phil Trinder

intra-systems seminar — january 2019

Background

Master of Software Science and Technology

2016–2018, Sorbonne Université, Paris, France



Background

Internships

- Optical flow computing optimisation on GPU

  2017, 2 months, Laboratoire d'Informatique de Paris 6, France

  supervisor: Lionel Lacassagne

- Efficient object tracking on heterogeneous and parallel architectures

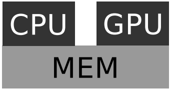
  2018, 6 months, Laboratoire d'Informatique de Paris 6, France

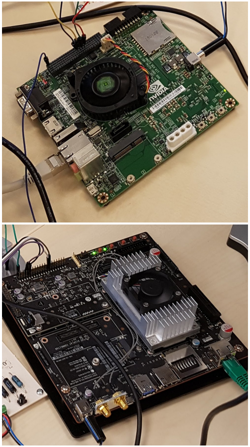
  supervisors: Lionel Lacassagne, Emmanuel Chailloux

Embedded Hardware

NVIDIA Jetson boards ~5-15W

Tegra K1, X1, X2 *(ARM CPU + NVIDIA GPU)*


CPU GPU
MEM



Power consumption comparison

Lenovo A485 ~*10-50W*

Intel i7 3370 ~*80W-140W*

GeForce GTX 970 ~*145W*

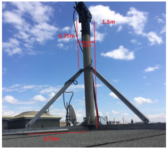
Optical Flow: application

The Meteorix project

- a nanosatellite to observe meteors from space
- limited resources, constrained environment
- communication device on an university tower
  - low satellite ↔ ground bandwidth
  - real-time meteor detection: ~ 30 fps
  - power consumption: ~ 1 Watt

CubeSat structure ©NASA

Meteor in space ©NASA



On the university roof

Optical Flow: application

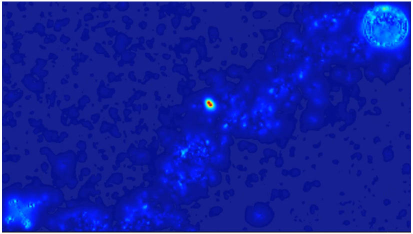
Processing pipeline example

- a meteor observed from the International Space Station

  provided by the METEOR project of the Chiba Institute of Technology

Optical Flow

- apparent motion of the pixels between two images



Optical Flow: objectives

My Objectives

- GPU implementation and optimisation *handwritten CUDA*
- can we respect the constraints? $\sim 30\ fps/W$

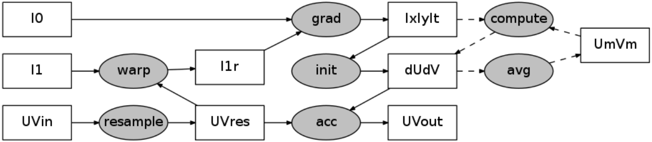
Why GPUs?

- compare to CPUs and FPGAs
- promising power efficiency
- fitting for image processing

Optical Flow: algorithm

Pyramidal Horn Schunck algorithm

- iterative computation of the optical flow in each pixel
- compute a coarse flow on a small resolution
- progressively scale up to bigger resolutions and refine



Processing a bigger resolution

Optical Flow: implementation

First results

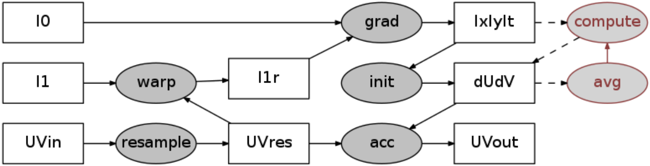
- Tegra X1, 1280×720 pixels, 4 levels, 6 iterations
- power consumption estimate: $\sim 10 Watts$

precision	time (ms)	fps	$fps/Watt$
f32 (single)	33.92	29.5	2.95
f16 (half)	20.88	47.9	4.79

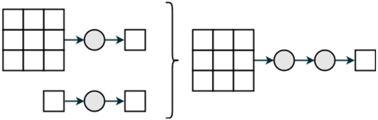
Optimisation

- focus on improving execution times
- lower the GPU frequency to reduce consumption

Optical Flow: optimisation



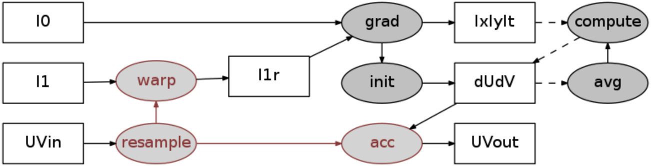
Fusing the average and the local flow computation



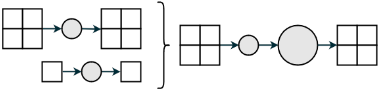
Producer-consumer model

precision	original time	fused time	local speedup
f32	2.42 ms	1.39 ms	×1.74
f16	1.42 ms	0.95 ms	×1.49

Optical Flow: optimisation



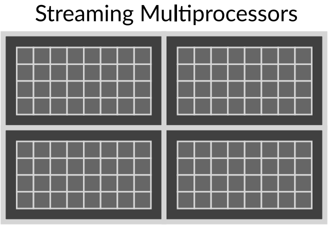
On-the-fly upscaling

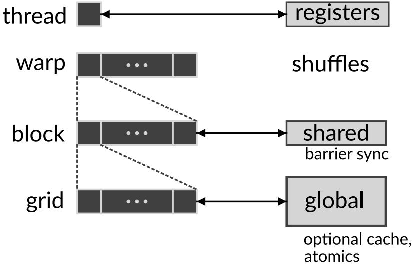


Producer-consumer model

precision	original time	fused time	local speedup
f32	2.75 ms	1.79 ms	×1.53
f16	1.67 ms	1.16 ms	×1.44

Optical Flow: optimisation





- *Single Instruction Multiple Threads*: minimize divergence
- memory transfers: aligned, coalesced, locality
- cover latencies with *Instruction Level Parallelism*
- tune block sizes

Optical Flow: results

Overall improvement: $\times 5.5$

precision	time (ms)	fps	speedup	fps/W
f32	$33.92 \rightarrow 9.90$	101	×3.4	10
f16	$20.88 \rightarrow 6.18$	162	×3.4	16

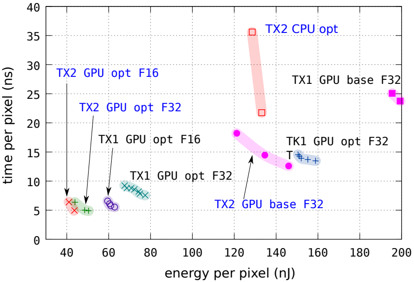
Constraints still not respected

- not the complete processing pipeline
- not in space, not radiation-hardened
- review the objectives, performance and quality constraints

Experimental Measures

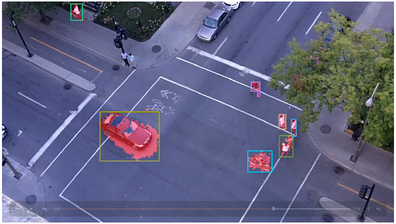
- comparing CPU and GPU efficiency

  non-pyramidal Horn-Schunck algorithm, embedded boards
- published to COMPAS *(French)* and DASIP
- Tegra X2 GPU: 4× faster 3× less energy



Pareto fronts for the best processors and versions, one point per frequency

Object Tracking: problematic



Deploying object tracking by *covariance matching*

- focus on embedded systems with integrated CPU/GPU
- real time, low energy consumption, good quality
- use the hardware efficiently
- development tools and abstractions

Object Tracking: algorithm

Covariance Matrix

- describe an object using various features
- compact, discriminant and robust

- a set of features, F
- a sample region
- statistical measure of the relations within F
- similarity metric between matrices

Tune features and algorithm specifically for the application

Object Tracking: algorithm

Object tracking from a motionless camera

selected processing pipeline

- movement detection

  ΣΔ *Sigma-Delta*
- noise filtering

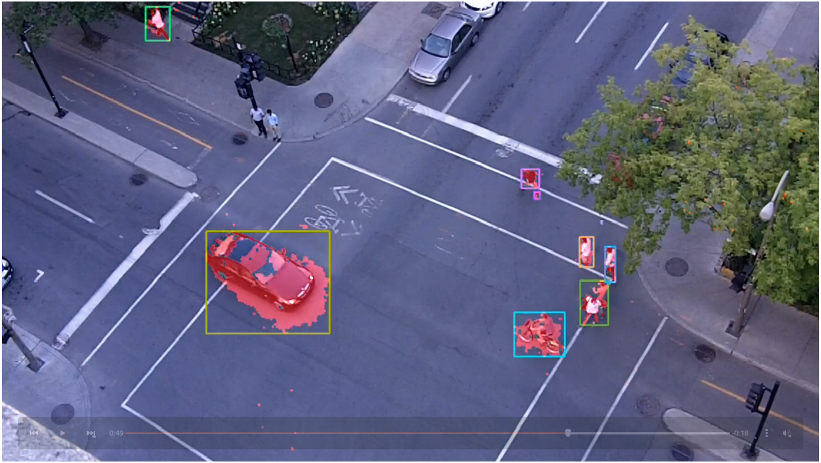
  mathematical morphology
- connected components and bounding boxes

  Light Speed Labeling
- description through covariance matrices

  position, intensity, texture → *7 features*
- tracking by greedy matching between frames

  Jensen-Bregman LogDet divergence

Object Tracking: algorithm

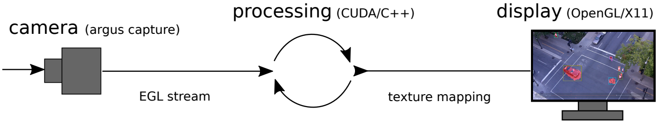


Processing the video *stmarc* from the Urban Tracker dataset

Object Tracking: low-level implementation

Tegra X2

Camera and display integration



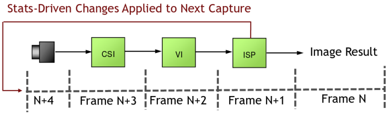
camera (argus capture)

processing (CUDA/C++)

display (OpenGL/X11)

EGL stream

texture mapping



fill the camera pipeline

maximise throughput despite latency

integration code: ~1 200 C++ lines

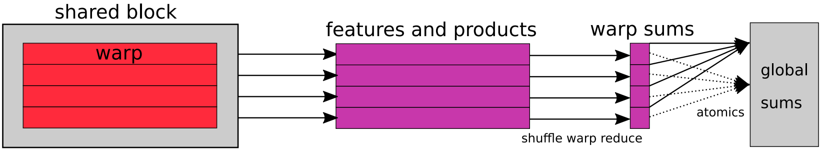
Object Tracking: low-level implementation

Computing covariance matrices

need to compute means, can use sums:

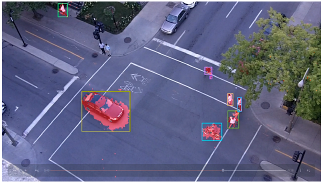
$$\operatorname{cov}(X, Y) = \mathrm{E}[(X - \mathrm{E}[X])(Y - \mathrm{E}[Y])] = \mathrm{E}[XY] - \mathrm{E}[X]\,\mathrm{E}[Y]$$

generating GPU code from elementary stencils:



Object Tracking: low-level implementation

Results



	average	range
labels	10.0	0 – 51
time (*ms*)	8.97	6.09 – 47.43
fps	111	21 – 166

Measures on the video (1280x720 pixels)

Issues

- maintain real-time processing *adapt quality to quantity, establish priorities*
- algorithmic alternatives to explore *conditional* $\Sigma\Delta$, *optical flow*
- more optimisations to explore
- not portable

Object Tracking: efficient abstractions

Simplify development to enable improvements

- Adapt *application prototyping and flexibility*
- Optimise *efficiency, explore the implementation space*
- Port *execute on multiple platforms*
- Verify *correct behavior, hold some properties*

C/CUDA/OpenCL are not suited, lack of productivity

- not portable, verbose, time consuming and error-prone
- not composable without performance loss *manual operator fusing*
- troublesome semantics, compiler lacks freedom and knowledge

Object Tracking: efficient abstractions

My rough idea

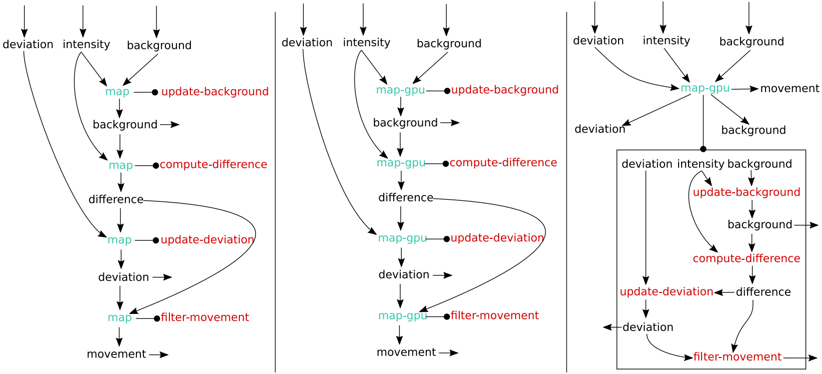
- high-level and low-level pattern graphs combining elementary functions

Inspiration

- algorithmic skeletons *map, zip, reduce*
- Structured Parallel Programming *Intel TBB, Cilk Plus, Intel ArBB*
- existing graphs and dataflow representations *TensorFlow, OpenVX*

Object Tracking: efficient abstractions

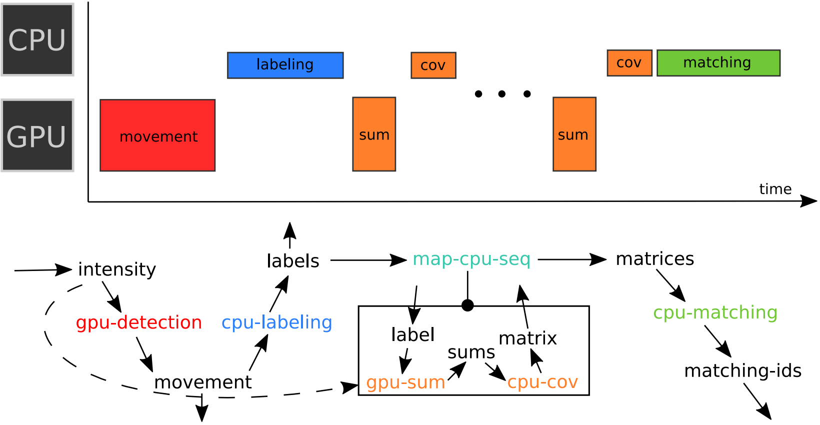
Example: ΣΔ movement detection



Object Tracking: efficient abstractions

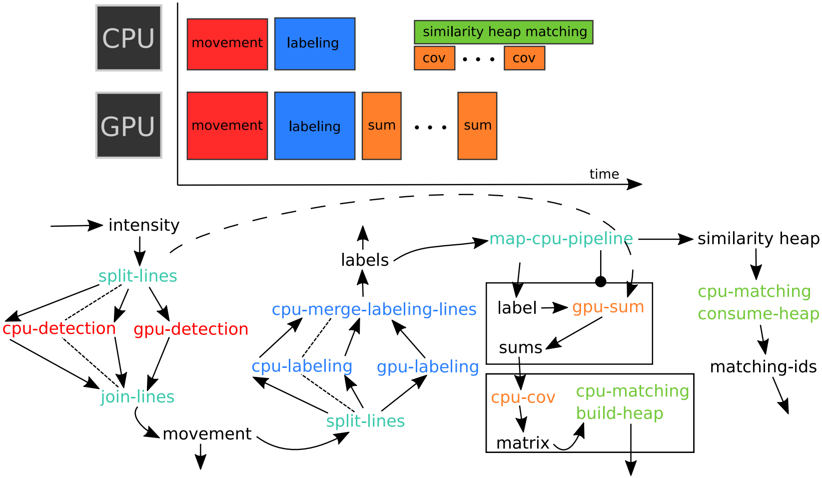
Complete system scheduling

same idea, predictable, low execution overhead



Object Tracking: efficient abstractions

Complete system scheduling



Object Tracking: efficient abstractions

Other approaches

- StarPU *runtime system for heterogeneous architectures*
  + task graph, data-aware scheduling
  - runtime overhead, requires specialised task implementations

- Halide *fast image processing, embedded in C++*
  + separate algorithm and scheduling, used in production
  - domain specific, mostly manual scheduling
  implicit internals, hard to troubleshoot

- LIFT *high-level fonctional language, rewrite system*
  + explicit rewrite rules and low-level expressions
  - not yet practical to use, rewriting takes time and setup

PhD Research

- simplify development of applications *adapt, optimise, port, verify*
- Lift approach elegant and promising
- not yet practical to use
- PhD offer on the Lift website

Practical development of efficient and portable image processing applications in Lift

Study applications to find limitations

blur, corner detection, optical flow, object tracking, etc

Research Questions

Focus

- How do we design a faster, more autonomous rewriting system?
  - How do we design a performance model serving this goal?
  - How beneficial is hardware and domain knowledge?

But also

- How should we integrate with non-processing tasks?
- How will we deploy the implementations on the architecture?
- How is the development workflow affected?

Thanks!

Thomas Kœhler
🌍 bastacyclop.gitlab.io
✉ t.koehler.1@research.gla.ac.uk

🌍 lift-project.org