

Methodology for mapping image processing algorithms on massively parallel processors

An NVIDIA GPU specific approach

Florian Guin Corinne Ancourt
firstname.name@mines-paristech.fr

MINES ParisTech – PSL Research University, Paris
Centre de Recherche en Informatique

22/06/2017

French community of compilation – 12th meeting – Saint Germain au Mont d'Or

Image processing domain

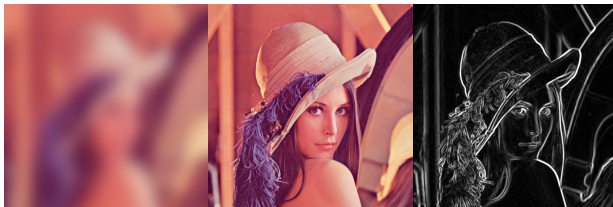
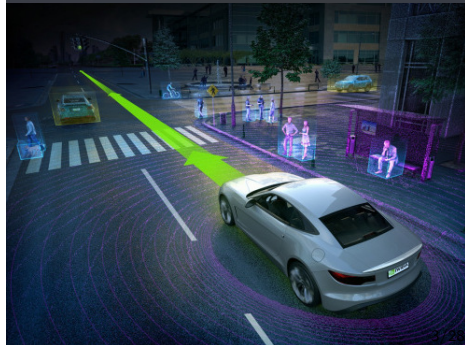
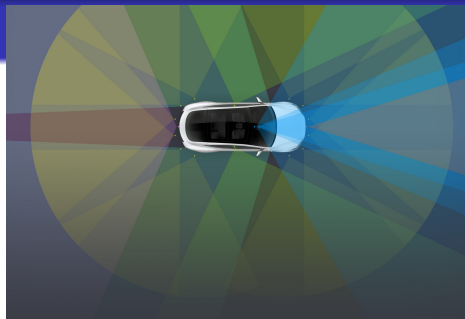


Figure: Image processing examples

Image processing domain

General tendencies for today and tomorrow:

- Data source volume is **growing exponentially**
- Data sources tend to be **multiplied**
- Available computing time tends to be **shorter** for real time processing
- Image processing algorithms are even **more complex**



Architectural evolution

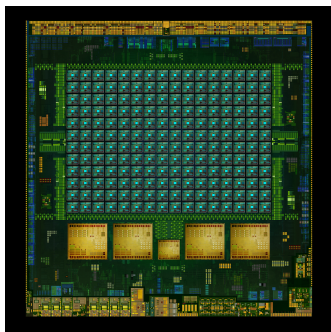


Figure: NVIDIA Kepler processor – 192 cores architecture

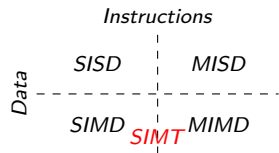
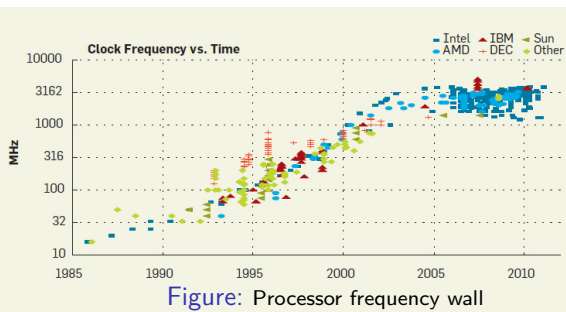


Figure: Flynn's taxonomy

Why do we need a methodology?

Parallel thinking is not trivial. The following methodology has been elaborated to provide:

- an assistance for GPU developers,
- an improvement of software production for industries,
- a support for other domain engineers,
- an assistance to optimise software for a specific GPU architecture.

Tools and compilers results can be limited in some cases:

- dynamic control code
- intensive function calls
- pointers arithmetic
- object oriented languages
- ...

Content

1 Application case

Content

1 Application case

2 Mapping methodology

Content

- 1 Application case
- 2 Mapping methodology
- 3 Experiments

Content

- 1 Application case
- 2 Mapping methodology
- 3 Experiments
- 4 Conclusion

Optical Flow: definition

Principle:

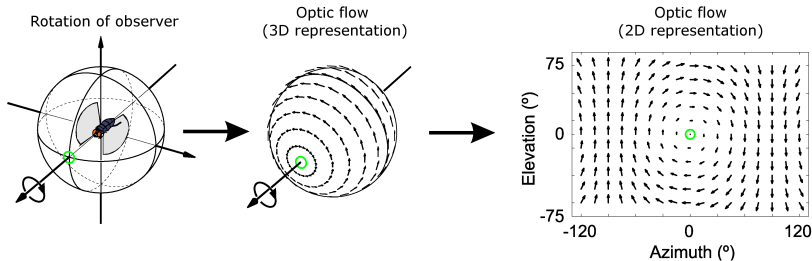
Motion quantification of each pixel taken from two distinct pictures.

Image processing application:

- spatial characterization
- temporal characterization

Examples of applications:

- Motion estimation
- Image stabilization
- Image segmentation
- Moving object tracking
- SLAM algorithms
- ...



Optical Flow: industrial application example



Figure: Example of motion flow analysis. Tesla Motor Company automatic drive.

Algorithm data

The SimpleFlow¹ algorithm is available in the OpenCV extensions.

- Approximately 600 lines of code
- Sequential algorithm
- Dynamic control code
- Approximative runtime for a couple of 2 million pixels images:
 - **200s** on a NVIDIA Jetson TX1
 - ARM Cortex A57(1.9GHz) + A53(1.3GHz)
 - **50s** on a desktop computer
 - Intel Core I7 4770S (8 logical cores at 3.1GHz)
 - Ideal runtime: **40ms**
- Language and library: C++ with the OpenCV library

¹Michael W. Tao et al. "SimpleFlow: A Non-iterative, Sublinear Optical Flow Algorithm". In: *Computer Graphics Forum (Eurographics 2012)* 31.2 (May 2012). URL: <http://graphics.berkeley.edu/papers/Tao-SAN-2012-05/>.

Simplified CallGraph

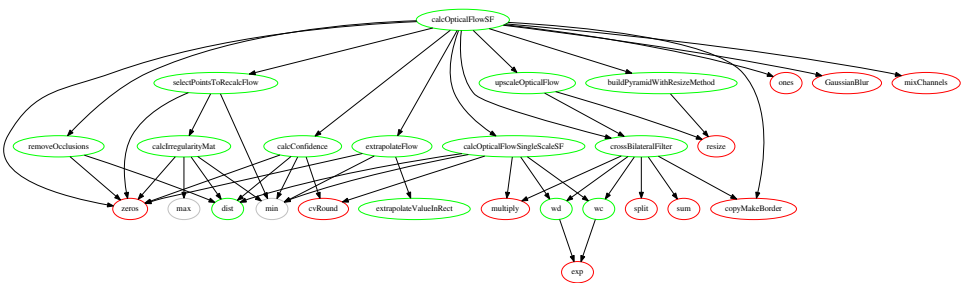


Figure: Simplified call graph. function is simpleflow one, function is openCV one and function comes from the C++ std library

Application example



Figure: Image 1 (t)



Figure: Image 2 ($t + \delta$)



Figure: X coordinate pixel motions

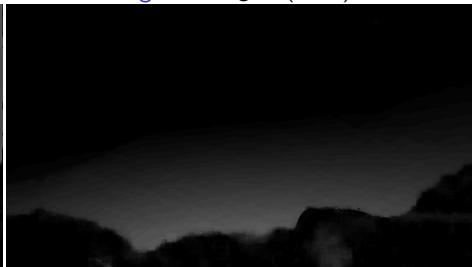
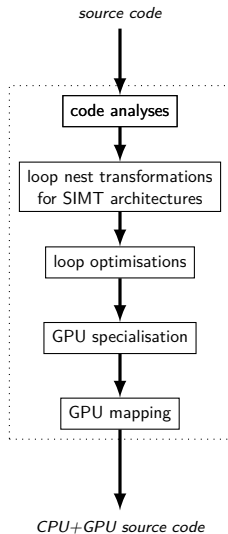
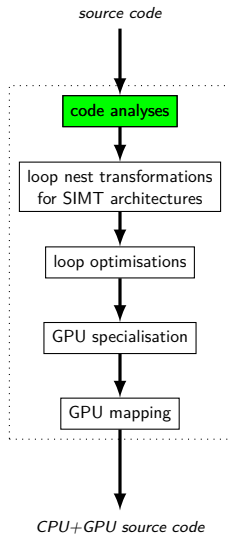
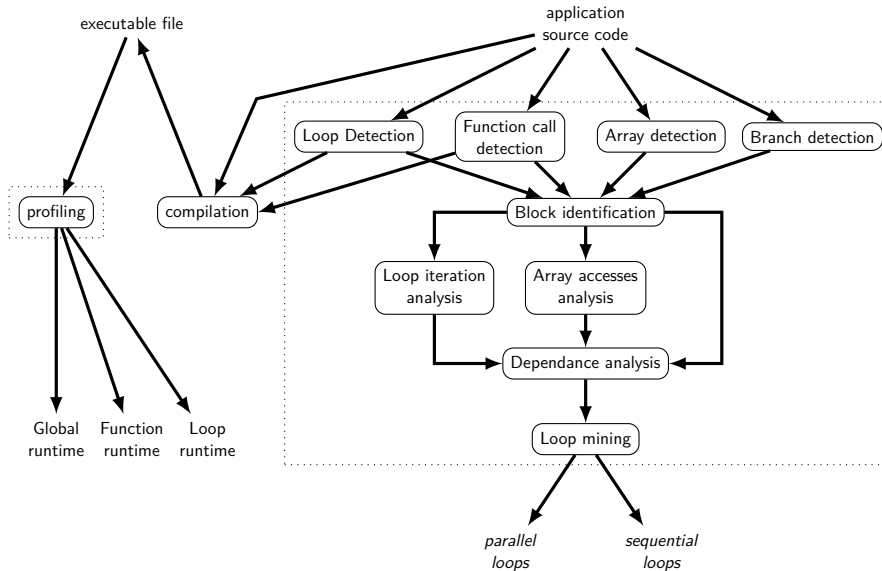


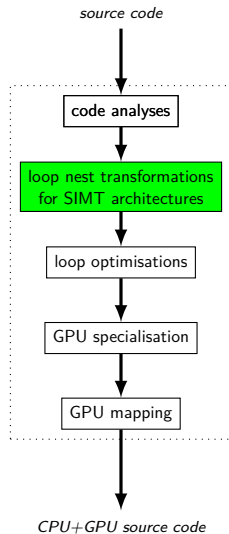
Figure: Y coordinate pixel motions

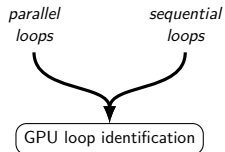




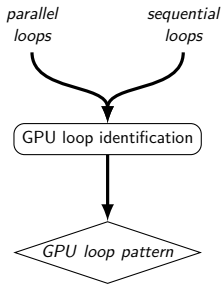
Code analyses



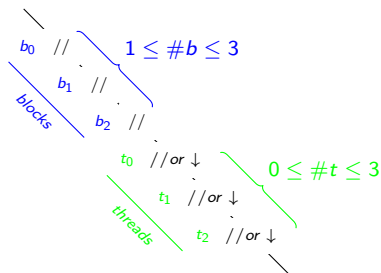




loop nest transformations for SIMT architectures



GPU loop pattern



loop nest transformations for SIMT architectures

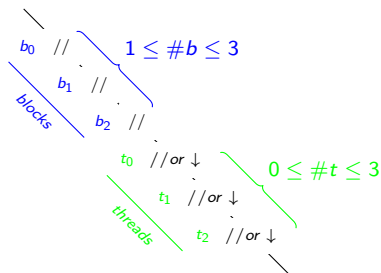
parallel loops sequential loops

GPU loop identification

GPU loop pattern

GPU loop size

GPU loop pattern

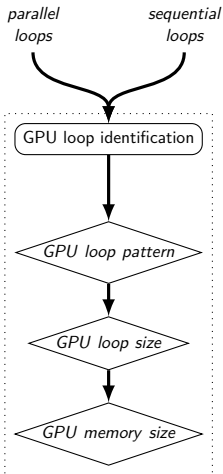


GPU loop size

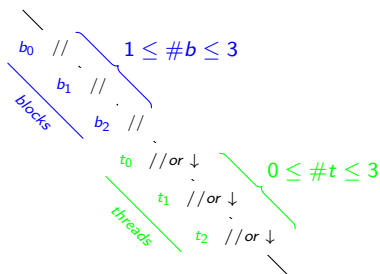
$$\left\{ \begin{array}{l} b = b_0 \times b_1 \times b_2 \\ b_0 < 2147483647 \\ b_1 < 65535 \\ b_2 < 65535 \\ b \gg t \end{array} \right.$$

$$\left\{ \begin{array}{l} t = t_0 \times t_1 \times t_2 \\ t < 1024 \\ t_0 < 1024 \\ t_1 < 1024 \\ t_2 < 64 \\ t \% 32 = 0 \\ t > 4 \times 32 \end{array} \right.$$

loop nest transformations for SIMT architectures



GPU loop pattern



GPU loop size

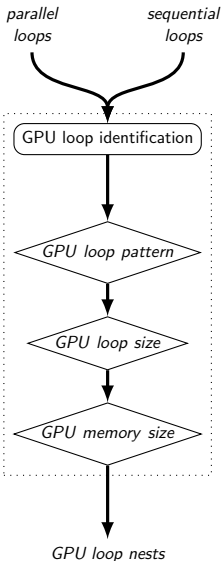
$$\left\{ \begin{array}{l} b = b_0 \times b_1 \times b_2 \\ b_0 < 2147483647 \\ b_1 < 65535 \\ b_2 < 65535 \\ b \gg t \end{array} \right.$$

$$\left\{ \begin{array}{l} t = t_0 \times t_1 \times t_2 \\ t < 1024 \\ t_0 < 1024 \\ t_1 < 1024 \\ t_2 < 64 \\ t \% 32 = 0 \\ t > 4 \times 32 \end{array} \right.$$

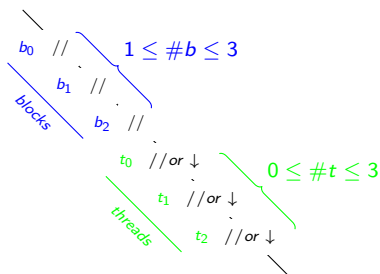
GPU memory size

$$\text{Global memory}_{\text{footprint}} < \text{GPU}_{\text{memory}}$$

loop nest transformations for SIMT architectures



GPU loop pattern



GPU loop size

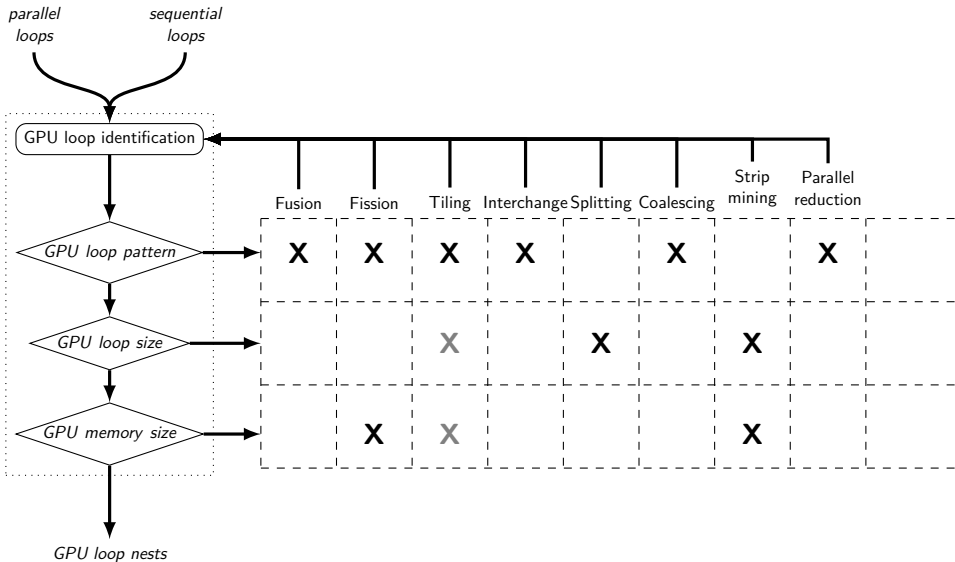
$$\left\{ \begin{array}{l} b = b_0 \times b_1 \times b_2 \\ b_0 < 2147483647 \\ b_1 < 65535 \\ b_2 < 65535 \\ b \gg t \end{array} \right.$$

$$\left\{ \begin{array}{l} t = t_0 \times t_1 \times t_2 \\ t < 1024 \\ t_0 < 1024 \\ t_1 < 1024 \\ t_2 < 64 \\ t \% 32 = 0 \\ t > 4 \times 32 \end{array} \right.$$

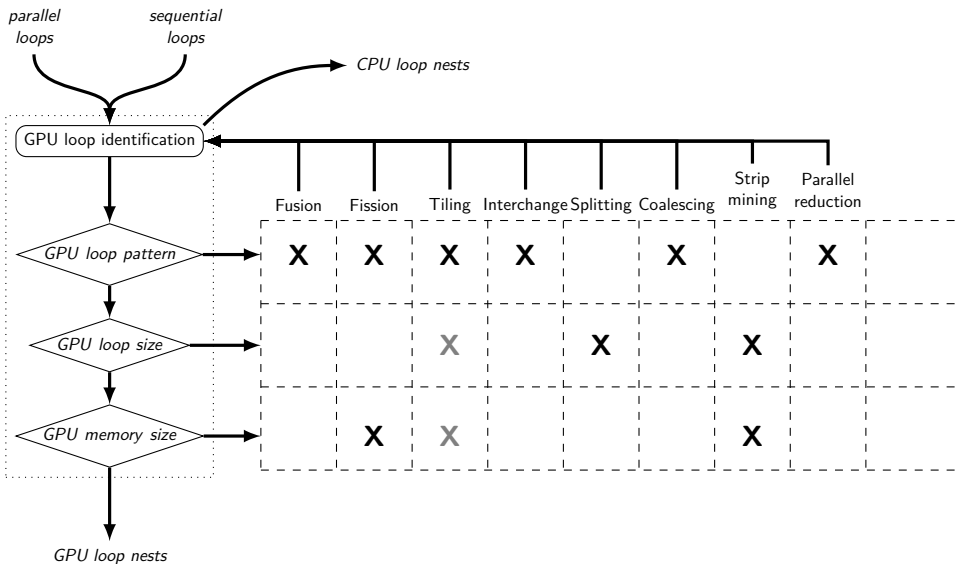
GPU memory size

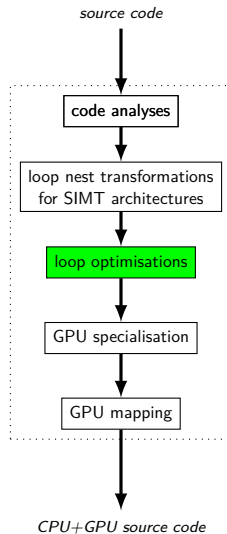
$$\text{Global memory}_{\text{footprint}} < \text{GPU}_{\text{memory}}$$

loop nest transformations for SIMT architectures

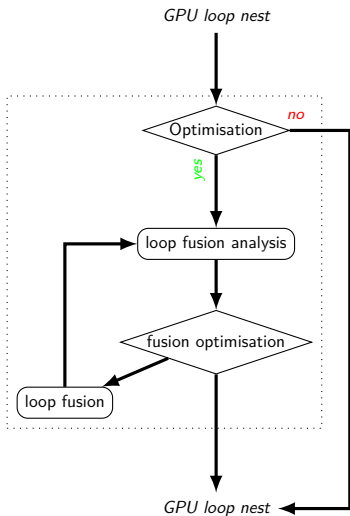


loop nest transformations for SIMT architectures

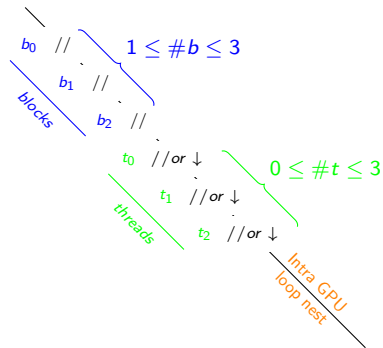




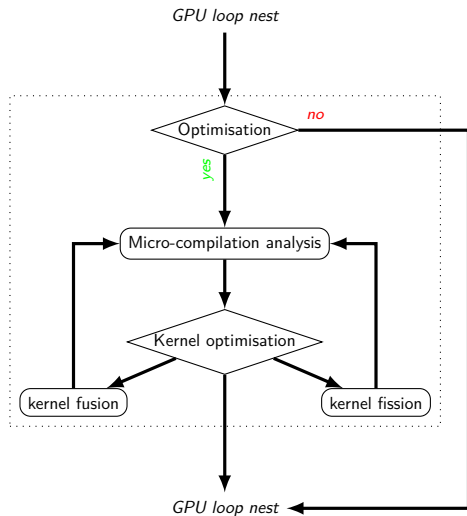
Intra GPU loop nest optimisation



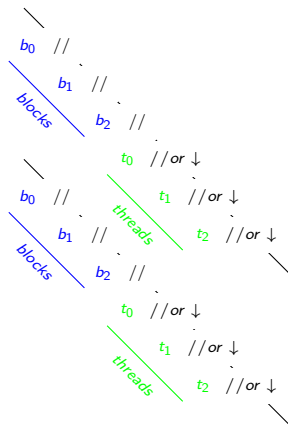
GPU loop pattern



Inter GPU loop nest optimisation

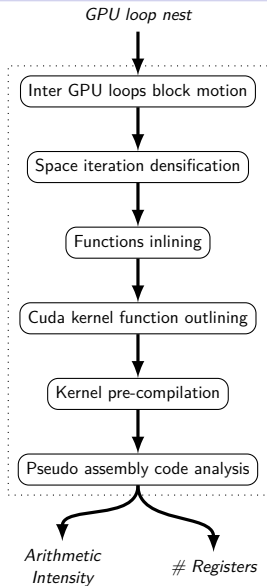
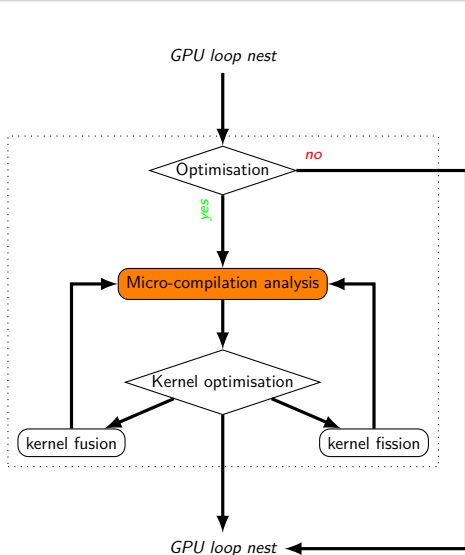


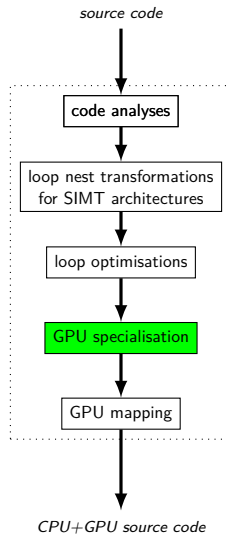
GPU loop pattern

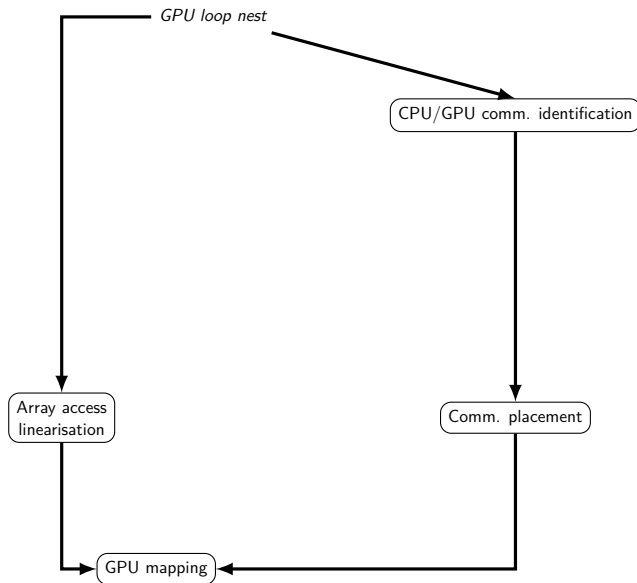


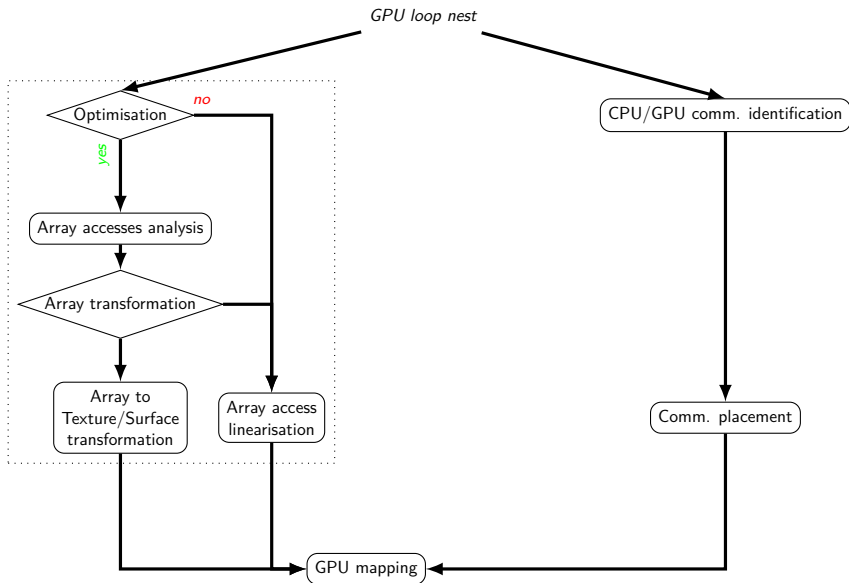
Inter GPU loop nests

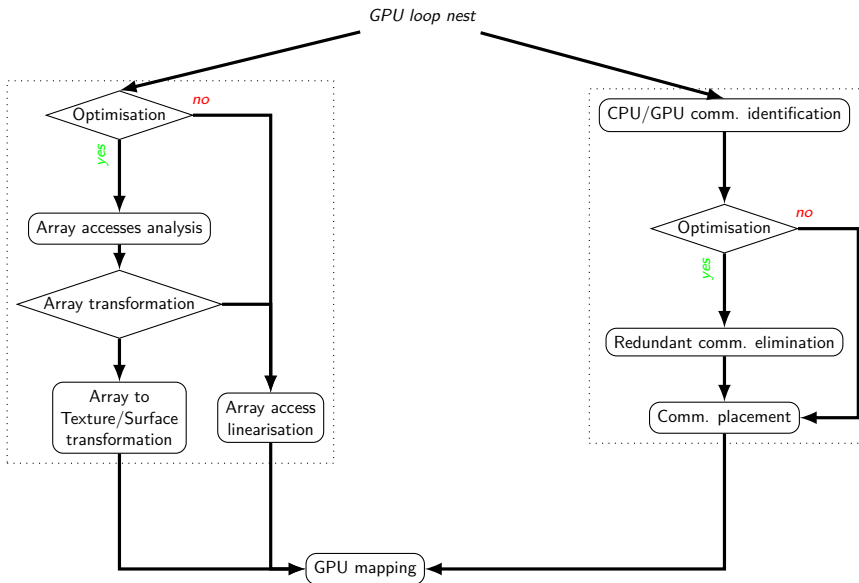
Inter GPU loop nest optimisation

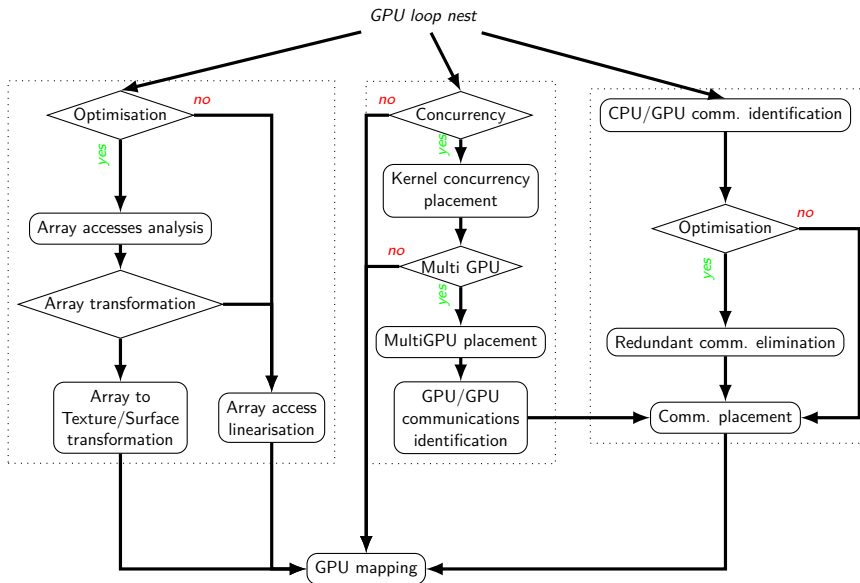


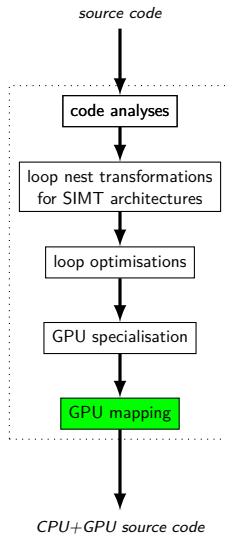


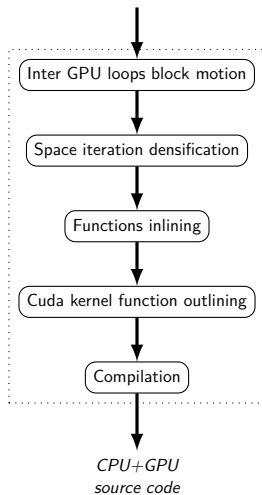


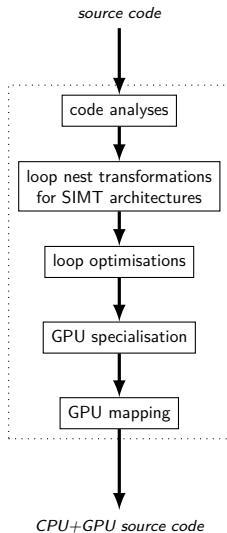


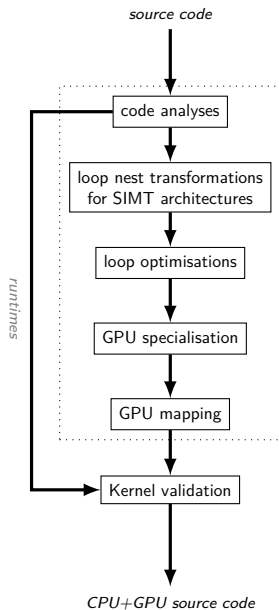


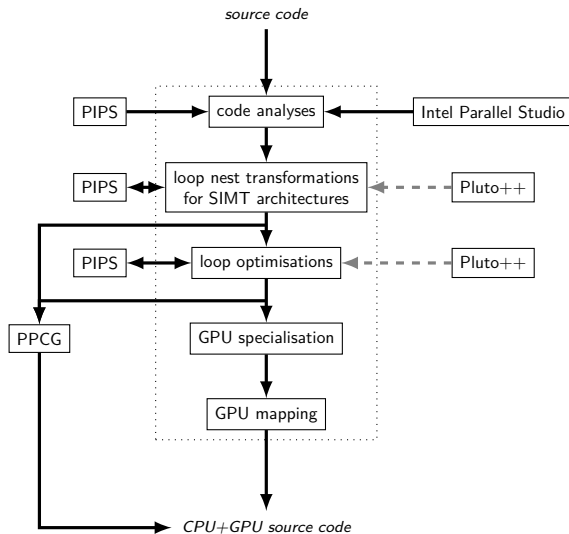












Validation

Methodology features:

- loop transformations applied:
 - strip mining, fusion, interchange, parallel reduction, tiling
- intra/inter GPU loop nest fusion
- kernel concurrency, multiGPU
- multidimensional array to texture/surface transformation
- CPU/GPU redundant communication elimination
- CPU/GPU asynchronous communications

Validation

Two applications benched:

- Threewise, a local variance computation algorithm
 - Time complexity: $O(N^2) \rightarrow O(N \log N)$
 - Runtime: $[3000ms, 100ms] \rightarrow 27ms$
 - Preserved output results
- SimpleFlow algorithm
 - Global runtime: $50s \rightarrow 6s$
 - Preserved output results

Contributions

- Methodology for GPU
 - with an optional architectural specialisation,
 - not domain specific (image processing),
 - based on arithmetic intensity metric (roofline model),
 - not language/API specific,
 - industrial application oriented.
- Methodology validation:
 - Threewise, a local variance computation algorithm
 - SimpleFlow algorithm
- Criteria developed for methodology driving
- Optimisation of a GPU parallel reduction pattern²
- Micro-compilation analysis developed
- Many frameworks have been evaluated:
 - Intel Parallel Studio, PIPS, PPCG, Pluto, ...

²Florian Guin, Corinne Ancourt, and Christophe Guettier. “Threewise: a local variance algorithm for GPU”. . In: *19th IEEE International Conference on Computational Science and Engineering (CSE 2016)*. 2016, pp. 257–262.

Perspectives

- ① Automate the methodology
- ② Benchmark (validation)
- ③ Extend to signal processing applications
- ④ Extend to other GPGPU architectures (AMD)
- ⑤ Extend to other parallel architectures (Intel Xeon Phi, Kalray MPPA, ...)