Methodology for mapping image processing algorithms on massively parallel processors

An NVIDIA GPU specific approach

Florian Gouin  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:** Processor frequency wall

**Figure:** Flynn’s taxonomy

**Instructions**

- **SISD**
- **MISD**
- **SIMD**
- **MIMD**

**SIMT**
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.
The SimpleFlow\textsuperscript{1} algorithm is available in the OpenCV extensions.

- Approximatively 600 lines of code
- Sequential algorithm
- Dynamic control code
- Approximative runtime for a couple of 2 million pixels images:
  - \textbf{200s} on a NVIDIA Jetson TX1
    - ARM Cortex A57(1.9GHz) + A53(1.3GHz)
  - \textbf{50s} on a desktop computer
    - Intel Core i7 4770S (8 logical cores at 3.1GHz)
  - Ideal runtime: \textbf{40ms}

- Language and library: C++ with the OpenCV library

SimpleFlow algorithm

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
Overview - Macroscopic scale

- Source code
- Code analyses
  - Loop nest transformations for SIMT architectures
    - Loop optimisations
      - GPU specialisation
        - GPU mapping
          - CPU+GPU source code
source code

code analyses

loop nest transformations for SIMT architectures

loop optimisations

GPU specialisation

GPU mapping

CPU+GPU source code
loop nest transformations for SIMT architectures
loop nest transformations for SIMT architectures

parallel loops

sequential loops

GPU loop identification
loop nest transformations for SIMT architectures

parallel loops

sequential loops

GPU loop identification

GPU loop pattern

\[
\begin{align*}
&b_0 \quad \text{// or} \quad 1 \leq \#b \leq 3 \\
&b_1 \quad \text{// or} \quad 0 \leq \#t \leq 3 \\
&b_2 \\
&t_0 \quad \text{// or} \quad 0 \leq \#t \leq 3 \\
&t_1 \quad \text{// or} \quad \text{// or} \\
&t_2 \\
\end{align*}
\]
loop nest transformations for SIMT architectures

parallel loops

sequential loops

GPU loop identification

GPU loop pattern

$1 \leq \#b \leq 3$

blocks

GPU loop size

$t \leq 32$

threads

$0 \leq \#t \leq 3$

GPU loop pattern

$G_{pu}$ loop size

$b = b_0 \times b_1 \times b_2$

$b_0 < 2^{31} - 1$

$b_1 < 2^{16} - 1$

$b_2 < 2^{16} - 1$

$t = t_0 \times t_1 \times t_2$

$t_0 < 128$

$t_1 < 128$

$t_2 < 64$

$t \% 32 = 0$

$t > 4 \times 32$
loop nest transformations for SIMT architectures

**GPU loop pattern**

\[
\begin{align*}
&b_0 \quad 1 \leq \#b \leq 3 \\
&b_0 < 2147483647 \\
&b_1 < 65535 \\
&b_2 < 65535 \\
&t \gg b \\
&0 \leq \#t \leq 3 \\
&t < 1024 \\
&t_0 < 1024 \\
&t_1 < 1024 \\
&t_2 < 64 \\
&t \% 32 = 0 \\
&t > 4 \times 32
\end{align*}
\]

**GPU memory size**

\[\text{Global memory footprint} < \text{GPU memory}\]
loop nest transformations for SIMT architectures

parallel loops

sequential loops

GPU loop identification

GPU loop pattern

GPU loop size

GPU memory size

GPU loop nests

**GPU loop pattern**

\[
\begin{align*}
1 \leq \#b & \leq 3 \\
b_0 & \text{ or } \ b_0 < 2^{31} \\
b_1 & \text{ or } \ b_1 < 65535 \\
b_2 & \text{ or } \ b_2 < 65535 \\
b & \gg t
\end{align*}
\]

**GPU loop size**

\[
\begin{align*}
t & = t_0 \times t_1 \times t_2 \\
t & < 1024 \\
t_0 & < 1024 \\
t_1 & < 1024 \\
t_2 & < 64 \\
t\%32 & = 0 \\
t & > 4 \times 32
\end{align*}
\]

**GPU memory size**

Global memory\(_{footprint}\) < GPU\(_{memory}\)
loop nest transformations for SIMT architectures

parallel loops

sequential loops

GPU loop identification

GPU loop pattern

GPU loop size

GPU memory size

GPU loop nests

Fusion  Fission  Tiling  Interchange  Splitting  Coalescing  Strip mining  Parallel reduction

X  X  X  X  X  X  X  X

X  X  X  X  X

X  X  X

X  X  X

X  X  X
loop nest transformations for SIMT architectures

parallel loops

sequential loops

CPU loop nests

GPU loop identification

GPU loop pattern

GPU loop size

GPU memory size

CPU loop nests

Fusion
Fission
Tiling
Interchange
Splitting
Coalescing
Strip mining
Parallel reduction

X X X X X X X

X X X X X X X

X X X X

X X X X

X X X X
Local – global optimisations

source code

- code analyses
- loop nest transformations for SIMT architectures
- loop optimisations
- GPU specialisation
- GPU mapping

CPU+GPU source code
Intra GPU loop nest optimisation

GPU loop nest

Optimisation

yes

no

loop fusion analysis

fusion optimisation

loop fusion

GPU loop nest

Intra GPU loop pattern

1 ≤ #b ≤ 3

0 ≤ #t ≤ 3

b0

b1

b2

//

//

//

//

// or

// or

// or

// or

// or

blocks

threads

Intra GPU loop nest

loop fusion analysis

fusion optimisation

loop fusion

GPU loop nest

Optimisation
Inter GPU loop nest optimisation

**GPU loop nest**

- Optimisation
  - no
  - yes
    - Micro-compilation analysis
      - Kernel optimisation
        - kernel fusion
        - kernel fission
        - GPU loop nest
Inter GPU loop nest optimisation

GPU loop nest

Optimisation

yes

no

Micro-compilation analysis

Kernel optimisation

kernel fusion

kernel fission

GPU loop nest

Inter GPU loops block motion

Space iteration densification

Functions inlining

Cuda kernel function outlining

Kernel pre-compilation

Pseudo assembly code analysis

Arithmetic Intensity

# Registers
Context and motivation

Application case

Mapping methodology

Experiments

Conclusion

GPU specialisation

source code

code analyses

loop nest transformations for SIMT architectures

loop optimisations

GPU specialisation

GPU mapping

CPU+GPU source code
Context and motivation

Application case

Mapping methodology

Experiments

Conclusion

GPU specialisation

**GPU loop nest**

- **Optimisation**
  - Yes
  - No
  - Array accesses analysis
  - Array transformation
    - Array to Texture/Surface transformation
    - Array access linearisation

- **CPU/GPU comm. identification**
  - Yes
  - No

- **Comm. placement**

- **GPU mapping**
GPU specialisation

**GPU loop nest**

- **Optimisation**
  - yes
  - no
  - Array accesses analysis
    - Array transformation
      - Array to Texture/Surface transformation
      - Array access linearisation
    - GPU mapping

- **Concurrency**
  - yes
  - no
  - Kernel concurrency placement
    - Multi GPU
      - MultiGPU placement
      - GPU/GPU communications identification
    - Optimisation
      - no
      - Redundant comm. elimination
      - Comm. placement
  - no
  - yes
  - yes
  - no
  - yes
  - no
Context and motivation
Application case
Mapping methodology
Experiments
Conclusion

GPU mapping

source code

code analyses

loop nest transformations for SIMT architectures

loop optimisations

GPU specialisation

GPU mapping

CPU+GPU source code
GPU mapping

Inter GPU loops block motion

Space iteration densification

Functions inlining

Cuda kernel function outlining

Compilation

**CPU+GPU source code**
Source code

- Code analyses
- Loop nest transformations for SIMT architectures
- Loop optimisations
- GPU specialisation
- GPU mapping

CPU+GPU source code
Summary

- Context and motivation
- Application case
- Mapping methodology
- Experiments
- Conclusion

**Source code**

Code analyses

Loop nest transformations for SIMT architectures

Loop optimisations

GPU specialisation

GPU mapping

Kernel validation

**CPU+GPU source code**
Summary

source code

- PIPS
- code analyses
- Intel Parallel Studio
- PIPS
- loop nest transformations for SIMT architectures
- Pluto++
- PIPS
- loop optimisations
- Pluto++
- PIPS
- GPU specialisation
- PPCG
- CPU+GPU source code
- GPU mapping

Application case

Mapping methodology
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 pattern\(^2\)

- Micro-compilation analysis developed

- Many frameworks have been evaluated:
  - Intel Parallel Studio, PIPS, PPCG, Pluto, ...

Perspectives

1. Automatize 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, ...)

28/28