Threewise: A Local Variance Algorithm for Graphical Processors

Florian Gouin[1,2], Corinne Ancourt[1], Christophe Guettier[2]

[1]MINES ParisTech PSL, Centre de Recherche en Informatique – Fontainebleau, France
[first name].[last name]@mines-paristech.fr

[2]SAFRAN Electronics & Defense – Massy, France

CSE 2016 – Paris, France – August 24-26, 2016

Variance application field

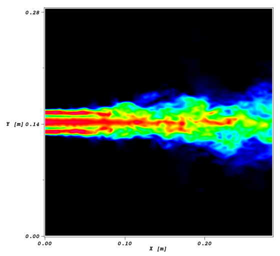
Non-exhaustive lists...

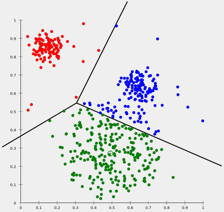
Domains :

- Statistics
- Business Intelligence
- Simulation

Application cases :

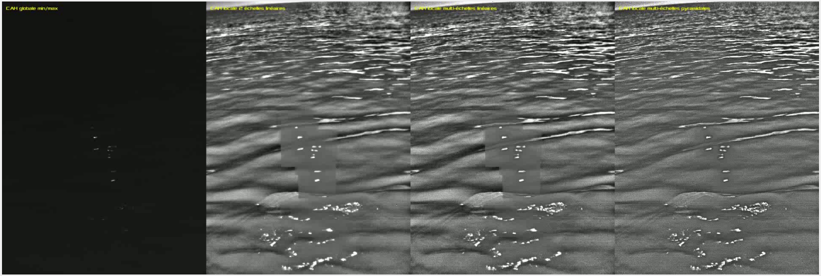
- Machine Learning
- Data Mining
- Clustering
- Anomaly detection





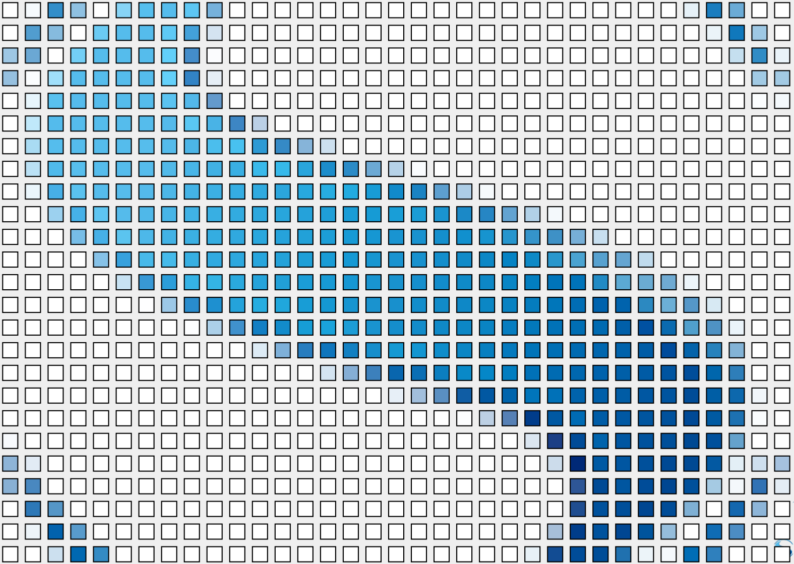
Domain : image processing

Application : local contrasts enhancement



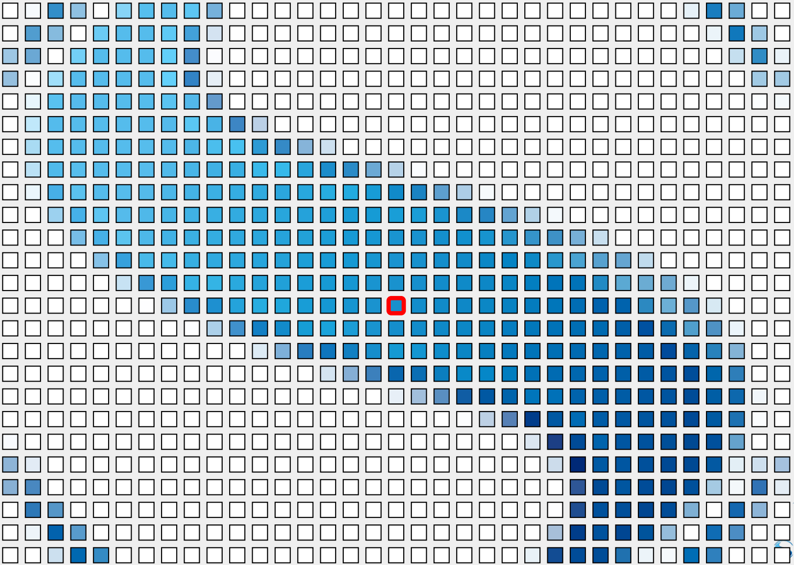
Local contrasts enhancement

Kernel or stencil computation principle



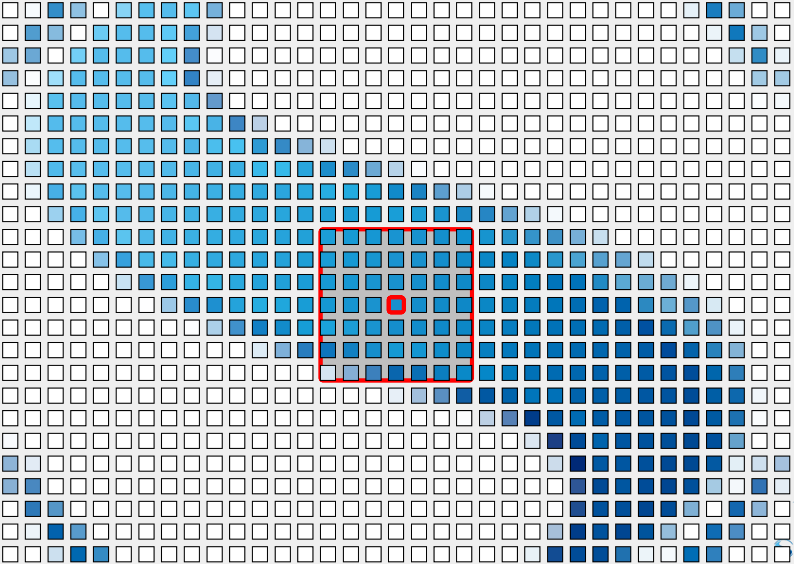
Local contrasts enhancement

Kernel or stencil computation principle



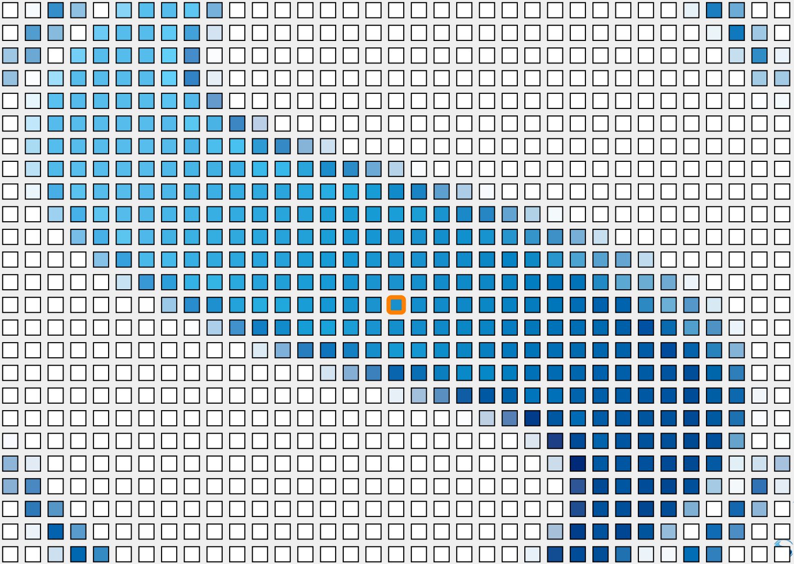
Local contrasts enhancement

Kernel or stencil computation principle



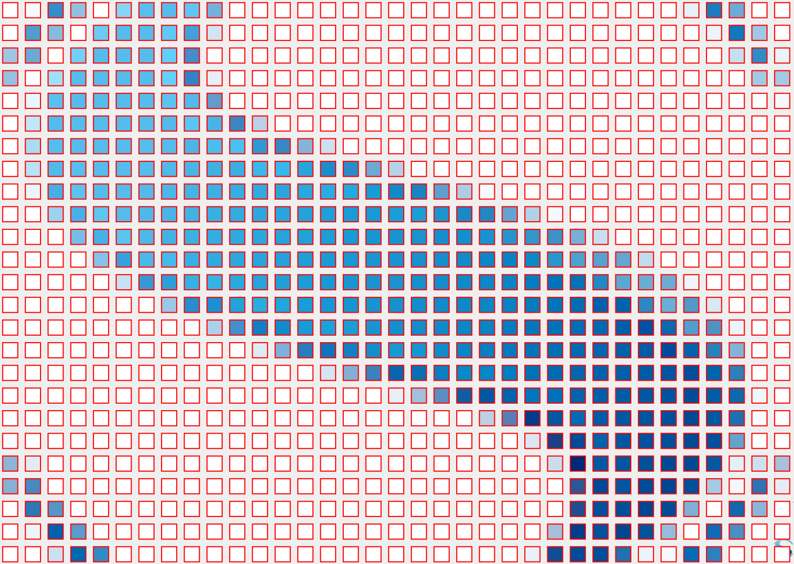
Local contrasts enhancement

Kernel or stencil computation principle



Local contrasts enhancement

Kernel or stencil computation principle



Local contrasts enhancement

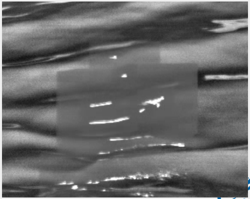
Features and issues

Algorithm features :

- No reduction
  - N input data for N output data
- Quadratic problem
- Image borders issue
  - mirror-like data replication

Visual artifacts :

- IEEE754 floating point encoding precision
  - numerical stability issue with some variance applications
- Halo phenomenon
  - Central weighting
  - Multi-sizes variance kernel computation



Variance computation solutions

State of the art

Usual formula

$$\sigma_\varphi^2 = \frac{\sum_{i=1}^{n} (\varphi_i - \mu_{\varphi,n})^2}{n} \qquad \mu_{\varphi,n} = \frac{\sum_{i=1}^{n} \varphi_i}{n}$$

Kœnig formula

$$\sigma_\varphi^2 = \mu_{\varphi^2,n} - \mu_{\varphi,n}^2 \qquad \mu_{\varphi^2,n} = \frac{\sum_{i=1}^{n} \varphi_i^2}{n}$$

Online algorithm

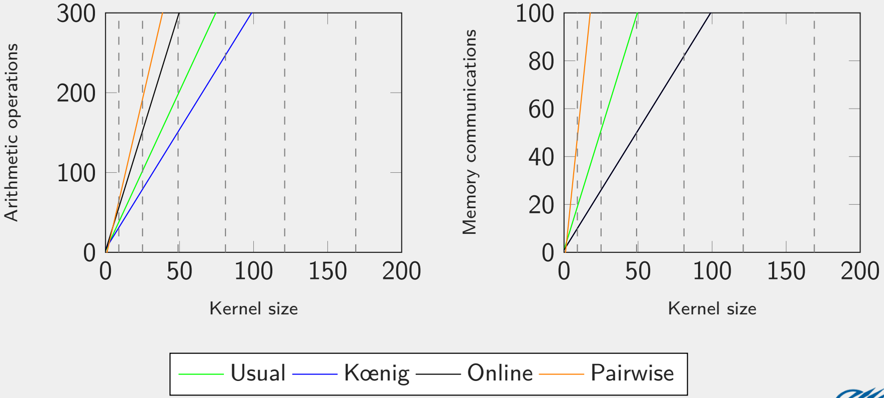
$$\sigma_\varphi^2 = \frac{M_{2,n}}{n} \qquad M_{2,n} = M_{2,n-1} + (\varphi_n - \mu_{\varphi,n-1}) \times (\varphi_n - \mu_{\varphi,n})$$

Pairwise algorithm

$$M_{2,\varphi_{1,2n}} = M_{2,\varphi_{1,n}} + M_{2,\varphi_{n+1,2n}} + \frac{1}{2n}\left(\sum_{i=1}^{n} \varphi_i - \sum_{i=n+1}^{2n} \varphi_i\right)^2$$

Cost functions

Memory communications and arithmetic operations



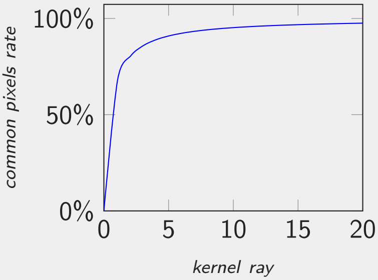
Variance kernel algorithm

Typical algorithm

```
/* Loops iterating through image elements */
for y ← 0 to HEIGHT do
    for x ← 0 to WIDTH do
        /* Loops iterating through kernel elements */
        for ky ← y − n to y + n do
            for kx ← x − n to x + n do
                /* Variance computation */
```

Optimisations

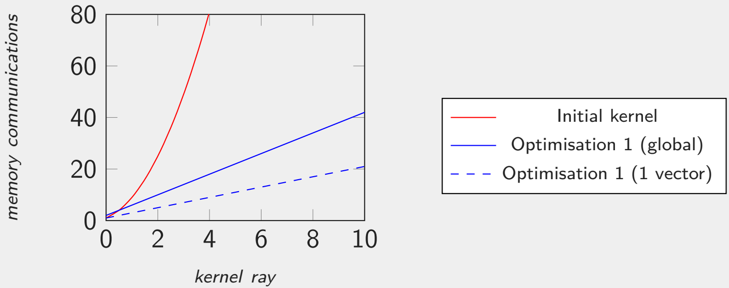
Evolution of the common pixels quantity for two kernels from contiguous pixels



Optimisations

1^{st} optimisation : Kernel separation

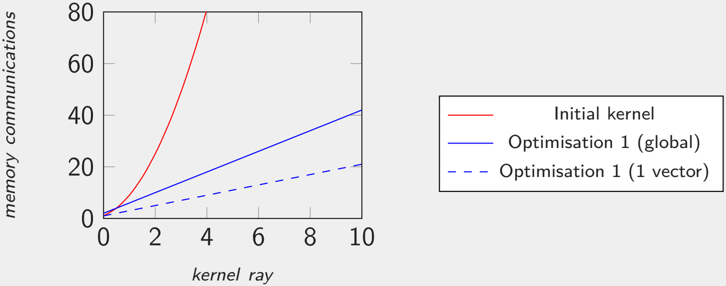
$$\begin{pmatrix} 1 & 2 & \dots & n & \dots & 2 & 1 \\ 2 & 4 & \dots & 2n & \dots & 4 & 2 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ n & 2n & \dots & n^2 & \dots & 2n & n \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 2 & 4 & \dots & 2n & \dots & 4 & 2 \\ 1 & 2 & \dots & n & \dots & 2 & 1 \end{pmatrix}$$



Optimisations

1^{st} optimisation : Kernel separation

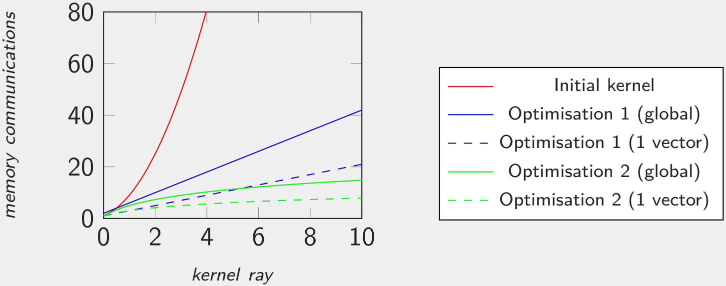
$$\begin{pmatrix} 1 & 2 & \dots & n & \dots & 2 & 1 \\ 2 & 4 & \dots & 2n & \dots & 4 & 2 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ n & 2n & \dots & n^2 & \dots & 2n & n \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 2 & 4 & \dots & 2n & \dots & 4 & 2 \\ 1 & 2 & \dots & n & \dots & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ \dots \\ n \\ \dots \\ 2 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 2 & \dots & n & \dots & 2 & 1 \end{pmatrix}$$



Optimisations

2^{nd} optimisation : Vector decomposition

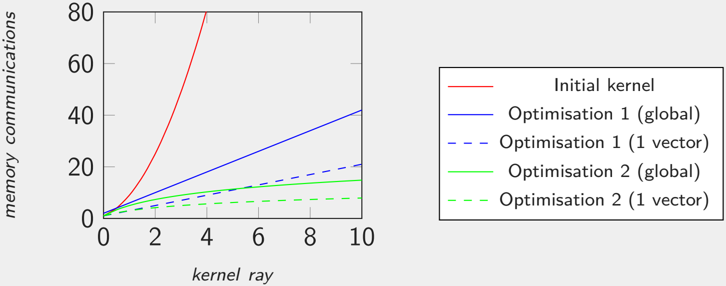
$$\begin{pmatrix} 1 & 2 & \dots & n & \dots & 2 & 1 \\ 2 & 4 & \dots & 2n & \dots & 4 & 2 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ n & 2n & \dots & n^2 & \dots & 2n & n \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 2 & 4 & \dots & 2n & \dots & 4 & 2 \\ 1 & 2 & \dots & n & \dots & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ \dots \\ n \\ \dots \\ 2 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 2 & \dots & n & \dots & 2 & 1 \end{pmatrix}$$



Optimisations

2^{nd} optimisation : Vector decomposition

$$\begin{pmatrix} 1 & 2 & \dots & n & \dots & 2 & 1 \\ 2 & 4 & \dots & 2n & \dots & 4 & 2 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ n & 2n & \dots & n^2 & \dots & 2n & n \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 2 & 4 & \dots & 2n & \dots & 4 & 2 \\ 1 & 2 & \dots & n & \dots & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ \dots \\ n \\ \dots \\ 2 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 2 & \dots & n & \dots & 2 & 1 \end{pmatrix}$$

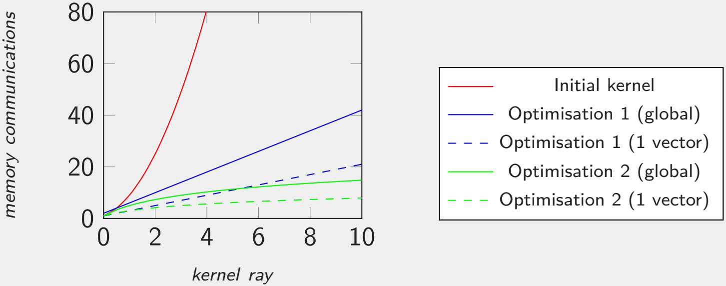


Optimisations

2^{nd} optimisation : Vector decomposition

$$\begin{pmatrix} 1 & 2 & \dots & n & \dots & 2 & 1 \\ 2 & 4 & \dots & 2n & \dots & 4 & 2 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ n & 2n & \dots & n^2 & \dots & 2n & n \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 2 & 4 & \dots & 2n & \dots & 4 & 2 \\ 1 & 2 & \dots & n & \dots & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \\ 2 \\ 0 \\ 1 \end{pmatrix} \otimes \dots \otimes \begin{pmatrix} 1 \\ 0 \\ \dots \\ 0 \\ 2 \\ 0 \\ \dots \\ 0 \\ 1 \end{pmatrix} \otimes$$

$$\begin{pmatrix} 1 & 2 & \dots & n & \dots & 2 & 1 \end{pmatrix}$$

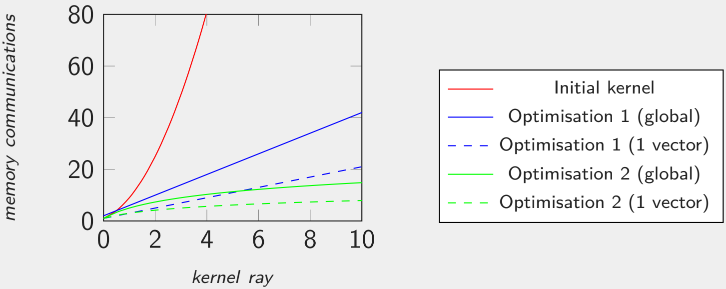


Optimisations

2^{nd} optimisation : Vector decomposition

$$\begin{pmatrix} 1 & 2 & \dots & n & \dots & 2 & 1 \\ 2 & 4 & \dots & 2n & \dots & 4 & 2 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ n & 2n & \dots & n^2 & \dots & 2n & n \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 2 & 4 & \dots & 2n & \dots & 4 & 2 \\ 1 & 2 & \dots & n & \dots & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \\ 2 \\ 0 \\ 1 \end{pmatrix} \otimes \dots \otimes \begin{pmatrix} 1 \\ 0 \\ \dots \\ 0 \\ 2 \\ 0 \\ \dots \\ 0 \\ 1 \end{pmatrix} \otimes$$

$$\begin{pmatrix} 1 & 2 & \dots & n & \dots & 2 & 1 \end{pmatrix}$$

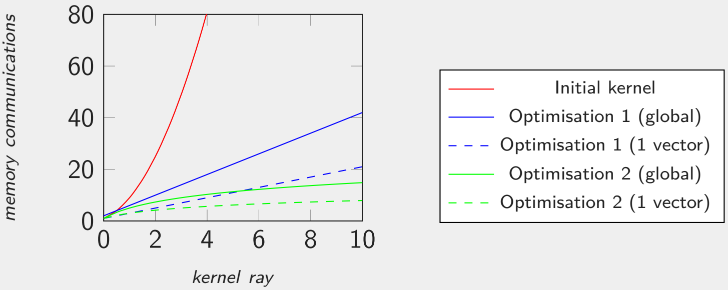


Optimisations

2^{nd} optimisation : Vector decomposition

$$\begin{pmatrix} 1 & 2 & \dots & n & \dots & 2 & 1 \\ 2 & 4 & \dots & 2n & \dots & 4 & 2 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ n & 2n & \dots & n^2 & \dots & 2n & n \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 2 & 4 & \dots & 2n & \dots & 4 & 2 \\ 1 & 2 & \dots & n & \dots & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \\ 2 \\ 0 \\ 1 \end{pmatrix} \otimes \dots \otimes \begin{pmatrix} 1 \\ 0 \\ \dots \\ 0 \\ 2 \\ 0 \\ \dots \\ 0 \\ 1 \end{pmatrix} \otimes$$

$$\begin{pmatrix} 1 & 2 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 2 & 0 & 1 \end{pmatrix} \otimes \dots \otimes \begin{pmatrix} 1 & 0 & \dots & 0 & 2 & 0 & \dots & 0 & 1 \end{pmatrix}$$



Variance computation algorithm
Optimised algorithm

```
/* Loop iterating through horizontal sparse vectors          */
for s ← 0 to n do
    /* Loops iterating through image elements                */
    for y ← 0 to HEIGHT do
        for x ← 0 to WIDTH do
            /* variance computation                          */
/* Loop iterating through vertical sparse vectors            */
```

Optimisations

3^{rd} Optimisation : Threewise algorithm

Pairwise algorithm

$$M_{2,\varphi_{1,2n}} = M_{2,\varphi_{1,n}} + M_{2,\varphi_{n+1,2n}} + \frac{1}{2n}\left(\sum_{i=1}^{n}\varphi_i - \sum_{i=n+1}^{2n}\varphi_i\right)^2$$

Threewise algorithm

$$M_{2,\varphi_{1,3n}} = M_{2,\varphi_{1,n}} + 2M_{2,\varphi_{n+1,2n}} + M_{2,\varphi_{2n+1,3n}} + \frac{\delta}{2n}$$

$$\delta = \left(\sum_{i=1}^{n}\varphi_i - \sum_{i=n+1}^{2n}\varphi_i\right)^2 + \frac{1}{2}\left(\sum_{i=1}^{n}\varphi_i - \sum_{i=2n+1}^{3n}\varphi_i\right)^2 + \left(\sum_{i=n+1}^{2n}\varphi_i - \sum_{i=2n+1}^{3n}\varphi_i\right)^2$$

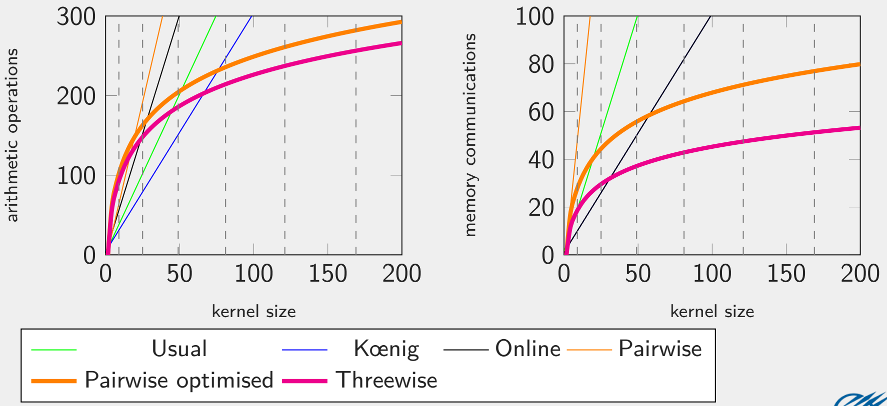
CUDA Kernel : local variance computation

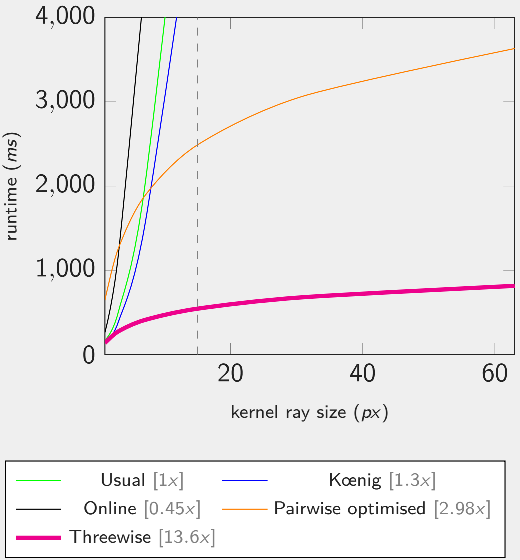
horizontal sparse vector computation

```
__global__ void varianceKernelX(float* inM, float* outM, float* inV, float* outV,
unsigned int width, unsigned int height, short delta) {

const unsigned short x = blockIdx.x * blockDim.x + threadIdx.x;
const unsigned short y = blockIdx.y * blockDim.y + threadIdx.y;

if (x<width && y<height){
    float variance;

    const unsigned short x1 = x>delta ? x-delta : delta-x;
    const unsigned short x2 = x+delta<width ? x+delta : 2*width-x+delta -1;

    const float m1 = inM[y * width + x1];
    const float m = inM[y * width + x];
    const float m2 = inM[y * width + x2];

    const float d1 = m1 - m;
    const float d2 = m - m2;
    const float d3 = m1 - m2;

    variance = inV[y * width + x1] + 2*inV[y * width + x]
               + inV[y * width + x2];
    variance += (2*d1*d1 + 2*d2*d2 + d3*d3)/4.0;
    variance /= 4.0;
    outV[y * width + x] = variance;
    outM[y * width + x] = (m1 + 2*m + m2) / 4.0;
}}
```

Cost functions

Memory communications and arithmetic operations





Experimental data

Image :

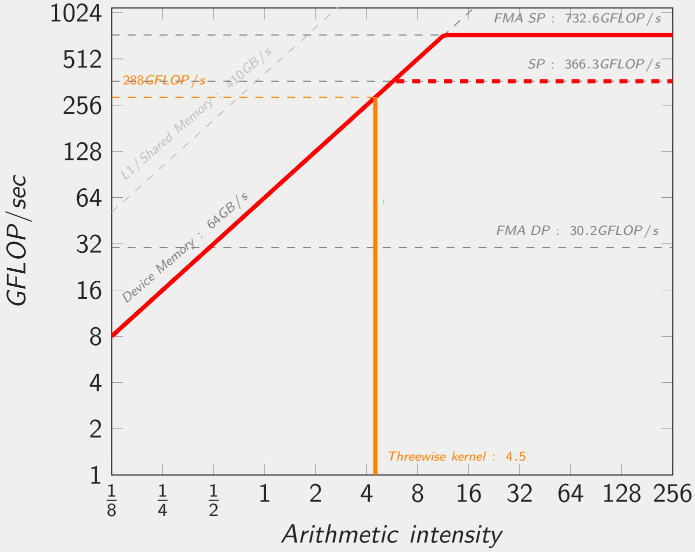
- resolution : 9688 × 8262 (80MPixels)
- depth : 8bits
- grayscale

Graphical card :

- architecture : NVIDIA Kepler
- reference : Quadro K2000
- computing units : 384 cores
- memory bandwidth : 64GBytes/s
- ECC : disabled
- L1 cache/shared memory : auto

Arithmetic intensity analysis

NVIDIA Quadro K2000



Conclusion

Conclusion :

- IEEE754 precision preserved
- free multi-scales management
- reduction of arithmetic operations
- reduction of memory communications
- runtime improvement (speedup : $\sim 4.0\times$)
- implementation nearly optimal for NVIDIA Quadro K2000

Further works :

- finer cache management
  - better use of shared memory
  - runtime improvement
- N–wise algorithm
  - balance definition between :
    - memory communications factorisation
    - loop unrolling