2016 IEEE International Conference on Computational Science and Engineering, IEEE International Conference on Embedded and Ubiquitous Computing, and International Symposium on Distributed Computing and Applications to Business, Engineering and Science

Threewise: a local variance algorithm for GPU

Florian Gouin
MINES ParisTech/CRI
PSL Research University,
35 rue Saint Honoré
77305 Fontainebleau - France
Email: florian.gouin@mines-paristech.fr

Corinne Ancourt
MINES ParisTech/CRI
PSL Research University,
35 rue Saint Honoré
77305 Fontainebleau - France
Email: corinne.ancourt@mines-paristech.fr

Christophe Guettier
SAFRAN Electronics & Defense,
100 avenue de Paris
91300 Massy - France

***Abstract*—Variance computation is commonly used in many fields like in image processing to improve local contrasts. This article is not only about developing and placing an algorithm of variance computation for graphical processors, it will also introduce its optimisation in terms of precision and computing time in relation to architectural constraints of graphical processors. Our algorithm enables to improve complexity in $O(N \log N)$ and brings a speedup of 112 compared to the classical formulation and of 4 regarding the optimized Pairwise algorithm.**

***Index Terms*—variance computation, kernel computation, GPU, image processing, local contrasts enhancement**

I. Significance of variance computation

Variance computation is commonly used in many fields. Simon and Litt [1] use it in the aerospace domain. Variance computation is therefore utilized to track aircraft engine parameters in real time to improve anomalies detection and subsequently the engine maintenance. For Restrepo and al. [2], variance computation is employed to identify objects through a 3D volume analysis. On the other hand we also have Stünckler and Behnke [3] who applied variance computation to a simultaneous localization and mapping algorithm. All these examples are a just a few samples but globally variance computation is seen in the ANOVA[1] domain where it plays an important role in data analysis and mining. More specifically in image processing, Singh and al. [4], [5], Chang and Wu [6] or Cvetkovic and al. [7]–[10] use the variance to improve local contrasts. This technic brings well contrasting images, where over-exposed and under-exposed areas are destorted. However, a certain attention must be drawn to problematics like adding noises in images or creating rings artefacts observable in the figure 1 from wrong variance utilization. This article shows the development of a variance computation algorithm, its setup on GPU[2] and its practical application to image processing to improve local contrasts. We draw a specific attention to how variance is computed to handle the noise and the ring artifacts detailed earlier. In the end, our solution considers:

- the precision of floating point operations which is a potential source of visual artifacts
- the amount of arithmetics operations and memory communications impacting the execution time on graphical processors

[1]**an**alysis **o**f **va**riance.
[2]**G**raphical **P**rocessing **U**nit

Although Benett and al. [11] showed some interest in parallel variance computation, their research was based on Intel Xeon computing clusters. As a result their architecture differs from the one for GPU.

The solution that we present here reduces the usual complexity from such algorithm in $O(N^2)$ to a complexity in $O(N \log(N))$ for both memory communications and arithmetics operations.

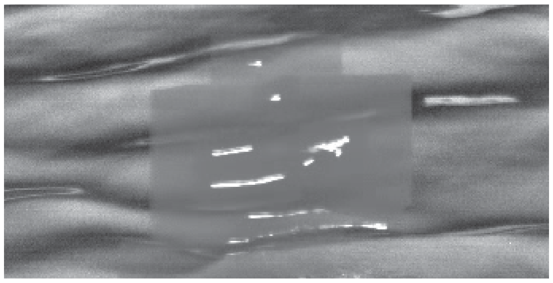
Figure 1. Example of ring artifacts

II. Context – local contrast improvement in image processing

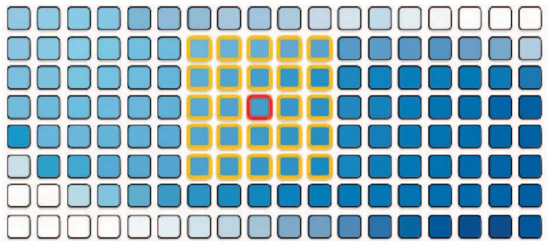
Figure 2. Two pixels kernel ray

In this article, the variance computation applies to an image processing algorithm to improve local contrasts. Our aim is to render all the computations in real time for HD video sequence at 25 images per second. That means we have to compose with 1920×1080 pixels in 40ms. We take into account Singh and al. [4], [5], Chang and Wu [6] and Cvetkovic and al. [7]–[10] solutions to limits noises et rings artefacts. By doing so, we need to compute several local variance kernels sizes simultaneously which will increase the quantity of needed computations and communications.

The local variance computation implies for each element of a given image, to compute the variance of a kernel with a specific size. The figure 2 shows a two-pixel ray kernel composed with orange circled pixels and a red one. The latter represents the kernel center as well as the element concerned by the kernel computation. So this computation is applied to each pixel of the image and as a consequence we have as much inner data as outter data for the representative algorithm. Eventually for red circled pixels near the border of the image, the orange circled pixels which are out of the image will be mirrored. This operation is left to GPU specialized hardware units.

III. Variance computation algorithms

Pébay [12] and Cha and al. [13] showed some interest in to different variance computation algorithms as well as there numerical stability. Variance computation formulas listed below are used for each element of an image to compute their local variance.

Algorithm 1 : usual variance formula

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

$$\mu_\varphi = \frac{\sum_{i=1}^{n}\varphi_i}{n} \tag{2}$$

For a given variance kernel, N represents the number of elements in this kernel, φ_i is the element number i, μ_φ the mean of the domain and σ_φ^2 is the squared deviation also called the variance.
With this algorithm, the local mean computation (2) is invariant in calculating the variance (1). With this in mind, it is advised to extract the carried loop calculating the equation (2) from the outter loop calculating the variance given by the equation (1). The dependence brought by this optimisation improves the global number of operations and communications. However it requires to execute these two loops sequentially. The kernel elements are read twice and as a consequence this doubles the quantity of memory accesses. Eventually, the distributivity of these two loops is poor due to the typical carried dependence of loop reductions. Communication and operation number cost functions extracted from our implementation are given by formulas (3) and (4).

$$Cost_{Comm} = Img_{size} \times (2N+1) \tag{3}$$

$$Cost_{Op} = Img_{size} \times (4N+2) \tag{4}$$

Algorithm 2 : Kœnig formula

To the opposite of the first algorithm, the Kœnig formula (5) presents two independent loops (2) and (6). These have the same number n of iterations and read the same data in the same order φ_i. So, we have all the requirements to apply a loop fusion, which improves the data locality and reduces memory communications. The result is clearly visible by comparing the cost functions (3) and (7).

$$\sigma_\varphi^2 = \mu_{\varphi^2} - \mu_\varphi^2 \tag{5}$$

$$\mu_{\varphi^2} = \frac{\sum_{i=1}^{n}\varphi_i^2}{n} \tag{6}$$

However Chan and al. [13] have demonstrated that the numerical precision of this algorithm is less stable than the first one. Indeed, the more the kernel size increases, the bigger and the closer the value of μ_{φ^2} and μ_φ^2. In consequence, the substraction of these two values creates a small value that is prone to high truncations. Kirk and al. [14] and Collange and al. [15] explain this phenomenon applied to the floating point computation units of the GPU. Finally, the solution proposed by Whithead and al. [16], which is to sort data by numerical order in the way to improve floating point computation precisions, can't be used here. If we were to use their solution, the consequence would be to increase the number of arithmetic operations and memory communications. The arithmetic cost function given by our implementation of the Kœnig formula is represented by the equation eqrefcoutOp2.

$$Cost_{Comm} = Img_{size} \times (N+1) \tag{7}$$

$$Cost_{Op} = Img_{size} \times (3N+4) \tag{8}$$

Algorithm 3 : the online *algorithm*

$$M_{2,n} = M_{2,n-1} + (\varphi_n - \mu_{n-1}) \times (\varphi_n - \mu_n) \tag{9}$$

$$\sigma_\varphi^2 = \frac{M_{2,n}}{n} \tag{10}$$

The *online* algorithm introduces a new formula (9) in the variance computation with $M_{2,n}$. This formula has the benefit of being more stable than Kœnig's and it always goes through the kernel data once. On the other hand, the dependence between $M_{2,n}$ and $M_{2,n-1}$ reduces the distributivity of this algorithm with the sequential scan of data. The recursive aspect of this approach has very few value for the SIMD architecture of the graphical processors. The equation (10) brings the final variance result. Finally, equations (11) and (12) explain the arithmetic operations and memory communications cost functions.

$$Cost_{Comm} = Img_{size} \times (N+1) \tag{11}$$

$$Cost_{Op} = Img_{size} \times (6N+2) \tag{12}$$

Algorithm 4 : the Pairwise *algorithm*

$$M_{2,\varphi_{1,2n}} = M_{2,\varphi_{1,n}} + M_{2,\varphi_{n+1,2n}} + \frac{1}{2n}\left(\sum_{i=1}^{n}\varphi_i - \sum_{i=n+1}^{2n}\varphi_i\right)^2 \quad (13)$$

The *Pairwise* algorithm proposed by Chan and al. [13] spreads the *online* algorithm. The formula (13) applied to our image processing application, make it possible to compute the global variance of two equal size subsets based on their own means and variance. This algorithm has a numerical stability equal to the *online* algorithm. However, it benefits from an improved parallelism given that $M_{2,\varphi_{1,n}}$ and $M_{2,\varphi_{n+1,2n}}$ present no data dependances. On the other hand, this method as described by Chan and al., is more adapted to variance computation of a global set. This data reduction type algorithm generates one result from a set of multiple data. In this case the data quantity is divided by two at each iteration to finally generates a single variance data. But as mentioned before our kernel based algorithm generates a variance result for each element of the image. In other words we generate as many output data as input data. As a consequence, the exact use of the *Pairwise* algorithm with local variance computation involves to duplicate kernels elements in the way to reduce them. To conclude, this approach slashes the memory communication rate for an improved parallelism. Memory and arithmetic operations cost functions are given by the equations (14) and (15).

$$Cost_{Comm} = Img_{size} \times \log_2 N \times 6 \quad (14)$$

$$Cost_{Op} = Img_{size} \times \log_2 N \times 8 \quad (15)$$

IV. An optimized algorithm for graphical processors

A. Kernel optimisation

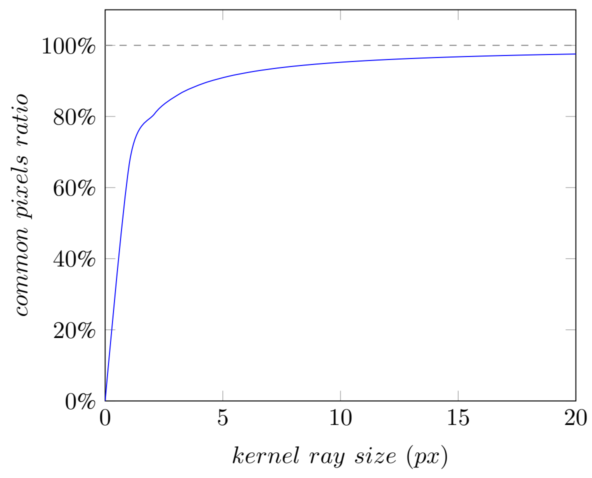

Figure 3. Common pixels rate for two adjacents kernels

The above figure 3 highlights the common pixels ratio for two consecutive kernels implied in variance computation. The algorithmic kind curve reveals that this quantity of redundants memory communications grows quickly to be close to 100% for a kernel size higher than 5 pixels. These redundants memory communications added to those of non consecutive kernel affect runtime performances of variance kernel algorithms by unnecesarily over using the memory bandwidth.

Moreover, as Hennessy and Patterson have noted in their well-known *Computer Architecture A Quantitative Approach* [17], for the last decades, memory performances have improved slowly compared to those of computing units. In consequence, data memory bandwidth in algorithms is now critical in modern high performance architectures like GPUs. Also, it is essential to minimize redundants memory communications in our variance kernel algorithm. With this aim in mind, we have used two optimisations. Both consider ring artefacts problems by applying highest weighting to central kernel element. Weightings decrease progressively as the concerned kernel element is far from the central element. So, the lowest weighting is applied to kernel extremities.

1) Kernel separation: In order to reduce the redundants memory communications, we firstly propose to apply the property defined by equation (16). This property enables to reduce the quantity of memory communication by separating a kernel of r length ray to two $2r+1$ length vectors. In this way, we have transformed $(2r+1)^2$ communications which corresponds to a $O(N^2)$ complexity to $2 \times (2r+1)$ communications represented by a $O(N)$ complexity. The first one corresponds to the blue curve in the figure 4 and the second to the red curve. The application of these two vectors is necessarily successive but the equation (16) is commutative. In other words, it doesn't matter if we start with the vertical vector or the horizontal one.

$$\begin{pmatrix} 1 & 2 & ... & n & ... & 2 & 1 \\ 2 & 4 & ... & 2n & ... & 4 & 2 \\ ... & ... & ... & ... & ... & ... & ... \\ n & 2n & ... & n^2 & ... & 2n & n \\ ... & ... & ... & ... & ... & ... & ... \\ 2 & 4 & ... & 2n & ... & 4 & 2 \\ 1 & 2 & ... & n & ... & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ ... \\ n \\ ... \\ 2 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 2 & ... & n & ... & 2 & 1 \end{pmatrix} \quad (16)$$

2) Vector decomposition: In addition to the kernel separation, we have used the principle exposed by the formula (17) in order to reduce again the memory communications. Our aim is to decompose the two variance vectors obtained in the previous optimisation into a serie of successive three elements sparse vectors. These sparse vectors are generated with a common pattern. For each vector central element we apply a weighting of two and for each vector extremity we apply a weighting of one. The gain obtained with this optimisation is represented by the brown curve in the figure 4. The red curve from the first optimisation is for two $2r+1$ length vectors. So, the number of memory communications associated is $2 \times (2r+1)$ which gives us a $O(N)$ complexity. By adding this second optimisation, each of these two vectors is decomposed to $2 \times \log_2(r+1)+1$ communications. The new complexity obtained is now $O(\log(N))$ for ech kernel. Finally, the formula (17) is commutative like the formula (16). This global commutativity is beneficial to multiple kernel sizes computation in order to reduce ring artefacts. In this way, by using the first and the smallest vertical sparse vector with the horizontal one, we can obtain the first and the smallest variance kernel. This kernel is the first scale. By using it with the seconds vertical and horizontal sparse vectors, we can obtain a second greater kernel scale and so on. As a consequence, the cost in terms of memory communications and arithmetics operations for this multi-scales computation is only the one for the greatest variance kernel scale.

$$\begin{pmatrix} 1 \\ 2 \\ \dots \\ n \\ \dots \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ \mathit{0} \\ 2 \\ \mathit{0} \\ 1 \end{pmatrix} \otimes \dots \otimes \begin{pmatrix} 1 \\ \mathit{0} \\ \dots \\ \mathit{0} \\ 2 \\ \mathit{0} \\ \dots \\ \mathit{0} \\ 1 \end{pmatrix} \tag{17}$$

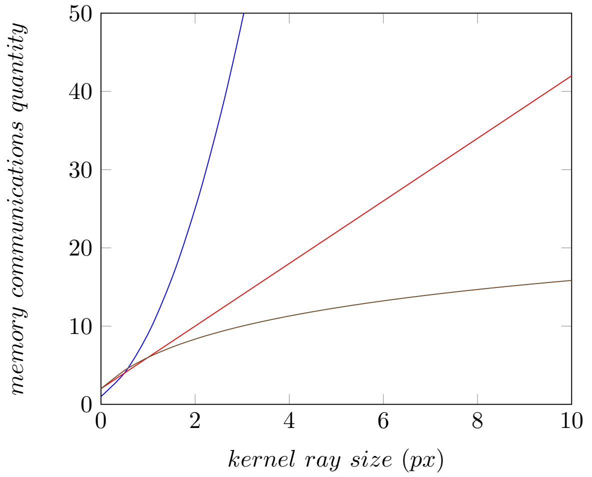

Figure 4. Effect of kernel separation and vector decomposition on memory communications. The blue curve is the initial form, the red one corresponds to the first optimisation and the brown one to the cumulation of first and second optimisations.

$$\begin{aligned} M_{2,\varphi_{1,3n}} = {} & M_{2,\varphi_{1,n}} + 2M_{2,\varphi_{n+1,2n}} + M_{2,\varphi_{2n+1,3n}} \\ & + \frac{1}{2}(\mu_{1,n} - \mu_{n+1,2n})^2 \\ & + \frac{1}{4}(\mu_{1,n} - \mu_{2n+1,3n})^2 \\ & + \frac{1}{2}(\mu_{n+1,2n} - \mu_{2n+1,3n})^2 \end{aligned} \tag{18}$$

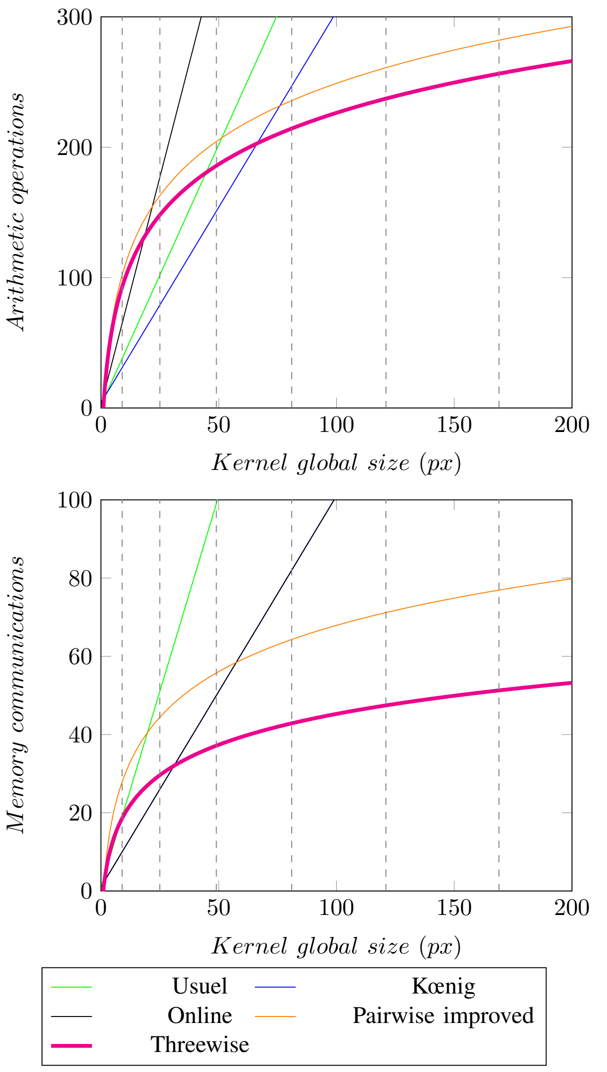

Figure 5. Arithmetic operations and memory communications complexities depending to kernel size. The dashed vertical lines represent the kernel ray size evolution.

B. Our Threewise algorithm

By modifying the *Pairwise* algorithm and taking into account the previous two optimisations, we have developed the formula (18). Due to the parallel aspect of the *Pairwise* algorithm and its numerical stability, our *threewise* algorithm is a good candidate to GPU architecture. Our formula is better appropriate to 3 elements sparse vectors with one of them having a weighting of two. Memory communications for each kernel are deducted from the cost function (19) and the arithmetic operations from the cost function (20). The resulting algorithm is described in Algorithm 1. Cost functions depending on kernel size is described in figure 5. In order to prove the benefit of the *threewise* algorithm, we have applied our sparse vectors double optimisations to the *pairwise* algorithm. We can observe that these two algorithms are most efficient than the others described previously in this article for a kernel ray greater than 3 pixels. Lastly, we can notice that the *threewise* algorithm is prone to lower arithmetic operations and memory communications quantity compared to the *pairwise* algorithm. This can be explained by the fact that we process with three elements per iteration for our solution as opposed to with only two elements for the *pairwise* solution.

$$Cost_{Comm} = Img_{size} \times 2 \times (\log_2(N+1) - 1) \times 4 \quad (19)$$

$$Cost_{Op} = Img_{size} \times 2 \times (\log_2(N+1) - 1) \times 20 \quad (20)$$

```
Input/Output: $mImg$ is a $WIDTH \times HEIGHT$ image initialized with original data
Input/Output: $vImg$ is a $WIDTH \times HEIGHT$ image initialized to 0
1  $delta \leftarrow 1$;
2  for $s \leftarrow 0$ to $STEPS$ do
       /* Horizontal vector pass */
3      for $y \leftarrow 0$ to $HEIGHT$ do
4          for $x \leftarrow 0$ to $WIDTH$ do
5              $deltaA \leftarrow mImg[x][y] - mImg[x-delta][y]$;
6              $deltaB \leftarrow mImg[x][y] - mImg[x+delta][y]$;
7              $deltaC \leftarrow mImg[x+delta][y] - mImg[x-delta][y]$;
8              $vImg2[x][y] \leftarrow vImg[x][y] + vImg[x-delta][y] + vImg[x+delta][y]$;
9              $vImg2[x][y] \leftarrow vImg2[x][y] + (2 \times detlaA \times deltaA + 2 \times detlaB \times deltaB + detlaC \times deltaC)/4$;
10             $mImg2[x][y] \leftarrow mImg[x][y] + mImg[x-delta][y] + mImg[x+delta][y]$;
       /* Vertical vector pass */
11     Same $x$ and $y$ loops are run here but $delta$ is applied to $y$. Data are read from mImg2 and vImg2. Results are written in mImg and vImg;
12     $delta \leftarrow delta \times 2$;
```

Algorithm 1: *Threewise* algorithm for variance kernel computation on GPU.

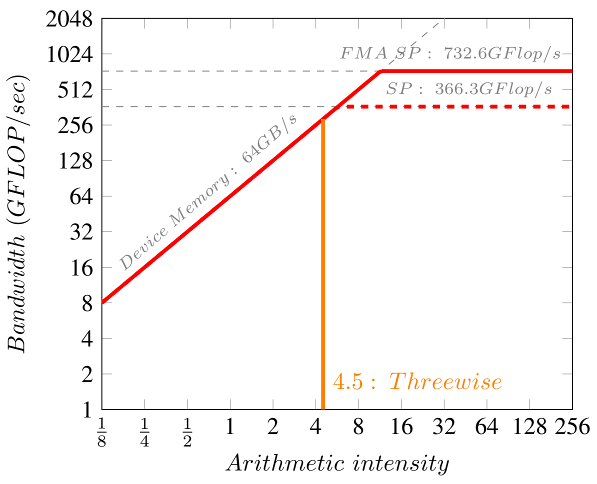

Figure 6. Theorical placement of the *Threewise* algorithm on the Patterson's roofline model for the NVIDIA Quadro K2000 architecture.

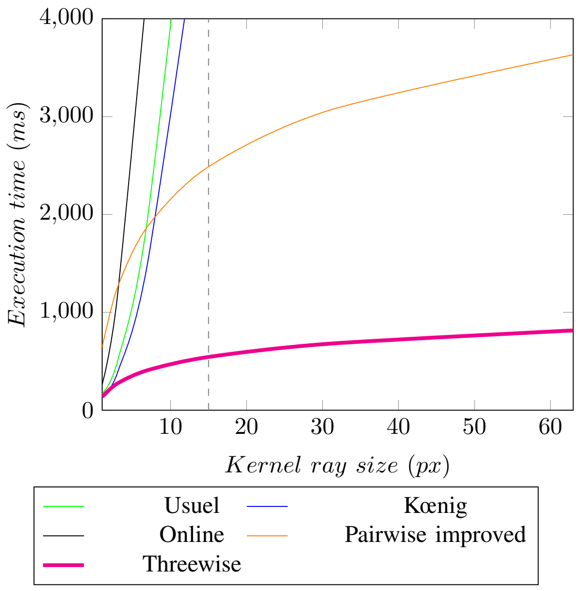

Figure 7. Variance algorithms execution times comparison for a 80MPixels image. The vertical dashed line represents a common 15 pixels kernel ray size.

V. Experimental Results

Our purpose is now to experimentally validate the benefits of our kernel variance algorithm on a GPU architecture. The algorithms listed in this article have been translated to CUDA 7.0 and we have used a NVIDIA Quadro K2000 GPU card for our experiments. The latter is composed of two NVIDIA Kepler SMX processors, each of them carrying 192 cores. The analysis of the *threewise* algorithm assembly code generated by the NVIDIA compiler brings us a ratio of 4.5 for the number of arithmetic instructions per data bytes communicated from memory. This metric defined as Arithmetic Intensity by Patterson [17], [18] can be used to define the limiting factor of an algorithm for a given architecture. We can notice in the figure 6 that our algorithm is limited by the memory bandwidth but is really close to the transition point. After that point, our algorithm would have been limited by computation capacities. As a consequence, we theorically almost use the full capacity of our GPU card and the execution time of our algorithm would depend on the memory communications optimisations. To compare our algorithm with the others, we have used a 80MPixels image and varried the kernel ray size. The corresponding execution times can be observed in the figure 7. We can note that the *pairwise* and the *threewise* algorithms have a $O(N\log(N))$ curve line unlike the others which have a $O(N^2)$ curve line as expected. The *pairwise* algorithm is more efficient for a kernel ray size higher that 7 pixels. However, our algorithm is almost the most efficient for every kernel ray size. This can be explained by a better instruction pipeline feeding which is typical to loop unrolling. For our application, we have then used a 1920 by 1080 pixels HD image with a 63 pixels kernel ray. The speedup and execution time for each algorithm can be observed in the figure 8. Our algorithm presents a speedup of 112 for a 27 ms runtime compared to the usual formula with a 3028 ms. Finally, our algorithm presents a speedup of 4 when compared to the optimized pairwise. To conclude, we remember that the *Threewise* algorithm has an execution time lower than 40 ms and as a consequence is a good candidate to real-time image processing for HD 25 fps video format as needed.

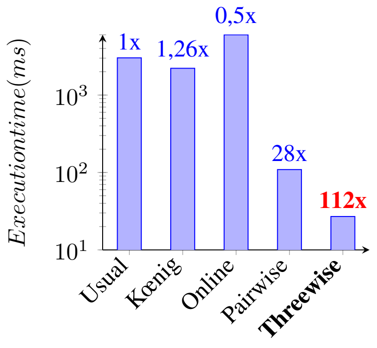

Figure 8. Variance algorithms execution times comparison for a HD image.

VI. Conclusion

The *Threewise* algorithm presented in this article is adapted to kernel variance computation for image processing on massively parallel graphical processors. This algorithm is deducted from the *Pairwise* algorithm and so takes over its numerical stability. We have applied two optimisations to these algorithms. The first one is the separation of the kernel in two vectors. The second one splits these two latter vectors in a serie of sparse vectors. These two optimisations used together contribute to reducing the global memory communications and the global arithmetic operations complexities from $O(N^2)$ to $O(N \log N)$. Moreover, the highly parallel orientation of our *Threewise* algorithm added to its intensive memory communications reduction, best fits to GPU architecture. Finally, the experimentation based on a NVIDIA Quadro K2000 and presented in this article, has comfirmed our theorical expectations. These optimisations effectively offer a 112 speedup compared to the common variance algorithm with only 27 ms execution time for a high definition image on this architecture.

References

[1] Simon, Donald L and Litt, Jonathan S, “A Data Filter for Identifying Steady-State Operating Points in Engine Flight Data for Condition Monitoring Applications,” *Journal of Engineering for Gas Turbines and Power*, vol. 133, no. 7, p. 071603, 2011.
[2] Restrepo, Maria I and Mayer, Brandon A and Ulusoy, Ali O and Mundy, Joseph L, “Characterization of 3-d volumetric probabilistic scenes for object recognition,” *Selected Topics in Signal Processing, IEEE Journal of*, vol. 6, no. 5, pp. 522–537, 2012.
[3] Stückler, Jörg and Behnke, Sven, “Multi-resolution surfel maps for efficient dense 3D modeling and tracking,” *Journal of Visual Communication and Image Representation*, vol. 25, no. 1, pp. 137–147, 2014.
[4] Singh, S Somorjeet and Singh, Th Tangkeshwar and Devi, H Mamata and Sinam, Tejmani, “Local contrast enhancement using local standard deviation,” *International Journal of Computer Applications*, vol. 47, no. 15, 2012.
[5] Singh, S Somorjeet and Singh, Th Tangkeshwar and Singh, N Gourakishwar and Devi, H Mamata, “Global-Local Contrast Enhancement,” *International Journal of Computer Applications*, vol. 54, no. 10, 2012.
[6] Chang, Dah-Chung and Wu, Wen-Rong, “Image contrast enhancement based on a histogram transformation of local standard deviation,” *Medical Imaging, IEEE Transactions on*, vol. 17, no. 4, pp. 518–531, 1998.
[7] Cvetkovic, Sascha D and Schirris, Johan and De With, Peter HN, “Locally-adaptive image contrast enhancement without noise and ringing artifacts,” in *Image Processing, 2007. ICIP 2007. IEEE International Conference on*, vol. 3. IEEE, 2007, pp. III–557.
[8] Cvetkovic, Sascha D and Schirris, Johan and others, “Multi-band locally-adaptive contrast enhancement algorithm with built-in noise and artifact suppression mechanisms,” in *Electronic Imaging 2008*. International Society for Optics and Photonics, 2008, pp. 68 221C–68 221C.
[9] Cvetkovic, S and Schirris, Johan and de With, PHN, “Non-linear locally-adaptive video contrast enhancement algorithm without artifacts,” *Consumer Electronics, IEEE Transactions on*, vol. 54, no. 1, pp. 1–9, 2008.
[10] Cvetkovic, S and Klijn, Jan and With, PHN de, “Tone-mapping functions and multiple-exposure techniques for high dynamic-range images,” *Consumer Electronics, IEEE Transactions on*, vol. 54, no. 2, pp. 904–911, 2008.
[11] Bennett, Janine and Grout, Ray and Pébay, Philippe and Roe, Diana and Thompson, David, “Numerically stable, single-pass, parallel statistics algorithms,” in *Cluster Computing and Workshops, 2009. CLUSTER’09. IEEE International Conference on*. IEEE, 2009, pp. 1–8.
[12] Pébay, Philippe, “Formulas for robust, one-pass parallel computation of covariances and arbitrary-order statistical moments,” *Sandia Report SAND2008-6212, Sandia National Laboratories*, vol. 94, 2008.
[13] Chan, Tony F and Golub, Gene H and LeVeque, Randall J, “Updating formulae and a pairwise algorithm for computing sample variances,” in *COMPSTAT 1982 5th Symposium held at Toulouse 1982*. Springer, 1982, pp. 30–41.
[14] Kirk, David B and Wen-mei, W Hwu, *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.
[15] Collange, Sylvain and Daumas, Marc and Defour, David, “État de l’intégration de la virgule flottante dans les processeurs graphiques.”
[16] Whitehead, Nathan and Fit-Florea, Alex, “Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs,” *rn (A+B)*, vol. 21, pp. 1–1 874 919 424, 2011.
[17] Hennessy, John L and Patterson, David A, *Computer architecture: a quantitative approach*. Elsevier, 2011.
[18] Williams, Samuel and Waterman, Andrew and Patterson, David, “Roofline: an insightful visual performance model for multicore architectures,” *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.