Investigating performance variations of an optimized GPU-ported granulometry algorithm

Vincent Boulos, Vincent Fristot, Dominique Houzet, Luc Salvo, P. Lhuissier

Investigating performance variations of an optimized GPU-ported granulometry algorithm

Vincent Boulos, Vincent Fristot, Dominique Houzet
GIPSA-lab
UMR5216 CNRS/INPG/UJF/U.Stendhal
F-38402 GRENOBLE CEDEX, France
firstname.lastname@gipsa-lab.grenoble-inp.fr

Luc Salvo, Pierre Lhuissier
SIMAP
UMR5266 CNRS/INPG/UJF
F-38402 GRENOBLE CEDEX, France
firstname.lastname@simap.grenoble-inp.fr

***Abstract*—In this article, we present an optimized GPU implementation of a granulometry algorithm which is used a lot in the study of material domain. The main contribution to this algorithm is the binarization of the input data which increases throughput while reducing data allocated memory space. Also, the optimized GPU implementation brings an order of magnitude speedup compared to a CPU multi-threaded implementation. Furthermore, we investigate the reasons why GPU performance drop for different input data dimensions. Three main factors are exposed: under-exploited threads, threadblocks and streaming multiprocessors. This study should help the reader understand the tight relation that exists between the CUDA programming paradigm and the gpu architecture as well as some main bottlenecks.**

I. Introduction

GPU architecture is well adapted to 2D and 3D image analysis. The potential of such architectures have been investigated in several studies and reveals large speedup for various 2D and 3D image analysis algorithms in comparison to a CPU implementation [1]–[3]. One of the most time consuming algorithm is the granulometry algorithm which is used to determine objects size in an image [4], [5]. In materials science, it is often used on 3D images obtained by tomography in metal foams. Pore size extraction and structure thickness are possible applications of the granulometry algorithm [6], [7]. This algorithm involves a large number of erosion and dilation with specific structural elements and therefore is quite time consuming [8], [9]. For example, granulometry on a metal foam performed with imorph [7], [10] on a volume 600 x 600 x 250 with maximum size objects of 50 pixels requires 15 minutes using a CPU implementation. Time optimization can be performed but with loss of accuracy. It is important to note that acquisition of 3D images has become very fast since it requires less than 1 second to acquire a volume of 1024 x 1024 x 1024 [11] and therefore there is a real need to increase the time to perform granulometry on such large volumes, which at the moment requires several hours. The aim of this paper is, firstly, to present a fast cpu exact computation of granulometry and, secondly, to implement this algorithm on the gpu architecture. At last, we present the gpu results and investigate the reasons of the performance variations for different sizes of input data.

II. Granulometry

Granulometry is the study of the statistical distribution of the sizes of a population of finite elements. In other words, it is the study of an image's objects sizes. In physics, that would resemble sieving (grain sorting): the image is filtered with a series of sieves with decreasing hole sizes. A more specific goal is to define the predominant size of objects in the image.

A. Algorithm description

Granulometry uses morphological opening operations. An opening is the combination of two mathematical morphology operators: it is an erosion followed by a dilation. These operations are filters and the mask used by these filters is called a structuring element (SE). When performing an opening on an image, all finite element that is smaller than the structuring element disappears. Thus, the granulometry application processes an input image by computing openings with an increasing structuring element size until all objects in the volume disappear *i.e.* the volume is empty. After each opening, we collect the number of positive pixels still present in the image. We then plot the results on a curve: the granulometric curve. The abscissa of this curve represents the number of openings and the ordinate shows the number of positive pixels left in the image. The discrete derivative of the granulometric curve is called the pattern spectrum and the abscissa of its peak is the predominant size of objects in the image (Figure 1).

B. Optimizations

Whereas a granulometry algorithm for grayscale images and rectangular structuring elements had already been ported to GPU [12], our implementation of the granulometry algorithm is adapted to a particular post-tomographic usage. In this context, three constraints need to be taken into consideration and will help for further choices in optimizations. Firstly, the input data goes through a binary threshold operation before being processed by the granulometry algorithm. Thus, the input data has binary values: black or white voxels. Secondly, as shown in Figure 2, the structuring element used is a 3D cross-shaped. Thirdly, boundary conditions are accounted.

1) Algorithmic optimizations:

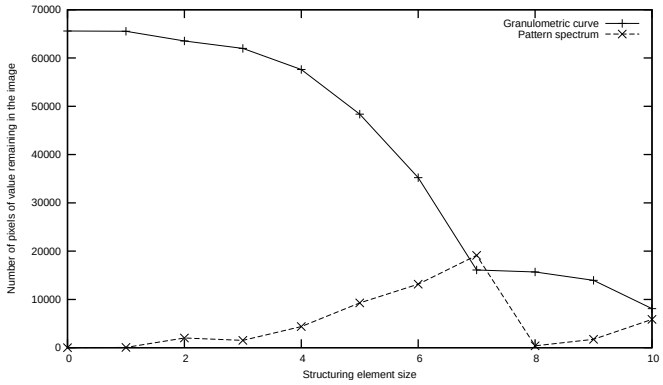


Fig. 1: Example of a granulometric curve. The pattern spectrum's extrema indicates the predominant size of the objects in the image.

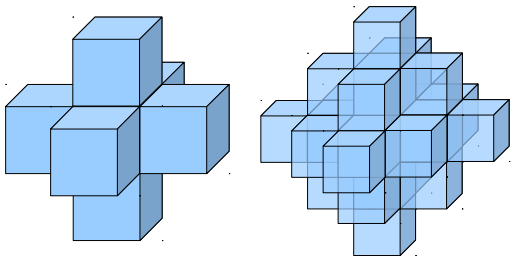
a) The structuring element shape: Any rectangular,cube-shaped structuring element is separable in 3 vector filters: one in each x,y,z direction. This property can be used to reduce the amount of accesses to memory and the amount of computations. However, the 3D cross-shaped structuring element is not a separable filter.

b) The structuring element size: An erosion/dilation with a structuring element of size $n \times x$ gives the same output as performing n successive erosions/dilations with a structuring element of size x. Also, while a 3D cross-shaped structuring element of size 1 (Figure 2a) has 6 neighboors and needs as much memory read requests and logical AND/OR operations, a structuring element of size 2 (Figure 2b) has 24 neighboors and thus needs 4x more memory read requests and computations. It is more efficient from a read/write requests count perspective to compute twice the output of an opening by a structuring element of size 1 than computing one opening with a structuring element of size 2. Therefore, one optimization consists in performing morphological operations with a constant structuring element size.

c) Redundant computations: An opening with a SE of size 1 is the succession of an erosion with the SE of size 1 and a dilation with the same SE of size 1. Based on the previous optimization, an opening with a SE of size n is the succession of n erosions with the SE of size 1 and n dilations with the same SE of size 1. In fact, in order to perform an opening with a SE of size n incremented by 1, only 1 erosion with a SE of size 1 needs to be computed followed by $n+1$ dilations with a SE of size 1. The n first erosions computations can be saved because they have already been computed on the previous step. In the end, the granulometry application is implemented as shown in Figure 3.

2) GPU implementation optimizations:

a) Maximize data reuse: In order to make the most out of each global memory access, we save in shared memory each voxel value that is used in computations more than once. When there is no bank conflict shared memory latency is nearly as quick as registers and is roughly 100x lower than global memory latency. Our approach resembles that



(a) SE of size 1 (b) SE of size 2

Fig. 2: 3D cross-shaped structuring element is not a separable filter.

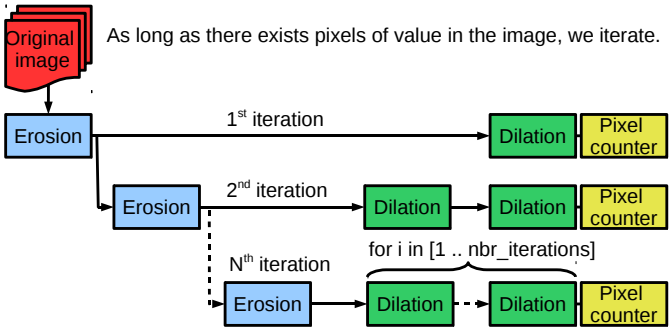


Fig. 3: The granulometry application decomposed.

of Paulius Micikevicius [13]. The naive approach to compute an erosion/dilation refetches from global memory all 6 neighboors in the SE. Our implementation reduces read latency by performing read requests from shared memory instead. Since there is not enough shared memory available per multiprocessor to store a significantly large 3D subdomain of a problem, a 2D tile is stored instead. If we assign a thread to compute output values for a given column along z (Figure 4), no additional shared memory storage is needed. Threads of a given threadblock coherently traverse the volume along z, computing output for each slice. Each thread saves its preceding/succeeding (x,y) voxel value on the z-dimension in a register (Figure 5).

b) Maximal throughput per instruction: Based on Figure 5, if each thread computes one output, the erosion equation is:

$$result = up \ \& \ down \ \& \ left \ \& \ curr \ \& \ right \ \& \ next \ \& \ previous$$

NVidia GPUs' integer ALUs perform 32-bit and 64-bit arithmetic operations [14]. However, 64-bit integer shift and bitwise operations are not natively supported in hardware. Therefore, the maximal throughput per instruction is attained when each thread processes 32-bit words. The input data is in RAW format: each voxel is encoded on one byte. However, in our context, the input data goes through a binary threshold operation before being processed by the granulometry algorithm. In other words, each voxel is a binary information. Thus, in

order to attain the maximal voxel throughput per instruction, the input data is binarized before processing: each voxel is encoded on one bit. Then, the erosion equation is slightly modified with some bit twiddling: when fetching right and left voxels, the 4-byte word is shifted by 1 bit:

$$left \;=\; ((current >> 1) \mid (right << 31))$$

$$right \;=\; ((current << 1) \mid (left >> 31))$$

Now, 32 voxels are output per thread per instruction. The same occurs with the dilation equation.

c) Minimize device memory allocation: Binarizing the input data – encoding a voxel from a byte to a bit – also divides its size by eight. This is not negligible when volumes of size 1024 voxels or more are processed on a GPU. Allocating 128 MB of device memory rather than 1 GB is a significant economy of device memory when you know that latest GeForce GPU cards embed between $1 \sim 2$ GB and approximately 2 GB for mid range Quadro GPU cards. That leaves a great amount of free device memory space for intermediate and final results buffers.

d) Optimal memory access: Processing 32-bit words per thread is also interesting from a memory access point of view. With 4-byte words, the segment size adressed by a half-warp (compute capability 1.x) or by a warp (compute capability 2.x) is maximal and thus bandwidth usage is optimal. Also, since threads access words in sequence, memory access is coalesced. Moreover, handling 4-byte words makes it easier to avoid shared memory bank conflicts since each 32-bit word is assigned to successive banks.

e) GPU-optimized reduction: The count of remainging valuable voxels is the last step after each opening iteration (Figure 3). This last step occurs within the same kernel as the one which computes the last dilation. We chose to incorporate these two computations (last dilation and counting voxels) within the same kernel in order to save time spent during global memory read/write requests. Since the result of the dilation is not used afterwards, this kernel dismisses saving the result back to global memory and saves time spent due to global memory latency. Instead, another shared memory space is allocated dedicated to counting voxel values. At first, each thread uses __popc() to count the number of voxels within the 4-byte word result value. Then, this count is progressively summed along z by each thread at its (x,y) position in shared memory. When the volume has been traversed along z, a parallel sum reduction [15] is done on the whole shared memory space for each threadblock. Finally, each threadbloc performs an atomic addition to a global memory counter.

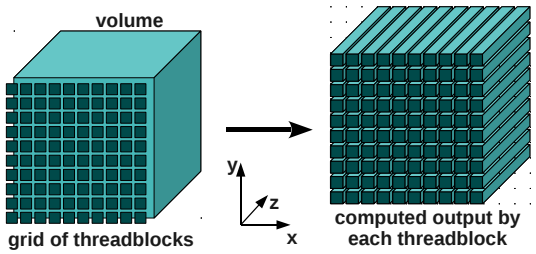


Fig. 4: All threadblocks in the grid compute one output slice while traversing the volume along z.

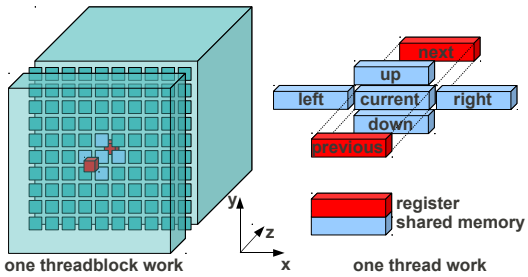


Fig. 5: All voxel neighboors in (x,y) dimension are stored in shared memory while neighboors along z are stored in registers.

III. Experimental results

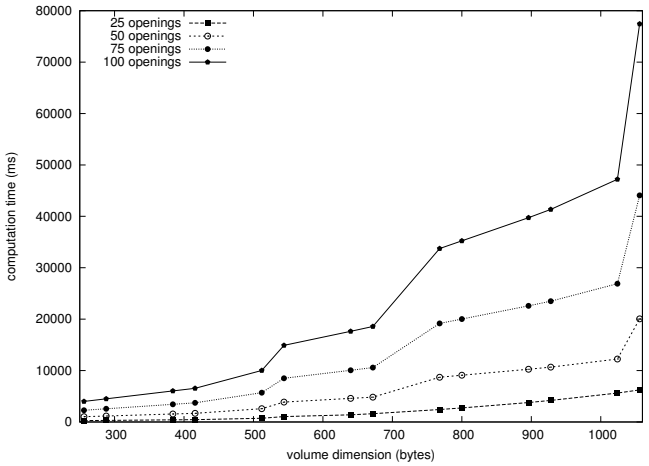
In this section, we present a comparison of the time consumption between a CPU multithread (4 cores/8 threads through HyperThreading) implementation and a GPU implementation of our granulometry algorithm. Both computing targets present the algorithmic optimizations presented in the previous section. The CPU implementation is a plugin developped for ImageJ [16] which is a well-known image processing software.

Working station configuration:

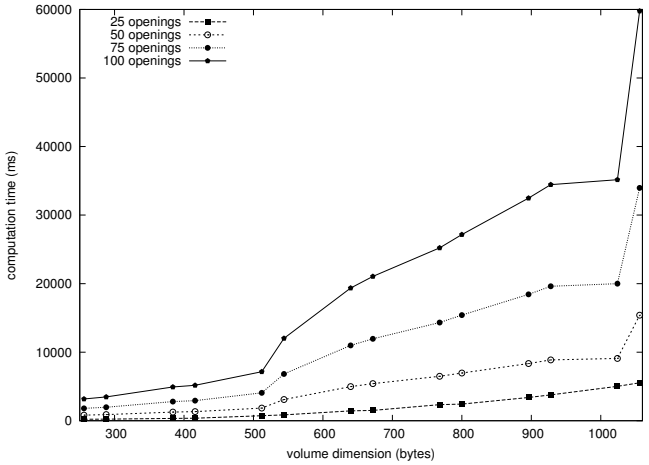
- CPU: Intel Core i7-920 (@2.67GHz)
- OS: CentOS release 5.6 (Final)
- Memory: 11.8 GBytes
- GPU: GTX285 & GTX480 & Quadro4000 through PCI-E 2.0
- CUDA Toolkit 4.0

Figure 6 shows the GPU time spent for different sizes of input data and Figure 7 shows the speedup ratio between the CPU's multithreaded implementation vs GPU's implementation for different architectures [17], [18]. The Quadro card is not as competitive as the GeForce cards because their clock frequency is diminished by half for better reliability during computations. One can note that the time needed to perform a granulometry on a volume of 1024 x 1024 x 1024 is less than 1 minute using GPU architecture and about 20 minutes for our CPU implementation, which is quite interesting in regard to time acquisition of data as presented in the introduction.

In the following sections, we investigate the reason why the computation time curve is not affine or exponential and make a link between the hardware and the CUDA programming



(a) GTX 285



(b) GTX 480

Fig. 6: Computation time for different dimensions of the input data.

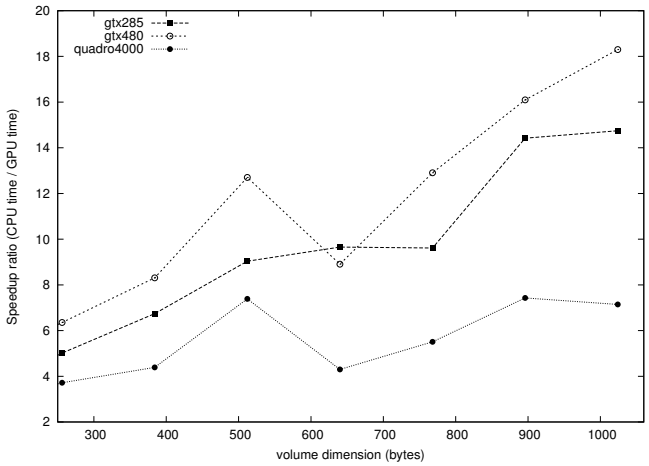


Fig. 7: Speedup ratio between CPU (4 cores,8 threads) and GPUs on a volume of different dimensions and containing objects of size 100 voxels maximum (50 openings are performed).

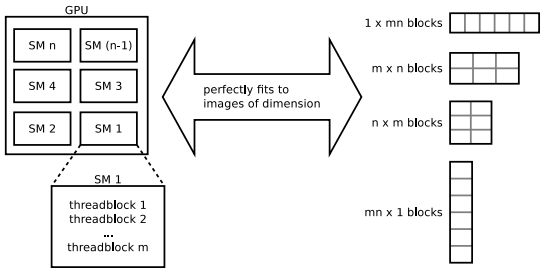


Fig. 8: If the device contains n SMs and each SM can process up to m simultaneously (m is occupancy dependant), at max workload per SM, mxn blocks are scheduled simultaneously on the device.

paradigm. The following study is conducted on a GTX285, with a fixed threadblock size of 16 x 16 and a volume of $variable$ x 512 x 512 voxels: one of the three dimensions is varying while the two others are kept unchanged and fixed to 512.

A. Streaming multiprocessors' workload: varying y-dimension

One of the big benefits of CUDA is its scalability: it automatically scales the number of threadblocks to be processed onto the number of Streaming Multiprocessors the GPU contains [14]. However, for a perfectly balanced workload, the number of blocks to process must be a multiple of the GPU's number of SMs (Figure 8).

While fixing dimBlock.y to 16, we made the y-dimension of the volume vary with a pace of 32 pixels thus increasing the number of threadblocks to be allocated by two for each pace increment. Since the used graphic card: GTX 285, possesses 30 SMs, one threadblock is affected per SM if the y-dimension is equal to $nb\_SMs \times dimBlock.y = 30 \times 16 = 480$. When the y-dimension exceeds 480, at least one SM will be attributed more than one threadblock.

This is observed on the GPU computations time on Figure 9 by a periodic time leap that appears every 480 pixels. Since all blocks do the same operations, their computation time is the same. Thus, the maximal number of blocks allocated per SM fixes the time spent for the whole GPU computations: it suffices that one SM is attributed one more threadblock for the whole processing time to be increased.

B. Threadblocks' workload: varying x-dimension

Let's consider a threadblock of size 16x16, each thread works on 4 bytes in order to optimize memory bandwidth (coalesced memory access) and each voxel is coded on one bit (Figure 10). Therefore, in ordrer for all threads of the x dimension in a threadblock to be useful, the x dimension of the image has to be a multiple of $blockDim.x \times 4bytes \times 8bits = 16 \times 4 \times 8 = 512$. Thus, the volume's width (x dimension) has to be a multiple of 512.

When the volume's width is inferior to 512 pixels, one threadblock is allocated to process 16 lines of the input

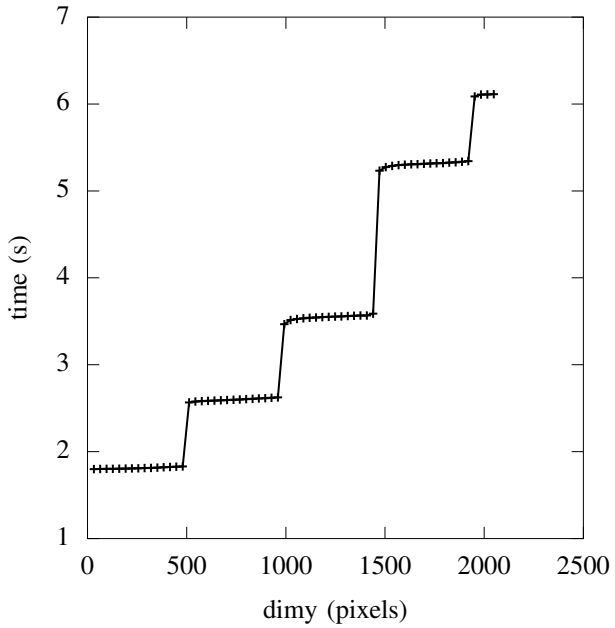


Fig. 9: Computation time for different y-dimension sizes.

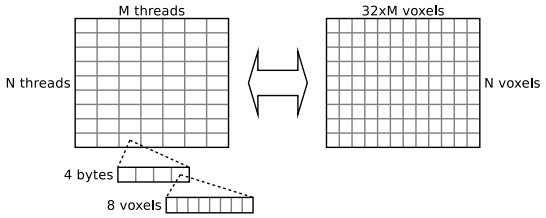


Fig. 10: One MxN threadblock maps to a (32xM)xN 2D tile of voxels.

volume ($dimBlock.y = 16$). The pourcentage of useful threads increases with the volume's width until it reachs $width = 512pixels$ and efficiency is thus maximal for the threadblock.

When the volume's width is inferior to 512, one part of the created threads is active while the other part is inactive. Nevertheless, whether threads are active/inactive doesn't influence the time spent by one warp. Since all 32 threads in a warp are tied together, the time spent by one warp is the addition, in case of divergence, of all paths' time.

When the volume's width exceeds 512 pixels another threadblock is allocated and thus adding more time to GPU computations. This is shown on Figure 11 by a periodic bounce every 512 pixels corresponding to a new allocated threadblock.

When there are idle threads, the GPU gain compared to the CPU drops. It drops proportionally to the variation of y,z-dimensions because:

- as seen in the previous section, dimy has an impact on the number of threadblocks that needs to be processed
- dimz represents the number of iterations along (x,y) computations

So the amount of idle threads and thus the amount of wasted

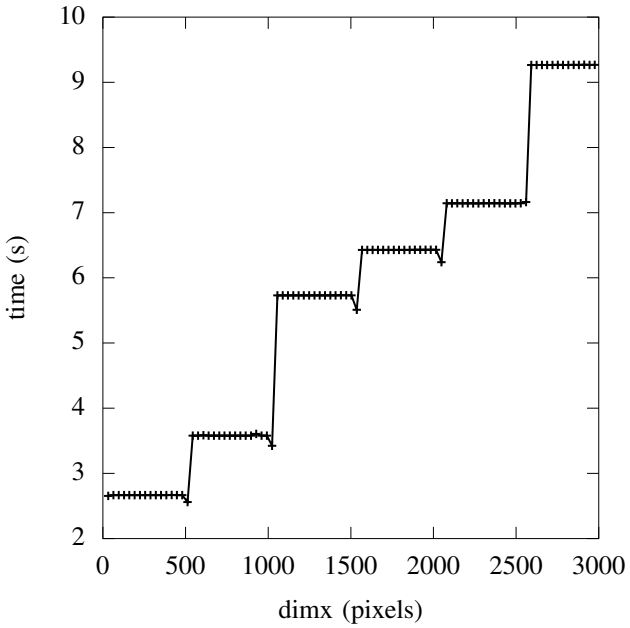


Fig. 11: Computation time for different x-dimension sizes.

time depends directly from (y,z) dimensions.

C. Threads' workload: varying z-dimension

While varying the z dimension, the amount of computations per thread proportionnaly increases and so does the computation time. This is clearly shown on Figure 12 with an affine curve.

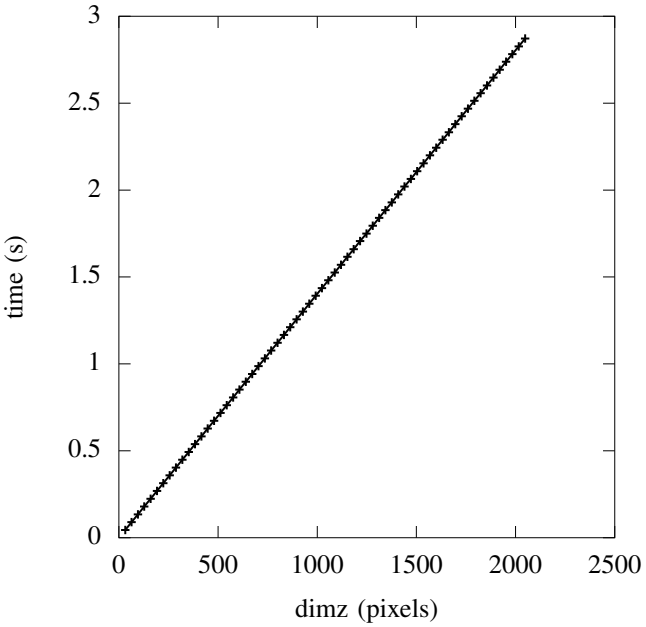


Fig. 12: Computation time for different z-dimension sizes.

IV. Conclusion

We implemented a highly performant granulometry algorithm for segmented black/white images. Thanks to the binarization technique, the CPU implementation is already very fast compared to other solutions the SIMAP lab used and which needed several hours of computations. The time needed to perform a granulometry on a volume of 1024 x 1024 x 1024 is about 20 minutes for our CPU implementation and less than 1 minute using GPU architecture.

We also observed the factors that impact the performance of the GPU implemented granulometry algorithm.

Note that this application was presented to illustrate the design flow described in [19]. However, its implemented optimizations and its performance variations were not discussed. This article presents the granulometry application with technical details.

References

[1] D. Castaño-Díez, D. Moser, A. Schoenegger, S. Pruggnaller, and A. S. Frangakis, "Performance evaluation of image processing algorithms on the gpu." *Journal of Structural Biology*, vol. 164, no. 1, pp. 153–160, 2008. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/18692140

[2] J. Cornwall, O. Beckmann, and P. Kelly, "Accelerating a c++ image processing library with a gpu," in *POHLL 2006: Workshop on Performance Optimization for High-Level Languages and Libraries (colocated with IPDPS06, Rhodes).*, 2006. [Online]. Available: http://pubs.doc.ic.ac.uk/Image-processing-with-gpu/

[3] B. Jaromír, "Algorithms of mathematical morphology for stream-oriented architectures," Ph.D. dissertation, École des Mines de Paris, Jul. 2006.

[4] G. MATHERON, *Elements pour une théorie des milieux poreux*. Paris: Masson, 1967.

[5] M. Coster and J. L. Chermant, *Precis d' Analyse d'Image*. Presses du CNRS, 1985,.

[6] N. Tuncer, G. Arslan, E. Maire, and L. Salvo, "Investigation of spacer size effect on architecture and mechanical properties of porous titanium," *Materials Science and Engineering: A*, vol. 530, no. 0, pp. 633–642, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0921509311011385

[7] E. Brun, "De limagerie 3d des structures à létude des mécanismes de transport en milieux cellulaires," Ph.D. dissertation, AIX-MARSEILLE UNIVERSITE, Sep. 2009.

[8] J. Serra, *Image Analysis and Mathematical Morphology*. Orlando, FL, USA: Academic Press, Inc., 1982.

[9] P. Soille, "On the morphological processing of objects with varying local contrast," in *Discrete Geometry for Computer Imagery*, ser. Lecture Notes in Computer Science, I. Nyström, G. Sanniti di Baja, and S. Svensson, Eds. Springer Berlin / Heidelberg, 2003, vol. 2886, pp. 52–61, 10.1007/978-3-540-39966-7_4. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-39966-7_4

[10] S. Project, "imorph." [Online]. Available: http://imorph.sourceforge.net/

[11] L. Salvo, M. DiMichiel, M. Scheel, P. Lhuissier, B. Mireux, and M. Suéry, "Ultra fast in situ x-ray micro-tomography: Application to solidification of aluminium alloys," *Materials Science Forum (Volumes 706 - 709)*, vol. THERMEC 2011, Jan. 2012.

[12] L. Domanski, P. Vallotton, and D. Wang, *Parallel van Herk/Gil-Werman image morphology on GPUs using CUDA*.

[13] P. Micikevicius, "3d finite difference computation on gpus using cuda," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 79–84. [Online]. Available: http://doi.acm.org/10.1145/1513895.1513905

[14] NVidia, "Nvidia cuda c programming guide 4.0," 2012.

[15] ——, "Nvidia cuda c sdk code samples." [Online]. Available: http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html

[16] N. I. of Health, "Imagej." [Online]. Available: http://rsbweb.nih.gov/ij/

[17] NVidia, "Nvidia geforce gtx 200 gpu architectural overview." [Online]. Available: http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf

[18] ——, "Nvidia fermi architecture whitepaper." [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[19] S. Huet, V. Boulos, V. Fristot, and L. Salvo, "DFG implementation on multi GPU cluster with computation-communication overlap," in *DFG implementation on multi GPU cluster with computation-communication overlap*, Tampere, Finlande, Nov. 2011, pp. pp.1–8. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00657536