https://ntrs.nasa.gov/search.jsp?R=20110012130 2019-08-29T18:31:11+00:00Z

2.4 GPU Accelerated Vector Median Filter

GPU Accelerated Vector Median Filter

Rifat Aras & Yuzhong Shen
Department of Modeling, Simulation, and Visualization Engineering
Old Dominion University
raras001@odu.edu yshen@odu.edu

Noise reduction is an important step for most image processing tasks. For three channel color images, a widely used technique is vector median filter in which color values of pixels are treated as 3-component vectors. Vector median filters are computationally expensive; for a window size of $n \times n$, each of the n^2 vectors has to be compared with other $n^2 - 1$ vectors in distances. General purpose computation on graphics processing units (GPUs) is the paradigm of utilizing high-performance many-core GPU architectures for computation tasks that are normally handled by CPUs. In this work, NVIDIA's Compute Unified Device Architecture (CUDA) paradigm is used to accelerate vector median filtering, which has to the best of our knowledge never been done before. The performance of GPU accelerated vector median filter is compared to that of the CPU and MPI-based versions for different image and window sizes. Initial findings of the study showed over 100x improvement of performance of vector median filter implementation on GPUs over CPU implementations and further speed-up is expected after more extensive optimizations of the GPU algorithm.

1.0 INTRODUCTION

Eliminating noise effectively is an important step for most image processing tasks. Median Filters [1] are one class of effective non-linear noise removal techniques for gray scale images thanks to their edge preserving capability. Another important property of median filters is that the output image from the filter does not contain any synthesized pixels, in other words all of the output pixels can be found in the input image. For three channel color image denoising, one way is to apply median filter to each channel separately (Marginal ordering vector median filter) [2-3]; however this technique results in the loss of no synthesized pixels property. Another widely used technique is vector median filter [4], in which color values of pixels are treated as 3-component vectors and the vector median of a filter kernel is computed to be the one that has the smallest sum of distances to other vectors in the kernel. By applying vector median filter to color images, the no synthesized pixel property is also satisfied. Although an effective filtering technique, median filters are computationally expensive. For an implementation with a kernel width of n, each of the n^2 vector has to be compared to other $n^2 - 1$ vectors in distances [5]. Each Euclidean distance (L2-norm) calculation involves 8 floating point operations and a square root operation. For a kernel window size 3x3, the total number of operations is equal to 576 floating point and 72 square root operations.

General purpose computation on graphics processing units (GPGPU) can be described as a paradigm of utilizing high-performance many-core graphics processing units (GPUs) for computation tasks that are normally handled by CPUs. With the transition from fixed to programmable graphics pipeline, software developers gained the ability to use multiple computational cores on a GPU for non-graphics data without the explicit need of managing parallel computation elements such as threads, shared memory, and message passing interfaces. Initially, GPGPU applications suffered from limitations and difficulties arising from using graphics API elements such as vertex and pixel shaders [6] to perform non-graphics computations. To address this issue, three widely accepted solutions have been proposed: the open industry standard Open Computing Language (OpenCL) [7] framework, Microsoft's DirectCompute, and NVIDIA's Compute Unified Device Architecture (CUDA) [8]. CUDA is an extension to the C programming language for massively parallel computing using GPUs. With the introduction of CUDA and the other architectures, software developers were able to perform GPGPU without the in-

depth knowledge of programmable graphics shaders.

In this work, our main contribution is implementing vector median filter using the CUDA programming paradigm and applying CUDA specific optimizations. The performance of the implemented filter is compared to the single thread CPU and multi-processor MPI versions with respect to different image and kernel sizes.

The remainder of the paper is organized as follows. Section 2 describes the definition of vector median filter, previous attempts for accelerating vector median filters, and CUDA and MPI implementations. Section 3 compares the different implementations of the filter in terms of performance. Finally, Section 4 concludes the paper and discusses future work.

2.0 BODY

2.1 Vector Median Filters

Non-linear filters such as Bilateral [9] and Median filters are important image processing techniques of gray scale and colored image processing because of their ability to preserve edge, line, and other image structures while removing noise artifacts. Vector median filter performs non-linear filtering by moving a window over a pixel (Fig. 1) (with RGB channels) and selects the pixel that has the smallest sum of distance to the other pixels in the window as the output [4].

Given a window that contains $N = n \times n$ pixels denoted by $W = \{x_1, x_2, \ldots, x_N\}$ the output of vector median filter x_{VM} is computed by Eq. (1).

$$\sum_{i=1}^{N} \|x_{VM} - x_i\| \leq \sum_{i=1}^{N} \|x_j - x_i\|, j = 1, \ldots, N, (1)$$

where $\|\cdot\|$ denotes the distance metric between the vectors. In this paper Euclidean distance (L2-norm) is used to determine the distance between two vectors. The Euclidean distance between two vectors u and v that are in $\mathbb{R}^p$ is given in Eq. 2.

$$\|u - v\| = \sqrt{\sum_{k=1}^{p} (u_{(k)} - v_{(k)})^2}. \qquad (2)$$

Figure 1. An example 5x5 window. This window slides over the target pixels and compute the output according to the 25 pixel values inside.

Variations of vector median filters have been developed to address wide variety of

problem domain. Weighted vector median filters [10] fuzzy vector median filters [11] are two variations of vector median filter that have been successfully deployed in a number of applications.

The computational complexity of vector median filter makes it very challenging to be used for large problems that have stringent time requirements. To address this issue, there have been numerous attempts to accelerate the vector median computation by different means.

In the work of Boudabous et al. [12], a parallel architecture of the vector median filter was implemented in an embedded system. The time consuming steps of the filter were implemented in hardware level, specifically by programming Field Programmable Gate Array (FPGA) devices with VHSIC Hardware Description Language (VHDL).

Another approach to accelerate the vector median filter was proposed by Barni and Cappellini [13]. In this work, the performance of the filter is increased by using the L1-norm distance metric instead of the L2-norm. In addition to using the simplified distance metric, another simplification the authors adopted is using a central color for comparisons. In other words, for a window of pixels, the output pixel is chosen to be the one that is closest to the central color that is obtained by component wise application (marginal) of median filter to the color channels. These simplifications increase the performance of the filter significantly, however at a cost of decreased quality of the output.

There are other attempts to accelerate vector median filters by utilizing different distance metrics. For a list of such work and their computational complexity, the reader is encouraged to refer to the work of Barni and Cappellini [5].

2.2 A Brief CUDA Primer

For more than two decades, the end users of computer systems with single central processing units (CPU) enjoyed the increasing performance of their applications with each new generation of CPUs. The equation was simple, as the clock frequency of the CPUs increased with each generation; the very same application was able to run faster on the new architecture. However, this profile of increasing speeds has slowed down in 2003 due to the power consumption and heat dissipation issues [14]. The responses of CPU manufacturers to address these limitations were producing multi-core CPUs having similar clock frequencies with previous generations. With this adopted strategy, the expectation of increased performance with new generation of CPUs vanished especially for the so called sequential applications that rely on a single CPU. In order to satisfy the performance demand of the end users, application developers need to delve into the art of parallel programming, which is typically performed on large-scale expensive parallel computers, such as clusters.

Since 2003, microprocessor manufacturers adopted two main strategies for their processor designs. The multi-core strategy (CPU designs) provides small number of large cores (typically 2-4 cores) that try to maximize the execution speed of sequential programs with their large control logic elements and cache structures. On the other hand, designs that adopt many-core strategy (GPU designs) try to maximize the floating point calculation throughput by devoting more processor area (typically 120-240 cores) to data processing units instead of flow control logic elements and large data caches [8]. Because of these design choices, the recent ratio of theoretical computation peak of GPUs over CPUs is roughly 6.5 to 1 (1300 Gflop/s to 200 Gflop/s). When the huge difference between the two architecture's memory bandwidth is included in the equation (Fig. 2), GPUs turn out to be well suited for massively parallel applications with high arithmetic intensity, in which the same instruction is applied to multiple data [8].

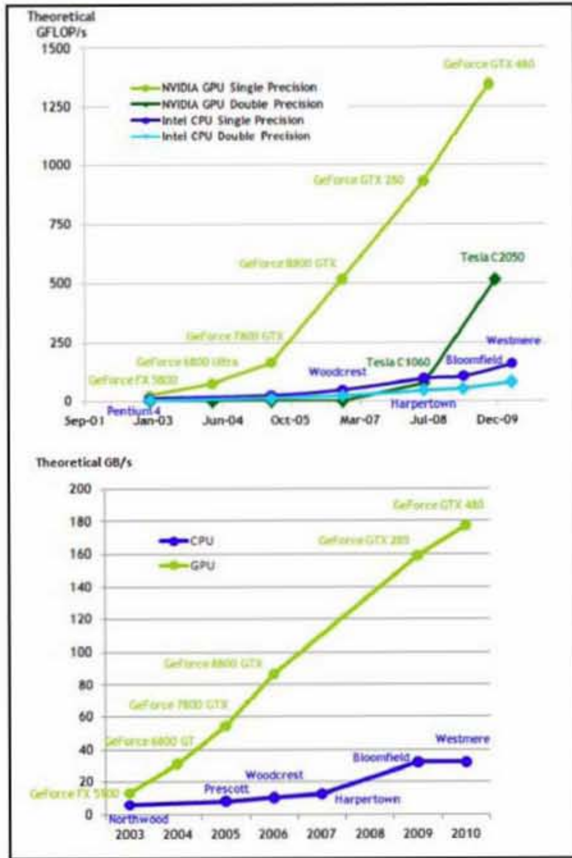


Figure 2. The comparison of GPU and CPU architectures in terms of computation power and memory bandwidth [8].

General-purpose programming using graphics processing units (GPGPU) became an active research area that studies the methods and algorithms for a wide range of problem domain such as signal and image processing, physically based simulations, computational finance, and computational biology [15-16]. The first generation of GPGPU applications were difficult to implement because programmers had to use interfaces that were primarily designed for computer graphics computations such as C for graphics (Cg) by NVIDIA, High Level Shading Language (HLSL) by Microsoft, and OpenGL Shading Language (GLSL) by Khronos Group. These computer graphics oriented APIs limited the kinds of applications that can work on GPUs [14].

To address this issue, in 2007 NVIDIA released Compute Unified Device Architecture (CUDA), which is not only an extension to the C programming language, but also adding additional hardware to the chip to facilitate the ease of parallel programming [14]. In the CUDA enabled chips, CUDA programs do not go through the graphics interface. CUDA requests are handled by a new general-purpose parallel programming interface located on the chip. CUDA relieves GPGPU application developers the necessity of having in-depth knowledge of graphics interfaces or programmable shaders.

CUDA also has significant advantages over classical parallel programming languages and models such as Message Passing Interface (MPI) for scalable cluster computing and OpenMP for shared-memory multiprocessor systems. MPI is for cluster systems, in which data sharing is done by explicit message passing, in other words processors in a cluster do not share memory. Parallel applications written using MPI model have been known to run on clusters with more than 100,000 processors. OpenMP, on the other hand, can support shared memory interface. However, it suffers from scalability issues. Parallel applications using OpenMP could not be able to scale beyond a couple hundred computing nodes mainly because of the thread management overheads [14]. Compared to these legacy models, CUDA provides a shared memory interface among the cores of a streaming multiprocessor (SM) along with higher scalability and low-overhead thread management properties.

CUDA enabled GPU architectures consist of arrays of streaming multiprocessors (SM). Different generations of GPUs contain different number of heavily threaded SMs. Each SM contains a total of 8 streaming processors (SPs) or in other words cores. Cores of the same SM share the control logic, instruction cache and fast access shared memory. GPUs support up to 4 GBs of graphics double data rate (GDDR) DRAM

that serves as the frame buffer and texture memory for 3D graphics applications. For general purpose computations, this memory space is referred as the global memory that resides off-chip and has very high bandwidth. Each SP has a multiply-add (MAD) unit, an additional multiply unit, and special function units performing floating point operations such as square roots. Because of massively threaded nature of SPs, thousands of concurrent threads can be handled for a GPGPU application. The recent GPUs with GT200 architecture can support 1024 threads per SM, which roughly sums up to about 30,000 threads for the entire chip [8].

In CUDA, fine-grained data parallelism is achieved by the massively threaded structure. Kernels are user created functions that contain statements that are executed by each individual thread. These threads are organized into a two-level hierarchy: 3D blocks consisting of individual threads and 2D grid consisting of blocks. The threads in the blocks are further divided into groups of 32 threads called warps. The notion of warp is important, because a warp is the unit of thread scheduling in SMs, i.e. when a warp is scheduled to run, all of the accompanying threads run the same single instruction. CUDA also scales transparently with the underlying hardware capabilities. Without changing the underlying code, the GPGPU application can run on different GPU hardware that may have different number of SMs or thread capacities. This important property of CUDA programs is achieved by allowing the execution of blocks in any order [8]. When a kernel is launched, the threads of the kernel are distributed among SMs on a block by block basis. Each SM can claim at most 8 blocks at a time. When more blocks are involved than the maximum number of resident blocks (# of SMs x 8) in a CUDA application, the maintained list of blocks that need to be executed is used and new blocks are assigned to SMs as they complete the execution of previously assigned blocks.

In order to obtain the maximum performance out of the CUDA capable GPUs, the memory hierarchy of CUDA has to be examined carefully. Global memory and constant memory are located at the bottom. Global memory (or device memory) supports high-bandwidth read-write access that has relatively higher access latency compared to system's RAM. Because of this relatively higher latency, it should be used wisely not to cause performance degradation. Constant memory supports short-latency high-bandwidth read-only cached access by the device. Shared memory is an on-chip memory that is allocated to thread blocks. Shared memory is a very fast parallel access enabled memory space, which is often used by the threads of the same block to cooperate by sharing their input data and intermediate results. Registers are at the top of the hierarchy. They are allocated privately for each thread and typically used to hold frequently accessed variables other than arrays. Besides these basic memory structures, CUDA also provides a texture memory space that resides in global memory but cached in texture cache. The texture cache makes use of 2D spatial locality, i.e. if threads of the same warp access texture addresses that are close in 2D space, they can achieve fast access rates [8, 14]. Special care has to be taken when accessing these memory spaces. Because of the underlying DRAM structure of global memory, it is used most efficiently when threads of the same warp access it in a certain pattern. Shared memory accesses also require caution for fast access. To support parallel access, shared memory space is divided into equally sized memory banks that can be accessed simultaneously as long as different threads in a warp access different shared memory banks. If different threads try to access memory addresses that reside in the same bank, a bank conflict occurs and the accesses are serialized. More information about memory access patterns can be found in NVIDIA CUDA C Programming Guide [8].

2.3 CUDA Implementation

To utilize the massively threaded nature of CUDA, we followed a divide and conquer approach on the input image. The image is divided into tiles of size 1 row and N columns that are assigned to thread blocks for processing (Fig. 6). Each thread therefore is responsible for accessing global memory, copying the pixel data to shared memory space (Fig. 3), and computing the median filter output for a single pixel.

```
__shared__ float dataR  [ROW_TILE_W + MEDIAN_RADIUS*2]
                        [MEDIAN_RADIUS*2+1];
__shared__ float dataG  [ROW_TILE_W + MEDIAN_RADIUS*2]
                        [MEDIAN_RADIUS*2+1];
__shared__ float dataB  [ROW_TILE_W + MEDIAN_RADIUS*2]
                        [MEDIAN_RADIUS*2+1];
```

Figure 3. Arrays located in shared memory space that store color information of pixels. The color channels are stored separately in order to access the shared memory without bank conflicts.

As median filter works in a window of pixels, individual threads are also responsible for accessing and copying the neighboring pixels. After the pixel data is copied to the shared memory by following the coalesced global memory access requirements (Fig. 4), each thread computes the median vector among the vectors in the surrounding window. The median vector computation is comprised of 4 nested loops (Fig. 5) and takes the major part of the running time. When the median vector is computed, it is written back to the global memory, which is finally read back to the CPU.

```
for(int row = -MEDIAN_RADIUS; row <= MEDIAN_RADIUS; row++)
{
    dataR[smemPos][MEDIAN_RADIUS+row] =
        tex2D(texImage, loadPos + 0.5f, blockIdx.y + row + 0.5f).x;
    dataG[smemPos][MEDIAN_RADIUS+row] =
        tex2D(texImage, loadPos + 0.5f, blockIdx.y + row + 0.5f).y;
    dataB[smemPos][MEDIAN_RADIUS+row] =
        tex2D(texImage, loadPos + 0.5f, blockIdx.y + row + 0.5f).z;
}
```

Figure 4. Loading the shared memory arrays with pixel color data. The pixels that are in MEDIAN_RADIUS neighborhood are also loaded.

```
//Cycle through median filter window, surrounding (x, y) texel
for(int j = -MEDIAN_RADIUS; j <= MEDIAN_RADIUS; j++)
    for(int i = -MEDIAN_RADIUS; i <= MEDIAN_RADIUS; i++)
    {
        sumOfDistance = 0;
        for(int m = -MEDIAN_RADIUS; m <= MEDIAN_RADIUS; m++)
            for(int n = -MEDIAN_RADIUS; n <= MEDIAN_RADIUS; n++)
                sumOfDistance += vecLenXYZ(
                    dataR[smemPos+j][MEDIAN_RADIUS+i],
                    dataG[smemPos+j][MEDIAN_RADIUS+i],
                    dataB[smemPos+j][MEDIAN_RADIUS+i],
                    dataR[smemPos+m][MEDIAN_RADIUS+n],
                    dataG[smemPos+m][MEDIAN_RADIUS+n],
                    dataB[smemPos+m][MEDIAN_RADIUS+n]
                );

        if(sumOfDistance < minDistance)
        {
            minDistance = sumOfDistance;
            clrX = dataR[smemPos+j][MEDIAN_RADIUS+i];
            clrY = dataG[smemPos+j][MEDIAN_RADIUS+i];
            clrZ = dataB[smemPos+j][MEDIAN_RADIUS+i];
        }
    }
```

Figure 5. The computation of the median vector is comprised of 4 nested loops.

2.4 MPI Implementation

To compare the performance results of the CUDA implementation, we chose to use MPI as the secondary development model. The program is executed with varying processor numbers. With the number of processors being N (1 master and N-1 slaves), the input image data is divided into N blocks and N-1 of these blocks are distributed among the accompanying slave processors (Fig. 6). After distribution is completed, the master and slave processors start computing the vector median filter. When slaves complete their assigned workloads, they send their output image data back to the master node. The outputs received from slaves are concatenated to obtain the final output image. In this parallel strategy, the slave processors do not need to communicate with each other, thus they only receive and send back a portion of the whole image.

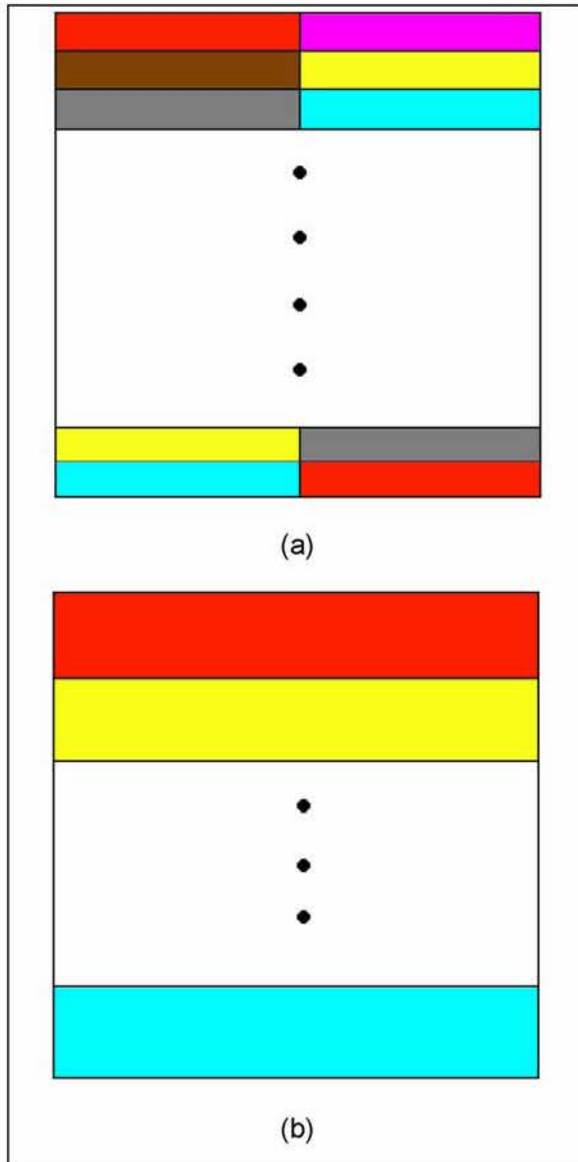


Figure 6. The comparison of parallel strategy of CUDA implementation vs. MPI implementation. (a) In the parallel strategy for CUDA, the input image is divided to equal sized tiles and assigned to individual thread blocks. (b) In MPI strategy, whole image is divided into relatively larger N parts that are distributed to accompanying processors.

3.0 DISCUSSION

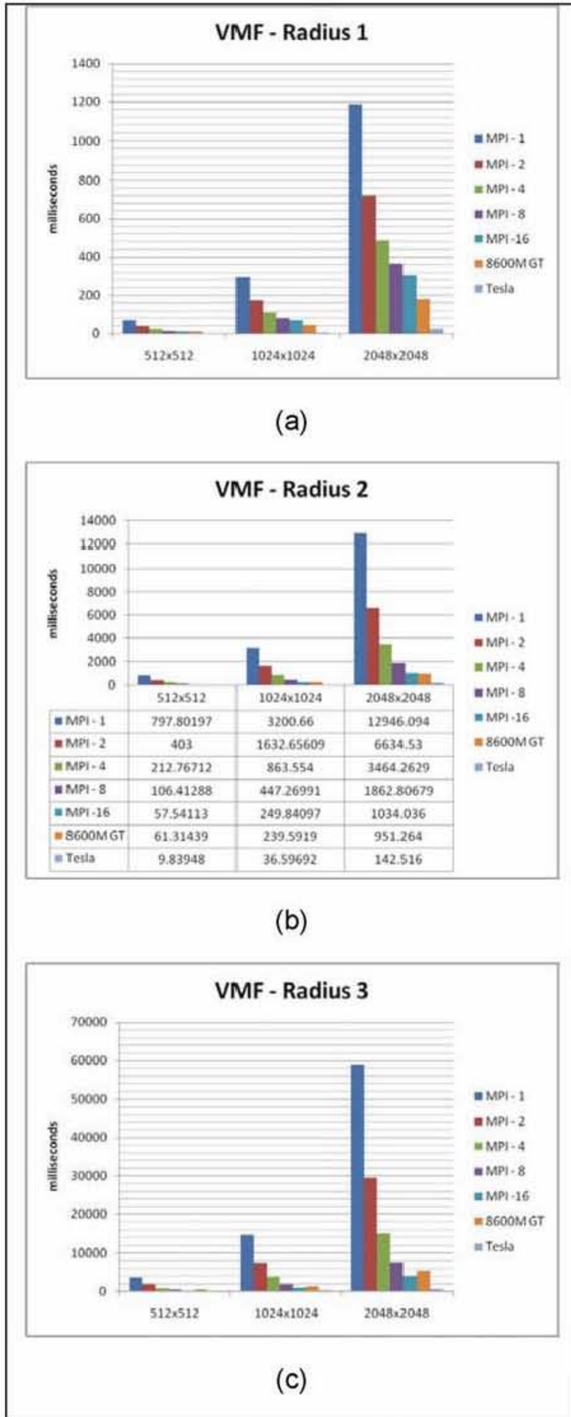
We chose different platforms to compare the performance of vector median filter. The two GPU platforms are the NVIDIA GeForce 8600M GT on a laptop computer and NVIDIA Tesla Personal Supercomputer on a workstation machine, whose specifications are given in Table 1.

Table 1. 8600M GT and TESLA GPU specifications.

	8600M GT	TESLA
# of SMs	4	30
# of Cores	32	240
Global Memory	256 MB	4 GB
Memory Bandwidth	9.25 GB/sec	102 GB/sec
Clock Rate	0.95 GHz	1.3 GHz

MPI parallel programming model is used as the platform of choice for sequential (single processor) and parallel CPU implementations. MPI code is run on Old Dominion University's High Performance Computing Cluster "Zorka" that has a number of Quad core 2.693 GHz AMD Opteron processors. In our experiments, we utilized 1, 2, 4, 8, and 16 processors.

The performance analysis of vector median filter is performed with varying image and kernel window radius sizes. Input images are 24-bit RGB images at the resolution of 512 x 512, 1024 x 1024, and 2048 x 2048 pixels. The applied kernel radius' are 1 (3 x 3), 2 (5 x 5), and 3 (7 x 7). Figure 7 presents the comparison results. Tesla platform outperformed MPI implementations in every case. We obtained close performance results when we utilized 16 MPI processors against the 8600M GT GPU. In other processor settings, the 8600M GT GPU outperformed MPI implementations.



	512x512	1024x1024	2048x2048
MPI - 1	797.80197	3200.66	12946.094
MPI - 2	403	1632.65609	6634.53
MPI - 4	212.76712	863.554	3464.2629
MPI - 8	106.41288	447.26991	1862.80679
MPI -16	57.54113	249.84097	1034.036
8600M GT	61.31439	239.5919	951.264
Tesla	9.83948	36.59692	142.516

Figure 7. The running times of Vector Median Filter (in milliseconds) on different platforms with varying image resolutions.

4.0 CONCLUSION

Vector median filter is a long used effective noise eliminating filter. In addition to removing noise and other artifacts, it is also capable of preserving edges and important features in an image. One drawback of vector median filter is its computational inefficiency. In this work, we demonstrated the extensive computing power of GPUs and harnessed this power to accelerate the Vector Median Filter computations with NVIDIA's Compute Unified Device Architecture (CUDA). For comparison purposes, filter was also implemented by using Message Passing Interface and run on a high performance computing cluster with varying processor numbers.

CUDA implementations were superior in most of the cases. The MPI implementation was faster than the 8600M GT laptop GPU when it was using 16 processors and working on smallest image size (512 x 512). The average speed-ups of Tesla GPU over MPI implementations are presented in Figure 8.

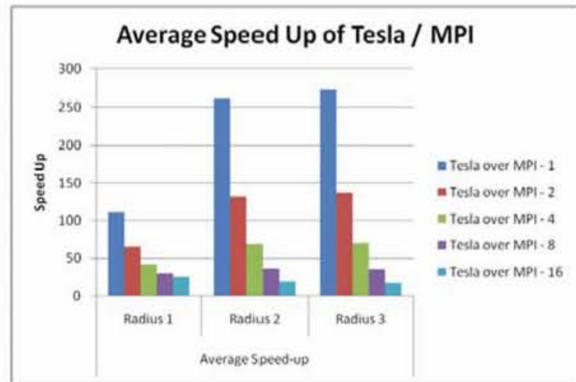


Figure 8. The average speed-ups of Tesla implementation over MPI implementations.

We have ported the computationally intensive Vector Median Filter to massively parallel GPU environment by using NVIDIA's CUDA. In order to obtain the maximum achievable performance, a CUDA developer not only has to take into account many parameters, but also has to rethink the algorithm in a parallel fashion. Although we believe that we produced decent results

for the vector median filter using CUDA, a near-term goal is to further optimize this implementation.

5.0 REFERENCES

[1] T. Sun and Y. Neuvo, "Detail-preserving median based filters in image processing," *Pattern Recognition Letters,* vol. 15, pp. 341-347, 1994.

[2] V. Barnett, "The ordering of multivariate data," *Journal of the Royal Statistical Society. Series A,* vol. 139, pp. 318-355, 1976.

[3] I. Pitas and P. Tsakalides, "Multivariate ordering in color image filtering," *Circuits and Systems for Video Technology, IEEE Transactions on,* vol. 1, pp. 247-259, 295-6, 1991.

[4] J. Astola, P. Haavisto, and Y. Neuvo, "Vector median filters," *Proceedings of the IEEE,* vol. 78, pp. 678-689, 1990.

[5] M. Barni and V. Cappellini, "On the computational complexity of multivariate median filters," *Signal Processing,* vol. 71, pp. 45-54, 1998.

[6] R. J. Rost, *OpenGL(R) Shading Language (2nd Edition)*: Addison-Wesley Professional, 2005.

[7] Khronos Group, *The OpenCL Specification Version 1.0*. Khronos Group, 2009.

[8] NVIDIA, "NVIDIA CUDA C Programming Guide," 2010.

[9] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*, Washington, DC, USA, 1998, p. 839.

[10] K. Oistamo, Q. Liu, M. Grundstrom, and Y. Neuvo, "Weighted vector median operation for filtering multispectral data," in *Systems Engineering, 1992., IEEE International Conference on*, 1992, pp. 16-19.

[11] Y. Shen and K. E. Barner, "Fuzzy vector median-based surface smoothing," *Visualization and Computer Graphics, IEEE Transactions on,* vol. 10, pp. 252-265, 2004.

[12] A. Boudabous, L. Khriji, A. B. Atitallah, P. Kadionik, and N. Masmoudi, "Efficient architecture and implementation of vector median filter in co-design context," *Radio Engineering,* vol. 16, pp. 113-119, 2007.

[13] M. Barni and V. Cappellini, "A computationally efficient implementation of the L_1 vector median filter," in *Digital Signal Processing Proceedings, 1997. DSP 97., 1997 13th International Conference on*, 1997, pp. 283-286 vol.1.

[14] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors*, 1 ed.: Elsevier, 2010.

[15] M. Pharr and R. Fernando, *Gpu Gems 2: Programming techniques for high-performance graphics and general-purpose computation*: Addison-Wesley Professional, 2005.

[16] K. Fatahalian and M. Houston, "GPUs: A closer look," *Queue,* vol. 6, pp. 18-28, 2008.

MODSIM WORLD Conference & Expo

October 13–15, 2010
Hampton, Virginia

GPU Accelerated Vector Median Filter

Rifat Aras and Yuzhong Shen

Department of Modeling, Simulation, and Visualization Engineering

Old Dominion University

October 15, 2010

1

MODSIM WORLD Conference & Expo

Outline

- Vector Median Filter
  - Working Principle
  - Formal Description
  - Computational Complexity
- Our Contribution
  - Why GPU Acceleration?
  - GPGPU
  - CUDA Paradigm
- Experiments – Results
- Conclusions

2

MODSIM WORLD Conference & Expo

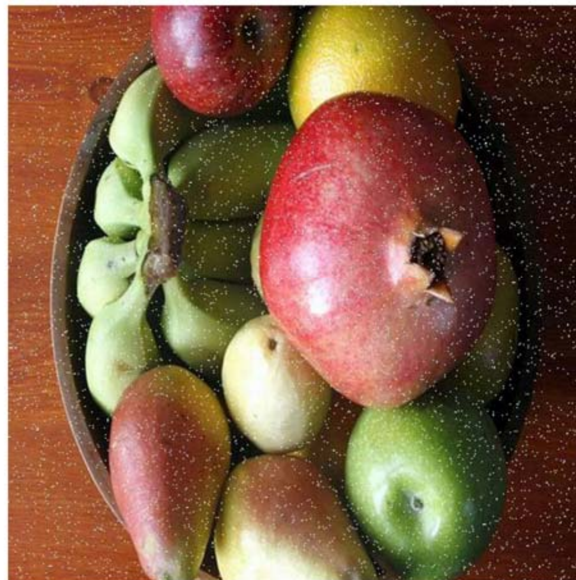
Vector Median Filter

- An effective noise elimination step for image processing tasks

T. Sun and Y. Neuvo, "Detail-preserving median based filters in image processing," *Pattern Recognition Letters,* vol. 15, pp. 341-347, 1994.

3

MODSIM WORLD Conference & Expo



4

MODSIM WORLD
Conference & Expo

Preserving edges, lines, and other image structures while removing noise artifact

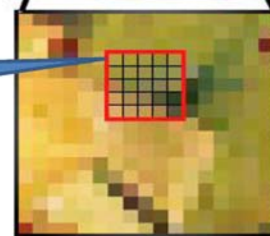
5

MODSIM WORLD
Conference & Expo

Working Principle

- For a window over a pixel, VMF selects the pixel that has the smallest sum of distance to the other pixels in the window.

5x5 Window (Window radius is 2)



MODSIM WORLD Conference & Expo

Formal Description

Given a window that contains N = nxn pixels denoted by $W=\{x_1,x_2,\ldots,x_N\}$ the output of vector median filter x_{VM} is computed by

$$\sum_{i=1}^{N}\|x_{VM}-x_i\| \le \sum_{i=1}^{N}\|x_j-x_i\|, j=1,\ldots,N, (1)$$

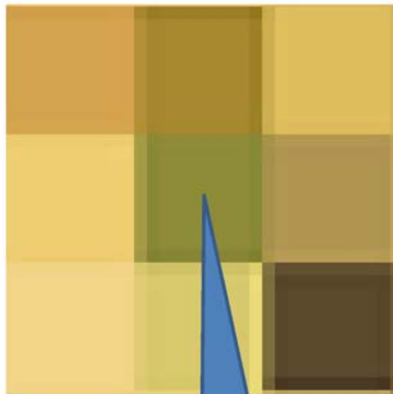
||.|| denotes the distance metric between two vectors. In this work, Euclidean distance (L2-norm) is used.

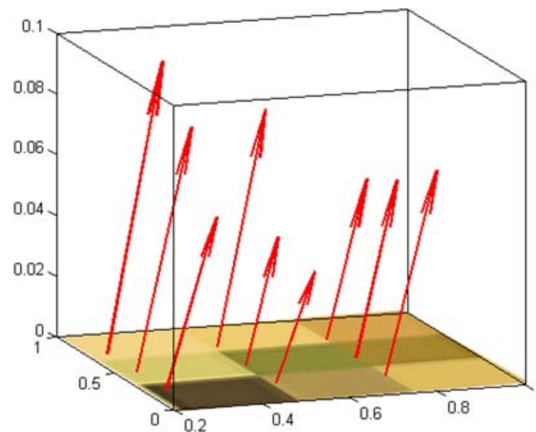
$$\|u-v\| = \sqrt{\sum_{k=1}^{p}(u_{(k)}-v_{(k)})^2}. \qquad (2)$$

7

MODSIM WORLD Conference & Expo

Vectors in an Image



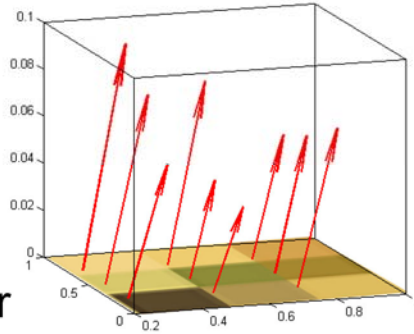


8

MODSIM WORLD
Conference & Expo

Computational Complexity

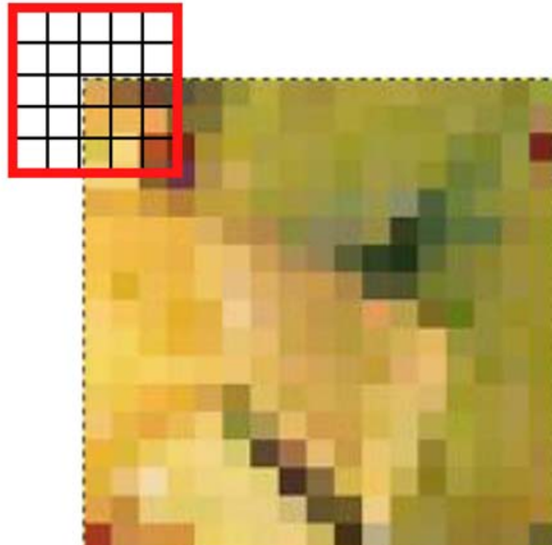
- For the nxn window case, each n^2 vector is compared to other n^2-1 vectors.
- Each Euclidean distance calculation involves 8 floating point operations and a square root.
- For 3x3 case, the total number of operations is equal to 576 floating points and 72 square roots.



9

MODSIM WORLD
Conference & Expo

Sequential Implementation



10

MODSIM WORLD Conference & Expo

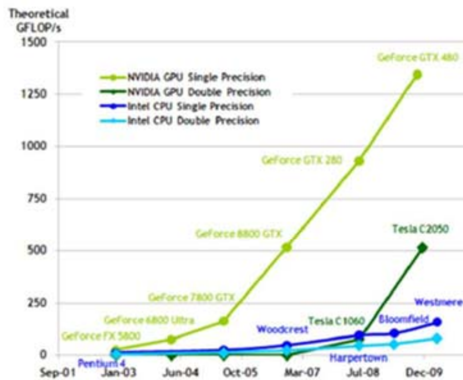
Main Contribution

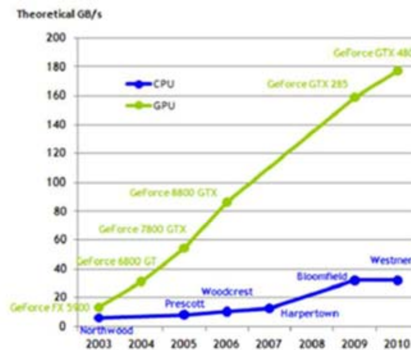
- Implementing VMF using the CUDA programming paradigm
- Comparing the performance of GPU accelerated implementation to the single thread CPU and multi-processor MPI versions

11

MODSIM WORLD Conference & Expo

Why GPU Acceleration?





12

MODSIM WORLD Conference & Expo

General-purpose Programming Using Graphics Processing Units (GPGPU)

- GPUs are well suited for massively parallel applications with high arithmetic intensity.
- Wide range of problem domain: signal and image processing, physically based simulations, computational finance, biology...
- Difficulties in first generation GPGPU applications

13

MODSIM WORLD Conference & Expo

CUDA Paradigm

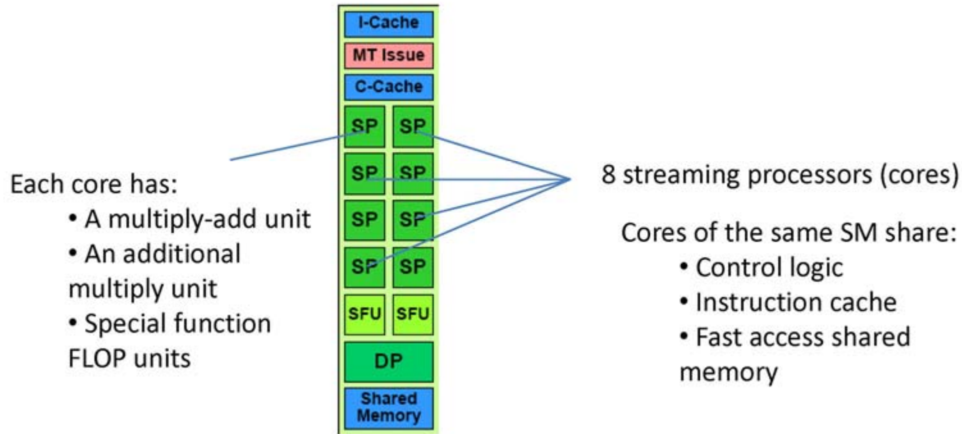
- In 2007, NVIDIA introduced Compute Unified Device Architecture (CUDA)
  - Extension to the C language
  - Additional hardware to the chip
- CUDA programs (kernels) do not go through the graphics interface.
- Allows programmers to implement parallel code on GPUs without direct knowledge of a graphical programming language.

14

MODSIM WORLD Conference & Expo

CUDA Enabled GPU Architectures

- Arrays of Streaming Multiprocessors (SMs)



GPUs with GT200 architecture can support 1024 concurrent threads per SM.

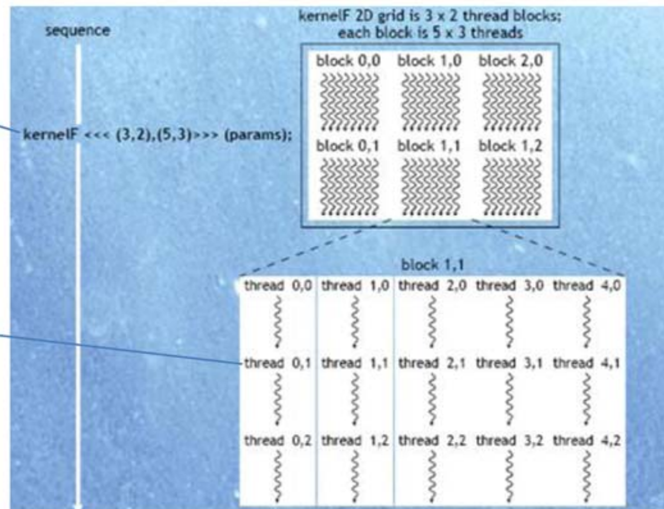
15

MODSIM WORLD Conference & Expo

Fine-grained Data Parallelism

Kernels: User created functions that are executed by each individual thread

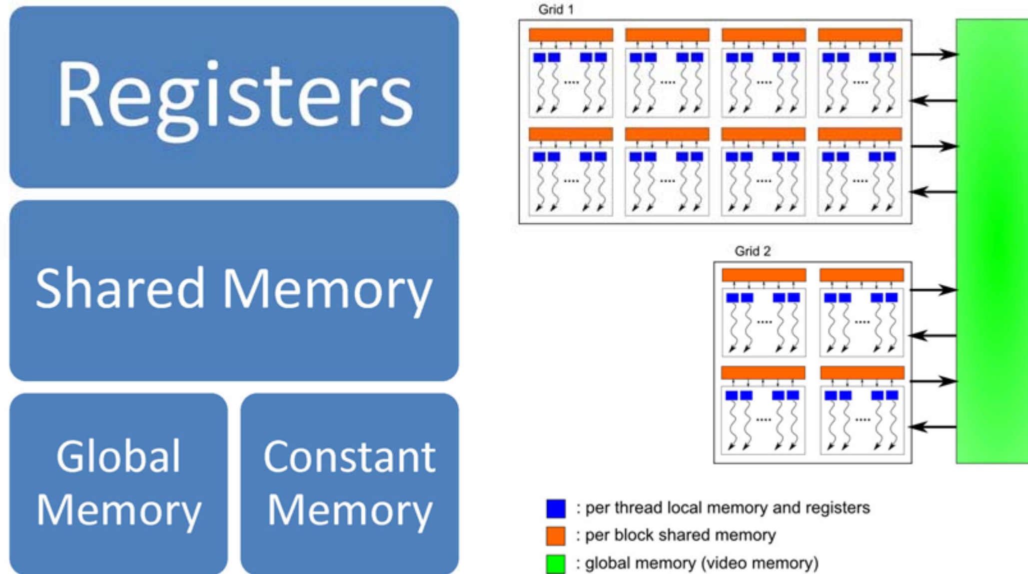
Threads are organized into a two-level hierarchy:
- 2D grid consisting of blocks
- 3D blocks consisting of threads



16

MODSIM WORLD Conference & Expo

CUDA Memory Hierarchy



17

MODSIM WORLD Conference & Expo

CUDA Examples

- **Accelerating SQL Database Operations on a GPU with CUDA** (Peter Bakkum / Kevin Skadron) **x70 Speed Up**
- **GPU Accelerated Analysis of Financial Markets** (Tobias Preis) **x80 Speed Up**
- **Syntetic Aperture Radar Range-doppler Algorithm using CUDA** (Carmine Clemente) **x15 Speed Up**

18

MODSIM WORLD Conference & Expo

Experiment Setup

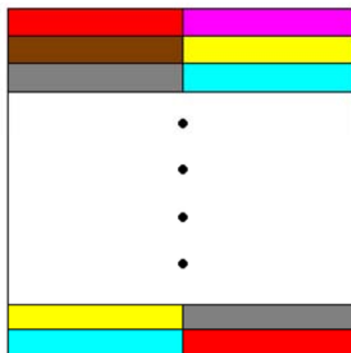
- 3 different image sizes: 512x512, 1024x1024, and 2048x2048
- 3 different window radius: 1(3x3), 2(5x5), and 3(7x7)
- Two GPU platforms
  - NVIDIA 8600M GT (Mobile GPU)
  - NVIDIA Tesla Personal Supercomputer
- 1, 2, 4, 8, and 16 MPI Processors (Quad core 2.693 GHz AMD Opterons).

19

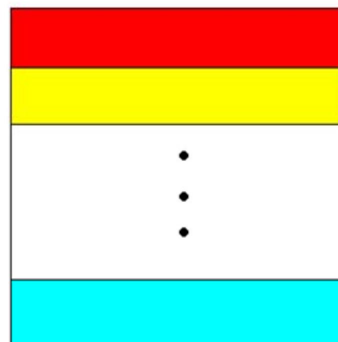
MODSIM WORLD Conference & Expo

Parallel Strategies

CUDA Strategy



MPI Strategy



20

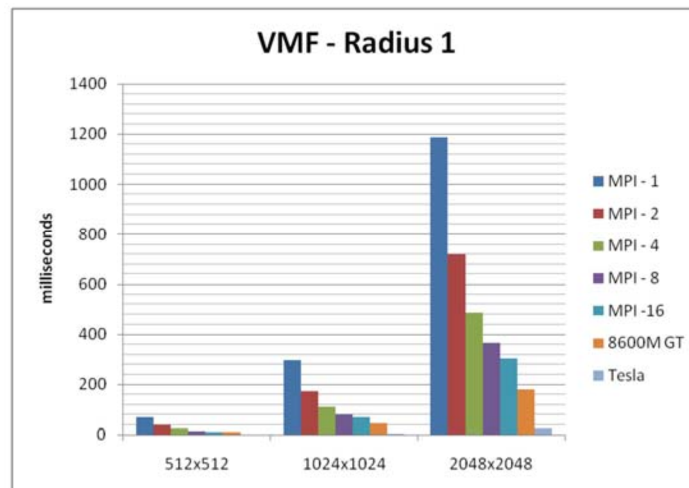
MODSIM WORLD Conference & Expo

GPU Specifications		
	8600M GT	**TESLA**
# of SMs	4	30
# of Cores	32	240
Global Memory	256 MB	4 GB
Memory Bandwidth	9.25 GB/sec	102 GB/sec
Clock Rate	0.95 GHz	1.3 GHz

21

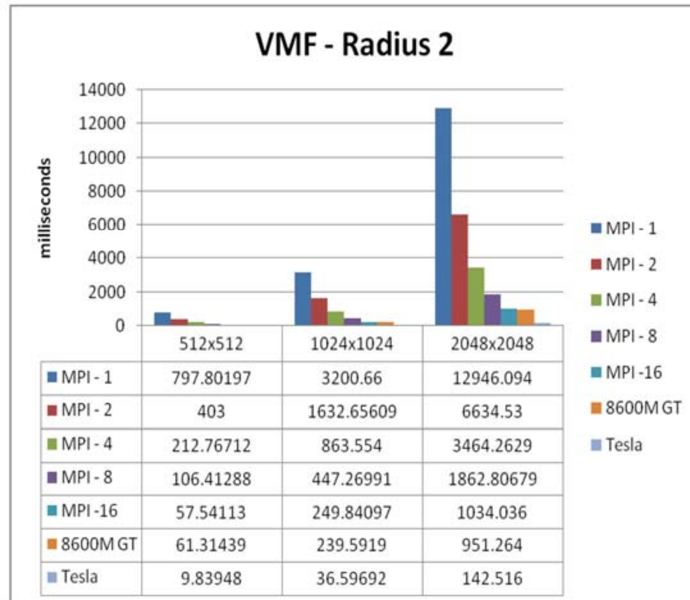
MODSIM WORLD Conference & Expo

Results – VMF Window Size: 3x3



22

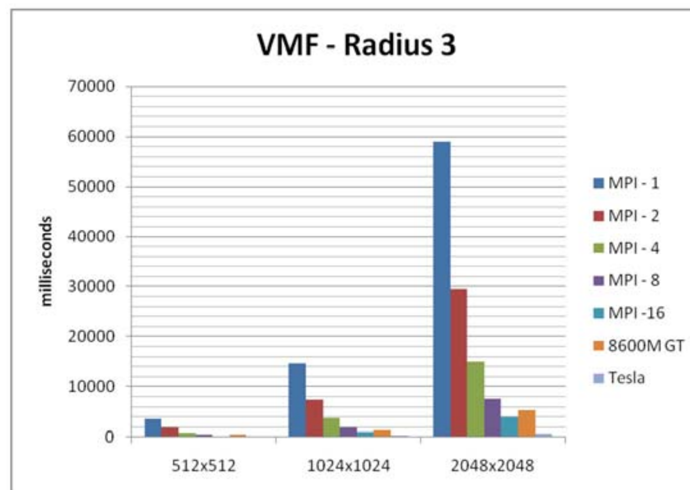
Results – VMF Window Size: 5x5



	512x512	1024x1024	2048x2048
MPI - 1	797.80197	3200.66	12946.094
MPI - 2	403	1632.65609	6634.53
MPI - 4	212.76712	863.554	3464.2629
MPI - 8	106.41288	447.26991	1862.80679
MPI -16	57.54113	249.84097	1034.036
8600M GT	61.31439	239.5919	951.264
Tesla	9.83948	36.59692	142.516

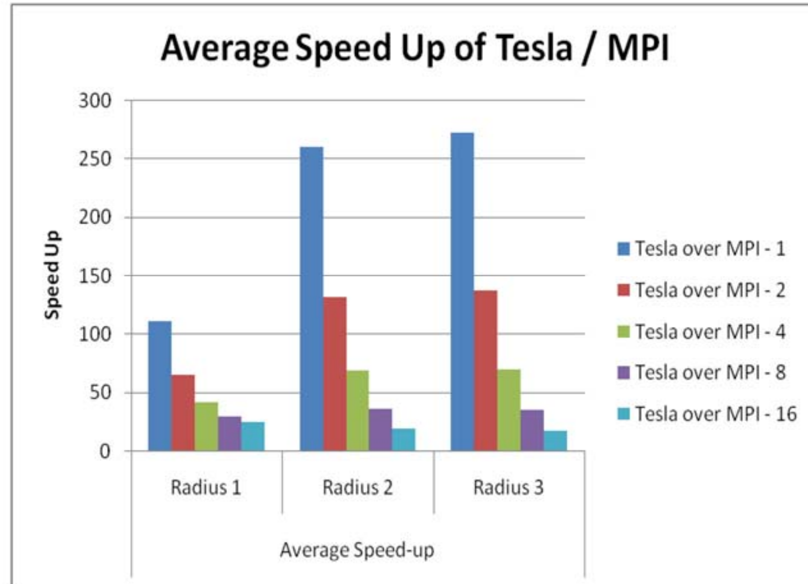
23

Results – VMF Window Size: 7x7



24

MODSIM WORLD Conference & Expo



25

MODSIM WORLD Conference & Expo

Conclusions

- Vector Median Filter: Removing noise, preserving edges and other important image features
- High computational complexity
- We have ported VMF to massively parallel GPU environment using CUDA.
- Obtained 100x – 275x of speed-up over single processor implementation

26

Thank you...

Questions and Comments?

27