A Novel Computation-to-core Mapping Scheme for Robust Facet Image Modeling on GPUs

Yong Cao Seung-In Park Layne T. Watson

***Abstract*— Though the GPGPU concept is well-known in image processing, much more work remains to be done to fully exploit GPUs as an alternative computation engine. The difficulty is not reformulating the algorithm and writing the code so that the program can run in parallel. The bigger challenge is achieving good GPU utilization, which requires a careful implementation armed with in-depth knowledge of the performance characteristics of the underlying architecture. This paper shows how to optimize the computational parallelism in robust facet image modeling to GPU architecture, using fine-grained block level parallelism achieved by assigning more GPU cores/threads to process one pixel, rather than pixel level parallelism. The mapping strategy dependence on the computational profile is characterized.**

Keywords: Facet image modeling, Robust estimation, GPGPU, Computation-to-core mapping.

I. Introduction

A long standing challenge to the field of image processing is that massive computational power is required in order to achieve higher speed. It is often the case that an image processing algorithm, such as robust facet image modeling, is theoretically sound but not useful for real-world applications due to the computational resource and time requirements. Many applications have successfully leveraged the computational power of graphics processing units (GPUs) toward their real-time requirements in image processing. It is well known that a GPU is a many-core processor capable of high performance parallel computation and data throughput, which is an ideal implementation platform for image processing algorithms.

Many image processing tasks perform the same operation on each pixel of the input image, a typical data parallel scenario. Therefore, they are readily mapped to the parallel architecture of GPUs in a fashion that the processing of each pixel is in parallel. In some cases, such a computation-to-core mapping scheme is simple and efficient. However, in some other cases, such as robust facet image modeling, the amount of computation for each pixel is too heavy for a GPU core to execute efficiently. Sometimes, such a mapping also results in a GPU resource deficit, which makes the processing of large images impossible.

This paper provides a detailed illustration of the disadvantages of a pixel-level mapping scheme on a GPU implementation of the robust facet image modeling algorithm. A novel computation-to-core mapping scheme that seeks fine-grained parallelism by asking multiple GPU cores to process one pixel is proposed. This novel type of parallelism, called *block-level facet processing*, greatly enhances the efficiency of GPU resource usage, resulting in a substantial performance gain. The proposed block-level processing implementation out-performs the standard pixel-level mapping scheme by a factor of 32, and the overall performance gain from the GPU implementation is a speedup of 159 compared with a standard CPU implementation.

The rest of the paper is organized as follows. Section 2 describes the architecture and programming model of GPUs. Several previous studies on image processing using GPUs are reviewed in Section 3. Section 4 illustrates the robust facet image modeling algorithm and analyzes the computational characteristics of the algorithm. Section 5 explores two implementation designs differing in the mapping of data elements to GPU processing units. Section 6 discusses how the mapping affects the performance and what mapping strategy should be taken depending upon the computational profile and purpose, and Section 7 summarizes and describes future work.

II. GPU Architecture

This work is concerned with programming GT200-series NVIDIA GPUs, which come with the CUDA (Compute Unified Device Architecture) programming framework and instruction set architecture. The CUDA device is built as a collection of streaming multiprocessors, each of which consists of eight SIMT (single-instruction, multiple-thread) stream processors. The SIMT architecture allows each stream processor in a multiprocessor to run the same instruction on different data independently, making it ideal for data-parallel computing. This section will discuss the thread organization of CUDA and GPU memory hierarchy. A detailed description of the programming model and architectural specification can be found in [1] and [8].

A. Thread Organization

CUDA manages a large number of computing *threads* and organizes them into a set of logic *blocks*. Each thread block can be mapped onto one of the multiprocessors for execution. The number of threads allowed in a block depends on some hardware limitation and the computational resources required by each thread in the block. Blocks are further organized into *grids*. The threads within one grid all execute the same *kernel function*, and the thread grids are scheduled to run sequentially on the GPU.

Yong Cao, Seung-In Park and Layne T. Watson are with the Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA, 24061. Layne T. Watson is also with the Department of Mathematics. Emails: {yongcao, spark80, ltw}@vt.edu. **Yong Cao** is the contact author. The conference name is **PDPTA'10**.

In terms of scheduling, a group of 32 threads forms a *warp*, which is the minimum thread set that is scheduled independently to run on multiprocessors in parallel. Since each multiprocessor has only one instruction fetch unit, all threads in a warp must execute the same instruction in a GPU clock cycle for the best performance. If a branch instruction causes the execution of diverged codepaths within a warp, all different codepaths have to be executed sequentially, which results in performance degradation.

B. Memory Hierarchy

Another important feature of CUDA is a memory hierarchy to hide memory and pipeline latency. CUDA threads can read from or write into multiple memory spaces at different access speeds during the execution: *local*, *shared*, or *global memory*. Local memory access is allowed exclusively for some automatic variables, therefore the user only has control over shared and global memory management. Shared memory is on-chip memory with very fast access that is partitioned among all the threads in the same block. Global memory can be accessed by all threads across all multiprocessors but is costly; global memory latency is 400 to 600 cycles while shared memory latency is 10 cycles. Consequently global memory access should be minimized as much as possible for the best performance. However, only 16KB of shared memory is given per multiprocessor, and if enough shared memory to process the kernel function in at least one thread block is not available, then the kernel launch will fail. In addition to these read-and-write memory spaces, threads also can access read-only *constant* and *texture memory* as shown in Figure 1.

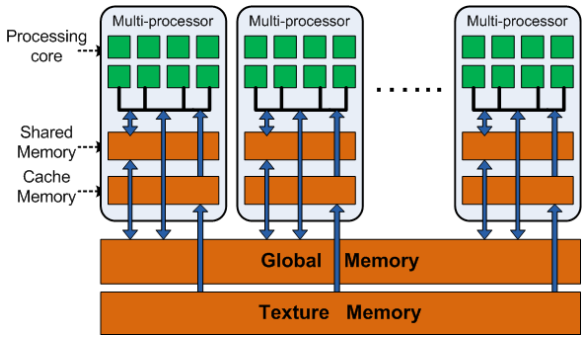

Fig. 1. CUDA memory hierarchy.

III. Related Work

Several applications of GPU technology for image processing have already been reported in the literature. Mizukami and Tadamura [7] proposed implementation of Horn and Schunck's regularization algorithm with a multiscale search method for optical flow computation based on the CUDA programming framework. Making use of on-chip shared memory, they were able to get speedup of approximately 16 using a NVIDIA GeForce 8800GTX card over a 3.2-GHz CPU on Windows XP. Bui and Brockman [3] presented a 2-D rigid image registration pyramid-based algorithm on CUDA and examined the performance efficiency of their work. They reported speedup of 90 with bilinear interpolation and speedup of 33 with bicubic interpolation. They profiled the data to identify performance bottlenecks of the CUDA platform and emphasized the need to manage memory resources carefully to obtain maximum speedup. Smelyanskiy et al. [9] implemented subvolume based ray-casting for volume rendering on an NVIDIA GTX280 and achieved speedup of 5 to 8 over the scalar baseline version running on a single core Harpertown. These efforts demonstrated that the GPU is an effective platform for image processing and computer vision, but did not present a strategy to port the image processing algorithm onto a GPU platform for optimal performance. Although Bui addressed the optimization in terms of memory management, this is already a generally known issue for GPU programming.

IV. Robust Facet Image Modeling

The concept of facet image modeling was introduced by Haralick and Watson [5]. Besl et al. [2] proposed a robust window operator to yield good model estimates for facets when the sample data are contaminated with more than one statistical distribution. The algorithm applies robust statistics to minimize the error between the underlying gray level model and the observed data from the image. While there are a wide range of applications of robust facet image modeling such as edge detection, background normalization, and image segmentation, the computational requirement is very high and increases rapidly as the order of the model increases. The remainder of this section sketches the algorithm.

A. M-estimation

The robust window operator estimates the parameters of the underlying facet model for a given two-dimensional $n \times m$ window centered at the pixel with local coordinates (0,0); the model function $f(r,c)$ at pixel (r,c) is a linear combination of (polynomial) basis functions ϕ_i,

$$f(r,c) = \sum_{i=1}^{p} a_i \phi_i(r,c), \tag{1}$$

where p is the dimension of the vector space generated by the ϕ_i.

To find the coefficient vector a of the fitting function, M-estimation minimizes the residual error

$$E(a) = \sum_{r=-n'}^{n'} \sum_{c=-m'}^{m'} \rho\left(\frac{d(r,c) - f(r,c)}{s}\right), \tag{2}$$

$$n' = \frac{n-1}{2}, \qquad m' = \frac{m-1}{2},$$

where $d(r,c)$ is observed data, ρ is a symmetric, monotone increasing function with $\rho(0) = 0$, and the scaling factor s is evaluated using the median absolute deviation (MAD).

The optimal coefficient vector a is found by minimizing $E(a)$. Choosing ρ so that its derivative is the Huber minimax

function, $\nabla E(a) = 0$ can be written as

$$\sum_{r=-n'}^{n'} \sum_{c=-m'}^{m'} \sum_{k=1}^{p} w(r,c)\phi_i(r,c)a_k\phi_k(r,c) = \sum_{r=-n'}^{n'} \sum_{c=-m'}^{m'} d(r,c)w(r,c)\phi_i(r,c), \quad i = 1, \ldots, p, \tag{3}$$

where the weight $w(r,c)$ is defined as $\rho'\big(e(r,c)\big)/e(r,c)$, $e(r,c) = \big(d(r,c) - f(r,c)\big)/s$. Equation (3) in matrix form is

$$\Phi^t W \Phi \; a = \Phi^t W \; d, \tag{4}$$

which is a nonlinear equation in a because the weight matrix W depends on these coefficients. Φ is a $nm \times p$ matrix whose rows are $\phi_1(r,c)$, ..., $\phi_p(r,c)$. W is a $nm \times nm$ diagonal matrix whose diagonal elements are $w(r,c)$, a is a p-vector whose entries are a_i, and d is a nm-vector whose entries are the observed image data. Iteratively reweighted least squares (IRLS) is used to solve this nonlinear matrix equation via the recurrence formula

$$a^{(t+1)} = \big(\Phi^t W(a^{(t)})\Phi\big)^{-1} \Phi^t W(a^{(t)}) d, \tag{5}$$

where t is the iteration number.

B. Iterative Reweighted Least Squares (IRLS)

The IRLS process for polynomial models of each order occurs iteratively. To initialize the iteration, an initial fit coefficient vector $a^{(0)}$ is needed. $a^{(0)}$ for zero-th order is the median value of the observed data, and is set with the previous order fit coefficient vector for higher order, e.g., the final planar fit initializes the quadratic fit.

IRLS uses the QR decomposition to solve Equation (5). A QR decomposition of an $m \times n$ matrix A is a factorization $A = QR$, where Q is an $m \times m$ orthogonal matrix and R is an $m \times n$ upper triangular matrix. Among three major QR factorization algorithms—modified Gram-Schmidt, Givens, and Householder—the Householder transformation algorithm outperforms the modified Gram-Schmidt algorithm in numerical stability, and requires fewer arithmetic operations than the Givens rotation algorithm [4]. Therefore the QR factorization is done with Householder transformations, a series of orthogonal transformations applied to the input matrix A to bring it into upper triangular form. The product of these orthogonal transformations is the matrix Q^t giving $Q^t A = R$.

C. The Algorithm

Figure 2 outlines the algorithm for IRLS-based robust facet image modeling, referred to as *Robust FIM* from now on. Given an $n \times m$ window, several different order robust surface fits for a pixel are computed up to a preselected maximum order. Here three is selected as the highest degree of the fitting polynomial function, since the complexity of the fitting function is adequate enough for the most commonly used window size, which is 5×5. If the IRLS iteration yields a zero MAD of the residual (a perfect fit), the algorithm will be terminated with the estimated coefficients for the fitting function. If the maximum iteration limit is reached without convergence, the next higher degree fit is computed. At the final step, the fit quality for each degree polynomial model is evaluated, and then the fitting function with the best fit quality is chosen. Note that facet image modeling begins by computing the median value of the window; the zero-th order model (constant fit) is initialized with the median value of the observed data without performing the IRLS process. Then the first set of residual errors, scale factor, and weights are computed from the zero-th order fit to initialize the planar fit.

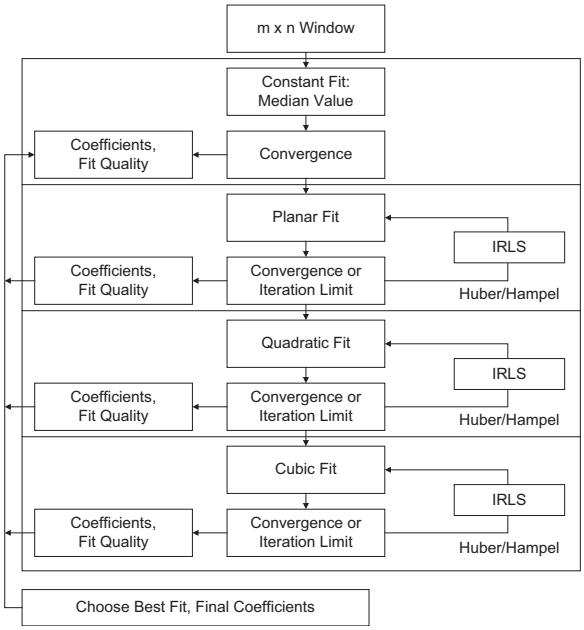

Fig. 2. Outline of IRLS-based robust facet image modeling algorithm.

Pseudo code for the *Robust FIM* algorithm can be found in Algorithm 1, illustrating the IRLS estimation process for a k-th degree fitting polynomial, which has p coefficients, for a single pixel. Algorithm 1 maintains a two-dimensional $n \times m$ observed data matrix $window$ for the pixel. The matrix E stores the residual error between the observed data and the approximation for each pixel in $window$. W is the weight matrix whose values are assigned with 'WeightFunction' of the residual error matrix E. A is a $nm \times p$ matrix, which is the multiplication of $W^{\frac{1}{2}}$ and the Gram matrix Φ of basis function values $\phi(r,c)$, and b is a nm vector, which is the multiplication of $W^{\frac{1}{2}}$ and $window$. Both A and b are needed to rewrite Equation (4) as the least squares problem $Aa \approx b$. Then A is factored into Q and R components, which are used to find coefficient vectors a with backward substitution. The function 'House' returns the transformation vector v, and the Householder reflection matrix H is computed from v. If x is an arbitrary column vector of dimension $q \leq nm$, then with $\alpha = -\text{sgn}(x_1)\|x\|$, the first $nm - q$ components of v are zero, and the remaining components of v are given by $\frac{x - \alpha e_1}{\|x - \alpha e_1\|}$, where e_1 is the canonical vector (1, 0, ..., 0)^T and $\|\cdot\|$ is the Euclidean norm. Transforming sequentially each

column of A yields an upper triangular matrix R. Details for the Householder QR decomposition algorithm can be found in [6]. The robust fit quality measure is given by the 'FitQuality' function of E, p, and $scale$. The process is repeatedly performed until yielding a zero MAD (only for a perfect fit) or reaching the maximum iteration limit.

Algorithm 1 k-th Degree Polynomial Fit for a Single Facet

Require: $window[n][m]$ of image data, $d[1:nm]$ is vector representation of $window$, p is number of coefficients
Ensure: coefficient vector $a_k[p]$

1: $E = |window - \Phi_{k-1}a_{k-1}|$
2: $i = 0$
3: **while** ($scale! = 0$ & $i <$ MAXITERATION) **do**
4: $W =$ WeightFunction$(r, scale)$
5: $A = W^{\frac{1}{2}} \times \Phi_k$
6: $b = W^{\frac{1}{2}} \times d$
7: $Q = I$
8: **for** $j = 1$ to p **do**
9: $v[1:nm] =$ House$(A[j:nm, j])$
10: $H = I - 2vv^t$
11: $Q^t = H \times Q^t$
12: $A = H \times A$
13: **end for**
14: $R = A$
15: $a_k[1:p] =$ BackwardSubstitution$(Ra_k = Q^t b)$
16: $E = |window - \Phi_k a_k|$
17: $scale = 1.4826\,$Median(E)
18: $i = i + 1$
19: **end while**
20: $fit[k] =$ FitQuality$(E, p, scale)$
21: return $a_k[1:p]$

V. Approach

This section introduces two different computation-to-core mapping schemes when implementing the robust facet image modeling algorithm, as described in Section IV, on GPUs. These two mapping schemes exhibit different levels of parallelism, thread-level facet processing and block-level facet processing. Each of the schemes has unique memory requirements, posing different hardware limitations with respect to the size of input data and the order of the fitting function. As a result, a substantial performance difference can be found between these two GPU implementations.

Consider first the thread-level facet processing scheme, since it is a straightforward choice for implementing image processing algorithms. Why the memory requirement of the thread-level mapping scheme is a limitation will be explained by providing a detailed presentation of how the memory resources in GPUs are allocated and shared between threads. The explanation of the limitation of the thread-level mapping scheme motivates the second mapping scheme, a block-level facet processing scheme, which provides an intuitive solution to the problem introduced by memory limitations of the first scheme. Performance analysis of the second mapping scheme, and the limitation of the scheme with a multiGPU processing approach are addressed. Detailed performance results for these two schemes can be found in Section VI.

A. Thread-level Facet Processing

As mentioned in Section II-A, the massive parallelism of a GPU is achieved by organizing a large number of concurrently executed threads, which are organized into thread blocks and run on the multiprocessors of the GPU. To determine the thread-block organization for a specific algorithm, the overall computation is segmented into units of operations that can be mapped onto each GPU thread. Among various criteria used for computation segmentation, independence is paramount. It is obvious that if two processing units can be executed independently, they can be scheduled to run in parallel without synchronization.

For *Robust FIM*, an independent computational unit is the processing of the facet image model of a pixel. Thus an input image with $width \times height$ pixels has $width \times heigh$ independent computational units, each of which calculates the facet model for a pixel. The first computation-to-core mapping scheme, thread-level facet processing, is based on such a computation segmentation—simply map the facet processing of one pixel onto a GPU thread.

1) Implementation of Kernel Function: We implement the overall algorithm *Robust FIM* in one kernel function that executes Algorithm 1 four times, once for each degree of the fitting function (from constant fit, $k = 0$, to cubic fit, $k = 3$). The implementation is straightforward since the computation within a CUDA kernel is sequential.All the CPU code, written in C, is simply copied for Algorithm 1 to this CUDA kernel function, using the GPU's device memory instead of the host CPU memory.

Table I lists all the required variables and their memory requirements in Algorithm 1, with two additional temporary variables T and t. The matrix T is used to store the result of matrix-matrix multiplication (at line number 11 and 12 in Algorithm 1), and the vector t is used to store the result of matrix-vector multiplication (at line number 6 in Algorithm 1). The matrix R shares the same memory space with A (A is not used after R). The $nm \times nm$ diagonal weight matrix W only requires $n \times m$ elements of memory space. p_k is the number of coefficients for the k-th degree polynomial model, e.g., $p_3 = 10$ for a cubic fit. Since *Robust FIM* calculates four fits, the vector a must hold all the coefficients from all fits. Therefore the space needed for a is $q = \sum_{i=0}^{3} p_i = 1 + 3 + 6 + 10 = 20$. Fit quality is evaluated for each polynomial model, $k = 4$. Table I also lists memory requirements for two window sizes, 5×5 and 7×7. Note that each element in the vectors and matrices has data type `float`.

Accessing shared memory is two orders of magnitude faster than global memory, so as many variables as possible should be allocated to shared memory space before reaching the space limit, which is $16K$ for each multiprocessor. The rest of the variables, mostly matrices, must use global memory.

Variable	Memory (bytes)	5×5	7×7
a	$q \times 4$	80	80
$window$	$n \times m \times 4$	100	196
E	$n \times m \times 4$	100	196
W	$n \times m \times 4$	100	196
b	$n \times m \times 4$	100	196
v	$n \times m \times 4$	100	196
t	$n \times m \times 4$	100	196
Φ	$nm \times p \times 4$	1,000	1,960
$A(R)$	$nm \times p \times 4$	1,000	1,960
H	$nm \times nm \times 4$	2,500	9,604
Q^t	$nm \times nm \times 4$	2,500	9,604
T	$nm \times nm \times 4$	2,500	9,604
fit	$k \times 4$	16	16
Total		10,196	34,004

TABLE I

MEMORY REQUIREMENT FOR EXECUTING THE *Robust FIM* ALGORITHM FOR A SINGLE FACET.

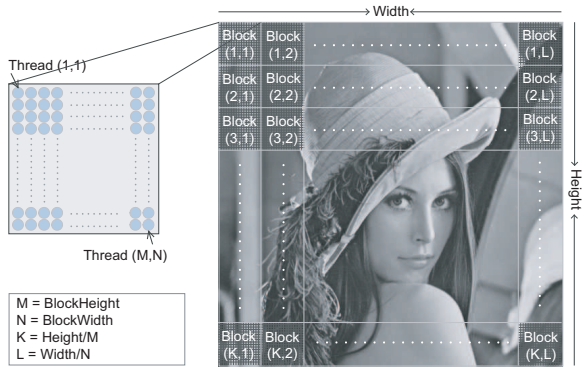

Fig. 3. Thread-Block configuration for thread-level facet processing.

2) Thread-Block Configuration: Since all threads in a block share the same shared memory space in a multiprocessor, there can only be a minimal number of threads in a block if each thread is to use as much of the shared memory space as possible for the variables listed in Table I. A 32-thread *warp* is the minimum thread set that can fully utilize the computational resources in a multiprocessor. Therefore, the thread-level facet processing approach generates 32 threads per block, and $\frac{width \times height}{32}$ blocks in total as our thread-block configuration, as shown in Figure 3.

3) Limitations: In the thread-level processing scheme, each thread requires a large amount of global memory space. For a 5×5 window, 8.7K of global memory is allocated for each thread for the variables $v, t, A(R), H, Q^t$, and T (the rest of the variables listed in Table I are allocated in shared memory). All threads are executed in parallel on the GPU, and global memory is pre-allocated for all threads before calling the CUDA kernel function. Each thread runs one instance of the algorithm on a single pixel, and the whole input image is processed with many instances of the algorithm running concurrently in their own global memory space. The required global memory for a large input image, therefore, can exceed the hardware limit. To run the thread-level processing implementation for a 5×5 window size on the GTX280 GPU, which has 1GB of global memory, the input image can not be larger than 351×351, a significant limitation for the application of *Robust FIM*.

B. Block-Level Facet Processing

To address the limitation of the thread-level mapping scheme, a novel computation-to-core mapping scheme is proposed. This mapping scheme seeks block-level parallelism where the estimation of a facet model is executed on a *block* of threads instead of only one thread. All the threads in a block work collaboratively to accelerate the linear algebra, such as matrix multiplication.

This block-level mapping scheme allows more variables of *Robust FIM* to fit into the shared memory space because only one instance of the algorithm is executed in a multiprocessor. Consequently, no or little global memory is required for each thread, obviating the limitation associated with the thread-level mapping scheme.

1) Implementation of Kernel Function: In Algorithm 1, each matrix and vector operation is segmented into computational units. These units are mapped onto different threads in a block and executed in parallel. For example, in matrix-matrix multiplication, a unit is defined as the calculation of an element in the resulting matrix, which is the inner product between a row vector from the first matrix and a column vector from the second matrix. For matrix-vector multiplication, a unit is defined as the calculation of an element in the resulting vector, which is also an inner product. All the other operations in Algorithm 1 are segmented into units of computation in a similar fashion—a block of threads covers the computation of all the units in the operation. In this implementation, each matrix/vector operation can be computed in parallel.

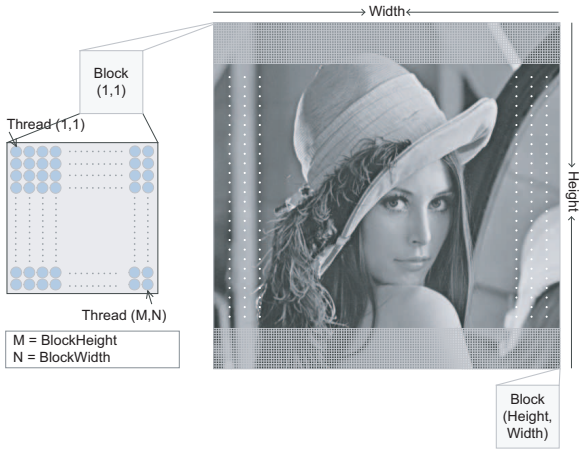

Fig. 4. Thread-Block configuration for block-level facet processing

For this block-level mapping scheme, all the variables in Algorithm 1 fit in high speed shared memory in multiprocessors. There is no global memory allocation for the

computation. At the beginning of the kernel function, we load the image data in shared memory and allocate shared memory space for the variables. During the *IRLS* iteration, all threads in the same block operate on the data in shared memory. Finally, the result is written back to global memory.

2) Thread-Block Configuration: The block-level mapping scheme generates one thread *block* for each pixel/facet, as shown in Figure 4. The block threads cover all the computational units of each matrix/vector operation. Among these operations, matrix-matrix multiplication has the largest number of units. For example, for a 5×5 window, the largest matrix is 25×25, which will result in 625 units of computation. Since the maximum number of threads per block is set to 512 as a hardware limitation, each thread has more than one unit to complete.

In CUDA, the 32-thread *warp* is the atomic resource unit that is scheduled by the GPU thread manager. The number of threads in a block should be a multiple of 32. Our experiments show that the thread block with size 16×26 yields the best performance in the block-level facet processing implementation.

3) Advantages over Thread-level Mapping Scheme: The proposed block-level mapping scheme has no global memory constraint and can be applied to any size image. This advantage over thread-level parallelism derives from the fact that the GPU manages global memory and shared memory differently.

Figure 5 illustrates this difference, where n thread blocks, TB_1, ..., TB_n, are scheduled to execute on m stream multiprocessors, SM_1, ..., SM_m. On the top half of the figure, CUDA first schedules m thread blocks, TB_1 to TB_m, assuming that only one TB can execute on a SM. On the bottom half of the figure, the second m TBs are scheduled, after the first m TBs are completed. Focusing on global memory and shared memory usage, notice that the shared memory space in SM_i is used by both TB_i and TB_{m+i}. Therefore, if the amount of shared memory in a SM is enough for a thread block, no additional memory is needed. In terms of global memory usage, however, no space can be shared between different TBs, because the scheduled order of the TBs is indeterminate. All TBs have to pre-allocate global memory space. If the image size $n >> m$, n is limited by the hardware constraint on global memory size.

VI. Result and discussion

To test the accuracy and efficiency of our GPU implementation, we perform experiments on a machine equipped with an Intel Core 2 Quad 2.33 GHz CPU, 4GB system memory, and a NVIDIA GTX280 GPU, which has 240 processor cores with 1.3 GHz clock for each core, and 1GB of device memory. To evaluate the performance on a multi-GPU system, we also use a system with 4 NIVIDA GTX 295 cards, each of which has 2 GPUs with the similar hardware specifications with GTX 280 card.

We have chosen 5 example images as our test cases. They are the canonical *Lena* picture and 4 other noisy images downloaded from the internet: *Cell*, *Airplane*, *Moon*, and *Fingerprint*. With a size of $2,048 \times 2,048$, *Lena* image is also used for performance experiments with different image sizes, where a portion of the image is used.

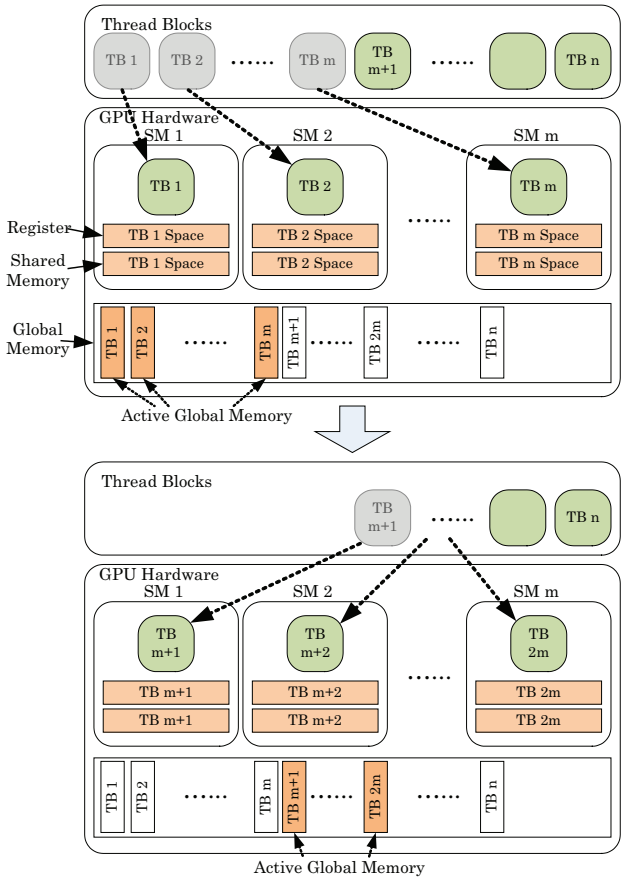

Fig. 5. CUDA Thread Block Scheduling. Top half: execution of first m thread blocks. Bottom half: execution of next m thread blocks.

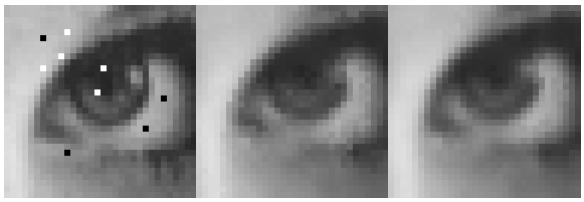

Fig. 6. Noisy input image (right), the processed images from both CPU implementation (middle) and GPU implementation (left).

A. Accuracy of the GPU Implementation

We shall first demonstrate the accuracy of our GPU implementation of the Robust FIM algorithm. In Figure 6, we show a noisy input image and two processed images resulting from CPU and GPU implementations of the algorithm. In this experiment, we use the *Lena* image with artificially added impulse noise. In Figure 6 the two resulting images are visually identical. For quantitive comparison, we calculated the normalized Root Mean Square (RMS) distance between processed images resulting from our CPU and GPU implementations. The results from all five test images are listed In Table II.

Image	RMS	CPU time (sec)	GPU time (sec)
Lena	0.0014	120.696	7.009
Cell	0.0028	142.301	7.009
Airplane	0.0014	138.882	7.003
Moon	0.0035	142.459	7.006
Fingerprint	0.0008	101.313	5.641

TABLE II
COMPARISON BETWEEN CPU AND GPU IMPLEMENTATION.

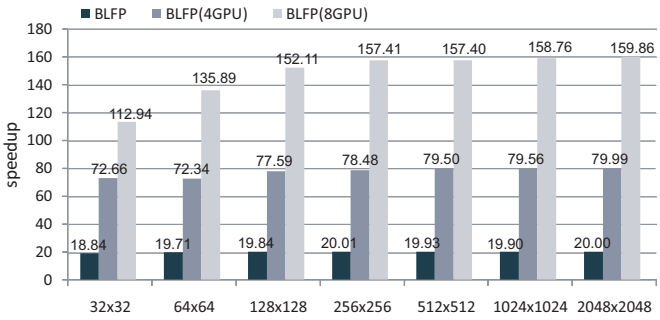

Fig. 8. Performance Gain Over CPU with process degree k=3

B. Comparison Between Two Mapping Schemes

Figure 7 shows the performance speedup of the block-level processing scheme over thread-level processing. The block-level mapping scheme greatly out-preforms thread-level scheme. For the image size of 256×256, block-level scheme runs 32.01 times faster than thread-level processing, largely due to the performance gain from accessing high speed shared memory instead of global memory.

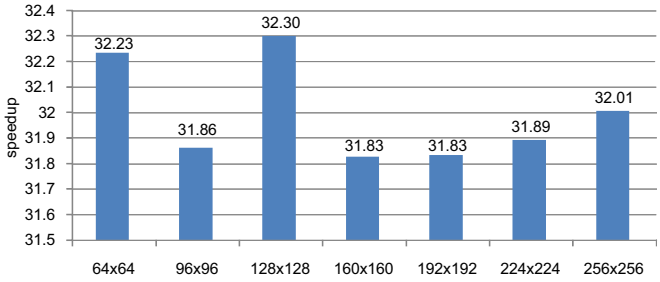

Fig. 7. Speedup of block-level processing over thread-level processing.

C. Performance Gain Over CPU

To demonstrate the performance gain of our proposed GPU approach over the standard CPU implementation, we developed a sequential version (single thread) of the *Robust FIM* algorithm on the CPU of our test machine. The CPU implementation is written in C++ and is compiled with highest optimization level (-O3 in gcc to include SSE options). Compared with the performance of the CPU implementation, our GPU algorithm exhibits a significant speedup, as shown in Figure 8. The block-level mapping scheme on a single GPU shows a speedup of 20 times for a $2,048 \times 2,048$ image. As the number of GPUs increase, GPU performance increases linearly; a 4-GPU implementation shows a speedup of 79.99, and a 8-GPU shows a speedup of 159.86.

VII. CONCLUSION AND FUTURE WORK

This paper propose a novel computation-to-core mapping scheme for the robust facet image modeling algorithm on GPUs. This mapping scheme shows a significant performance gain over the standard pixel-based mapping scheme. From the experience we have gained in this experiment, the following two principles should be considered when optimizing an application for the GPU platform.

Firstly, when considering the level of parallelism for implementation on a GPU, we will choose the level that results in simple and compact kernel functions, so that each thread can work efficiently with limited hardware resources, such as shared memory. In the *Robust FIM* algorithm, estimating a local facet model for a pixel is too much work for a thread, which makes the execution inefficient.

Secondly, GPU memory resources are allocated and used differently in a thread block for global memory and for shared memory. It's important to consider memory constraints when deciding on the level of parallelism for the application. In the *Robust FIM* algorithm, a large amount of global memory is required for thread level parallelism, which makes large input images impossible.

In the future, we plan to address memory limitation problems associated with a larger window-size, such as 7×7. In this case, the shared memory is not enough for the computation of a single facet even in our block-level processing scheme. The solution of this problem can be found in other computation-to-core mapping schemes, e.g. multiple blocks per facet.

REFERENCES

[1] NVIDIA's Compute Unified Device Architecture. `http://developer.nvidia.com/object/cuda.html/`.

[2] P.J. Besl, J.B. Birch, and L.T. Watson. Robust window operators. *Machine Vision and Applications*, 2(4):179–191, 1989.

[3] P. Bui and J. Brockman. Performance analysis of accelerated image registration using GPGPU. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 38–45, New York, NY, USA, 2009. ACM.

[4] G.H. Golub and C.F. Van Loan. *Matrix Computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[5] R.M. Haralick and L.T. Watson. A facet model for image data. *Computer Graphics Image Processing*, 15(2):113–129, 1981.

[6] A.S. Householder. Unitary triangularization of a nonsymmetric matrix. *J. ACM*, 5(4):339–342, 1958.

[7] Y. Mizukami and K. Tadamura. Optical flow computation on compute unified device architecture. In *ICIAP '07: Proceedings of the 14th International Conference on Image Analysis and Processing*, pages 179–184, Washington, DC, USA, 2007. IEEE Computer Society.

[8] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[9] M. Smelyanskiy, D. Holmes, J. Chhugani, A. Larson, D.M. Carmean, D. Hanson, P. Dubey, K. Augustine, D. Kim, A. Kyker, V.W. Lee, A.D. Nguyen, L. Seiler, and R. Robb. Mapping high-fidelity volume rendering for medical imaging to CPU, GPU and many-core architectures. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1563–1570, 2009.