J Real-Time Image Proc
DOI 10.1007/s11554-012-0272-7

ORIGINAL RESEARCH PAPER

Performance analysis of a novel GPU computation-to-core mapping scheme for robust facet image modeling

Seung In Park · Yong Cao · Layne T. Watson · Francis Quek

Received: 4 November 2011 / Accepted: 11 August 2012

Abstract Modern graphics processing units (GPUs) are commodity data-parallel coprocessors capable of high performance computation and data throughput. It is well known that the GPUs are ideal implementation platforms for image processing applications. However, the level of efforts and expertise to optimize the application performance is still substantial. This paper investigates the computation-to-core mapping strategies to probe the efficiency and scalability of the robust facet image modeling algorithm on GPUs. Our fine-grained computation-to-core mapping scheme achieves a significant performance gain over the standard pixel-wise mapping scheme. With in-depth performance comparisons across the two different mapping schemes, we analyze the impact of the level of parallelism on the GPU computation and suggest two principles for optimizing future image processing applications on the GPU platform.

Keywords Facet image modeling · Robust estimation · GPGPU · Computation-to-core mapping

S. I. Park · Y. Cao (✉) · L. T. Watson · F. Quek
Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA
e-mail: yongcao@vt.edu

S. I. Park
e-mail: spark80@vt.edu

L. T. Watson
e-mail: ltw@cs.vt.edu

F. Quek
e-mail: quek@vt.edu

L. T. Watson
Department of Mathematics, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA

1 Introduction

Modern graphics processing units (GPUs) are commodity data-parallel coprocessors capable of high performance computation and data throughput. GPUs are ideal implementation platforms for image processing algorithms. The majority of image processing algorithms belong to the class of embarrassingly parallel problems in which the same operation on each pixel of the input image is performed and the operations are completely independent. It is therefore conceptually simple to map the processing of each pixel to the parallel architecture of GPUs. In most cases, such simple computation-to-core mapping can be effective [12, 17, 33, 37]. However, when an algorithm involves a large linear system to solve, for example, robust facet image modeling [9] and surface analysis [35, 36] algorithms, the amount of computation for each pixel is too heavy for a single GPU core to execute efficiently. Applying such a pixel-wise mapping of computation to graphics hardware for this type of algorithms may result in a GPU resource deficit, and limit the scalability of an application.

In this paper we investigate the computation-to-core mapping strategies to achieve the efficiency and scalability of the image processing applications on GPUs. This paper is an extended version of our earlier work in [30]. The previous work mainly focused on developing a computation-to-core mapping scheme to realize fine-grained parallelism for the robust facet image modeling algorithm. We showed our mapping scheme, called *block-level facet processing*, achieved an efficient GPU resource utilization and thus substantial performance gains over the standard pixel-level mapping scheme. On top of this, attempts are made to better understand the impact of parallelism granularity on numerically oriented computation within image

Published online: 25 September 2012

processing. Our facet-based model involves a numerical method known as QR decomposition. We did comparative implementation of the QR decomposition computation for both the pixel and block-level approaches. We provide in-depth performance comparisons across these implementations, and draw two principles for optimizing future image processing applications on the GPU platform.

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 Sect. 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 cores. Section 6 introduces an algorithmic optimization for robust image modeling and how the optimization affects the two implementations discussed in the previous section. Section 7 discusses how the mapping affects the performance and scalability. Section 8 summarizes what mapping strategy should be taken depending upon computational profile and purpose, and describes a limitation and future work.

2 GPU architecture

GPUs have evolved from fixed function graphics pipelines to fully programmable, massively parallel architectures for general purpose computational applications. With the advent of NVIDIA's Compute Unified Device Architecture (CUDA) in 2007, developers were liberated from need to frame computation within the structure of shaders and graphics APIs. CUDA allowed researchers from many disciplines to exploit the low-cost massively parallel computational power of GPUs with support for random memory access and C programming SDK.

Our paper is about the use of modern GPUs with the CUDA programming framework and instruction set architecture for numerically-based computation and image processing. We employed NVIDIA's GT200-series GPUs as our test platform. These GPUs are 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 the GPU memory hierarchy. A detailed description of the programming model and architectural specification for CUDA can be found in [21] and [22].

2.1 Thread organization

CUDA manages a large number of computing *threads* by organizing them into a set of logical *blocks*. Each thread block can be mapped onto one of the multiprocessors for execution. The number of threads allowed in a block depends on the hardware limitation of each specific device 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.

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.

2.2 Memory hierarchy

Another important feature of CUDA is a memory hierarchy to hide memory and pipeline latency. Figure 1 illustrates the memory resources available to CUDA, and how each resource may be accessed. Each processing core within a multiprocessor has its own *local memory* (not shown in figure) that is allowed exclusively for some automatic variables, and is not in programmatic control of the developer. Each multiprocessor has a set of *shared memory* and *cache memory* resources, and a set of memory resources that is shared across multiprocessors: *global, constant*, and *texture memory*. The cache memory is closely related to texture memory. Texture and constant memory are accessible as read-only memory from within CUDA. These memory resources may be written from the host CPU program, and serve as a way to pass information

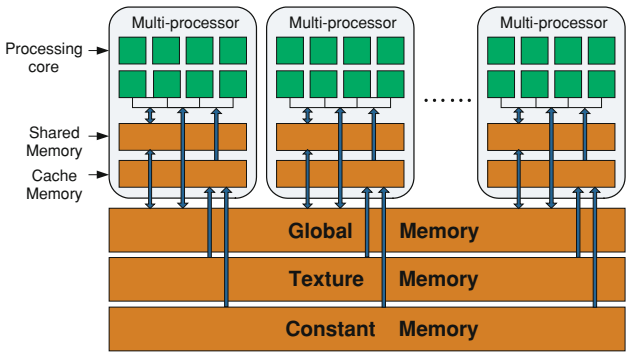


Fig. 1 Overview of the CUDA memory hierarchy [22]

into the multiprocessor program of the GPU from the CPU. The difference, as the name implies, is that texture memory also serves the purpose of declaring image arrays to be used by the GPU.

The most flexible memory available to CUDA are the shared and global memory, subject to size limitations. Shared memory is on-chip memory with very fast access that is shared 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 400–600 cycles while shared memory latency is ten cycles. Consequently global memory access should be minimized as much as possible for the best performance. However, only 16 KB 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.

3 Related work

Owens et al. [25] surveyed the latest developments and trends in general purpose application of GPUs. The focus here is on a subset of the work in image processing. A GPU-based stereo matching technique combining the sum of squared differences dissimilarity measure and multiresolution approach was introduced in [39] and [40]. Histogram equalization and tone mapping using a vertex shader was presented in [32]. Sinha et al. [33]. described the shader based implementation of the Kanade–Lucdas–Tomasi feature tracking and SIFT feature extraction algorithms on GPUs. GPUs were designed to enhance the performance of graphics related applications specifically, and significant efforts were required to use them outside of a graphics rendering context. Therefore, the focus of image processing on GPUs was reformatting of the target algorithm to be mapped to the computing of vertex transformation or pixel illumination.

With the CUDA programming framework, using the GPU for image processing has found its way into a wide variety of applications. Just to name a few of those studies, Mizukami and Tadamura [20] proposed implementation of Horn and Schunck's regularization algorithm with a multiscale search method for optical flow computation. They were able to achieve speedup of approximately 16 on a NVIDIA GeForce 8800 GTX card over a 3.2-GHz CPU. Bui and Brockman [5] presented a 2-D rigid image registration algorithm using CUDA and 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. The general optimization strategies include utilizing many threads and maximizing memory and instruction throughput through a set of techniques such as global memory coalescing, reducing shared memory bank conflicts, and reducing divergent branching [22]. Both of [20] and [5] applied these strategies to achieve optimal performance. However, they simply take the processing of each pixel as a computation unit for parallelization, none of them explored the influence of level of parallelism on the optimization.

Focusing on program optimization aspects of CUDA, Ryoo and his colleagues [28] demonstrate the significance of the optimization principles by analyzing the relative performance among different configurations on a suite of applications. Because of the enormous computing power of GPUs, orders of magnitude performance difference can exist between well optimized and poorly optimized codes of an application. In Ryoo's experiment, speedups between 10.5 and 457 were achieved by the code optimization. Later, Ryoo et al. developed their ideas further on efficient GPU program optimization, and revealed that optimizing an application for maximum performance goes beyond simply application of a set of optimization techniques to code. The difficulty comes from the fact that the interactions among the underlying architectural and programming model constraints affect performance in a non-linear fashion [29]. They modeled GPU programming for maximum performance as a multivariable optimization problem. The complete optimization space consists of memory bandwidth, dynamic instruction reduction, threads occupancy, instruction level parallelism, memory latency hiding, and work redistribution. To avoid inefficient empirical search of the large optimization space, they proposed program optimization carving that prunes those variables down to a set of configurations that bring the best performance. In order to do that, metrics that capture the instruction efficiency of the kernel code and utilization of the compute resources were developed and used to reduce the search space.

Yixun [41] took a novel perspective on CUDA program optimization. They presented how program inputs affect the effectiveness on the optimization. Then G-ADAPT (GPU adaptive optimization framework), a compiler-based framework, was presented to help decision making on the optimal code configuration. As Yixun revealed, some GPU applications are affected by certain optimization factors besides well-known ones. In this work, we explore the influence of computation-to-core mapping strategy on the image processing applications.

4 Robust facet image modeling

We selected *facet image modeling* (FIM) as the test algorithm because it represents aspects of computation that are

both image oriented (and therefore naturally replicated), and numerical (involving mathematical approaches that employ large computational arrays and numerical optimization).

The concept of FIM was first introduced by Haralick and Watson [9]. The basic idea is to divide images into larger regions that are homogeneous with respect to some high level criterion that allows these regions to be handled similarly. For example, an image of a polyhedral object may be best represented by a set of planes, each describing a surface of the polyhedron that exhibits the same surface normal (and hence similar shading characteristics). Besl et al. [4] 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. We call this *robust FIM*.

FIM has been applied to many different applications such as edge detection [11, 19, 27], background normalization [16], and image segmentation [10]. The massive amount of computation inhibits the algorithm from being used more widely in real world applications.

In an earlier attempt to employ parallel computation for FIM, Pathak et al. [26] focused on an efficient implementation of quadratic facet modeling with algorithm transformation. The optimized implementation on a MediaStation 5000 improved the performance by a factor of 7 over the direct implementation on a SUN SparcStation 10/41 for quadratic facet modeling. Our research applies parallelization on the more readily available and faster GPU computational architecture.

4.1 Algorithm overview

Facet-based modeling requires accurate parametric representation of each image facet so that it reveals the structure of the underlying whole image. As such, precise parameter extraction is at the heart of the algorithm. Since FIM employs numerical fitting of such parametric models to the image data, it is essential that the resulting parameters are not contaminated by image outliers. Our algorithm employs a robust estimation technique known as *M-estimation*. The results of the M-estimation process is fed to an *iterative reweighted least squares* algorithm that performs the actual parameter estimation. Because FIM employs polynomial basis functions for the modeling, one needs to know the order of the polynomial to apply to each facet. Obviously, this cannot be known a priori. Hence the algorithm employs a *variable order* approach where the order of the polynomial is estimated through a series of iterations beginning with lower order polynomials and advancing to higher orders. We will discuss the mathematics behind the algorithm and flush out the details of the algorithm thereafter.

4.2 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) located at the rth row and the cth column 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

$$\begin{aligned} E(a) &= \sum_{r=-n'}^{n'} \sum_{c=-m'}^{m'} \rho\left(\frac{d(r,c) - f(r,c)}{s}\right), \\ n' &= \frac{n-1}{2}, \quad m' = \frac{m-1}{2}, \end{aligned} \tag{2}$$

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 [15], $\nabla E(a) = 0$ can be written as

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

where the weight $w(r,c)$ is defined as $\rho'(e(r,c))/e(r,c), e(r,c) = (d(r,c) - f(r,c))/s$. 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 $n \cdot m \times p$ matrix whose rows are $\phi_1(r,c), \ldots, \phi_p(r,c)$. W is a $n \cdot m \times n \cdot m$ diagonal matrix whose diagonal elements are $w(r,c)$, a is a p-vector whose entries are a_i, and d is a $n \cdot m$-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

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

where t is the iteration number. Detailed derivation of these equations can be found in [4].

4.3 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 Eq. (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 [7]. 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 $Q^{\mathrm{t}}A = R$.

4.4 The algorithm

Figure 2 outlines the algorithm for IRLS-based Robust FIM. Given an $n \times m$ window, several different order robust surface fits for a pixel are computed up to a preselected maximum order. Here the highest degree of the fitting polynomial function is set at 3 because the complexity of this fitting function is adequate for the most commonly used window size, which is 5×5.

If the IRLS iteration yields a residual MAD below some *epsilon* threshold, the fit is termed 'good enough' and 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.

The pseudo code for the Robust FIM algorithm in Algorithm 1 illustrates the IRLS estimation process for a

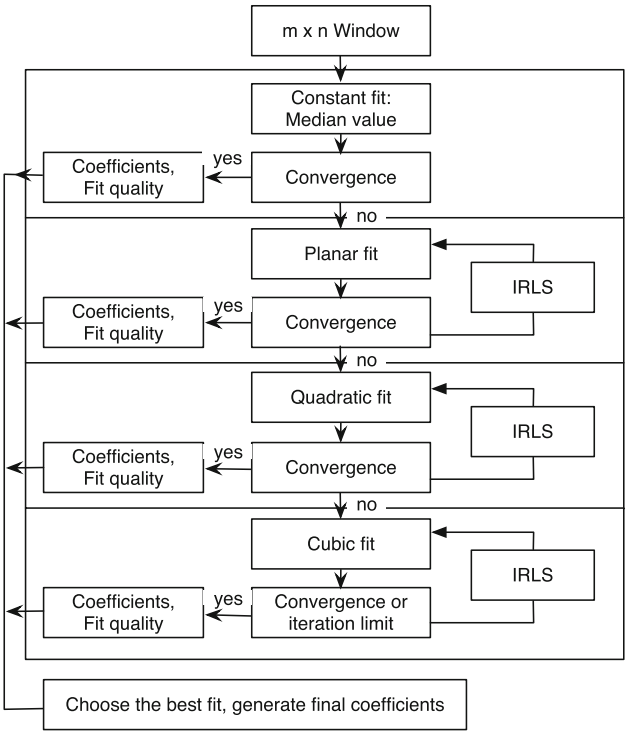


Fig. 2 Overview of the robust facet image modeling algorithm

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 $n \cdot m \times p$ 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 $n \cdot m$ vector, which is the multiplication of $W^{\frac{1}{2}}$ and *window*. Both A and b are needed to rewrite Eq. (4) as the least squares problem $Aa \approx b$. Then A is factored into Q and R components, which are used to find coefficient vector a by 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 $q \leq n \cdot m$, then with $\alpha = -\mathrm{sgn}(x_1)\|x\|$, the first $n \cdot m - q$ 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 basis vector $(1, 0, \ldots, 0)^{\mathrm{T}}$ 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 [14]. The robust fit quality measure is given by the 'FitQuality' function whose parameters are E, p, and *scale*. The process is repeatedly performed until MAD $\leq \epsilon$ or the maximum iteration limit has been reached.

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

Require: $window[n][m]$ of image data, $d[1 : n \cdot m]$ is vector representation of $window$, p is number of coefficients

Ensure: coefficient vector $a_k[p]$

$E = |window - \Phi_{k-1} a_{k-1}|$; $i = 0$
while $(scale! = 0$ & $i <$ MAXITERATION) **do**
  $W = \text{WeightFunction}(r, scale)$
  $A = W^{\frac{1}{2}} \Phi_k$; $b = W^{\frac{1}{2}} d$; $Q = I$
  for $j = 1$ to p **do**
    $v = \text{House}(A[j : n \cdot m, j])$
    $H = I - 2vv^t$; $Q^t = HQ^t$; $A = HA$
    $b = Hb$
  end for
  $R = A$
  $a_k[1 : p] = \text{BackwardSubstitution}(Ra_k = Q^t b)$
  $E = |window - \Phi_k a_k|$
  $scale = 1.4826\,\text{Median}(E)$
  $i = i + 1$
end while
$fit[k] = \text{FitQuality}(E, p, scale)$
return $a_k[1 : p]$

5 Approach

This section introduces two different computation-to-core mapping schemes when implementing the robust FIM algorithm 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 distinct 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.

We consider the thread-level facet processing scheme first because it is a straightforward choice for implementing image processing algorithms. Our results show that the memory requirements of this thread-level mapping scheme poses a significant limitation that may be explained by the way in which memory resources in GPUs are allocated and shared between threads. The explanation of the limitation of the thread-level mapping scheme motivates the block-level facet processing scheme that provides an intuitive solution to the problem introduced by memory limitation of the first scheme. Performance analysis of the second mapping scheme shows that it properly addresses memory limitation. We further discuss an optimization strategy and block-level facet scheme for multiGPU processing in Sect. 6.

5.1 Thread-level facet processing

As mentioned in Sect. 2.1, the massive parallelism of a GPU is achieved by organizing a large number of concurrently executed threads into thread blocks that are 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. Consider a model where 'each pixel sits on its own facet' such that we use its neighborhood pixels for the facet computation. For robust FIM, an independent computational unit is the facet image modeling of a pixel. Thus an input image with width $\times$ height pixels has width $\times$ height 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.

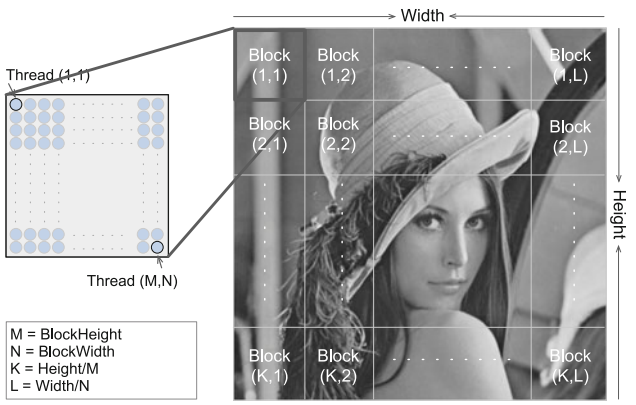


Fig. 3 Thread organization for thread-level facet processing

Figure 3 illustrates a thread organization for thread-level facet processing in detail. width $\times$ height threads are generated to process width $\times$ height pixels on the image. The threads are grouped into a block of width N and height M as on the left of the figure. The number of threads $N \times M$ in a block is determined by the resource usage of an individual thread. Consequently, width$/N \times$ height$/M$ of thread blocks run on the image as shown on the right of the figure.

Because of the uniformity of the algorithm across the entire image, the algorithm may be implemented as a single CUDA kernel function that is executed for all facets. Each block of the input image is loaded onto the shared memory array of the multiprocessor assigned to process the facet associated with that block. The kernel function performs Algorithm 1 four iterations of IRLS for each degree of fitting function (constant fit, $k = 0, \ldots, k = 3$). The implementation is straightforward since the computation within a kernel is sequential. However, the challenge is on dealing with memory hierarchy to conserve memory bandwidth and reduce the memory latency for the optimal performance.

5.1.1 Thread-block configuration

We analyze the memory requirement for the robust FIM to describe the thread-level facet processing in detail. Table 1 lists all the required variables and their memory usage in Algorithm 1, with two additional temporary variables T and t. The matrix T is used to store the intermediate result of matrix-matrix multiplication (at line number 7 in Algorithm 1), and the vector t is used to store the intermediate result of matrix-vector multiplication (at line number 4 in Algorithm 1). Robust FIM evaluates four polynomial models from constant fit to cubic fit in one kernel function to yield the best estimation, the vector a must hold all the coefficients from all fits. Therefore, the space needed for a is $q = \sum_{k=0}^{3} p_k = 1 + 3 + 6 + 10 = 20$, where p_k is the number of coefficients for the k-th degree polynomial model, e.g., $p_3 = 10$ for a cubic fit. The weight matrix W only requires $n \times m$ elements of memory space, because only diagonal elements contain an effective value. The matrix R shares the same memory space with A (A is not used after R). Fit quality is evaluated for each polynomial model, and stored in *fit*. Since d is a vector representation of the *window* data, d requires no extra memory space. Table 1 shows memory requirements for two window sizes, 5×5 and 7×7. Note that each element in the vectors and matrices has data type float.

As shown in Table 1, the overall memory required for all four fits (from constant to cubic) is 10,196 bytes for the window size of 5×5 and 34,004 bytes for 7×7. A thread's resource usage in low latency shared memory should be maximized to improve the performance of an individual thread. However, the total number of threads running on each streaming multiprocessor decreases as each thread's shared memory usage increases, because of the 16K space limit. This decrease in thread count leads to a less thread-level parallelism, and results in GPU underutilization. It is necessary to have enough threads to hide the long latency of global memory accesses and multicycle arithmetic operations such as division and reciprocal square root. The tradeoff of the performance of an individual thread and the degree of concurrency among all threads should be considered when we decide which variables in Table 1 are allocated in shared memory and how many threads are assigned in each block.

Because of the high memory requirements of the robust FIM algorithm, we minimize the use of high latency global memory as it is one of the priority principles in CUDA program optimization [23]. The thread-level facet processing execution is set to have 32 threads per block and $(\text{width} \times \text{height})/32$ blocks on the image. The most frequently revisited space throughout the robust FIM algorithm, i.e., $a, window, E, W, v$, and *fit*, can reside in shared memory with this configuration. One might consider reducing the number of threads per block to increase the amount of shared memory available for each thread. However, the 32-thread *warp* is the atomic resource unit that is scheduled by the GPU thread manager in CUDA. The number of threads in a block should be a multiple of 32 threads, to achieve optimal computing efficiency and facilitate coalescing.

Table 1 Memory requirement for executing the *robust FIM* algorithm for a single facet

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	$n \cdot m \times p \times 4$	1,000	1,960
$A(R)$	$n \cdot m \times p \times 4$	1,000	1,960
H	$n \cdot m \times n \cdot m \times 4$	2,500	9,604
Q^t	$n \cdot m \times n \cdot m \times 4$	2,500	9,604
T	$n \cdot m \times n \cdot m \times 4$	2,500	9,604
fit	4×4	16	16
Total		10,196	34,004

5.1.2 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 $b, t, A(R), H, Q^t$, and T (Φ is stored in constant memory and the rest of the variables are allocated in shared memory). All threads are executed in parallel on the GPU, and global memory is preallocated 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 GTX295 GPU, which has 876 MB of global memory, the input image cannot be larger than 328×328, a significant limitation for the application of robust FIM. If we raise the number of threads per block for a higher concurrency, the global memory usage per thread increases because of a decline in the amount of shared memory available per thread. This results in further reduction of the allowed image size.

Even for a small size image, the large amount of high latency global memory access by each thread causes a performance issue. If there are not enough threads to achieve a full multiprocessor occupancy, the multiprocessor will be forced to idle, and results in performance degradation. We discuss this issue with performance experiment results in Sect. 7.2.2.

5.2 Block-level facet processing

To address the limitation of the thread-level mapping scheme, we propose a fine-grained computation-to-core mapping scheme. This mapping scheme seeks block-level parallelism where the estimation of a facet model is executed on a *block* of threads instead of a single thread. All the threads in a block work collaboratively to accelerate the linear algebra, such as matrix multiplication. The configuration of block-level facet processing is shown in Fig. 4. Each block consists of $N \times M$ threads as on the left of the figure, and width $\times$ height blocks are created to process width $\times$ height pixels on the image as on the right of the figure.

Each matrix and vector operation in Algorithm 1 is further 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 cooperatively.

This block-level mapping scheme allows all the computation in Algorithm 1 of robust FIM to stay in the shared memory space because only one instance of the algorithm is executed in a multiprocessor. The image data is loaded into shared memory at the beginning of the kernel function execution. 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. No global memory allocation for the computation is needed, the limitation associated with the thread-level mapping scheme is obviated.

5.2.1 Thread-block configuration

The block-level mapping scheme generates one thread *block* for each pixel/facet. The 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. The hardware limitation for the maximum number of threads per block is 512 in the GTX295. Therefore, each thread has more than one unit to complete with the maximum thread allocation granularity.

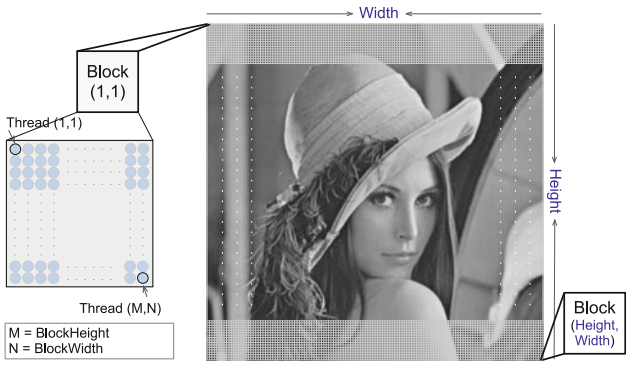


Fig. 4 Thread organization for block-level facet processing

To find an optimal thread block configuration, we varied the number of threads per block and evaluated the performance iteratively. This empirical optimization with varying configurations is a typical approach in GPU programming because a general performance prediction model for a GPU architecture is not available due to the complexity of its parallel programming model [2, 28, 29]. Our experiments showed that the thread block with size 16×26 yields the best performance in the block-level facet processing implementation. We provide the experimental results with varying thread counts in Sect. 7.2.2.

5.2.2 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, $\mathrm{TB}_1, \ldots, \mathrm{TB}_n$, are scheduled to execute on m stream multiprocessors, $\mathrm{SM}_1, \ldots, \mathrm{SM}_m$. 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 $n \gg m, n$ is limited by the hardware constraint on global memory size.

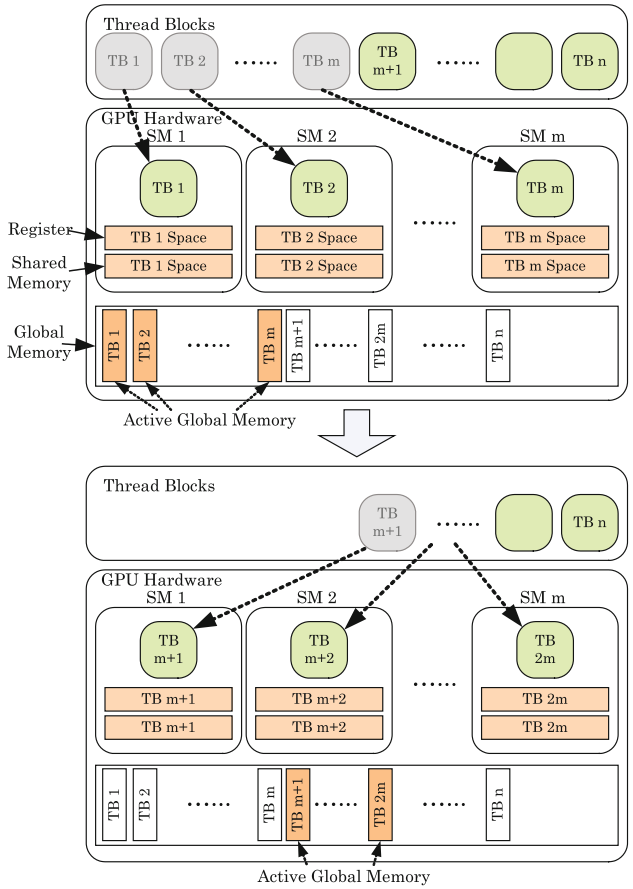


Fig. 5 CUDA thread block scheduling. *Top half* execution of first m thread blocks. *Bottom half* execution of next m thread blocks

6 Optimization

Further optimization of memory utilization for the QR decomposition is possible. First, a Householder reflection matrix H can be formed implicitly. H applied to a column vector x of A in place, overwriting the column vector, has the form $Hx = (I - 2vv^{\mathrm{t}})x = x - 2(v^{\mathrm{t}}x)v$. Second, since Householder transformations are applied to both sides of the equation $Aa \approx b$, Q^{t}need not be explicitly computed or stored. The reflection vector v is saved instead at each step, and products of the form $Q^{\mathrm{t}}A$ or $Q^{\mathrm{t}}b$ can be computed efficiently. A detailed explanation is found in [34]. Consequently, T for the intermediate result of matrix-matrix multiplication in Algorithm 1 is not used. Algorithm 2 illustrates the IRLS estimation process for a k-th degree fitting polynomial, with the efficient QR decomposition. Table 2 lists all the required variables and their memory requirements for Algorithm 2. In the remainder of this section, the implementation and thread-block configuration for thread-level and block-level processing with the algorithm modification are revisited. Then multi-GPU processing with the block-level scheme is introduced to show that further performance gain can be easily obtained with the hardware extension.

Algorithm 2 Modified k-th Degree Polynomial Fit for a Single Facet.

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

$E = |window - \Phi_{k-1}a_{k-1}|$; $i = 0$
while ($scale! = 0$ & $i <$ MAXITERATION) **do**
  $W = \mathrm{WeightFunction}(r, scale)$
  $A = W^{\frac{1}{2}}\Phi_k$; $b = W^{\frac{1}{2}}d$
  for $j = 1$ to p **do**
    $v = \mathrm{House}(A[j : n \cdot m, j])$
    for $k = 1$ to p **do**
      $A[:, k] = A[:, k] - (2v^tA[:, k]/(v^tv))v$
    end for
    $b = b - (2v^tb/(v^tv))v$
  end for
  $R = A$
  $a_k[1 : p] = \mathrm{BackwardSubstitution}(Ra_k = Q^tb)$
  $E = |window - \Phi_k a_k|$
  $scale = 1.4826\,\mathrm{Median}(E)$
  $i = i + 1$
end while
$\mathrm{fit}[k] = \mathrm{FitQuality}(E, p, scale)$
return $a_k[1 : p]$

6.1 Thread-level facet processing

6.1.1 *Thread-block configuration*

As shown in Table 2, the overall memory required for all four fits (from constant to cubic) is 2,596 bytes for the window size of 5×5 and 5,092 for 7×7. Again, Φ is stored in constant memory. If the rest of the variables $a, window, E, W, b, v, A$, and fit in Table 2 are assigned in shared memory, only ten threads are allowed in a block. To maximize the number of variables that are allocated in shared memory while having enough threads in a block, all variables except A, b, and Φ are placed in shared memory. A and b must use global memory. Then 32 threads are generated per block, resulting in $(\mathrm{width} \times \mathrm{height})/32$ blocks in total. Comparing to the first trial of thread-level facet processing in Sect. 5.1, the shared memory layout of the optimized thread-level facet processing is exactly the same. However, the global memory usage is 1.1K per thread.

Though global memory usage decreased from 8.7K to 1.1K per thread with this approach, the required global memory for a large image can still exceed the hardware limit. The input image cannot be larger than 921×921 with the GTX295 GPU.

6.2 Block-level facet processing

6.2.1 *Thread-block configuration*

In the original block-level facet processing in Sect. 5.2, multiple threads operate on a single multiprocessor to

Table 2 Memory requirement for executing the *robust FIM* algorithm for a single facet with the more efficient QR decomposition

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
Φ	$n \cdot m \times p \times 4$	1,000	1,960
$A(R)$	$n \cdot m \times p \times 4$	1,000	1,960
fit	4×4	16	16
Total		2,596	5,092

execute the robust FIM algorithm. This can be done because all the variables needed for the computation of a facet are the same across all the threads, and can be efficiently represented in the shared memory of the block. Building on this idea, the new algorithm keeps all variables associated with a facet in shared memory with the exception of t, H, Q^{t}. and T because they do not have to be explicitly computed. The threads cover all the computational units of each matrix/vector operation cooperatively. Furthermore, since H and Q^{t} are not explicitly computed or stored, the number of matrix-matrix multiplication units decreases significantly. The largest computation unit is A, which is the product of the weight matrix W and the basis function (Gram) matrix Φ. For a 5×5 window, the matrix A is 25×10, which will result in 250 units of computation. With the iterative performance evaluation in Sect. 7.2.2, we found a thread block of 8×8 yields the best performance.

6.2.2 *MultiGPU processing*

The use of multiple GPUs can bring more parallelism. The block-level mapping scheme can be extended for multiGPU processing, since the implementation can be easily adapted to use multiple devices without modifying kernel code. Once the input data is distributed among several devices, each device runs the kernel function for block-level facet processing. By hardware design, device code can be executed on only one device at any given time. To use multiple CUDA devices, host threads in the CPU are required to launch a device code. As many CPU threads as the number of GPUs are created. Each host thread feeds each device with input data and launches the device code. As shown in Fig. 6, the thread organization is the same as that for block-level facet processing. The only difference is that the input image is segmented into a number of pieces equalling the number of GPUs, and each input segment is processed by each GPU. 8×8 is chosen as the thread block size through the iterative optimization process.

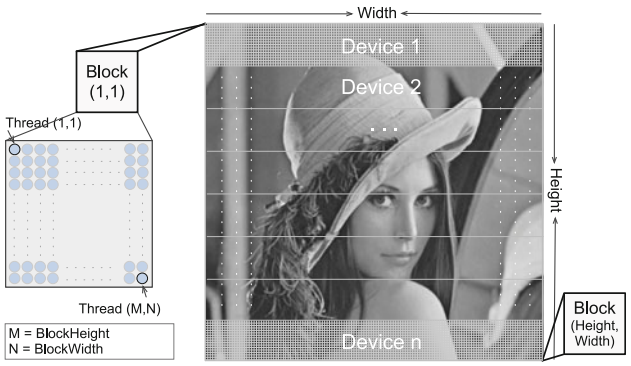


Fig. 6 Thread organization for block-level facet processing with multiple GPUs

7 Result and discussion

We measured the performance of our GPU implementations on a NVIDIA GTX295. The GTX295 is a dual-GPU based graphics card, and each GT200 GPU has 240 processor cores with a 1.24 GHz clock, 896 MB of device memory, and compute capability 1.3. Each GPU implementation of different mapping schemes on Algorithms 1 and 2 is tested using a single GT200 among the two GPUs. A multiGPU implementation is tested on a system with four GTX295 cards, which allows up to 8 GPUs to run concurrently. For comparison, a standard CPU implementation (single thread) of the robust FIM algorithm is written in C++ and is compiled with the highest optimization level. The CPU implementation is tested on a computer with Intel Core i7-920 2.67 GHz CPU and 11.9 GB system memory.

The implementation of robust FIM is tested with the canonical *Lena* picture, and two other synthetic input images, *S1* and *S2*. The *S1* and *S2* images contain step edges, four intersecting roof edges, and other complicated geometric characteristics. Vertical stroke noise is added to the *S1* image, and impulse noise in addition to the vertical stroke noise is synthesized in the *S2* image. The Lena image is also used for performance experiments with different image sizes in Sects. 7.2 and 7.3. To ensure a similar pattern of computation is given to the multiprocessors, the 64×64 *Lena* image is mosaicked to form a set of larger size images. The facet model window size is 5×5 throughout.

7.1 Accuracy of the GPU implementation

In this section the correctness of the GPU implementation of the *robust FIM* algorithm is demonstrated. Figures 7

and 8 show 256 × 256 input images of *S1* and *S2*, and the processed images resulting from CPU and GPU implementations of Algorithm 2. The resulting outputs in Fig. 7 confirm that the typical action of smoothing and feature preservation of the *robust FIM* algorithm [4, 9, 18] is present in our CPU and GPU implementations. The processed outputs in Fig. 8 show that the robust computation (removing outliers) worked correctly in both implementations.

Figure 9 shows a noisy input image (*Lena* with impulse noise added) and two processed images resulting from CPU and GPU implementations. 64 × 64 selected parts of the 512 × 512 images are provided to show details of input and output images for our ensuing discussion.

As observed on the right in Fig. 9, the GPU version appears to apply slightly greater smoothing than the CPU version. This effect results from the differences in the CPU and GPU floating point hardware. Floating point operations between different devices are not guaranteed to be identical owing to rounding errors, different size of FP registers, different order of instructions, etc. [6]. In GT200 architecture based GPUs with compute capability 1.3 and below, some single precision arithmetic operations do not follow the IEEE 754 standard, and denormal numbers are flushed to zero [38]. Consequently, GPU operations such as division and square root may not always yield floating point values of the same accuracy as CPU computations. Furthermore, the GPU computes in single or double precision only, while the CPU may use an extended precision for intermediate results. Due to these differences between hardware, mathematical function libraries, and instruction sets, GPU and CPU implementations do not yield identical

Fig. 7 Synthetic test image with stroke noise (*left*), processed images from CPU (*middle*) and GPU (*right*) implementations

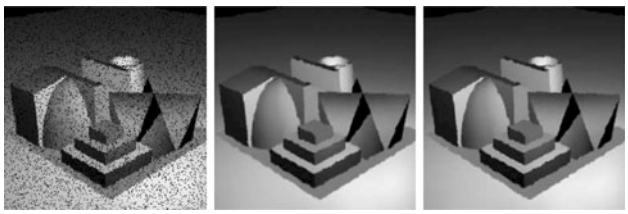


Fig. 8 Synthetic test image with stroke and impulse noise (*left*), processed images from CPU (*middle*) and GPU (*right*) implementations

Fig. 9 *Lena* with impulse noise added (*left*), processed images from CPU (*middle*) and GPU (*right*) implementations

arithmetic results. Because the robust facet algorithm employs extensive numerical computation for facet estimation, there are inevitable differences in the images resulting from the GPU and CPU implementations. We performed the paired t-test to compare the CPU and GPU result images, and the *t* test results showed that the images processed by the CPU and GPU implementations are not significantly different (*p* values of 0.999 for all the three test images).

The results illustrate important computational trade-offs when using GPUs for numerical computation in image processing. Most of these trade-offs arise from numerical precision issues in the generation of GPUs used in our studies, as discussed in [38]. Fortunately, these issues are being addressed in newer generations of CUDA architecture, Fermi [24], as we expect the performance gains over CPU computation for these new GPUs to be maintained or increased.

7.2 The level of parallelism and performance

For the sake of brevity, we shall use the acronyms shown in Table 3.

Both the T1 and T2 implementations use 15,872 bytes of shared memory per block and 40 registers per thread. Each thread uses 8.7K of global memory in T1 and 1.1K in T2. The B1 implementation uses 8,824 bytes of shared memory per block and 20 registers per thread. B2 runs with 1,524 bytes of shared memory per block and 19 registers per thread. Both B1 and B2 do not access the global memory for the facet computation. C1 and C2 process computations with a single CPU thread.

Various image sizes ranging from 32 × 32 to 256 × 256 are used to compare the performance of T1, B1, and C1, and T2, B2, and C2. For this test, the images are limited to relatively small sizes due to the scalability issue of the GPU thread-level mapping, as discussed in Sect. 5.1.2. Section 7.3 presents the performance evaluations of B1, B2, C1, and C2 with larger images up to 2,048 × 2,048. Table 4 provides speedups of T1 and B1 over C1, B1 over T1, T2 and B2 over C2, and B2 over T2. The speedup values are rounded up to the second decimal place. In the rest of this section, we present the performance characterizations of robust FIM with different mapping schemes.

Table 3 Acronyms for different computations

	Algorithm 1	Algorithm 2
GPU thread-level	T1	T2
GPU block-level	B1	B2
CPU	C1	C2

7.2.1 Impact of mapping scheme on execution time

Figure 10 shows the comparison of execution times for T1, B1, and C1. For all image sizes, B1 runs ≈20 times faster than C1, and ≈32 times faster than T1. The execution time difference between the two GPU implementations is largely due to the performance gain from accessing high speed shared memory instead of global memory in B1.

Interestingly, C1 has a faster execution time than T1 for all image sizes. This is because only one *warp* is resident on a multiprocessor for the T1 computation. Since 15,872 bytes of shared memory are consumed by a 32 thread block, the maximum number of thread blocks per multiprocessor is limited to one. Each thread of the warp accesses a large amount of global memory, and no computation is done while data is accessed from global memory. This low concurrency in computation causes a failure to hide the long latency of global memory access, hence deteriorates the performance of T1. From this result we observe that a parallel implementation can be slow when execution resources are saturated.

The execution times of T2, B2, and C2 are shown in Fig. 11. In comparison to C2, T2 and B2 achieve speedups from 2.04 to 3.21 and ≈4.2 respectively. B2 runs ≈1.3 times faster than T2. The performance gap between the thread-level and block-level mappings is not as significant as for Algorithm 1, mostly because: (1) A decrease of 7.6K in global memory usage in T2 promotes a performance improvement in the thread-level mapping approach, by alleviating the memory latency issue. (2) There is not as much data parallel computation in Algorithm 2 as in Algorithm 1. The chunks of 625 matrix-matrix multiplication units of the QR decomposition in Algorithm 1 have decreased to chunks of 25 matrix-vector multiplications units in Algorithm 2. This result indicates that the granularity of parallelism would not have a great impact on execution time if there were not a significant latency problem due to the global memory access and multicycle operations.

7.2.2 Impact of mapping scheme on scalability

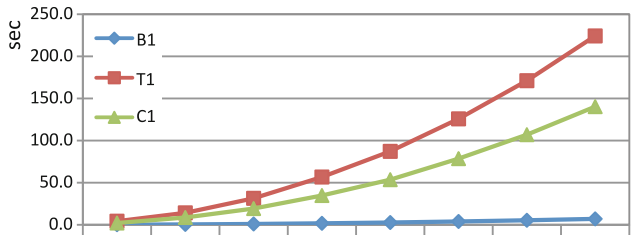
As for the CPU computation, all four GPU implementations exhibit a trend of less performance gain with small image sizes, see Table 4. For example, a speedup of 4.17 is obtained with a 32×32 image while one of 4.28 is gained on a 256×256 input with B2. Though T1 has larger execution times than C1, this trend toward better performance for larger images persists. This trend is more pronounced for thread-level mapping (going from 0.49 to 0.63 for T1:C1, and 2.04 to 3.21 for T2:C2) than for block-level mapping (going from 19.41 to 20.01 for B1:C1, and 4.17 to 4.28 for B2:C2).

In Fig. 12, the speedup values of each GPU implementation in Table 4 are normalized with respect to its speedup factor on a 32×32 image. The speedups of B1 and B2 over the CPU implementation remain almost constant through varying image sizes, while those of T1 and T2 show upward trends as the image size increases. When T1 and T2 evaluate a small image, e.g., 32×32, they launch 1,024 threads and each thread processes its own pixel individually. A GPU on the GTX295 has 30 multiprocessors, with each capable of carrying out a maximum of 8 active blocks at a time, and a total of 240 potentially active blocks are available. Given a 32×1 thread block size 1,024 threads are grouped into 32 blocks, which is far less number than the 240 available blocks. Hence, T1 and T2 suffer from the GPU underutilization problem. However, a 256×256 image requires 2,048 blocks with 65,536 threads for 65,536 pixel computations, and gives enough parallelism to the GPU. From this observation we can characterize thread-level parallelism as not efficient for small input images due to the GPU resource underutilization.

To further understand the impact of the level of parallelism on the performance, the block-level mapping implementations are tested with varying thread block sizes

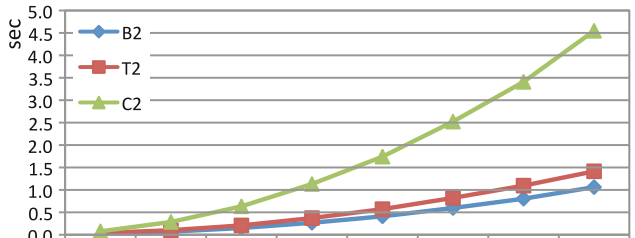
Table 4 Speedups of each algorithm with respect to the other

Img. size	T1:C1	B1:C1	B1:T1	T2:C2	B2:C2	B2:T2
32×32	0.49	19.41	39.26	2.04	4.17	2.04
64×64	0.61	19.71	32.25	2.81	4.16	1.48
96×96	0.61	19.58	31.90	3.04	4.22	1.39
128×128	0.61	19.84	32.36	3.04	4.24	1.39
160×160	0.61	19.86	32.31	3.05	4.24	1.39
192×192	0.62	19.91	31.88	3.07	4.23	1.38
224×224	0.63	19.92	31.86	3.11	4.24	1.35
256×256	0.63	20.01	31.98	3.21	4.28	1.33



	32^2	64^2	96^2	128^2	160^2	192^2	224^2	256^2
B1	0.1087	0.4397	0.9855	1.7526	2.6986	3.9454	5.3705	7.0123
T1	4.2683	14.1810	31.4415	56.7196	87.2014	125.787	171.093	224.238
C1	2.1097	8.6683	19.2926	34.7793	53.5898	78.5626	107.002	140.292

Fig. 10 Execution time of T1, B1, and C1



	32^2	64^2	96^2	128^2	160^2	192^2	224^2	256^2
B2	0.0183	0.0685	0.1502	0.2672	0.4107	0.5965	0.8037	1.0621
T2	0.0374	0.1016	0.2083	0.3720	0.5709	0.8219	1.0951	1.4145
C2	0.0763	0.2849	0.6336	1.1318	1.7412	2.5228	3.4088	4.5468

Fig. 11 Execution time of T2, B2, and C2

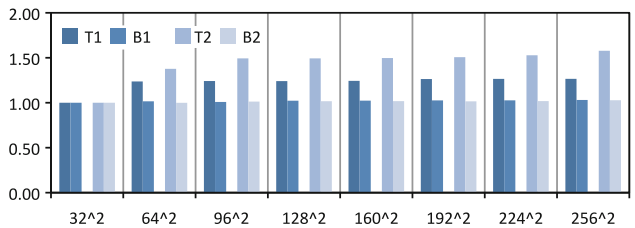


Fig. 12 Normalized speedup of T1, B1, T2, and B2 with respect to CPU implementations

on three sizes of inputs, $32 \times 32, 512 \times 512$, and $1{,}024 \times 1{,}024$. Figure 13 shows plots of execution time as a function of the number of threads per block for a $1{,}024 \times 1{,}024$ image for both B1 and B2. We show only a graph for the largest image ($1{,}024 \times 1{,}024$) for both implementations in Fig. 13, since exactly the same pattern is observed with the other two images, with respect to each approach. For B1, the execution times improve rapidly as the number of threads per block increase from 64 to 160, and then it levels off up to 384. Above 416 threads per block, the computation times are virtually constant. For B2, the execution times increase steadily as the number of threads per block increase from 64 to 288. It then levels off up to 384 threads per block. There is a large performance hit as we go to 416 threads-per-block after which the execution times plateau.

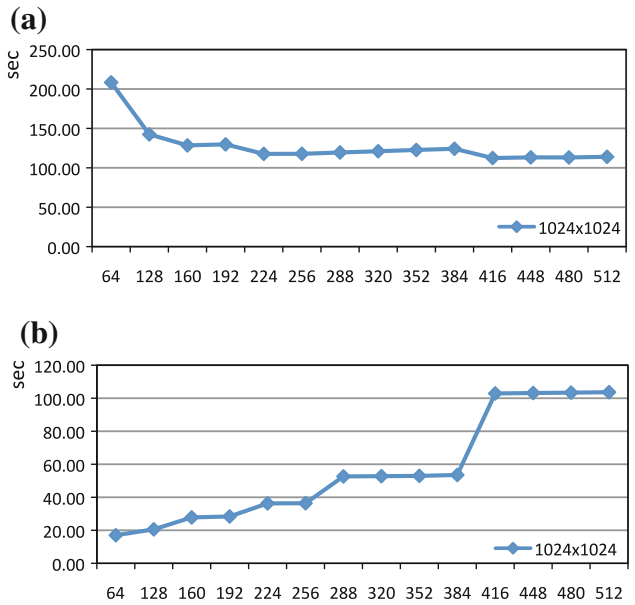


Fig. 13 Execution times as a function of number of threads per block for a $1{,}024 \times 1{,}024$ image for **a** B1, and **b** B2

In B1, since 8,824 bytes of shared memory are required per block, only one thread block can reside on a multiprocessor at a time, owing to the 16K space limit. More threads in a block add more parallelism to the computation. As such, the execution times for B1 go downwards but plateau after $416\,(=16 \times 26)$ threads. In B2, a configuration of $64\,(=8 \times 8)$ threads produces the best performance. A thread block size of 64 allows 8 blocks resident on a multiprocessor, given 1,524 bytes of shared memory usage per block in B2, the GTX295 hardware specification of a maximum of 512 threads per block. The 64 threads are grouped into 2 warps, and 16 warps reside on a multiprocessor in total. Compared to this, with a thread block size of 512, we have a single block and 16 warps per multiprocessor. Though the occupancy is the same, this configuration results in a lower number of thread blocks and therefore takes a longer execution time. When a lower number of threads per block is specified for the algorithm B2, CUDA assigns more blocks to the computation. This presents the opportunity for greater concurrency since if one block is waiting (e.g., for multicycle arithmetic operations), the other blocks can keep operating (this is automatically scheduled by CUDA). This explains the increase in performance for lower thread per block in B2.

7.3 Performance gain over CPU, and multiGPU processing

In this section the performance evaluations of GPU and CPU implementations are presented. We consider only the execution time for the facet computation in this

Table 5 Execution time measured in second of CPU and GPU implementations

Img. size	C1	B1 (1 GPU)	B1 (4 GPU)	B1 (8 GPU)
32 × 32	2.11	0.11	0.03	0.02
64 × 64	8.67	0.44	0.12	0.06
128 × 128	34.78	1.75	0.45	0.23
256 × 256	140.29	7.01	1.79	0.89
512 × 512	558.99	27.94	7.03	3.55
1,024 × 1,024	2,233.40	111.63	28.03	14.03
2,048 × 1,024	8,977.32	448.79	112.23	56.15
Img. size	C2	B2 (1 GPU)	B2 (4 GPU)	B2 (8 GPU)
32 × 32	0.08	0.02	0.01	0.01
64 × 64	0.28	0.07	0.02	0.01
128 × 128	1.13	0.27	0.07	0.04
256 × 256	4.5	1.06	0.29	0.14
512 × 512	18.24	4.25	1.12	0.57
1,024 × 1,024	73.06	17.00	4.49	2.26
2,048 × 1,024	292.44	68.05	17.96	8.94

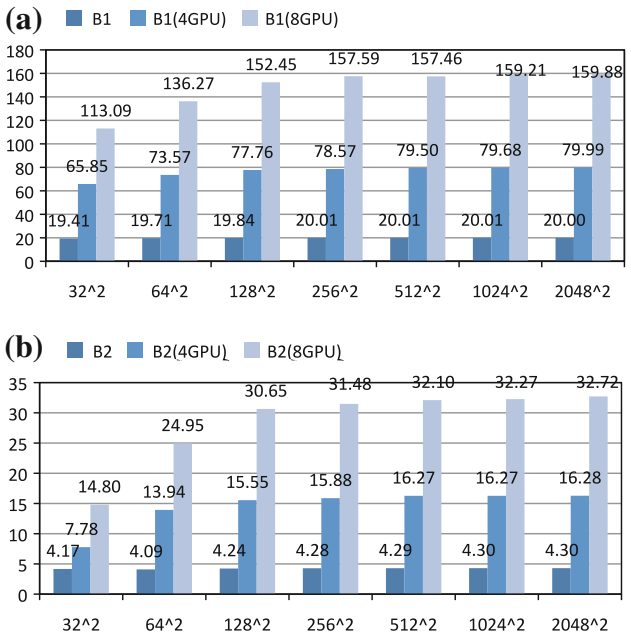


Fig. 14 GPU performance gain over CPU for **a** Algorithm 1, and **b** Algorithm 2

comparison. The times are measured in seconds and rounded up to the second decimal place in Table 5. Figure 14a compares the performance of B1 and C1. B1 on a single GPU shows a speedup of 20 for a 2,048 × 2,048 image. As the number of GPUs increases, the performance increases linearly; a four-GPU implementation shows a speedup of 79.99, and the eight-GPU one shows a speedup of 159.88.

The performance comparison of the B2 and C2 implementations is presented in Fig. 14b. B2 on a single GPU shows a speedup of 4.1 for a 2,048 × 2,048 image. As the number of GPUs increases, the performance increases linearly; a four-GPU implementation shows a speedup of 16.28 and the eight-GPU one shows a speedup of 32.72.

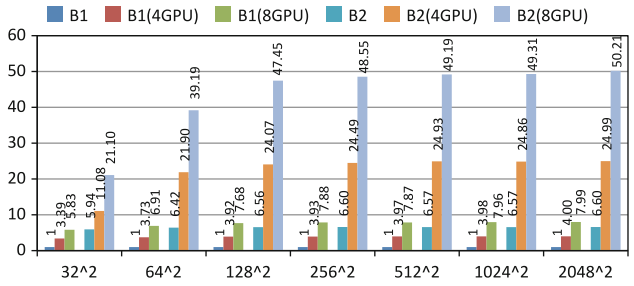


Fig. 15 The performance comparison of B1 and B2 with multiGPUs

Overall performances for Algorithms 1 and 2 are significantly different as shown in Fig. 15. The speedup graph is plotted normalized for the B1 speedup on a single GPU. B2 on a single GPU shows a speedup of 6.60 over B1 for a 2,048 × 2,048 image. For this size of image, the block-level mapping scheme with Algorithm 2 on the four-GPU and eight-GPU systems shows speedups over Algorithm 1 of 24.99 and 50.21, respectively.

8 Conclusion and future work

This paper investigated the computation-to-core mapping strategies to probe the efficiency and scalability of the robust facet image modeling algorithm on GPUs. Our

fine-grained mapping scheme showed a significant performance gain over the standard pixel-based mapping scheme. This work suggests two principles for optimizing future image processing applications on the GPU platform. Firstly, when considering a parallel decomposition (the problem to processor mapping) for implementation on a GPU, choose the mapping 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 heavy a workload for a thread, which makes the execution inefficient. Secondly, since GPU memory resources are allocated and used differently in a thread block for global memory and for shared memory, it is important to consider memory constraints when deciding on the parallel decomposition 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 to process.

Our test results on the multiGPU implementations show that multiGPU systems have great potential for enhancing the performance of an algorithm with computationally intensive workloads. However there is a caveat to consider in the adoption of a multiGPU cluster for a real-time performance. In a multiGPU system, several GPUs are connected to a host CPU through a shared bus. As the number of GPUs attached to the shared bus grows, the increased pressure on the bus affects the transfer latencies, and can result in an overall performance degradation. For example, our system has four GTX295s installed in a single motherboard, where each card has two GT200 GPUs. When a program launches, the CPU communicates with the eight-GPUs to distribute the input image, and this causes an initial overhead delay. If an application stores a processed image onto a disk rather than directly displays it to a screen, each GPU has to transfer its result back to a CPU, an extra overhead.

The performance bottleneck due to data transfer between the CPU and GPU is a serious concern in hybrid computing [8, 31], and a next generation of GPU architectures will address this problem. For example, AMD has been in development of Fusion to integrate the CPU and GPU on the same silicon die, and this will allow unified address space and fully coherent memory access between the CPU and GPU [1, 3].

Our future work will 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 the block-level processing scheme. The solution to this problem may be found in other computation-to-core mapping schemes, e.g., multiple blocks per facet and controlled block scheduling.

References

1. AMD Inc.: AMD Accelerated Processing Units. Retrieved Feb. 2012 (2011). http://www.amd.com/us/products/technologies/fusion/Pages/fusion.aspx
2. Archuleta, J., Cao, Y., Scogland, T., Feng W.: Multi-dimensional characterization of temporal data mining on graphics processors. In: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS '09), IEEE Computer Society, pp. 1–12 (2009)
3. Branover, A., Foley, D., Steinman, M.: AMD's Llano Fusion APU. IEEE Micro **99** (PrePrints, 2012)
4. Besl, P., Birch, J., Watson, L.: Robust window operators. Mach. Vis. Appl. **2**(4), 179–191 (1989)
5. Bui, P., Brockman, J.: Performance analysis of accelerated image registration using GPGPU. In: Proceedings of 2nd workshop on General Purpose Processing on Graphics Processing Units, ACM, pp 38–45 (2009)
6. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput Surv **23**(1), 5–48 (1991)
7. Golub, G., Van Loan, C.: Matrix Computations, 3rd edn. Johns Hopkins University Press, Baltimore (1996)
8. Gregg, C., Hazelwood, K.: Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In: 2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 134–144 (2011)
9. Haralick, R.M., Watson, L.T.: A facet model for image data. Comput. Graph. Image Process. **15**(2), 113–129 (1981)
10. Haralick, R.M., Watson, L.T., Laffey, T.J.: The topographic primal sketch. Int. J. Robot. Res. **2**(1), 50–72 (1983)
11. Haralick, R.M.: Digital step edges from zero crossing of second directional derivatives. IEEE Trans. Pattern Anal. Mach. Intell. {\bf PAMI-6}(1):58–68 (1984)
12. Harish, P., Narayanan, P.: Accelerating large graph algorithms on the GPU using CUDA. In: Proceedings of the 14th International Conference on High, Performance Computing (HiPC'07), pp. 197–208 (2007)
13. Huang, J., Ponce, S., Park, S.I., Cao, Y., Quek, F.: GPU-accelerated computation for robust motion tracking using the CUDA framework. In: 5th International Conference on Visual Information Engineering, VIE 2008, pp. 437–442 (2008)
14. Householder, A.: Unitary triangularization of a nonsymmetric matrix. J. ACM **5**(4), 339–342 (1958)
15. Huber, P.J.: Robust estimation of a location parameter. Ann. Math. Stat. **35**(1), 73–101 (1964)
16. Jankowski, M.: Iterated facet model approach to background normalization. SPIE **2238**, 198–206 (1994)
17. Luo, Y.M., Duraiswami, R.: Canny edge detection on NVIDIA CUDA. In: 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, Vol. 43(1), pp. 1–8 (2008)
18. Mainguy, J., Birch, J.B., Watson, L.T.: A robust variable order facet model for image data. Mach. Vis. Appl. **8**, 141–162 (1995)
19. Matalas, I., Benjamin, R., Kitney, R.: An edge detection technique using the facet model and parameterized relaxation labeling. IEEE Trans. Pattern Anal. Mach. Intell. **19**, 328–341 (1997)
20. Mizukami, Y., Tadamura, K.: Optical flow computation on compute unified device architecture. In: ICIAP 07: Proceedings of the 14th International Conference on Image Analysis and Processing, pp. 179–184 (2007)
21. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue **6**(2), 40–53 (2008)
22. NVIDIA Corporation: NVIDIA's Compute Unified Device Architecture. Retrieved Feb. 2012 (2010). http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

23. NVIDIA Corporation: NVIDIA CUDA Best Practices Guide. Retrieved Feb. 2012 (2009). http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf
24. NVIDIA Corporation: NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Retrieved Feb. 2012 (2010). http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
25. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krűger, J., Lefohn, A.E., Purcell, T.J.: A survey of general purpose computation on graphics hardware. Comput. Graph. Forum **26**(1), 80–113 (2007)
26. Pathak, S.D., Kim, Y., Kim, J.: Efficient implementation of facet models on a multimedia system. Opt. Eng. **35**(6), 1739–1745 (1996)
27. Qiang, J., Haralick, R.M.: Efficient facet edge detection and quantitative performance evaluation. Pattern Recognit. **35**(3), 689–700 (2002)
28. Ryoo, S., Rodrigues, C., Baghsorkhi, S., Stone, S., Kirk, D., Hwu, W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, pp. 73–82 (2008a)
29. Ryoo, S., Rodrigues, C.I., Stone, S.S., Stratton, J.A., Ueng, S.Z., Baghsorkhi, S.S., Hwu, W.: Program optimization carving for GPU computing. J. Parallel Distrib. Comput. **68**(10), 1389–1401 (2008b)
30. Park, S.I., Cao, Y., Watson, L.T.: A novel computation-to-core mapping scheme for robust facet image modeling on GPUs. In: The 2010 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2010), pp. 189–195 (2010)
31. Schaa, D., Kaeli, D.: Exploring the multiple-GPU design space. In: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS '09), pp. 1–12 (2009)
32. Scheuermann, T., Hensley, J.: Efficient histogram generation using scattering on GPUs. In: Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games, pp. 33–37 (2007)
33. Sinha, S., Frahm, J.M., Pollefeys, M., Genc, Y.: Feature tracking and matching in video using programmable graphics hardware. Mach. Vis. Appl., **22**(1), pp. 207–217 (2007)
34. Trefethen, L.N., Bau, D.: Numerical linear algebra. SIAM Press, Philadelphia (1997)
35. Terzopoulos, D.: The computation of visible-surface representation. IEEE Trans. Pattern Anal. Mach. Intell. **10**(4), 417–438 (1988)
36. Torr, P.H.S., Zisserman, A.: MLESAC: a new robust estimator with application to estimating image geometry. Comput. Vis. Image Underst. **78**(1), 138–156 (2000)
37. Vineet, V., Narayanan, P.J.: CUDA cuts: Fast graph cuts on the GPU. In: 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, pp. 1–8 (2008)
38. Whitehead, N., Fit-Florea, A.: Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs, White Paper, NVIDIA Corporation (2011)
39. Yang, R., Pollefeys, M.: Multi-resolution real-time stereo on commodity graphics hardware. In: Proceedings of the 2003 IEEE Computer Society Conference on Computer Vision and, Pattern Recognition (CVPR'03), pp. 211–217 (2003)
40. Yang, R., Pollefeys, M., Li, S.: Improved real-time stereo on commodity graphics hardware. In: Proceedings of the 2004 Conference on Computer Vision and Pattern Recognition workshop (CVPRW'04), p. 36 (2004)
41. Yixun, L., Zhang, E.Z., Shen, X.: A cross-input adaptive framework for GPU program optimizations. In: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing, pp. 1–10 (2009)