Journal of Computer and Communications, 2016, 4, 18-50
http://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

An Approach to Parallelization of SIFT Algorithm on GPUs for Real-Time Applications

Raghu Raj Prasanna Kumar, Suresh Muknahallipatna, John McInroy

Department of Electrical Engineering, University of Wyoming, Laramie, USA
Email: raghuraj19@gmail.com, sureshm@uwyo.edu, mcinroy@uwyo.edu

How to cite this paper: Kumar, R.R.P., Muknahallipatna, S. and McInroy, J. (2016) An Approach to Parallelization of SIFT Algorithm on GPUs for Real-Time Applications. *Journal of Computer and Communications*, **4**, 18-50.
http://dx.doi.org/10.4236/jcc.2016.417002

Received: October 25, 2016
Accepted: December 26, 2016
Published: December 29, 2016

Abstract

Scale Invariant Feature Transform (SIFT) algorithm is a widely used computer vision algorithm that detects and extracts local feature descriptors from images. SIFT is computationally intensive, making it infeasible for single threaded implementation to extract local feature descriptors for high-resolution images in real time. In this paper, an approach to parallelization of the SIFT algorithm is demonstrated using NVIDIA's Graphics Processing Unit (GPU). The parallelization design for SIFT on GPUs is divided into two stages, a) Algorithm design-generic design strategies which focuses on data and b) Implementation design-architecture specific design strategies which focuses on optimally using GPU resources for maximum occupancy. Increasing memory latency hiding, eliminating branches and data blocking achieve a significant decrease in average computational time. Furthermore, it is observed via Paraver tools that our approach to parallelization while optimizing for maximum occupancy allows GPU to execute memory bound SIFT algorithm at optimal levels.

Keywords

Scale Invariant Feature Transform (SIFT), Parallel Computing, GPU, GPU Occupancy, Portable Parallel Programming, CUDA

1. Introduction

Image matching is a fundamental aspect needed to solve computer or machine vision problems, including object or scene recognition, 3D structure modeling, stereo image correspondence, motion tracking, etc. Objects in images have features that can be extracted and used for comparing across images. A large number of such features for various objects can be extracted from images with effi-

cient algorithms. An example of such an algorithm is Scale Invariant Feature Transform (SIFT) [1], which detects and extracts local feature descriptors from images. The features extracted using the SIFT algorithm are invariant to rotation, scaling and illumination and hence applicable to scene modeling, recognition and tracking [2]. A simple sequential implementation of SIFT on lower resolution images is shown to utilize huge memory with high computation times [3], making the use of SIFT in real-time applications infeasible.

High-resolution image processing is used in various application domains like remote sensing, traffic monitoring, unmanned air vehicles, mars expeditions, etc. SIFT has been shown to provide good tracking capability in all the above domains [4] [5] [6] [7]. However, the large computational burden prevents the use of SIFT in real-time applications in the above domains. Feng *et al.*, in their multi-core parallelization work [8] have demonstrated the potential of parallelizing the SIFT algorithm. A number of researchers [3] [9] [10]-[15] [35] [36] [37] [38] [39] have attempted to parallelize the SIFT algorithm to make it applicable for real-time. In particular, the research effort by Yao *et al.*, has demonstrated a parallelized SIFT algorithm [13] with the capability to process an image of resolution 640 × 480 pixels in 31 ms. The 31 ms processing time would be sufficient to process video of frame rate 24 frames per second with a resolution of 640 × 480. However, the current application domains require processing of images with resolutions ranging from 1280 × 720 (720 p) to 8192 × 4608 (8 k) known as high-resolution images in real-time. Recent researches [16] [17] [18] [19] [20] have attempted to parallelize SIFT for high-resolution images. However, all of these attempts either have high execution time or fail to provide reproducible implementation, making current approaches of parallelizing the SIFT algorithm incompatible for real-time applications.

In this paper, we discuss a two phase: a) algorithm design and b) implementation design, parallelization strategies to obtain a better and efficient parallelization of SIFT, which lowers the compute time enabling real-time processing of high-resolution images. While, the algorithm design phase strategies demonstrate parallelization of the SIFT algorithm, the implementation design phase strategies define the values of machine parameters, such as blocks and threads, to implement the parallelized SIFT algorithm on to the underlying GPU architecture. The two-phase parallelization approach facilitates porting of parallelized SIFT algorithm to different GPU architectures. The research work presented in this paper is based on the preliminary research presented by authors in [21]. In preliminary research, the parallelization of only the first stage of the SIFT algorithm was presented. This paper presents the complete parallel implementation of SIFT suitable for real-time applications.

The paper is organized as follows: In Section 2, we provide a brief description of the SIFT algorithm based on the research work by Lowe in [1]. Section 3, presents the mathematical steps in the SIFT algorithm. In Section 4, an intro-

duction to the GPU architecture and its constraints are presented. In Section 5, we provide the two phase parallelization strategies and demonstrate the parallelization of SIFT algorithm. Section 6 provides the performance of the parallelized SIFT algorithm. We discuss the future work and conclude the paper in Section 7.

2. Description of SIFT

SIFT algorithm discussed in the research work by Lowe has four stages to obtain the features of an image [1]. The four stages of the SIFT algorithm are shown in Figure 1. A discussion of the four stages is provided below:

1) Scale-space Extrema Detection (SSED): The SSED stage consists of two sub-stages namely the formation of the Difference of Gaussian (DoG) pyramid and extrema keypoint detection.

a) Difference of Gaussian Pyramid: The first stage of computation is to build a pyramid of images having different blurring levels and varying resolutions. First, the initial image is repeatedly convolved with different Gaussian functions characterized by their varying standard deviations known as scaling, to produce a set of images having same resolution but varying in their blurring levels as shown in Figure 2. In Figure 2, this set of images is marked as Octave 1 representing the same resolution but different blurring levels. The top image of octave 1 has

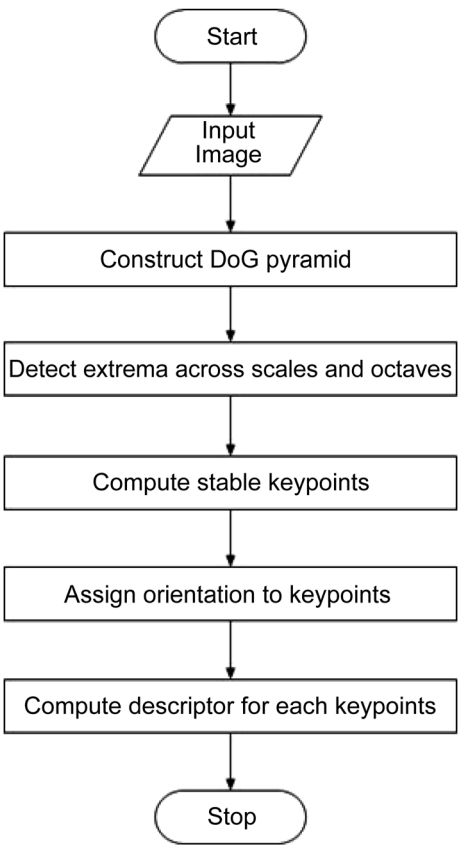

Figure 1. Flowchart of SIFT algorithm.

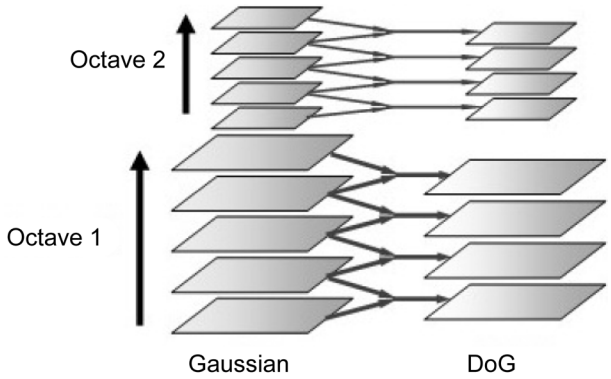

Figure 2. DoG pyramid structure.

the maximum blurring. Next, the resolution of the top image in Octave 1 is reduced by half to form the bottom image in Octave 2. Again, this bottom image in Octave 2 is repeatedly convolved with different scaled Gaussian functions to produce the set of images for Octave 2. This convolution and resolution reduction is repeated to build the different layers of the pyramid. Next, the DoGs of an octave are constructed by taking the difference between any two consecutive images of the same. The left side in **Figure 2** depicts the different octaves and scaled images in each octave, and the right side depicts the corresponding DoG images. Interest points for SIFT are obtained using the DoG images.

b) Extrema detection: The pixels of a DoG image are the difference in intensity values of consecutive blurred images. Keypoints are pixels identified as local maxima or minima of the DoG images across octaves. This is obtained by comparing a pixel in the DoG image to its neighboring pixels at the current and adjacent scales [1]. If the pixel is greater than or lesser than all its neighbors, it is considered as the local maxima or minima respectively for the given octave level. If the local minima or maxima for the first octave, continues to be a local minima or maxima for all octaves, then it is selected as a candidate keypoint.

2) Keypoint localization: The selected candidate keypoints (CKs) are filtered to eliminate unstable CKs. CKs having low contrast or being a part of an edge in an image are considered as unstable keypoints [1]. A 3D quadratic function fitting is performed on the selected CKs to evaluate its contrast with respect to neighboring points [22]. By setting a threshold for evaluation, low contrast CKs are eliminated [1]. A second-order derivative of the Hessian matrix [1] is used to eliminate the CKs that form part of an edge.

3) Orientation assignment: The CKs remaining after the localization step are considered as keypoints. One or more orientations are assigned to each keypoint based on local image gradient directions, *i.e.*, the contribution of each neighboring pixel is weighted by a gradient magnitude as discussed in [1]. The peaks of the histogram of orientations indicate dominant orientations. Once a keypoint orientation has been selected, the feature descriptor is computed as a set of orientation histograms represented in the form of a vector [1].

3. Steps in SIFT

A step-by-step mathematical discussion of SIFT, as seen in [1], is provided below. It should be noted that we are reiterating the description of SIFT from [1], without adding any new terminologies or techniques.

a) Gaussian convolution: Consider a Gaussian matrix $G^s \in R^{3\times 3}$, given by:

$$G_{ij}^s = \left(\frac{1}{2\pi\sigma_s^2}\right) e^{\frac{-\left(i^2+j^2\right)}{2\sigma_s^2}} \tag{1}$$

where,

$\sigma_s = \bar{\sigma} 2^{\frac{1}{s}}$ is the standard deviation at every scale,

$\bar{\sigma} = 0.8$ is the predetermined standard deviation for the SIFT algorithm, and s is the scaling level,

$i, j = \{-1, 0, 1\}$ are the indices of G^s, with $(0,0)$ being the element undergoing convolution operation. $(0,0)$ is the index of the middle element of G^s matrix.

G^s is convolved with the grayscale image $I^o \in R^{m^o \times n^o}$, where o is the octave level, m^o, n^o is the resolution of the image for the octave level o. The convolution is performed by sliding G^s as a window over I^o. The convolved image $L^{so} \in R^{m^o \times n^o}$, is obtained as follows:

$$L_{xy}^{so} = G_{ij}^s * I_{xy}^o \tag{2}$$

where, x, y in I_{xy}^o and L_{xy}^{so} is the co-ordinate of a pixel in the image.

Since the image values are discrete numbers, the convolution is performed as shown below:

$$L_{xy}^{so} = \sum_{i=1}^{3}\sum_{j=1}^{3} G_{ij}^s I_{(x-i)(y-j)}^o \tag{3}$$

b) DoG computation: The difference between L^{so} of two consecutive scale levels, s1 and s2, provides a corresponding DoG D^{s21o}, as shown below:

$$D_{xy}^{s21o} = L_{xy}^{s2o} - L_{xy}^{s1o} \tag{4}$$

This operation can be merged as shown in Equation (5) eliminating one multiplication per pixel in I^o.

$$D_{xy}^{s21o} = \left(G_{ij}^{s2} - G_{ij}^{s1}\right) * I_{xy}^o \tag{5}$$

c) Scaling: Once D^{s21o} for all scales of an octave are computed, the last L^{so} is scaled down by 2 to obtain I^{o+1}.

d) Pyramid construction: The steps (a), (b) and (c) are executed for I^{o+1} to obtain I^{o+2} and so on. This whole process is repeated till a preset octave level is obtained. For the purpose of this paper $o = 4$ and $s = 4$, is used based on the discussions in [1].

e) Extrema detection: Since s is set to four, in each octave, there are three DoG

images. For every DoG image in an octave, the maxima and minima are detected by comparing a pixel (indicated by a triangle) to its twenty six neighbors comprising of 3 × 3 × 3 region at the current and adjacent scales (indicated by circles) as shown in Figure 3.

f) Low contrast keypoint elimination: The extrema detection provides a set of CKs that include points having low contrast or are poorly localized along an edge, and hence are considered unstable [1]. To eliminate low contrast CKs, an evaluation of CKs' contrast with respect to its neighboring points needs to be determined. This is done by fitting a 3D quadratic function to the CK as described in [1] [22]. The 3D quadratic function is given by:

$$D(\mathrm{X}) = D + \left(\frac{\partial D}{\partial \mathrm{X}}\right)^{\mathrm{T}} \mathrm{X} + \frac{1}{2}\mathrm{X}^{\mathrm{T}} \frac{\partial^2 D}{\partial \mathrm{X}^2}\mathrm{X} \tag{6}$$

where,

$\mathrm{X} = \begin{bmatrix} x & y & \sigma \end{bmatrix}^{\mathrm{T}}$,

$D \in D_{xy}^{s21o}$,

$\frac{\partial D}{\partial \mathrm{X}} = \begin{bmatrix} \frac{\partial D}{\partial x} & \frac{\partial D}{\partial y} & \frac{\partial D}{\partial \sigma} \end{bmatrix}$,

$$\frac{\partial^2 D}{\partial \mathrm{X}^2} = \begin{bmatrix} \frac{\partial^2 D}{\partial x^2} & \frac{\partial^2 D}{\partial x \partial y} & \frac{\partial^2 D}{\partial x \partial \sigma} \\ \frac{\partial^2 D}{\partial x \partial y} & \frac{\partial^2 D}{\partial y^2} & \frac{\partial^2 D}{\partial y \partial \sigma} \\ \frac{\partial^2 D}{\partial x \partial \sigma} & \frac{\partial^2 D}{\partial y \partial \sigma} & \frac{\partial^2 D}{\partial \sigma^2} \end{bmatrix}$$

Since the image values are discrete numbers, the derivatives have to be computed numerically. Equations (7)-(9) provide the finite difference method to calculate the partial derivatives $\frac{\partial D}{\partial x}, \frac{\partial^2 D}{\partial x^2}$ and $\frac{\partial^2 D}{\partial x \partial y}$ respectively. The (x, y, σ) are cyclic, and hence can be swapped in Equations (7)-(9) to obtain other derivatives:

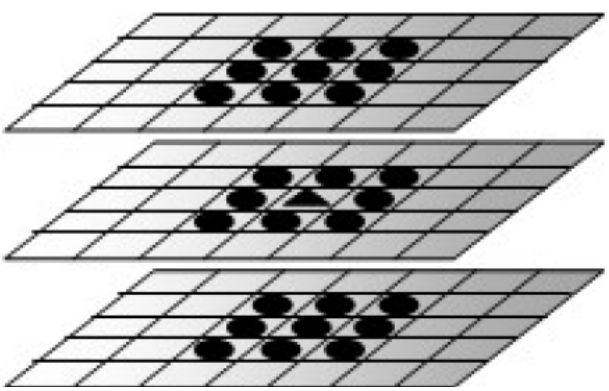

Figure 3. Extrema detection example for a pixel.

$$\frac{\partial D}{\partial x} = \left(L_{xy}^{so} - L_{(x-1)y}^{so}\right) \quad \text{or} \quad \left(L_{(x+1)y}^{so} - L_{xy}^{so}\right) \tag{7}$$

$$\frac{\partial^2 D}{\partial x^2} = L_{(x+1)y}^{so} + L_{(x-1)y}^{so} - 2L_{xy}^{so} \tag{8}$$

$$\frac{\partial^2 D}{\partial x \partial y} = \frac{L_{(x+1)(y+1)}^{so} - L_{(x+1)y}^{so} - L_{x(y+1)}^{so} + 2L_{xy}^{so} - L_{(x-1)y}^{so} - L_{x(y-1)}^{so} + L_{(x-1)(y-1)}^{so}}{2} \tag{9}$$

The contrast $\left(\left|D\left(\hat{\mathrm{X}}\right)\right|\right)$ of the CK w.r.t. neighboring pixels is evaluated by calculating the function value at the extremum position $\left(\hat{\mathrm{X}}\right)$ as shown in Equations (10)-(11):

$$D\left(\hat{\mathrm{X}}\right) = D + \frac{1}{2}\left(\frac{\partial D}{\partial \mathrm{X}}\right)^{\mathrm{T}} \hat{\mathrm{X}} \tag{10}$$

$$\hat{\mathrm{X}} = -\left(\frac{\partial^2 D}{\partial \mathrm{X}^2}\right)^{-1} \frac{\partial D}{\partial \mathrm{X}} \tag{11}$$

The CKs are considered low contrast if they satisfy the inequality shown in Equation (12) and are discarded.

$$\left|D\left(\hat{\mathrm{X}}\right)\right| < 0.03 \tag{12}$$

g) Edge keypoint elimination: The DoG has a strong edge response, *i.e.*, points on an edge is detected as an extrema. This is due to the principal curvature values for DoG tend to be higher along an edge. However, the principal curvature value for DoG along the perpendicular direction of an edge is low in magnitude. This property is used to eliminate the CKs that are a part of the edge. The eigen values of the Hessian matrix (H) computed on D, as shown in Equation (13), are proportional to the principal curvature values.

$$H = \begin{bmatrix} \frac{\partial^2 D}{\partial x^2} & \frac{\partial^2 D}{\partial x \partial y} \\ \frac{\partial^2 D}{\partial x \partial y} & \frac{\partial^2 D}{\partial y^2} \end{bmatrix} \tag{13}$$

Therefore, the ratio (r) of the maximum eigen value (α) to the minimum eigen value (β) can be used to determine whether a CK belongs to an edge. Using the relationship between trace (Tr) and determinant (Det) w.r.t the eigen values, as listed in Equations (14), (15), the candidate keypoint must satisfy the constraint as seen in Equation (16) to qualify as a keypoint.

$$Tr(H) = \alpha + \beta \tag{14}$$

$$\mathrm{Det}(H) = \alpha\beta \tag{15}$$

$$\frac{Tr(H)^2}{\mathrm{Det}(H)} < \frac{(r+1)^2}{r} \tag{16}$$

As described in [1], $r = 10$ serves as a good value to eliminate edge points.

h) Orientation assignment: The keypoints can be described relative to local orientation of its neighboring pixels rendering them invariant to image rotation. The orientation is assigned by calculating two parameters a) gradient magnitude (m) and b) gradient orientation (θ). For a given keypoint (x, y), orientation is assigned as follows:

$$m_{xy} = \sqrt{\left(L^{so}_{(x+1)y} - L^{so}_{(x-1)y}\right)^2 + \left(L^{so}_{x(y+1)} - L^{so}_{x(y-1)}\right)^2} \tag{17}$$

$$\theta_{xy} = \tan^{-1}\left(\frac{L^{so}_{x(y+1)} - L^{so}_{x(y-1)}}{L^{so}_{(x+1)y} - L^{so}_{(x-1)y}}\right) \tag{18}$$

A descriptor is generated using this orientation by constructing a histogram of orientations of sample points around the keypoints. Each sample point added to the histogram is weighted by its gradient magnitude and Gaussian-weighted circular window proportional to the σ of the keypoint scale. This generates a vector of values termed as keypoint descriptor. The keypoint descriptor is the scale, orientation, and location invariant data set obtained as the output of SIFT algorithm.

From an implementation perspective, steps (a), (b), (e), (f), (g) and (h) involve arithmetic computations, while steps (c) and (e) involve assignment and comparison operations. Since, assignment involves a memory write, requiring more cycles than a single floating point operation, every write in step (c) is considered as one computation for the purpose of this paper. Steps (e) - (h) are dependent on the number of CKs and keypoints computed in an image. On a per keypoint basis, the number of computations involved for steps (e) - (h) are in the order of hundreds. Steps (a) - (c) have large number of computations, independent of the CKs and keypoints generated. Since, the number of CKs and keypoints computed are fewer in number, steps (a) - (c) form a significant overhead compared to steps (e) - (h). The computations involved in steps (a) - (c) are shown in **Table 1** for a standard high definition image with resolution of $m^1 = 1080$, $n^1 = 1920$ at octave level 1. From the table, the computational complexity for steps (a) - (c) can be formulated as $432 + \sum_{o=1}^{4} 18\left(m^o - 2\right)\left(n^o - 2\right) + m^{o+1}n^{o+1}$. If the steps (a) - (d) generate k keypoints, then computations for (e) - (h) are $\mathrm{O}(100k)$. Hence, the computations shown in **Table 1** indicate that our parallelization should focus more on steps (a) - (c).

Table 1. Step-wise computations for SSED for octave level 1.

Step	Computational Complexity	Computations
a	$108 + 17\left(m^o - 2\right)\left(n^o - 2\right)$	35149376
b	$\left(m^o - 2\right)\left(n^o - 2\right)$	2067604
c	$m^{o+1}n^{o+1}$	518400

4. Compute Unified Device Architecture

General-Purpose Computing on GPU (GPGPU) is a new parallelization domain to accelerate scientific and engineering computations in vision algorithms [23]. NVIDIA's CUDA is the earliest and most widely adopted programming model for GPGPU [24]. Since the beginning of parallelization on GPGPU, NVIDIA has introduced several GPU architectures to improve the performance of parallelized programs. An overview of the three common features of GPU architecture namely thread, memory and execution hierarchies [25], are provided below.

GPU thread hierarchy is characterized by three parts namely the grid, block (B) and thread (T). Each grid contains 1D, 2D or 3D arrangement of blocks and each block contains 1D, 2D or 3D arrangement of threads. The number of threads per block cannot vary across blocks. Hence the total number of threads spawned is the product of total blocks and threads per block.

GPU memory hierarchy has three classifications namely the register (ρ), shared (ψ) and global memories. The memory hierarchy and its accessibility with respect to threads and blocks are shown in Figure 4.

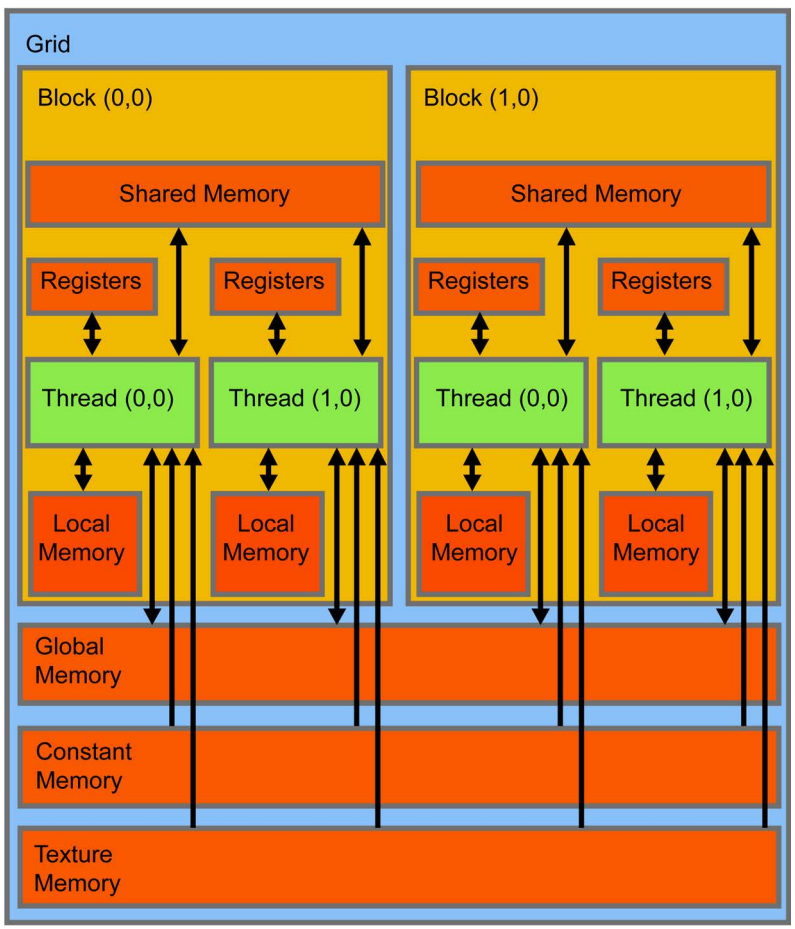

Figure 4. NVIDIA's representation of grids, blocks, and threads their memory accessibility.

Each thread within a block has access to private register memory, and all the threads within a single block have access to the shared memory, through which they share data. The threads distributed across multiple blocks have access to global memory, but sharing or visibility of modified data across blocks is not guaranteed. The memory access latency increases non-linearly from register memory to global memory thereby an excessive use of global memory leads to computational penalties. However, the amount of memory also decreases non-linearly from global to register memory. Furthermore, it is possible to achieve efficient memory bandwidth usage if the data on global memory is arranged for coalesced access.

In addition to the above three types of memories, there are two additional read-only memory types accessible by all threads known as constant and texture memories. The constant and texture are GPU memories that are cached and optimized for read access. While the constant memory is optimized to access data elements of limited size, the texture memory is optimized for scattered access of large data.

The instruction execution hierarchy of GPUs consists of a set of streaming multiprocessors (SMs) with each SM containing multiple execution cores (C). Based on the number of cores, each SM can concurrently execute a limited number of threads known as a warp (w). The warp organization represents the Single Instruction Multiple Threads (SIMT) architecture. Hence, combining the number of warps per SM and the number of SMs per GPU, increases compute parallelization and effective utilization of memory bandwidth. However, due to SIMT architecture, SM executes instructions with a granularity of a warp, *i.e.*; all threads within a warp execute the same instruction (φ). However, conditional flows due to the nature of the algorithm may introduce conditional branches within a single warp contributing to the increase in computation time.

The scalability of CUDA lies in the execution model. While blocks and threads form the partitioning of workload, the execution model provides a feature to map the threads to SMs. Based on the number of blocks and threads per block, a scheduler allocates one or more blocks to each SM. The number of blocks that can execute concurrently on a SM is based on the number of warps that can concurrently execute on the SM which, in turn, is based on the amount of maximum warps supported by the SM, and the amount of shared and register memory required by a block. As the memory requirement per block increases, the number of warps that can concurrently execute on a SM decreases. An SM is said to be at maximum efficiency if the number of warps concurrently executing is equal to maximum warps supported by the SM.

Like any other machine, GPUs also have constraints. Each SM, B, T and w have its own constraints. For example, each SM has a constraint on maximum active blocks (currently executing blocks), maximum block allocation, maximum active warps, maximum executing threads, etc. Similar constraints apply to B, T and w.

Constraints, for the purpose of this paper, are represented symbolically as a function shown below:

$$\eta(\alpha,\beta,\gamma)=\delta \tag{19}$$

where α represents the variable to which the constraint applies, β represents the scope of the α variable, γ represents the type of restriction imposed on α and δ is the output that provides the numerical value of the restriction. γ can take the following values listed in the Table 2 below.

Table 3 summarizes all the thread and execution hierarchy based constraint functions:

Table 2. List of restrictions imposed on constraint functions.

Description	Symbol
Minimum allocation or allocation granularity	$\vee$
Maximum allocation	$\wedge$
Allocation limited by algorithm input or user	χ
Allocation restricted by maximum warps	χ_w
Allocation restricted by shared memory available	χ_ψ
Allocation restricted by register memory available	χ_ρ

Table 3. List of execution hierarchy based constraint functions.

Description	Function
Maximum SMs per GPU	$\eta(SM,GPU,\wedge)$
Allocated blocks per GPU	$\eta(B,GPU,\chi)$
Maximum active blocks per SM	$\eta(B,SM,\wedge)$
Maximum warps per SM	$\eta(w,SM,\wedge)$
Maximum active threads per SM	$\eta(T,SM,\wedge)$
Allocated blocks per SM	$\eta(B,SM,\chi)$
Allocated warps per SM	$\eta(w,SM,\chi)$
Allocated warps per SM based on χ_w	$\eta(w,SM,\chi_w)$
Allocated warps per SM based on χ_σ	$\eta(w,SM,\chi_\psi)$
Allocated warps per SM based on χ_ρ	$\eta(w,SM,\chi_\rho)$
Maximum threads per block	$\eta(T,B,\wedge)$
Allocated warps per block	$\eta(w,B,\chi)$
Allocated threads per block	$\eta(T,B,\chi)$
Maximum threads per warp	$\eta(T,w,\wedge)$
Allocated Operations per thread	$\eta(\varphi,T,\chi)$

Similar to *SM*, *B*, *T* and *w*, the memory is also subjected to constraints. Table 4 summarizes all the memory hierarchy based constraint functions:

5. Parallelization

There have been a number of GPU parallelization strategies that have been proposed over the last few years. A few relevant to SIFT are presented in [26]-[34]. While [26] [27] focus on parallelization strategies for stencil codes similar to SIFT, [28]-[34] focus on implementation specific strategies for parallelization. None of the articles provide algorithm design strategies, as we propose in this paper, to utilize the benefits of GPU architecture for parallelization.

The parallelization strategies, proposed in this paper, are divided into two phases for separating SIFT and GPU related parallelization strategies. The algorithm design phase strategies focus on parallelizing SIFT based on data size, data usage, and data organization. The implementation design phase strategies focus on how the parallelized SIFT design can achieve maximum execution efficiency on a GPU. The remainder of this section focuses on the description of the two phases and their application on SIFT.

The algorithm design phase strategies involve parallelization techniques based on three guidelines.

a) Reduce: As seen in [35], the bottleneck of algorithm execution is always found to be the memory accesses-read and write operations. The memory operations require higher time reducing the performance of the algorithm. Hence, the higher the number of memory operations, the lower the performance of the algorithm. However, not all algorithms' performance is bound by memory. This is because algorithms may have a lot more computations than memory operations. In order to differentiate memory bound algorithms, a ratio, termed as computational intensity (CI), is defined as the number of floating point computations performed for each memory operation. This provides a measure to understand the impact of memory operations on algorithms. It is given by

Table 4. List of memory hierarchy based constraint functions.

Description	Function
Maximum shared memory per SM	$\eta(\psi, SM, \wedge)$
Maximum registers per SM	$\eta(\rho, SM, \wedge)$
Allocated registers per SM	$\eta(\rho, SM, \chi)$
Maximum shared memory per block	$\eta(\psi, B, \wedge)$
Allocated shared memory per block	$\eta(\psi, B, \chi)$
Minimum granularity of shared memory allocation per block	$\eta(\psi, B, \vee)$
Minimum granularity of register memory allocation per warp	$\eta(\rho, w, \vee)$
Maximum registers per thread	$\eta(\rho, T, \wedge)$
Allocated registers per thread	$\eta(\rho, T, \chi)$

$$\text{CI} = \frac{\text{Number of floating point computations}}{\text{Number of memory operations}} \tag{20}$$

The CI determines if an algorithm is memory or compute bound. If the ratio is less than 1, then it is said to be memory bound. For $\text{CI} \gg 0$, the algorithm tends to become compute bound.

The reduce guideline states that for a memory bound algorithm, the number of memory operations should be reduced to increase CI. In other words, for every floating point operation, the number of memory operations should be minimized. This can be achieved either by a) computing values rather than loading or b) reducing memory latency by grouping algorithm's data. While (a) replaces the memory operations with floating point computations, (b) groups floating point operations such that they operate on common memory elements.

The SIFT algorithm, inferring from steps (a) - (c) and (e) - (g) in section III, is a memory bound algorithm. For example, step (e) alone in section III requires 27 loads for detecting extrema.

b) Reuse: Commonly known as cache blocking, this guideline states that reusability of memory elements should be maximized, *i.e.*, a memory element loaded should not have to be reloaded during any part of the execution. This can be achieved by grouping algorithmic operations such that all computations associated with a memory element are performed together.

The SIFT algorithm has high data reusability. For example, step (c) seen in section III shows that 25% of the image data is re-used for every octave.

c) Re-arrange: Scattered memory operations result in poor utilization of memory bandwidth. This guideline recommends rearranging instructions and data to avoid branching of execution and increase memory bandwidth usage respectively.

SIFT algorithm is observed to have both branching and scattered data access. For example, while step (d) in section III requires twenty six comparisons, leading to branching, step (d) scatters data elements for an increase in octave level.

These guidelines are not rigid rules but rather provide a methodology to parallelize programs. The SSED parallelization, steps (a) - (e) in section III has been parallelized in our previous work-in-progress publication [21]. However, the adoption of guidelines has improved upon the parallelization strategies as shown below:

a) SSED provides a pyramid structure as shown previously in **Figure 2**. The pyramid structure emphasizes the dependency that exists between each octave. If the pixels in the lower most octave are considered as granularity of parallelization, it provides huge parallelization for threads to work. However, it also increases dependency across threads for each increment in octave level. This violates the guidelines by increasing memory operations, reloading similar memory elements and having scattered accesses. It has been observed through our simulations that this significantly deteriorates performance, especially for ultra-

high resolution images. Adhering to our guidelines, we propose top-down approach for parallelization of SSED; we group data based on the top DoG image in the highest octave. To determine if D_{xy}^{s21o} at the highest octave and scale is an extrema, it has to be compared to eight neighboring pixels of the same scale, and eighteen neighboring pixels of adjacent scales within the same octave level. Adopting reduce guideline, all scales within an octave can be computed simultaneously. Therefore SSED would require loading nine elements-eight neighboring pixels and the D_{xy}^{s21o} itself. Adopting the reuse guideline-instead of reloading the elements for each octave, reusing these elements across octave is recommended. However, data dependency between neighboring pixels from a lower octave level is observed, *i.e.*, in order to retain the eight neighboring pixels in the current octave, thirty six data neighboring pixels are required from the previous octave. Similarly, dropping down to lower octave levels, a fourfold increase in data elements is observed as shown in **Figure 5**. With $o = 4$, this would be 24 × 24. Adopting the re-arrange and reduce guideline, if threads can be grouped to share data, then the neighboring elements are grouped together to have common elements.

b) Next, finding the minima/maxima requires twenty six comparisons. The twenty six comparisons at every octave level results in divergence between the threads belonging to the same warp. This will prevent concurrent execution of threads in a warp [36] and hence increases the computational time. Hence, adopting the re-arrange guideline, the twenty six comparisons are replaced with a single comparison reducing the multiple divergences to a single divergence. Consider a pixel $\overline{a} \in N$ which needs to be compared with twenty six other pixels represented as a_x, where $x = 1, 2, \cdots, 26$ and $a_x \in N$. To check whether $\overline{a}$ is maxima or minima, the two factors given in Equations (7), (8) are computed:

$$\alpha = \left\lfloor \frac{\overline{a}^{26}}{\prod_x a_x} \right\rfloor \tag{21}$$

$$\beta = \left\lfloor \frac{\prod_x a_x}{\overline{a}^{26}} \right\rfloor \tag{22}$$

Next, If $\alpha > 0$ and $\beta = 0$ then $\overline{a}$ is the local maxima and if $\beta > 0$ and $\alpha = 0$ then $\overline{a}$ is the local minima.

c) Since the extrema detection can yield either a selection of candidate keypoint or rejection of pixels, the execution has to proceed only with CKs. This leads to load imbalance which can significantly impact performance on a SIMT architecture which executes at warp granularity. Therefore, to avoid load imbalance between threads, it is recommended to synchronize the execution after obtaining CKs.

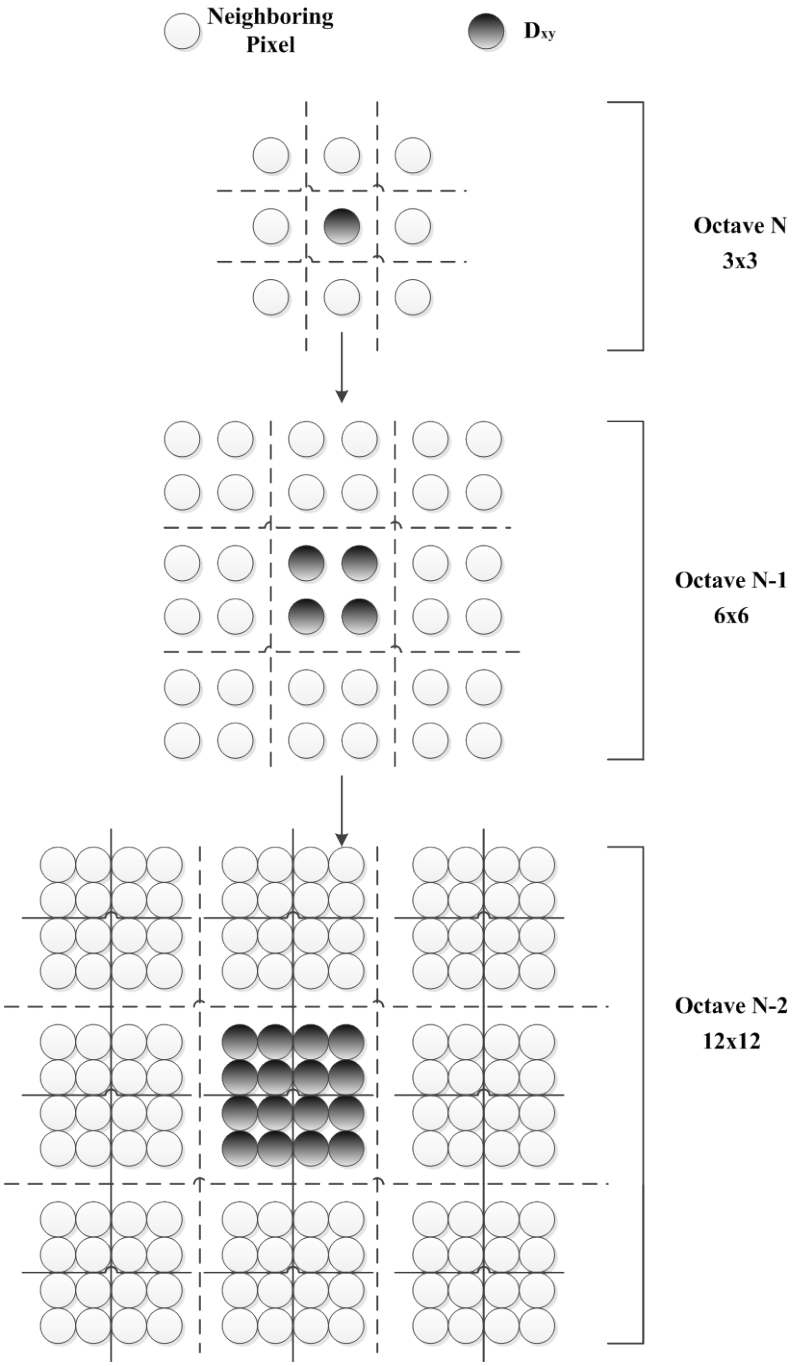

Figure 5. Top down view of number of data elements needed to compute the extrema for three octave levels.

d) The synchronization forces re-load of data elements on memory. Since the CKs are scattered, the computations on the CKs have poor memory bandwidth utilization due to scattered memory operations. This leads to performance deterioration. Adopting the re-arrange guideline, all CKs along with their neighboring pixels needed to compute partial derivatives and hessian matrix are merged

together into a new matrix structure, eliminating all irrelevant data. The organization of the new matrix structure can vary based on the underlying architecture support. For example, if the underlying architecture supports interleaved memory accesses between thread, similar to GPU SIMT architecture, the elements can be arranged in an interleaved fashion, *i.e.*, first data element accessed by every thread are grouped together followed by second element used by all threads and so on. If the underlying architecture supports consecutive memory accesses per thread, similar to CPU SIMD architecture, then all elements used by single thread are grouped together. A map between the new data layout and the original image is retained to re-map the descriptors back to original image.

Since there are no dependencies observed in steps (e) - (h) described in section III, the computation at hand is embarrassingly parallel with respect to each keypoint. Moreover, each keypoint requires no more than 8 neighboring pixels for all computations through steps (e) - (h) per scale per octave leading to a high CI.

As mentioned earlier, this algorithm design phase facilitates for parallelization of SIFT algorithm across architectures. The second phase of parallelization involves implementation design parallelization strategies. There are several different approaches to designing parallelization strategies for GPU. Two popular approaches, as seen in [26]-[34], are a) Maximizing occupancy and b) Maximizing memory bandwidth. Our approach for implementation design phase is maximizing occupancy as opposed to maximizing memory bandwidth. This is because our approach to eliminating memory bottlenecks at algorithm design phase complements a maximizing occupancy in implementation phase.

There are few research publications, such as [37], which conclude that occupancy is not a measure of effective utilization of GPU resources. The work conducted in [37] shows that the author pursued maximizing occupancy alone, without considering other dependent resources that are affected by increasing occupancy. Moreover, the results in [37] show that good performance can be achieved at lower occupancy. However, no proof has been provided to show increasing occupancy lowers performance either in [37] or any other publications.

The usual approach to maximizing occupancy on GPUs is to profile the code, and then add modifications such as increase the number of threads, or decrease the memory being used. Our approach to maximizing occupancy differs with others because we adopt a mathematical approach to maximize occupancy. Unlike the usual approach, the mathematical model connects occupancy with warps, shared memory, register memory, blocks and threads creating a multi-dependency model. Therefore, we do not set occupancy to maximum value and then compute the other dependencies; rather we try to calculate maximum achievable occupancy while tuning other dependencies. The idea is to combine optimization techniques to the underlying architecture mathematically, so as to obtain a near optimal implementation.

The two guidelines for implementation design guidelines:

a) Select threads per block to maximize occupancy: GPUs can spawn huge number of threads—a minimum of one thread per block to a maximum of $\eta(T,B,\wedge)$ threads per block. Consider $\eta(w,SM,\wedge)$ which dictates the maximum active threads per SM $\eta(T,SM,\wedge)=C\eta(T,w,\wedge)$. Occupancy (ζ) is the ratio of the number of warps executing on an SM ($\eta(w,SM,\chi)$), to the maximum number of warps that can execute on a SM ($\eta(w,SM,\wedge)$) as shown below:

$$\zeta=\frac{\eta(w,SM,\chi)}{\eta(w,SM,\wedge)} \tag{23}$$

The $\eta(w,SM,\chi)$ is constrained by three parameters, namely, the χ_w, χ_ψ and χ_ρ. Since $\eta(w,SM,\chi)$ is constrained, its value of is calculated by:

$$\eta(w,SM,\chi)=\min\left(\eta(w,SM,\chi_w),\eta(w,SM,\chi_\psi),\eta(w,SM,\chi_\rho)\right) \tag{24}$$

$\eta(w,SM,\chi_w)$ depends on the limitation of warps per SM and the maximum active blocks per SM as observed in NVIDIA's documentation. $\eta(w,SM,\chi_w)$ is decided as the minimum of two values: a) The number of warps needed to execute the allocated blocks and b) Maximum warps on a SM. It is computed using the Equation (25) below:

$$\eta(w,SM,\chi_w)=\min\left(\left\lfloor\frac{\eta(w,SM,\wedge)}{\eta(w,B,\chi)}\right\rfloor\eta(w,B,\chi),\eta(w,SM,\wedge)\right) \tag{25}$$

where,

$$\eta(w,B,\chi)=\left\lceil\frac{\eta(T,B,\chi)}{\eta(T,w,\wedge)}\right\rceil \tag{26}$$

Equation (25) calculates number of warps allocated based on the allocated blocks using threads per block and the number of threads per warp.

$\eta(w,SM,\chi_\psi)$ is the number of warps that can be allocated based on the shared memory required by the program and its availability per SM. Hence it is calculated using two values a) Shared memory available per SM and b) amount of shared memory needed per block as shown below:

$$\eta(w,SM,\chi_\psi)=\left\lfloor\frac{\eta(\psi,SM,\wedge)}{\text{ceil}\left(\eta(\psi,B,\chi),\eta(\psi,B,\wedge)\right)}\right\rfloor \tag{27}$$

Due to hardware constraints, shared memory has allocation granularity, *i.e.*, the allocation of shared memory is always an integral multiple of $\eta(\psi,B,\vee)$. The ceil function rounds the first argument to the immediate next multiple of the second argument. In Equation (27), the allocation of shared memory per block is rounded to allocation granularity using ceil function.

Similarly, $\eta(w,SM,\chi_\rho)$ is the number of warps that can be allocated based on the registers needed by the algorithm and its availability per SM. It is calcu-

lated as shown below:

$$\eta\left(w, SM, \chi_{\rho}\right) = \left\lfloor \frac{\eta\left(\rho, SM, \chi\right)}{\eta\left(w, B, \chi\right)} \right\rfloor \eta\left(w, B, \chi\right) \tag{28}$$

where,

$$\eta\left(\rho, SM, \chi\right) = \left\lfloor \frac{\eta\left(\rho, SM, \wedge\right)}{\text{ceil}\left(\eta\left(\rho, T, \chi\right)\eta\left(T, w, \wedge\right), \eta\left(\rho, w, \vee\right)\right)} \right\rfloor \tag{29}$$

As derived from NVIDIA documentation and seen in Equations (28), (29), the number of warps allocated based on registers depends on the registers allocated per thread, number of warps allocated per block and allocation granularity of register memory.

In the equations listed to calculate $\eta\left(w, SM, \chi_{w}\right)$, $\eta\left(w, SM, \chi_{\psi}\right)$ and $\eta\left(w, SM, \chi_{\rho}\right)$, the only unknown is $\eta\left(T, B, \chi\right)$. The rest of the values are available as calculated in algorithm design phase, *i.e.*, $\eta\left(\psi, B, \chi\right)$ and $\eta\left(\rho, T, \chi\right)$ can be calculated by algorithm's design. $\eta\left(T, B, \chi\right)$ is chosen such that $\zeta \to 1$. This provides a range of $\eta\left(T, B, \chi\right)$ values that can be used.

b) Select blocks to maximize occupancy: The blocks on a GPU tend to increase the number of floating point operations performed rather than hide the memory latency. Hence choosing higher blocks improves compute bound algorithms performance. The feasible blocks is calculated using:

$$\eta\left(B, \text{GPU}, \chi\right) = \frac{N}{\min\left(\eta\left(T, B, \chi\right), \eta\left(N, B, \chi\right)\right)} \tag{30}$$

where, N represents the total number of data elements. Since SIFT is a memory bound algorithm, we only focus on the scenario when CI < 1. If CI < 1 then selecting $\eta\left(T, B, \chi\right)$ with the highest value, and its corresponding $\eta\left(B, \text{GPU}, \chi\right)$ would provide better performance as increased threads per block promote memory latency hiding.

The above guidelines are based on the assumption that the total number of data points $N \le \eta\left(B, \text{GPU}, \chi\right)\eta\left(T, B, \chi\right)$. If otherwise, it is recommended to increase $\eta\left(\varphi, T, \chi\right)$ till Equation (31) provides blocks and threads within GPU constraints.

$$\eta\left(B, \text{GPU}, \chi\right) = \frac{N}{\eta\left(T, B, \chi\right)\eta\left(\varphi, T, \chi\right)} \tag{31}$$

The two guidelines in the implementation design phase provide the exact thread and block numbers that maximizes performance on GPU. Since the calculations depend on the GPU used, image resolution and the number of keypoints generated, its calculation for the entire SIFT is not demonstrated in the paper. However, calculations of thread and block numbers for SSED are demonstrated for a 720 p resolution greyscale image on C2075 Tesla architecture GPU.

The algorithm design phase demonstrated that for o = 4, each thread would

need to handle 24 × 24 data elements. Assuming four bytes per data element, this would require 2,304 bytes of memory for one thread. However, since we used the reduce guideline, each neighboring element can be processed by a thread sharing memory with an additional 768 bytes of data. For C2075,

$$\eta(\psi, B, \wedge) = 16128 \text{ bytes} \tag{32}$$

Therefore,

$$2304 \le \eta(\psi, B, \chi) \le 16128 \tag{33}$$

can be rewritten as,

$$\eta(\psi, B, \chi) = 2304 + i \times 768, \qquad i = 0,1,2,\cdots,18 \tag{34}$$

The code implementation showed the requirement of 8 registers per thread, each of size 32 bytes. Hence,

$$\eta(\rho, T, \chi) = 256 \text{ bytes} \tag{35}$$

Using Equations (32), (34), (35) along with C2075 architectural specifications, we can calculate Equations (25), (27), (28). Plugging in the results back to Equation (24) and Equation (23), it is seen that $\eta(w, SM, \chi_\psi)$ is the limiting factor for ζ. It is observed that ζ is maximum when $2304 \le \eta(\psi, B, \chi) \le 6144$. Since this is a memory bound problem, memory latency hiding is preferred. To hide memory latency, higher number of elements per block is preferred. Therefore the configuration with 6144 memory bytes supporting 6 data elements per block is chosen. Hence, $\eta(T, B, \chi)$ is chosen to be 256. The blocks are calculated to be 2400 using Equation (30), where, for $o = 4, N = 14,400$ elements.

6. Performance

The research work of Heymann *et al.*, on parallelization of the SIFT algorithm [3] has shown that the SSED stage computations are the major computational and memory bottlenecks in the SIFT algorithm. Feng *et al.* has shown that 40% - 60% of the total computational burden [8] is encountered in performing the SSED.

Moreover, performance analysis and comparison of parallelized SIFT algorithm is challenging. A brief overview of the complexity behind analysis and comparison with other parallel SIFT versions is provided below:

a) Image variation: The sizes of each image vary. Hence the workload varies with a change in image resolution.

b) Number of CKs generated: Each image generates different CKs, varying in number. This leads different workload for each image.

c) Trading accuracy for performance: Dropping accuracy, *i.e.*, not all keypoints are generated, can significantly boost the performance of parallel SIFT algorithm. For example, dropping precision model, using fast math libraries, lowering the number of octaves and scales in SSED can improve the performance but generate lower number of keypoints. It may also contain false keypoints that are not produced by Lowe's SIFT algorithm. Therefore, a given image can gen-

erate different number of keypoints.

d) Architecture and algorithmic constraints: SIFT has been a popular algorithm for over a decade. This has led to many different versions of the algorithm. The implementation of SIFT algorithms on NVIDIA GPGPUs were after 2008. Over the years the GPGPU have changed their architecture significantly. Therefore, obtaining the performance benchmark of parallelized Lowe's SIFT algorithm on a given NVIDIA GPU architecture is very challenging.

Therefore, this section presents results for analysis and comparison of SIFT algorithm in 4 phases:

1) Performance comparison of SSED: A sequential and parallel version of Lowe's SIFT algorithm was implemented on Matlab for comparing SSED performance. This helps in assessing the speedup from an average sequential and parallel program to our implementation

2) Performance analysis of SIFT-optimized for speed: For real time applications, it is essential for SIFT to perform within a duration. As discussed before, trading accuracy for execution time, allows SIFT to execute faster but with less accurate results. An analysis of the performance and accuracy of our GPU parallel version of SIFT (PSIFT), compiled for speed, is presented.

3) Performance analysis of SIFT-tuned to generate accurate keypoints: For certain real-time applications, speed and accuracy are both vital. This phase provides performance for PSIFT when tuned to provide accurate keypoints.

4) Effectiveness of Parallelized SIFT: GPU can spawn high number of threads. Our approach of selecting thread numbers for optimizing occupancy needs to be validated. Moreover, as mentioned before, it is challenging to find a parallel Lowe's SIFT algorithm benchmarked for an identical GPU. Therefore, using parallel profiling tools, the effectiveness of PSIFT is demonstrated by monitoring the occupancy and cache misses on the GPU for the duration of execution of PSIFT.

6.1. Phase 1

In order to compare the performance of our parallel implementation PSSED, we implemented Matlab based sequential (MSSED) and parallel (MPSSED) versions of the SSED. The MSSED version is implemented using standard Matlab functions whereas the MPSSED version is implemented using the Matlab Parallel Computing Toolbox (MPCT). The MPCT provides the capability to execute the standard convolution functions on a GPU. The PSSED was coded using CUDA C and executed on NVIDIA's C2075 GPU card hosted on a workstation equipped with Intel Xeon Phi Processor and 32 GBs of System RAM. The MSSED and MPSSED both were executed on the same host. The average computational time (ACT) for the PSSED, MSSED and MPSSED were collected with images of resolution increasing from 720 p to 8 K. The average and standard deviation of the measured computational time is determined by performing hundred trials of the

simulations.

In Figure 6, the MSSED ACT and the corresponding standard deviation (numbers on the graph) for each image resolution are shown.

The ACT for MSSED is less than a second for images with resolution less than 1080 P. Even at this low resolution, only two frames can be processed in a second using the MSSED implementation. This high value of ACT makes the MSSED implementation unsuitable for real-time applications. The MSSED ACT is comparable to that reported in the research work [3] by Heymann *et al.* In [3], the ACT for a 640 × 480 resolution image was 312 msecs compared to 401 msecs for a 1280 × 720 resolution image. Next, as the image resolution increases the ACT increases tremendously. The ACT of MSSED can be reduced further by using multiple cores of the Intel Xeon processor. It can be observed in Figure 6; the standard deviation is very less.

In Figure 7, the MPSSED ACT and the corresponding standard deviation for each image resolution are shown. Based on the ACT, the MPSSED is capable of processing 720 p and 1080 p images in real time.

In Figure 8, the PSSED ACT and the corresponding standard deviation for each image resolution are shown.

With PSSED implementation, the ACT is less than 16 msecs even with the 8 K-resolution image. This is comparable ACT of the convolution of 9 K-resolution images using GPUs, described in [40] to be 10 msecs. The low ACT would allow

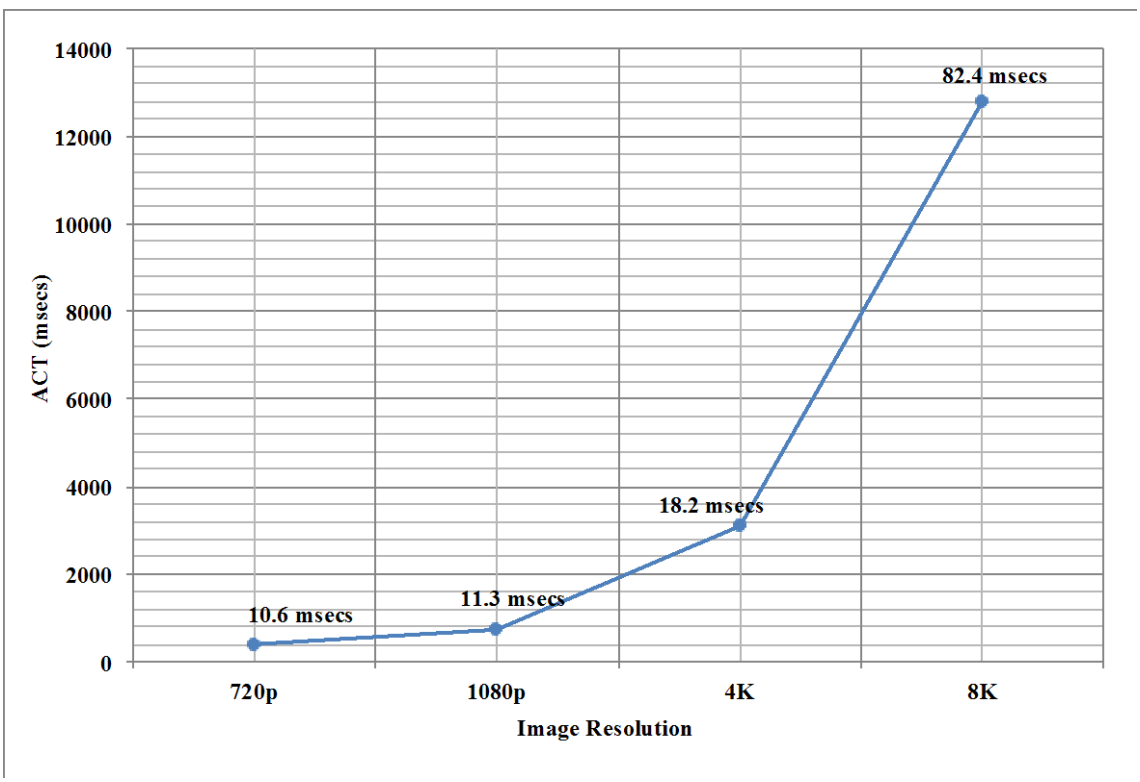

Figure 6. MSSED ACT with varying image resolutions.

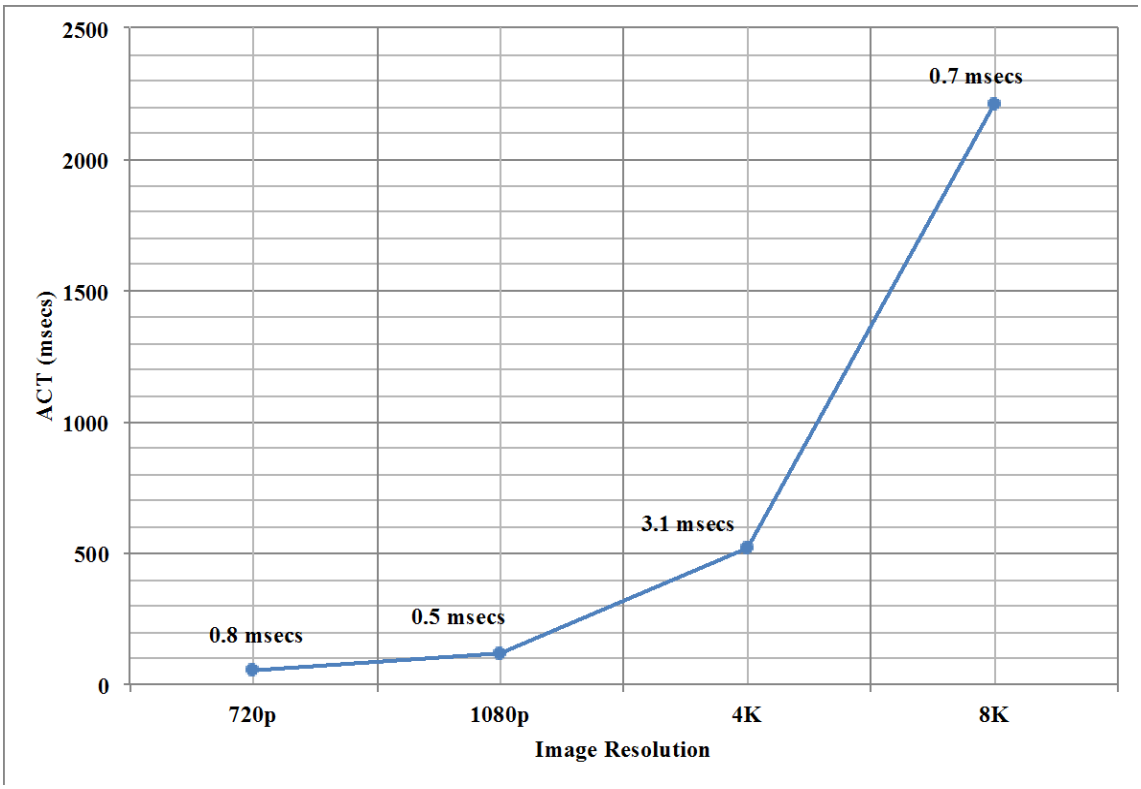

Figure 7. MPSSED ACT with varying image resolutions.

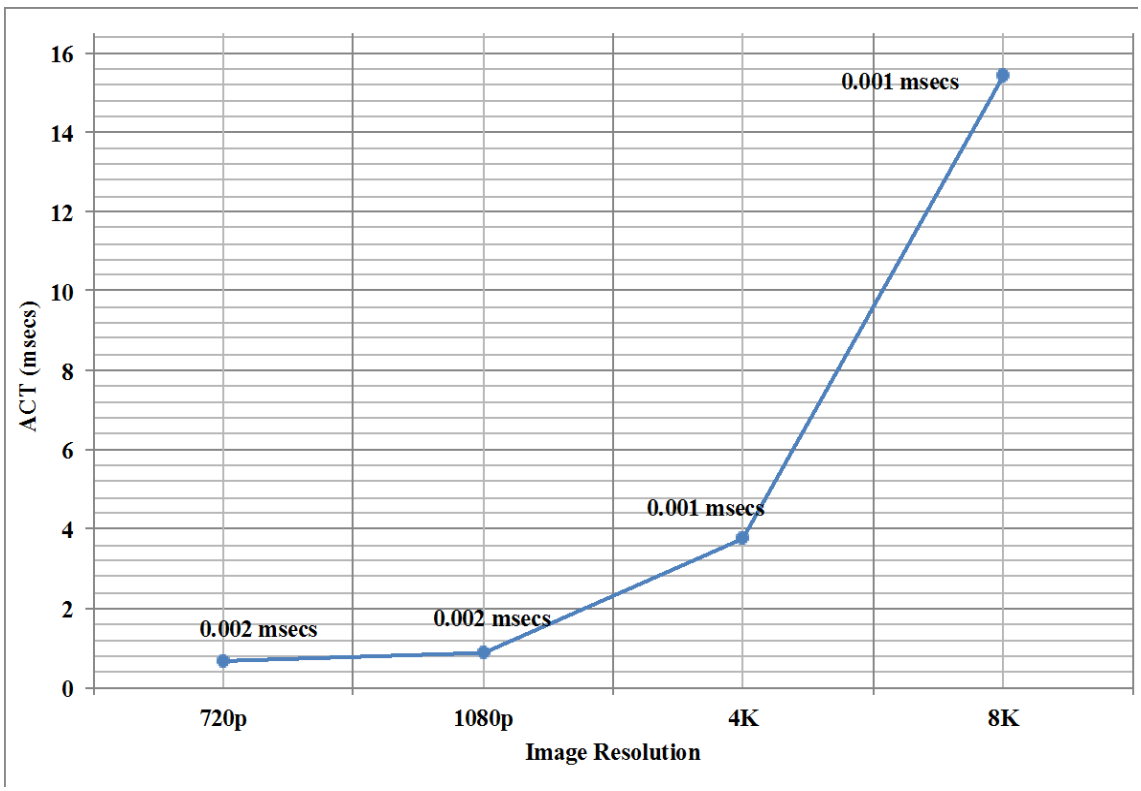

Figure 8. PSSED ACT with varying image resolutions.

processing of the 8 K resolution images at 60 fps making the PSSED implementation suitable for real-time applications. In the research work [12] it has been shown with homogenous multi-core DSP implementation a processing rate of 45 fps of 640 × 480 resolution images is achievable.

In Figure 9, the speedup of PSSED over MSSED and MPSSED are shown. The speedup is found to vary from 592× to 829× and 78× to 143× over MSSED and MPSSED respectively with increasing image resolutions. It is seen that the speedup increases initially and reaches saturation. This is due to the finite amount of memory and cores available on the GPU for concurrent execution. The optimized CUDA C PSSED implementation performance is still significantly better compared to the Matlab based GPU implementation of the SSED.

The test image used for comparing performance is shown in Figure 10, and the corresponding final DoG image by PSSED is shown in Figure 11. Identical DoG images are created by the MSSED, MPSSED and PSSED.

6.2. Phase 2

The ACT of PSIFT algorithm and its speed up w.r.t. sequential (SWS) Matlab SIFT (SSIFT) is provided in Figure 12. The ACT for a 720 p image is nearly 2.5 msecs making SIFT a feasible algorithm for real time applications. However, the ACT increases non-linearly with an increase in image resolution. The number of

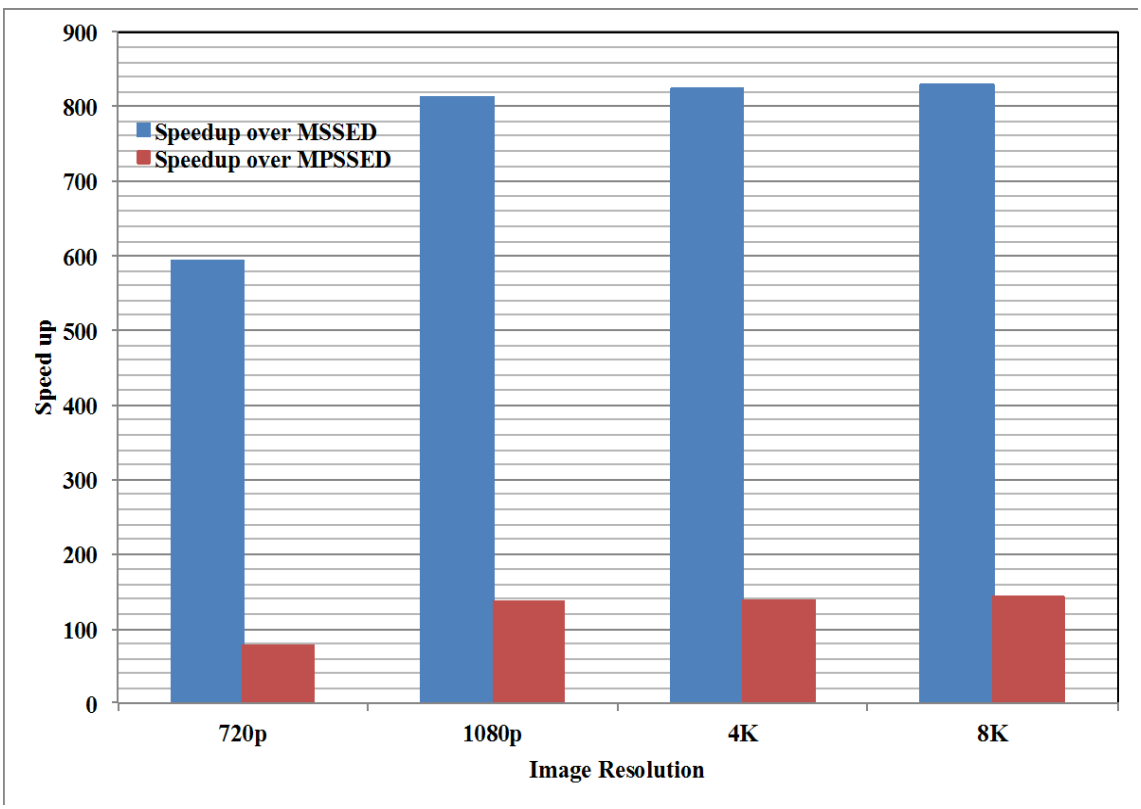

Figure 9. Speedup of PSSED over MSSED and MPSSED.

Figure 10. Test image.

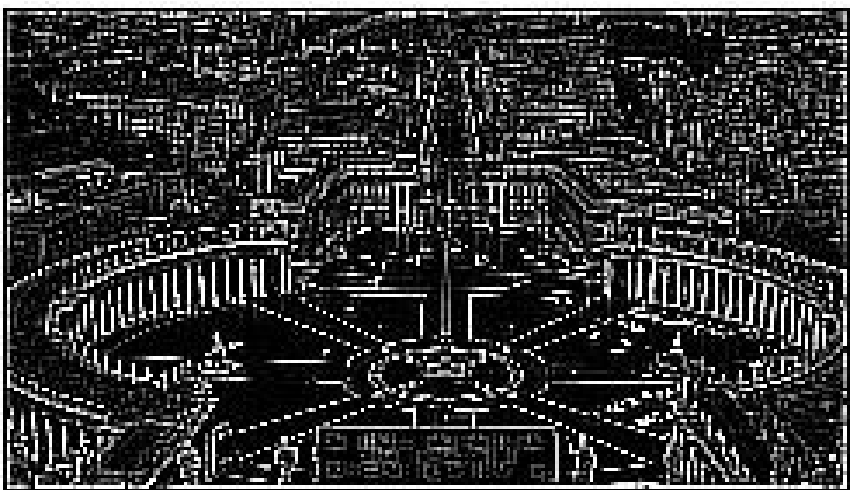

Figure 11. Final image obtained from PSSED.

image points increases with increasing image resolution causing code to execute a block of points sequentially instead of parallel execution due to the unavailability of idle SMs. Therefore, it reduces the speed up w.r.t. SSIFT. The speed up of PSIFT over SSIFT shown in **Figure 12** is only for a set of 4 images of different resolutions. A change in an image, leads to a change in CKs and keypoints, leading to different workload.

Figure 13 shows the impact of keypoints on PSIFT's performance for 720 p resolution images. The keypoints, expressed in percentage in **Figure 13**, are taken as a fraction of the total pixels of an image. As seen in **Figure 13**, the performance of PSIFT scales linearly for an increase in keypoints. Once beyond a threshold, 8% in case of **Figure 13**, the GPU has all SMs occupied. Hence, execution of slices of code is delayed till SMs are available. This is observed in the non-linear rise in ACT from 8% - 16% in **Figure 13**. **Figure 13** also displays the SWS, which are consistent up to 8%, and decrease thereafter.

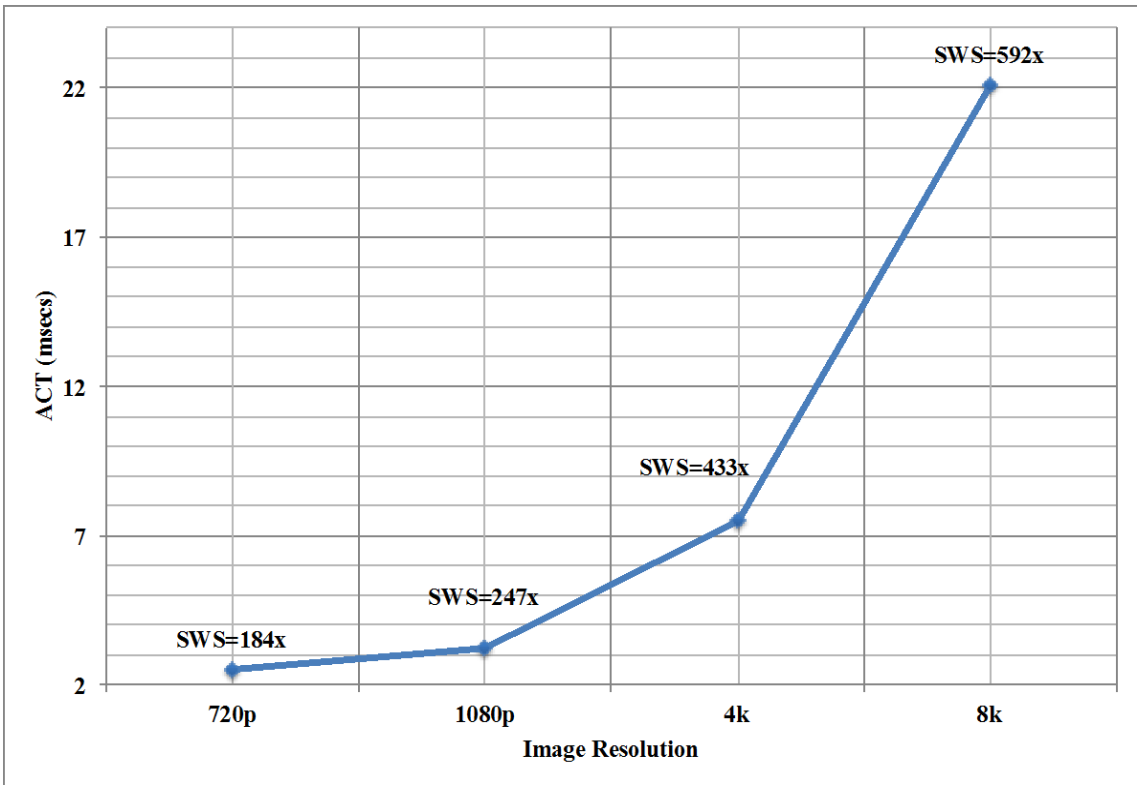

Figure 12. PSIFT performance with varying image resolutions.

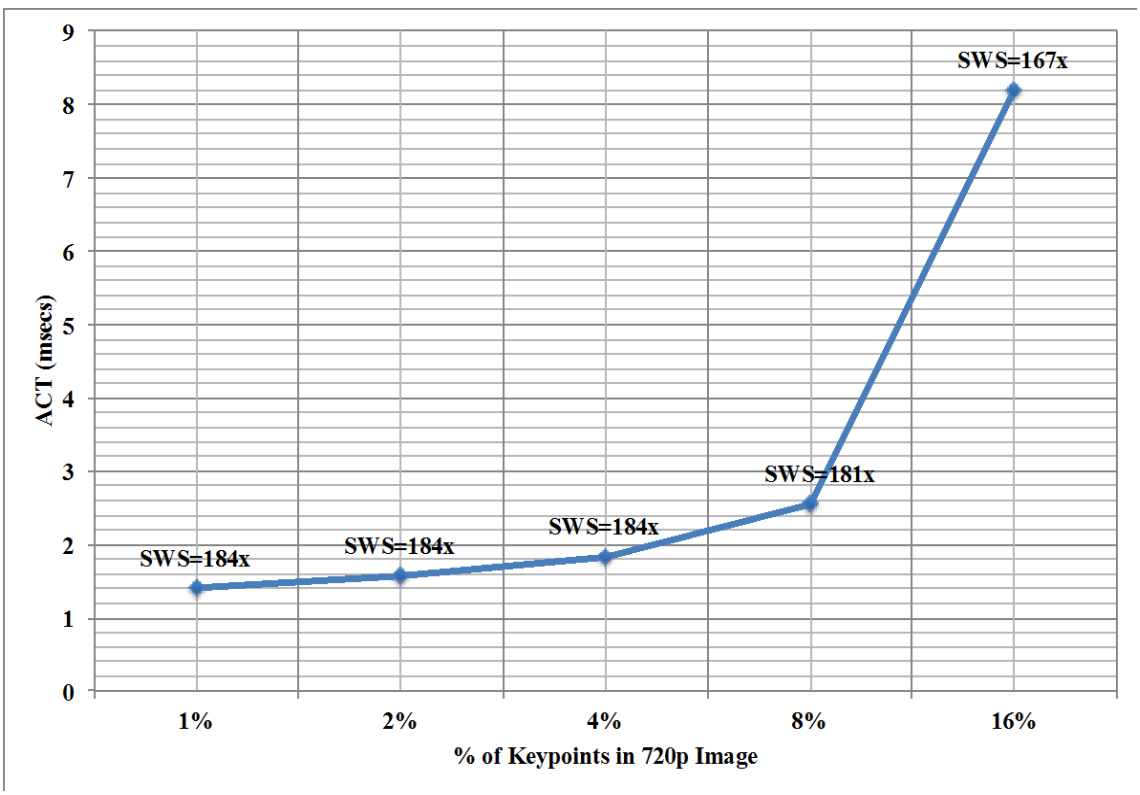

Figure 13. PSIFT performance for varying keypoints in 720 p image.

As mentioned before, phase 2 provides performance results of PSIFT tuned for faster execution. This performance boost is achieved by lowering the accuracy of the SIFT algorithm. Figure 14 shows the accuracy of PSIFT in terms of the percentage of matching keypoints generated in comparison to SSIFT, which is adopted from Lowe's algorithm. Figure 14 shows steep decrease in accuracy with an increase in performance. However, 89% keypoints matching SSIFT for 8 k images is an acceptable number of keypoints.

6.3. Phase 3

Since the accuracy of results are lower in phase 2, it is necessary to compare PSIFT optimized for ACT and PSIFT tuned for 100% accuracy. Figure 15 provides the SWS for the two approaches. It is evident that tuning PSIFT for accuracy does not scale well for higher resolution.

Figure 16 provides the SIFT descriptors generated for image. The figure shows a number of keypoints are generated as an output for the SIFT algorithm. It matches with the keypoints generated by SSIFT, verifying the accuracy.

6.4. Phase 4

In this phase, our approach to select the number of threads is analyzed. The optimal thread number for SSED is 256 threads. Figure 17 compares of performance of SIFT with optimal threads with threads lower and higher than optimal

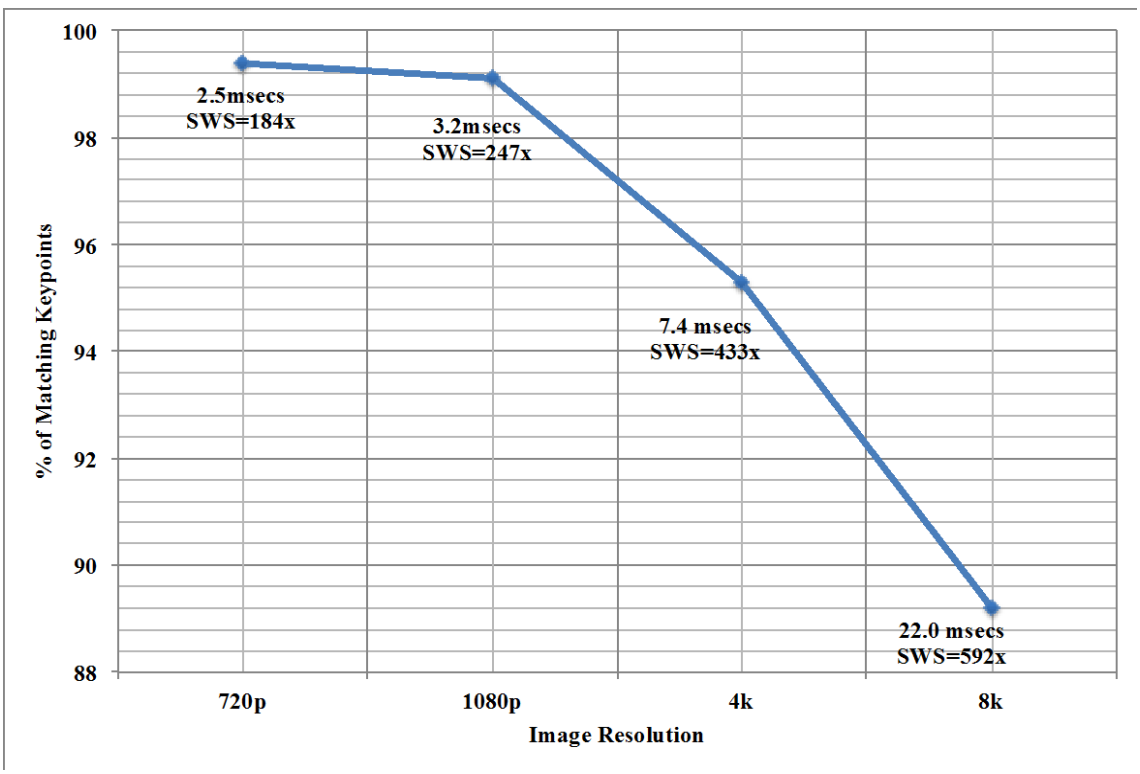

Figure 14. PSIFT accuracy for various image resolutions.

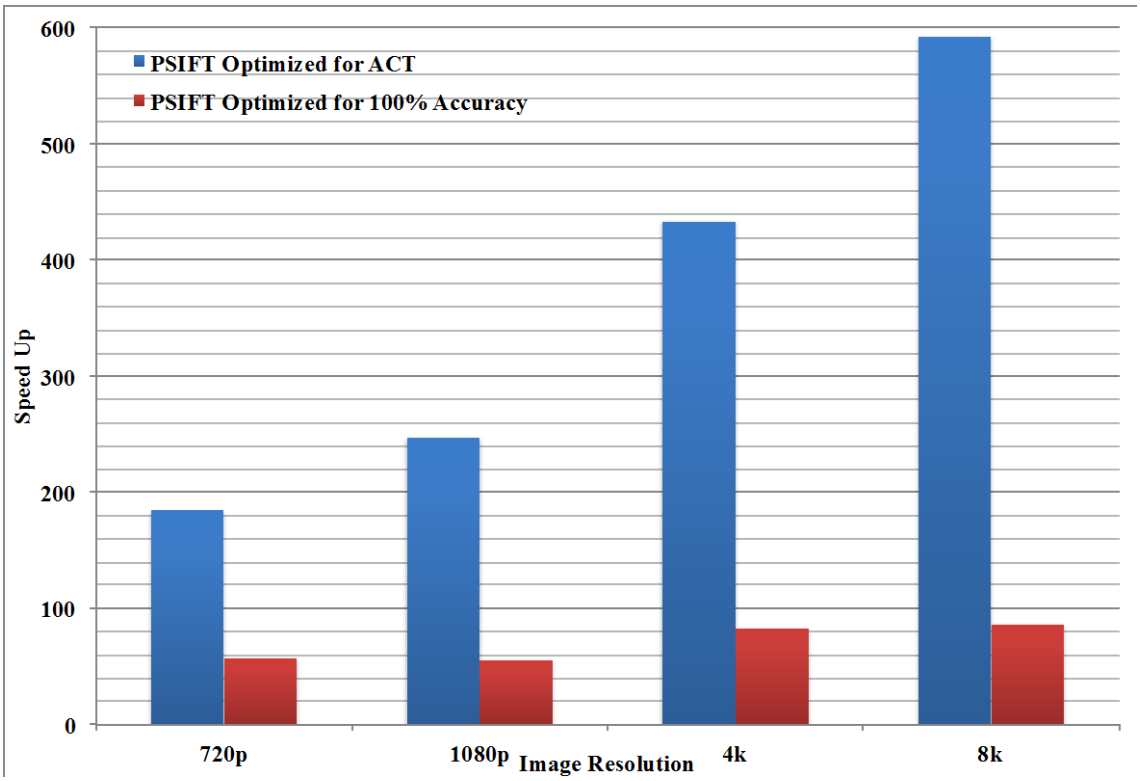

Figure 15. PSIFT accuracy for various image resolutions.

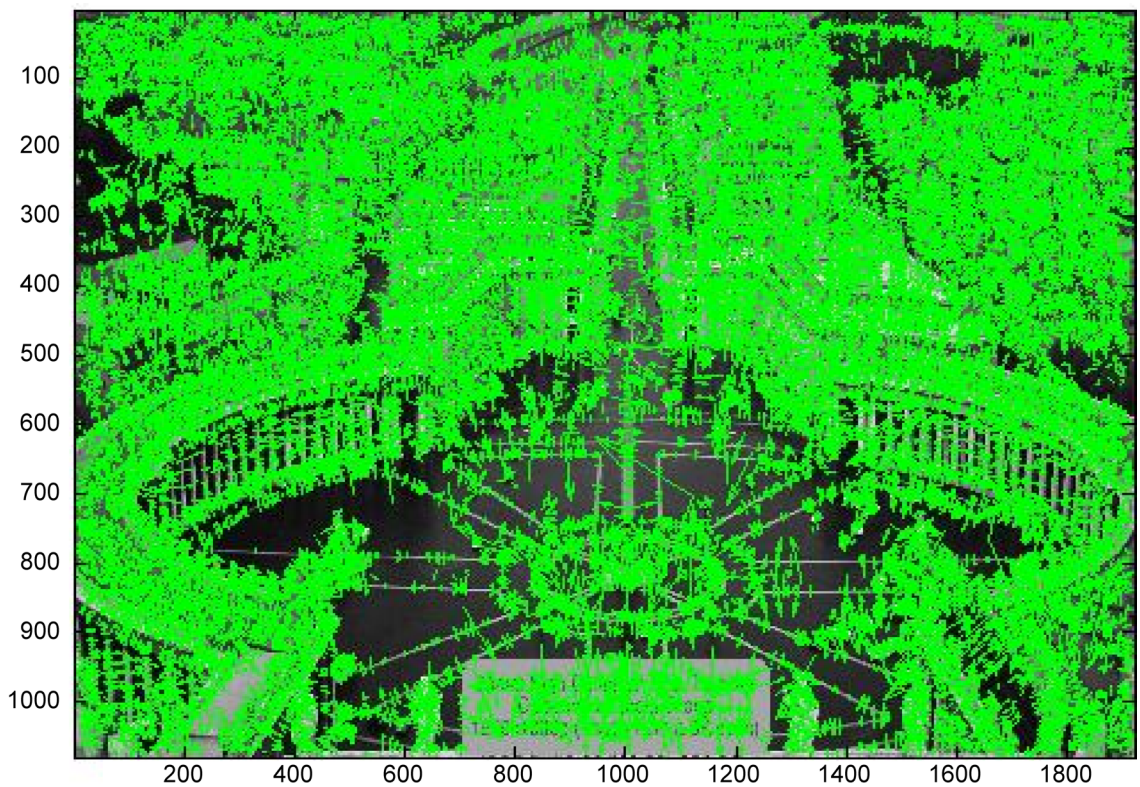

Figure 16. Descriptors generated by PSIFT.

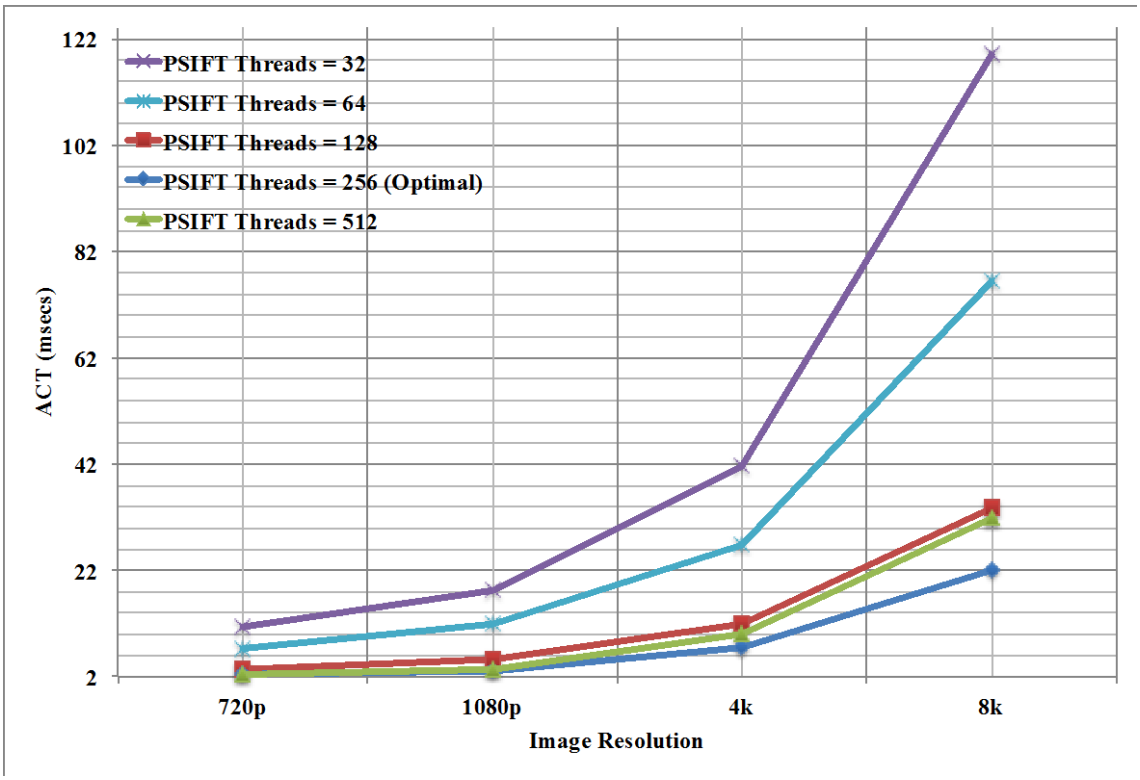

Figure 17. Performance variation in PSIFT due to variation in threads.

values. It is seen that our approach provides the best performance of the five cases. Lower threads fail to hide memory latency and hence have worse performance. Higher threads hide memory latency as much as optimal case, but for higher data sizes requires more memory than optimal case. Hence performance deteriorates as seen in **Figure 17**.

With no parameters for comparison, it is difficult to compare the effectiveness of parallelization approaches without having the same algorithm, device, and dataset. BSC has developed parallel program profiling tools that help analyze parallelization strategies and performance of the program. Two such tools are Extrae and Folding. Extrae is for collecting raw performance data from execution of software. Extrae allows parallel program developers to instrument the code to collect statistics on PAPI [39] counters to track resources within a GPU. However, doing so for a single thread does not provide insight for parallelization strategies. Therefore, the data is collected for 1000 sample runs of SIFT for 128 threads spread across different SMs. The counters are read periodically, and data is generated based on the counter. The folding tool helps merge the 1000 samples across 128 threads to provide an average view of execution on GPU. The smoother the curve folding generates, the better the reproducibility of the program. For PSIFT, PAPI counters for MIPS and LCM were monitored on a K20X GPU card. Readings were obtained from the counter for every 1 usecs, and the results are presented in **Figure 18**. It is seen that for the first 30% of the execution

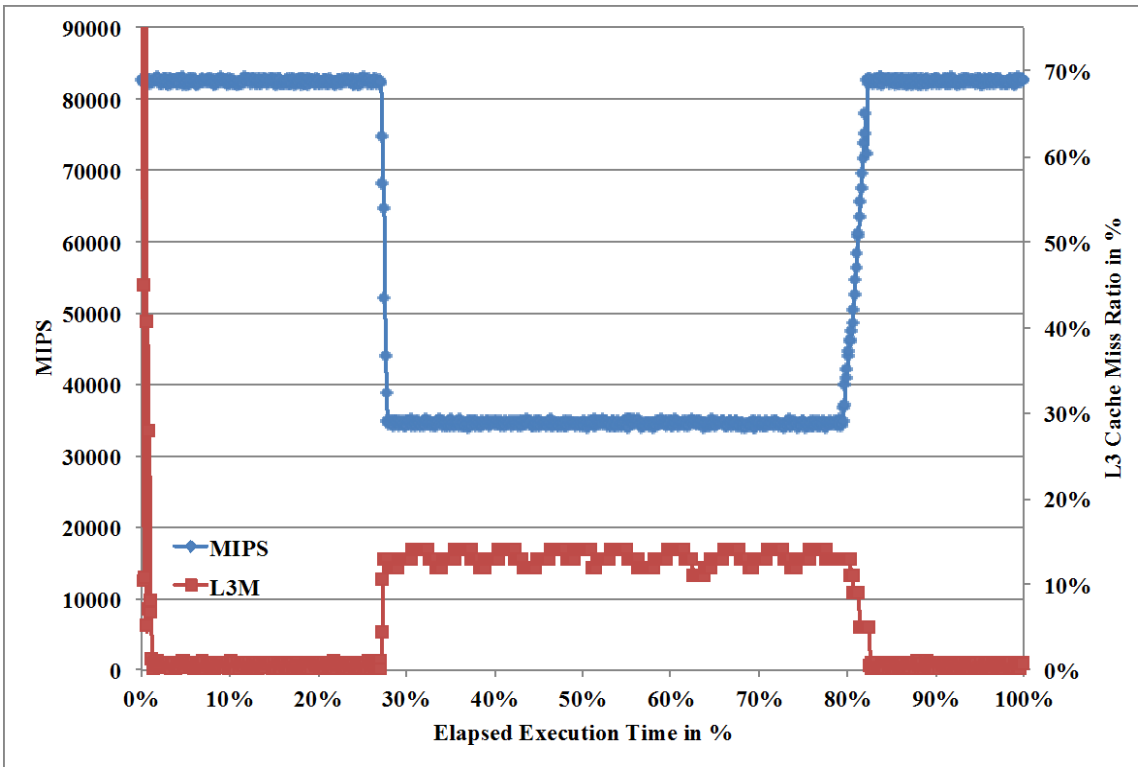

Figure 18. PSIFT GPU resource utilization.

time, the MIPS are high while LCM is lower than 2% except for the initial load. This marks the SSED phase where data is pre-fetched by the compiler reducing LCM and increasing MIPS. During the data rearrangement phase, the MIPS are cut down by 50%. The LCM during this phase is observed to be between 0% - 15% indicating scattered memory access. This phase lasts till 85% of the execution time, followed by the keypoint localization and descriptor generation, which are similar to SSED. The plot shows that the GPU MIPS is high throughout the execution of the algorithm, indicating that our parallelization strategy is effective.

7. Conclusions and Future Work

This paper demonstrates a new two-phase parallelization approach used to implement the SIFT algorithm. Our approach to the two-phase parallelization has an algorithm design phase—where parallelization strategies are based on data size, usage and organization, and an implementation design phase—where parallelization strategies are based on achieving maximum execution efficiency on a GPU. The algorithm design phase involves reduction in data usage and computations, reusing image data points to maximize cache blocking and, rearranging data to facilitate faster memory operations. The implementation design phase maximizes occupancy by adopting a mathematical model that connects occupancy with warps, shared memory, register memory, blocks, and threads. This model is then solved to get a range of GPU parameters such as number of blocks

and threads.

The performance of the parallelized SIFT thus obtained is presented, compared and analyzed. The following observations were made:

a) PSIFT performs better than the sequential and parallel Matlab implementations.

b) Accuracy of the SIFT algorithm, in terms of number of keypoints generated, can be traded to obtain lower execution time.

c) PSIFT can process high-resolution images under 25 msecs when optimized for low execution time.

d) The number of keypoints in a given image has a significant impact on the execution time of PSIFT.

e) The implementation design phase strategies provided the optimal number of threads and blocks for PSIFT.

f) The two-phase parallelization strategies did show effective use of GPU resources.

Our approach to parallelization facilitates SIFT algorithm to be used for processing high-resolution images in real time. The future work would be to provide a multi-GPU implementation.

References

[1] Lowe, D.G. (1999) Object Recognition from Local Scale-Invariant Features. *The Proceedings of the Seventh IEEE International Conference on Computer Vision*, **2**, 1150-1157. https://doi.org/10.1109/ICCV.1999.790410

[2] Skrypnyk, I. and Lowe, D.G. (2004) Scene Modelling, Recognition and Tracking with Invariant Image Features. *Third IEEE and ACM International Symposium on Mixed and Augmented Reality*, 110-119. https://doi.org/10.1109/ISMAR.2004.53

[3] Heymann, S., Muller, K., Smolic, A., Frohlich, B. and Wiegand, T. (2007) SIFT Implementation and Optimization for General-Purpose GPU. *Proceedings of the International Conference in Central Europe on Computer Graphics*, *Visualization and Computer Vision*, **144**.

[4] Li, L.Y., Tong, X.H., Jin, Y.M., Chen, P., Liu, S.J. and Hong, Z.H. (2012) Bundle Adjustment of High Resolution Stereo Camera on Mars Express. *Second International Workshop on Earth Observation and Remote Sensing Applications*, 243-245. https://doi.org/10.1109/EORSA.2012.6261174

[5] Yu, X.P., Liu, T., Li, P.X. and Huang, G.M. (2011) The Application of Improved SIFT Algorithm in High Resolution SAR Image Matching in Mountain Areas. *International Symposium on Image and Data Fusion*, 1-4.

[6] Chen, K., Lin, C., Chiu, T., Chen, M. and Hung, Y. (2011) Multi-Resolution Design for Large-Scale and High-Resolution Monitoring. *IEEE Transactions on Multimedia*, **13**, 1256-1268. https://doi.org/10.1109/TMM.2011.2165055

[7] Turner, D., Lucieer, A. and Watson, C. (2012) An Automated Technique for Generating Georectified Mosaics from Ultra-High Resolution Unmanned Aerial Vehicle (UAV) Imagery, Based on Structure from Motion (SfM) Point Clouds. *Remote Sensing*, **4**, 1392-1410. https://doi.org/10.3390/rs4051392

[8] Feng, H., Li, E., Chen, Y. and Zhang, Y. (2008) Parallelization and Characterization

of SIFT on Multi-Core Systems. *IEEE International Symposium on Workload Characterization*, 14-23.

[9] Feng, L., Xu, T., Huang, Q., Wang, X. and Wang, P. (2010) SIFT Implementation Based on Parallel Computation. 6*th International Conference on Wireless Communications Networking and Mobile Computing*, 1-4. https://doi.org/10.1109/wicom.2010.5600662

[10] NVIDIA Developer Forum (2015). https://developer.nvidia.com/search/gss/SIFT

[11] Warn, S., Emeneker, W. and Cothren, J. (2009) Accelerating SIFT on Parallel Architectures. *International Conference on Cluster Computing and* Workshops.

[12] Liu, X., Chen, W.J., Ma, T. and Xu, L.S. (2011) Real-Time Algorithm for Sift Based on Distributed Shared Memory Architecture with Homogeneous Multi-Core dsp. 2*nd International Conference on Intelligent Control and Information Processing*, **2**, 839-843. https://doi.org/10.1109/icicip.2011.6008366

[13] Yao, L., Feng, H., Zhu, Y., Jiang, Z., Zhao, D. and Feng, W. (2009) An Architecture of Optimised SIFT Feature Detection for an FPGA Implementation of an Image Matcher. *International Conference on Field-Programmable Technology*, 30-37. https://doi.org/10.1109/fpt.2009.5377651

[14] Kim, J., Park, E., Cui, X., Kim, H. and Gruver, W.A. (2009) A Fast Feature Extraction in Object Recognition Using Parallel Processing on CPU and GPU. *IEEE International Conference on Systems*, *Man and Cybernetics*, 3842-3847.

[15] Jiang, G., Zhang, G. and Zhang, D. (2010) A Distributed Dynamic Parallel Algorithm for SIFT Feature Extraction. 3*rd International Symposium on Parallel Architectures*, *Algorithms and Programming*, 381-385. https://doi.org/10.1109/PAAP.2010.58

[16] Kang, S.H., Lee, S.J. and Park, I.K. (2014) Parallelization and Optimization of Feature Detection Algorithms on Embedded GPU. *International Workshop on Advanced Image Technology*, **108**, 164-167.

[17] Huang, M. and Lai, C. (2014) Parallelizing Computer Vision Algorithms on Acceleration Technologies: A SIFT Case Study. *IEEE China Summit & International Conference on Signal and Information Processing*, 325-329.

[18] Fassold, H. and Rosner, J. (2015) A Real-Time GPU Implementation of the SIFT Algorithm for Large-Scale Video Analysis Tasks. *SPIE/IS&T Electronic Imaging*, International Society for Optics and Photonics.

[19] Mohammadi, M.S. and Rezaeian, M. (2014) Towards Affordable Computing: Sift-CU a Simple but Elegant GPU-Based Implementation of SIFT. *International Journal of Computer Applications*, **90**.

[20] Marinelli, M., Mancini, A. and Zingaretti, P. (2014) GPU Acceleration of Feature Extraction and Matching Algorithms. *IEEE/ASME* 10*th International Conference on Mechatronic and Embedded Systems and Applications*, 1-6. https://doi.org/10.1109/mesa.2014.6935620

[21] Kumar, R.P., Muknahallipatna, S.S. and McInroy, J.E. (2014) SIFT's Scale-Space Extrema Detection on GPU for Real-Time Applications (WIP). *Proceedings of the* 2014 *Summer Simulation Multiconference*, 67.

[22] Brown, M. and Lowe, D.G. (2002) Invariant Features from Interest Point Groups. *Proceedings of the British Machine Vision Conference*, **1**. https://doi.org/10.5244/c.16.23

[23] Fung, J. and Mann, S. (2004) Using Multiple Graphics Cards as a General Purpose

Parallel Computer: Applications to Computer Vision. *Proceedings of the* 17*th International Conference on Pattern Recognition*, **1**, 805-808. https://doi.org/10.1109/ICPR.2004.1334339

[24] NVIDIA, NVIDIA CUDA. http://www.nvidia.com/object/cuda_home_new.html

[25] NVIDIA, NVIDIA CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide

[26] Meng, J., Morozov, V.A., Vishwanath, V. and Kumaran, K. (2012) Dataflow-Driven GPU Performance Projection for Multi-Kernel Transformations. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 82.

[27] Zhang, Y. and Mueller, F. (2012) Auto-Generation and Auto-Tuning of 3D Stencil Codes on GPU Clusters. *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 155-164. https://doi.org/10.1145/2259016.2259037

[28] Meng, J. and Skadron, K. (2011) A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations. *International Journal of Parallel Programming*, **39**, 115-142. https://doi.org/10.1007/s10766-010-0142-5

[29] Unat, D., Cai, X. and Baden, S.B. (2011) Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. *Proceedings of the International Conference on Supercomputing*, 214-224. https://doi.org/10.1145/1995896.1995932

[30] Maruyama, N., Sato, K., Nomura, T. and Matsuoka, S. (2011) Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers. *International Conference for High Performance Computing, Networking, Storage and Analysis*, 1-12. https://doi.org/10.1145/2063384.2063398

[31] Stromme, A., Carlson, R. and Newhall, T. (2012) Chestnut: A Gpu Programming Language for Non-Experts. *Proceedings of the* 2012 *International Workshop on Programming Models and Applications for Multicores and Manycores*, 156-167. https://doi.org/10.1145/2141702.2141720

[32] Nugteren, C. and Corporaal, H. (2012) Introducing "Bones": A Parallelizing Source-to-Source Compiler Based on Algorithmic Skeletons. *Proceedings of the* 5*th Annual Workshop on General Purpose Processing with Graphics Processing Units*, 1-10. https://doi.org/10.1145/2159430.2159431

[33] Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C. and Catthoor, F. (2013) Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization*, **9**, 54. https://doi.org/10.1145/2400682.2400713

[34] Brodtkorb, A.R., Hagen, T.R. and Sætra, M.L. (2013) Graphics Processing Unit (GPU) Programming Strategies and Trends in GPU Computing. *Journal of Parallel and Distributed Computing*, **73**, 4-13. https://doi.org/10.1016/j.jpdc.2012.04.003

[35] Liu, L., Li, Z. and Sameh, A.H. (2008) Analyzing Memory Access Intensity in Parallel Programs on Multicore. *Proceedings of the* 22*nd Annual International Conference on Supercomputing*, 359-367. https://doi.org/10.1145/1375527.1375579

[36] Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B. and Hwu, W.M.W. (2008) Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. *Proceedings of the* 13*th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 73-82. https://doi.org/10.1145/1345206.1345220

[37] Volkov, V. (2010) Better Performance at Lower Occupancy. *Proceedings of the GPU Technology Conference*, **10**, 16.

[38] Iandola, F.N., Sheffield, D., Anderson, M.J., Phothilimthana, P.M. and Keutzer, K. (2013) Communication-Minimizing 2D Convolution in GPU Registers. *IEEE International Conference on Image Processing*, 2116-2120. https://doi.org/10.1109/icip.2013.6738436

[39] Papi. http://icl.cs.utk.edu/papi/

[40] Vedaldi, A. (2007) An Open Implementation of the SIFT Detector and Descriptor. UCLA CSD.