OPTIMIZING HARRIS CORNER DETECTION ON GPGPUs USING CUDA

A Thesis

presented to

The Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Masters of Science in Electrical Engineering

by

Justin Loundagin

March 2015

COMMITTEE MEMBERSHIP

TITLE:	Optimizing Harris Corner Detection on GPGPUs Using CUDA
AUTHOR:	Justin Loundagin
DATE SUBMITTED:	March 2015
COMMITTEE CHAIR:	Jane Zhang, PhD Professor of Electrical Engineering, Associate Dept. Chair
COMMITTEE MEMBER:	Lynne Slivovsky, PhD Professor of Computer Engineering
COMMITTEE MEMBER:	Dennis Derickson, PhD Professor of Electrical Engineering, Department Chair

ABSTRACT

Optimizing Harris Corner Detection on GPGPUs Using CUDA

Justin Loundagin

The objective of this thesis is to optimize the Harris corner detection algorithm implementation on NVIDIA GPGPUs using the CUDA software platform and measure the performance benefit. The Harris corner detection algorithm—developed by C. Harris and M. Stephens—discovers well defined corner points within an image. The corner detection implementation has been proven to be computationally intensive, thus realtime performance is difficult with a sequential software implementation. This thesis decomposes the Harris corner detection algorithm into a set of parallel stages, each of which are implemented and optimized on the CUDA platform. The performance results show that by applying strategic CUDA optimizations to the Harris corner detection implementation, realtime performance is feasible. The optimized CUDA implementation of the Harris corner detection algorithm showed significant speedup over several platforms: standard C, MATLAB, and OpenCV. The optimized CUDA implementation of the Harris corner detection algorithm was then applied to a feature matching computer vision system, which showed significant speedup over the other platforms.

Keywords: Harris Corner Detection, NVIDIA GPGPU, NVIDIA CUDA

ACKNOWLEDGMENTS

I would like to express my gratitude towards my thesis advisor and committee member Dr. Zhang for her knowledge and guidance. This thesis would not have been possible without the knowledge I have acquired from her expertise in the fields of image processing and computer vision. I would like to thank Dr.Derickson and Dr.Slivovsky for serving as members on my thesis committee.

I would like to thank my family, especially my father James Loundagin, who helped me discover my passion for electrical engineering. My family has always pushed me to go farther with my education—for that I can never thank them enough.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: Introduction

1.1 Thesis Introduction

The goal of computer vision is to model and replicate the human visual system through computer software and hardware and build autonomous systems [1]. Replicating the human visual system on a computational platform has proven to be inherently difficult. Computer vision is the field of understanding the 3D world from 2D images, however details of the 3D world are lost during image formation, thus making computer vision difficult. High-level computer vision systems rely on low-level processes, such as corner detection, to perform accurately [1].

GPUs (graphics processing unit) have become increasingly programmable over the past few decades. NVIDIA has led the field in parallel computing with their intuitive software, CUDA (Compute Unified Device Architecture), and highly optimized GPGPU (general purpose graphics processing unit) hardware. This thesis discusses the implementation of the Harris corner detection algorithm on the NVIDIA GPGPU by utilizing the CUDA software platform. Corner detection is computationally intensive, thus a realtime implementation has proven to be difficult. High-speed corner detection is in high demand for computer vision systems in applications such as motion detection, video tracking, augmented reality, and object recognition [2]. The objective of this thesis is to analyze the performance benefit of implementing and optimizing the Harris corner detection algorithm on the NVIDIA GPGPU platform using CUDA.

1.2 Thesis Organization

Chapter 1 introduces the background of the Harris corner detection algorithm and the history of the GPGPU computing platform. The chapter discusses the related work of utilizing CUDA for Harris corner detection which has been done prior to this work.

Chapter 2 presents an overview of the NVIDIA GPGPU hardware architecture and the CUDA architecture. An overview on how the CUDA software architecture runs on the GPGPU hardware will be explained, which will later justify parallel optimization strategies made to the Harris corner detection implementation. General CUDA and GPGPU optimizations will be discussed to provide a basis for general speed enhancements for parallel algorithm implementations.

Chapter 3 will present an overview of the mathematical representation of the Harris corner detection algorithm. The Harris corner detection algorithm will then be decomposed into a software architecture representation. Each stage in the software architecture will then be briefly discussed, along with its purpose and algorithmic function.

Chapter 4 will discuss the naive and optimized CUDA implementations of each stage in Harris corner detection software architecture: convolution, corner detection, and non-maxima suppression. At each stage, different optimization strategies will be discussed, and the performance will be compared to other platforms: standard C, MATLAB, and naive CUDA. Once each stage has been fully optimized to run on the GPGPU hardware, the performance of the overall Harris corner detection implementation will be analyzed and compared to other platforms.

In Chapter 5, Harris corner detection will be applied to a feature matching computer vision system. The performance benefit gained by incorporating the optimized CUDA Harris corner detection implementation into the feature matching system will be compared against several platforms.

Chapter 6 will describe future work for GPGPU Harris corner detection and will conclude this thesis.

1.3 Related Work

A paper published in 2011, “Low Complexity Corner Detector Using CUDA for Multimedia Applications”, investigated the performance benefit gained by implementing the Harris corner detection algorithm using CUDA [3]. Rajah Phull, Pradip Mainali, and Quiong Yang from the Institute of BroadBand Technology optimized the Harris corner detection algorithm by utilizing several different optimization strategies: shared memory, coalesced memory accesses, and thread occupancy. The paper focused on optimizing the LoCoCo (Low Complexity Corner) detector rather than the traditional Harris corner detection algorithm for added performance benefit. The LoCoCo detection algorithm sacrifices accuracy to increase performance by approximating the Gaussian derivative with a box filter. This implies that integral images can be utilized to reduce the number of arithmetic operations required for image convolution.

The performance results of the CUDA LoCoCo implementation were compared to the CPU implementation. The CUDA LoCoCo was designed to run on the NVIDIA GeForce 280 GTX GPGPU, specifications shown in Table 1.1. The performance analysis revealed that their CUDA LoCoCo implementation had around a 14 times faster speedup over the CPU implementation [3]. The paper was the first to report the findings of CUDA performance benefit when applied to the corner detection. The paper showed that their implementation had a significant performance improvement, however it didn’t not fully utilize the GPGPU nor advanced optimizations to further increase performance. This thesis will utilize a modern GPGPU (specification located in Appendix A) and the algorithm will be tuned for its specification. This thesis will explore in-depth CUDA optimization strategies for each stage of the Harris corner detection algorithm to maximize performance benefit without compromising precision.

GeForce 280 GTX Specifications	
CUDA Cores	240
Clock Rate	1.40 GHz
SM Count	30
Warp Size	32
Shared Memory	16 KB
Constant Memory	64 KB

Table 1.1: GeForce 280 GTX Specifications

A paper published in 2008, “Accelerated Corner-Detector Algorithms”, investigated the performance benefit of implementing corner detection algorithms on GPGPUs [4]. Lucas Teixeira, Waldemar Celes, and Marcelo Gattass, from Tecgraf (Technical Scientific Software Development Institute) designd a template for the KLT and Harris corner detector to run on the GPGPU. The paper focused on the GPGPU compression of the corner response to reduce memory bandwidth in the non-maxima suppression (NMS) algorithm. Their method to increase performance was to reduce the number of global memory reads during the NMS process [4]. The corner response compression was implemented by performing a reduction on all 2x2 neighborhood in the corner response, effectively decreasing the corner response size by a factor of 2. The compression was implemented by iterating a 2x2 window over all pixel locations with even parity (skipping every other pixel), and executing the neighborhood reduction shown in Equation 1.1. The result of the compression is an output image which represents all of the 2x2 neighborhood maxima in the original input, example shown in Figure 1.1.

$$\overline{p}(x,y) = \max\{p(2x,2y), p(2x+1,2y), p(2x,2y+1), p(2x+1,2y+1)\}$$

Equation 1.1: Corner Response Compression Equation

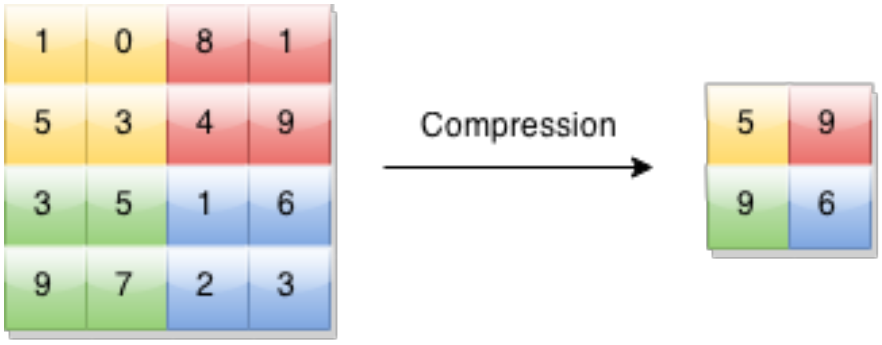


Figure 1.1: Corner Response Compression Example

The compression of the corner response reduces the memory bandwidth overhead from the GPGPU to the host by a factor of 2, thus increasing performance. Their GPGPU implementation was implemented to run on the NVIDIA GeForce 8800 GTX GPGPU, specifications shown in Table 1.2. Their performance findings for GPGPU corner response compression yielded a precision error of roughly 0.02 for Harris corner detection [4]; however, their GPGPU speedup resulted in NMS processing times not exceeding 6 ms for image dimensions of 1024 x 768. Their implementation achieves higher performance by sacrificing precision in the corner response calculation. This thesis will investigate alternative optimizations to achieve realtime performance without compromising corner response precision.

GeForce 8800 GTX Specifications	
CUDA Cores	128
Clock Rate	1.35 GHz
SM Count	16
Warp Size	32
Shared Memory Per SM	16 KB
Constant Memory	64 KB

Table 1.2: GeForce 8800 GTX Specifications

Both papers proposed CUDA implementations of the Harris corner detection algorithm which yielded higher performance over the CPU implementation. This thesis will implement more advanced CUDA optimizations to further increase performance without compromising precision.

Chapter 2: NVIDIA GPGPU and CUDA

2.1 GPGPU Introduction

Beginning in the late 1990's, the NVIDIA GPU (graphics processing unit) had become increasingly programmable. Since the revolution of the GPU platform, many developers were adapting GPU hardware into their preexisting graphical systems to increase performance. Programmers were also able to achieve performance increases on non-graphical systems by embedding their algorithms within the vertex and fragment shaders in GPU graphics pipeline. However, this was nontrivial, for programmers had to map their non-graphic algorithms into a graphics pipeline which focused primarily on triangles and polygons. In 2003, Ian Buck unveiled the first generic extension to C which allowed for parallel constructs—the Brooke compiler. NVIDIA coupled the Brooke language extension into their specialized hardware and created the first ever solution to general purpose parallel computing.

Parallel computation has been gaining popularity in the past few decades due to the performance benefits over sequential computation. NVIDIA states, "Driven by the insatiable market demand for realtime, high-definition 3D graphic, the Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational horsepower" [5]. Massive GPGPU parallelism is achieved through the massive replication of simple SIMD (single instruction multiple data) processors, known as streaming multiprocessors [6]. NVIDIA was the first to integrate an intuitive parallel software model into their highly optimized GPGPU hardware. Alternative software parallel constructs exists for parallel computing (openCL, openACC); however, CUDA has been the flagship software platform for GPGPU computation due to its intuitive nature, and its coupling with optimized NVIDIA hardware. CUDA was developed by NVIDIA with several goals in mind: provide a small set of extensions to standard

programming languages (C/C++), support heterogenous computation where applications can utilize both the CPU and GPGPU hardware [7].

NVIDIA GPGPUs are parallel processing units which have the capability of running thousands of concurrent threads in parallel. The GPGPU streaming multiprocessors (SM) have shared resources and on-chip memory which allows for parallel tasks to run with higher performance [7]. The difference between a GPU and GPGPU is that a GPU only allows for graphic mono-directional data transfers from the host CPU to the GPU. GPGPUs allow for bidirectional data transfers from the host CPU to the GPGPU and vice-versa through the PCI express bus to perform generic parallel algorithm computations. No prior knowledge of the graphics pipeline is required for CUDA programming and general algorithms can be decomposed into thousands of concurrent threads, executed in parallel, to achieve a significant performance benefit. Section 2.2 will discuss CUDA algorithm scalability between hardware configurations, and why it allows for contemporary CUDA implementations. Sections 2.3-2.4 will give an introduction into the GPGPU platform and its basic hardware components: streaming multiprocessor, and memory types. Section 2.5-2.6 will discuss a CUDA overview and the general optimizations which can be applied to all CUDA implementations.

2.2 GPGPU Scalability

The basis for CUDA popularity is due to automatic scaling of threads to GPGPU hardware configurations. Rob Farber, CEO of TechEnablement and CUDA expert, states that the “software abstraction of thread blocks translates into a natural mapping of the kernel onto an arbitrary number of SMs” [6]. The abstraction between thread blocks and GPGPU hardware allow CUDA implementations developed today to eventually scale to hardware configurations with higher performance specifications. Scalability allows for CUDA programmers to create general parallel implementations, and by updating the GPGPU hardware, the programmer can expect an improved performance benefit.

2.3 GPGPU Streaming Multiprocessor

The parallel architectural building block for the NVIDIA GPGPU is the streaming multiprocessor (SM), for the number of SMs on a GPGPU determines the degree of physical parallelism possible. The massive set of CUDA threads are partitioned into fixed sized thread blocks in the execution configuration. CUDA threads are grouped into blocks, and CUDA blocks are configured into a grid. Each SM is assigned blocks of threads which the SM is responsible for executing. The SM will further partition the blocks into warps, where each warp will be scheduled independently to run all of its threads with lock-step level parallelism. Threads within a thread block are guaranteed to run on the same SM, therefore threads within the same block can utilized local on-chip memory types: shared memory, and L1 cache. The scheduling of thread blocks to particular SMs is the job of the NVIDIA global scheduler, which will base its scheduling on the number of thread blocks, and the number of thread per a single block in the execution configuration. Multiple thread blocks can be scheduled to the same SM, if the number of thread blocks outweigh the number of SMs on the GPGPU. Figure 2.1 shows an example of how CUDA thread blocks are mapped to streaming multiprocessors on the NVIDIA GPGPU.

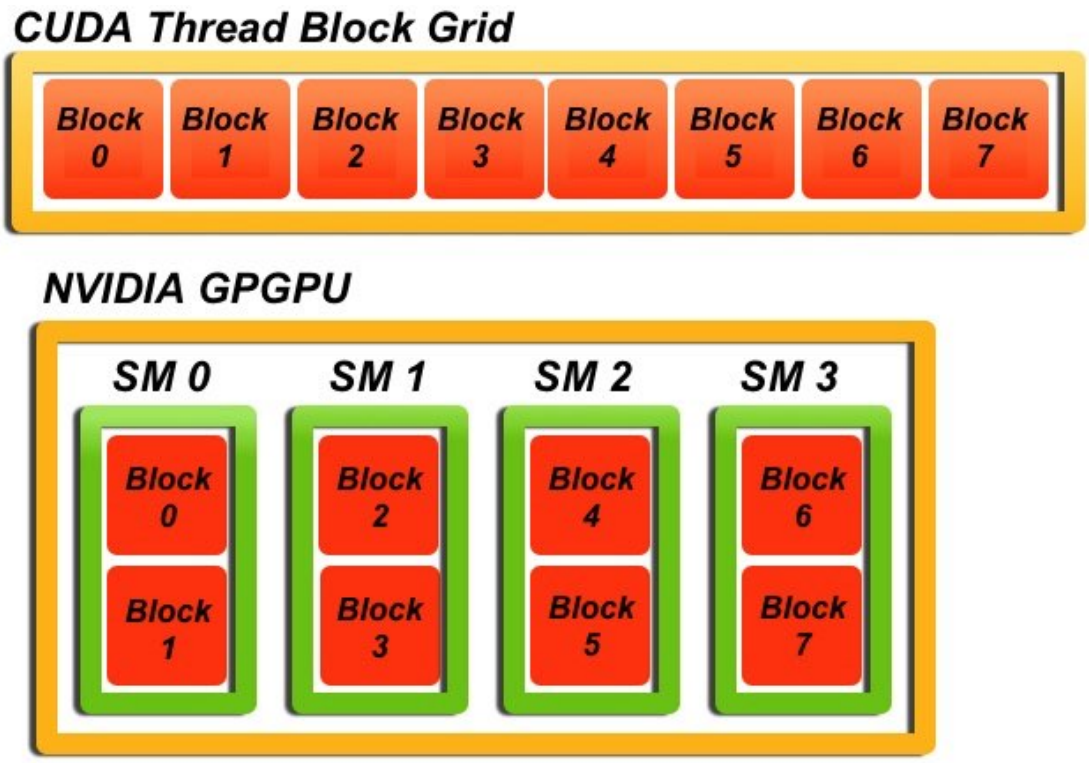


Figure 2.1: CUDA Thread Block Configuration on GPGPU

The SM contains a large array of SIMD (single instruction multiple data) processing cores. SIMD implies that the processing units within an SM will run the same instruction in lock-step level parallelism on different data. As stated earlier, the SM further partitions the scheduled block of threads into units called warps. A warp is the fundamental unit of parallelism defined on NVIDIA GPGPU hardware. Since the CUDA cores have an SIMD architecture, each thread within a warp must run the same instruction, or have to idle. The GPGPU Kepler architecture uses a quad warp scheduling scheme, where each SM is capable of executing four warps, of size 32 threads, in parallel. This implies a single SM on the Kepler architecture has the capability to execute 128 SIMD threads concurrently. Table A.3 in Appendix A shows the specific SM architecture for the NVIDIA GeForce 660 Ti—the GPGPU used to conduct this thesis research.

2.4 GPGPU Memory Types

NVIDIA GPGPUs contain various types of memory, each of which have their own performance characteristics. The fastest, however least abundant memory types on the GPGPU are the L2 cache, shared memory, and registers, for they are embedded directly onto the streaming multiprocessors. The slowest memory type on the GPGPU is global memory, however it is the most abundant memory on the GPGPU. The memory hierarchy shown in Figure 2.2 shows the basic memory layout of a generic NVIDIA GPGPU. Memory performance is inversely proportional to the size of the memory on the GPGPU, for slower off-chip memory types are more abundant than faster on-chip memory types. Table 2.2 shows the characteristics of some of the different memory types on the NVIDIA GPGPU, ordered from fastest to slowest performance memory.

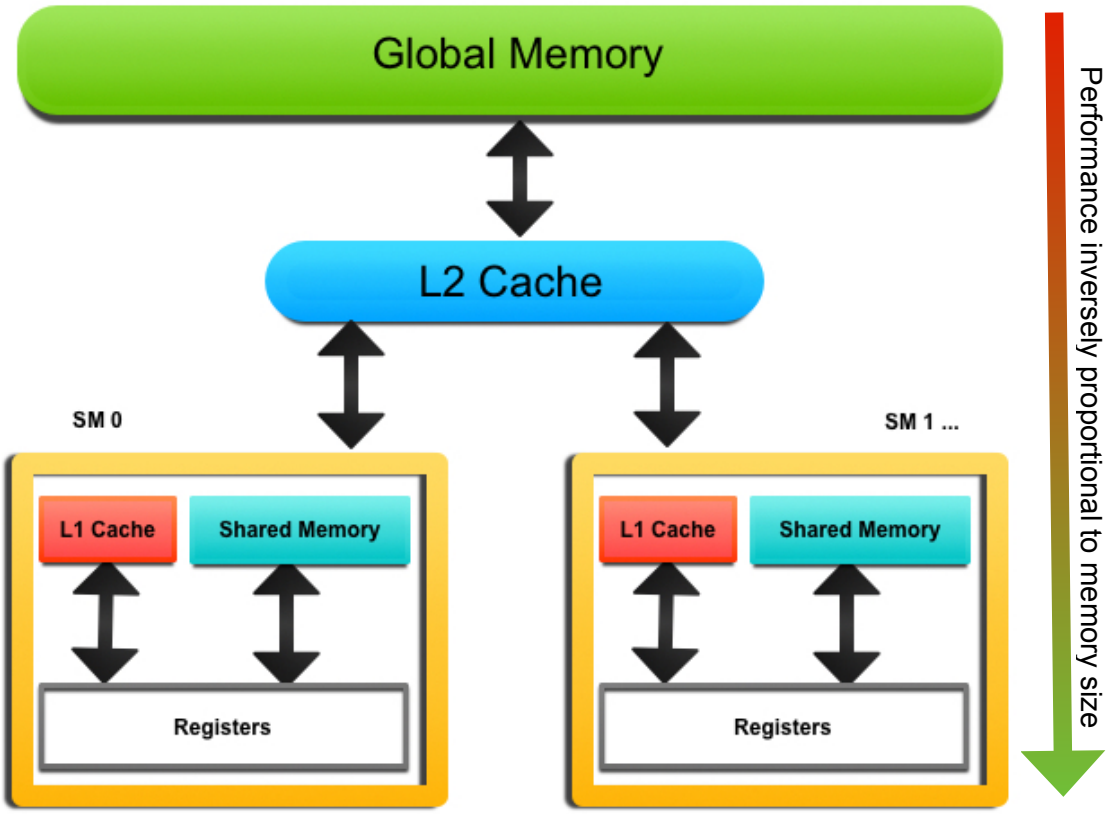


Figure 2.2: NVIDIA GPGPU Memory Hierarchy

Memory Type	Size	Cached	On Chip	Scope
Register Count	65536	No	Yes	Single Thread
L1 Cache	64 KB	N/A	Yes	Single Block
Shared Memory	48 KB	No	Yes	Single Block
L2 Cache	384 KB	N/A	No	All Threads
Global Memory	2048 MB	Yes	No	All threads

Table 2.1: Table of Various GPU Memory Characteristics

When optimizing parallel CUDA implementations, the programmer should always strive to utilize local on-chip memory that is directly integrated onto the streaming multiprocessor, specifically shared memory. Farber states, “Managing the significant performance difference between on-board and on-chip memory is the primary concern of a CUDA programmer” [6]. The avoidance of global memory accesses is typically the first optimization when programing NVIDIA GPGPUs. Every CUDA algorithm implementation can be benchmarked by its CGMA (compute to global memory access) ratio; thus, higher the ratio implies more computation for a single global memory access.

2.5 CUDA Overview

As mentioned in the earlier sections, CUDA is the software platform which allows users to interface with the NVIDIA GPGPU hardware. CUDA is not a programming language itself, rather it is a C/C++ extension which enables parallel constructs. The CUDA platform provides three key abstractions: thread group hierarchy, shared memories, and barrier synchronization [5]. CUDA revolves around the idea of a kernel, or GPGPU function, which is executed for every thread, in every block, within the configured grid.

Invoking a CUDA kernel involves firstly creating a thread hierarchy composed of the thread blocks, and threads per a block. As mentioned earlier, threads are grouped into what are called thread blocks. A thread grid is formed by first building a N-dimensional array of blocks, then defining how many threads exist in each block in N-dimensions. Figure 2.3 shows the typical grid configuration used for image processing (2 dimensional grid of blocks, 2 dimensional blocks of threads). In the case for image processing, the grid size would be dependent on the image's dimension. For example, if an image of size 1024x1024 were to be processed, and the number of threads per a block was defined as 32x32 (1024 threads per block), then the CUDA grid would contain 32x32 thread blocks to process each pixel individually.

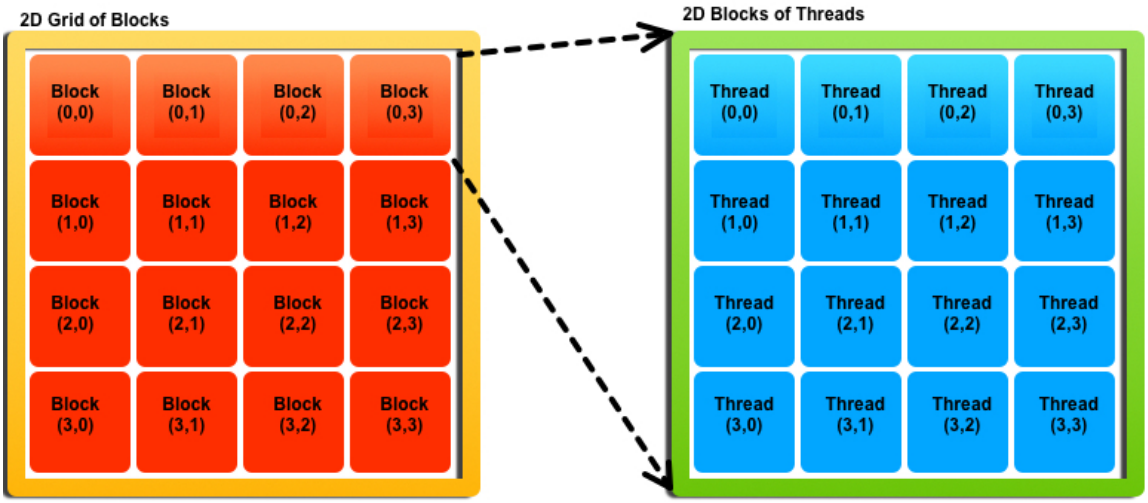


Figure 2.3: Typical CUDA Thread Configuration for Image Processing

2.6 General CUDA Performance Optimizations

Tuning CUDA algorithms for specific hardware configurations that they run on can highly improve the performance of the algorithm implementation. The performance of NVIDIA GPGPUs can be benchmarked by several specifications, shown in Table 2.2.

GPGPU Specification	How to Increase Performance
SM Count	Increase the number of streaming multiprocessors to increase the number of concurrent threads executing in parallel
Warp Size	Increase the warp size to increase the number of threads running in parallel within a single SM
Shared Memory Size	Increase the shared memory size per block to allow for higher SM thread occupancy
Warps Per SM	Increase the number of warps in a SM to increase the number of threads executing in parallel.

Table 2.2: GPGPU Hardware Performance Quantities

By knowing the GPGPU specifications for the hardware being programmed, an algorithm's implementation can be optimized to utilize all resources on the specific GPGPU.

Optimizing the GPGPU platform naively can sometimes produce worse performance over the CPU implementation. Correct optimization strategies must be known by the programmer in order to maximize the parallel performance. The purpose of running algorithms in parallel is to maximize algorithmic performance; therefore, hardware knowledge is imperative. Many factors should be considered when optimizing CUDA algorithms: data-bus overhead, memory caching, faster memory utilizations, and warp divergence.

2.6.1 Data-bus Overhead

The GPGPU memory is segregated from the host CPU memory space, therefore the GPGPU must communicate with the host CPU over the external PCI express bus. The overhead between transferring memory between the host CPU to the GPGPU and vice-versa can be significant if the data transfers are implemented naively. As the data

transfer overhead increases in a parallel implementation, the performance benefit of utilizing the GPGPU decreases.

Transferring memory between host CPU and GPGPU over the PCI express bus is typically the largest bottle neck in GPGPU algorithms. The CUDA driver API can only transfer memory from the host CPU to the GPGPU memory and vice-versa if the host memory is pinned (non-paged). By default, host memory allocations are pageable, thus the host CPU must perform a copy from pageable memory to pinned memory before copying the memory to the GPGPU global memory space. The transparent overhead of memory transfers can lead to poor performance when dealing with high bandwidth memory transfers, such as high resolution images or video processing.

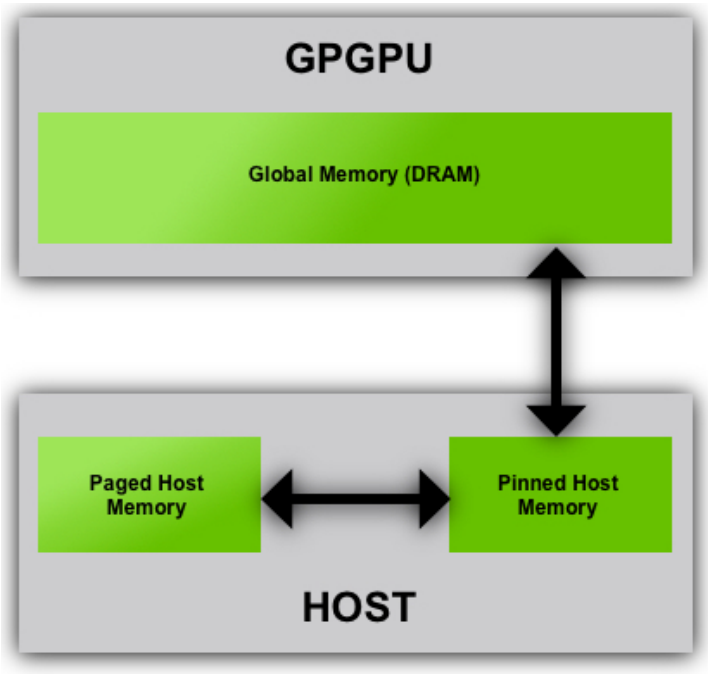


Figure 2.4: Memory Transfers Form Host to GPGPU and Vice-Versa

The CUDA API allows for allocating pinned memory to avoid the implicit host internal memory transfers from paged to pinned memory and vice-versa. Pinned memory transfers from the host CPU to the GPGPU and vice-versa have the highest bandwidth [8]. By avoiding paged host memory, the internal data transfer from paged to

pinned and vice-versa is avoided, thus increasing implementation performance. However, pinned memory should not be overused, for pinned memory allocations are computationally expensive, thus only a single allocation of pinned memory should be made and used as a staging area before memory transfers.

2.6.2 Cache Utilization

The L1 and L2 caches presented on the NVIDIA GPGPU hardware are transparent to the programmer, thus direct access is not possible. Knowledge of caching locality can greatly improve the performance of CUDA algorithms by avoiding global memory accesses. The L2 cache is the most abundant cache memory on the GPGPU, and it resides a single memory access away from global memory. The L2 cache greatly improves global memory access performance if memory accesses are based on spatial or temporal locality access pattern. Global memory accesses by threads within a single warp can be reduced if all the threads within the warp access spatially near portions of the input data [8].

Every streaming multiprocessor (SM) has its own dedicated on-chip L1 cache, which are exclusively designed for spatial locality. The L1 caches do not utilize an LRU (least recently used) caching scheme, and temporal access pattern will invoke cache misses, thus decreasing memory performance [6]. If temporal access patterns exists within the CUDA software, then memory should reside locally in shared memory on the SM in order to guarantee that data is kept on-chip.

2.6.3 Shared Memory Utilization

As mentioned earlier, a CUDA implementation’s memory performance can be quantified by its CGMA ratio. The CGMA ratio represents the compute calculations compared to the number of global memory accesses. When optimizing CUDA algorithm implementations, the global memory bandwidth typically becomes the bottleneck of the

performance. Increasing the CGMA ratio will effectively increase the CUDA implementation's performance. The strategy to increase the CGMA ratio involves utilizing other types of GPGPU memory, typically shared memory. Shared memory is streaming multiprocessor on-chip memory, normally with sizes 16 KB - 64 KB. Each block of threads has its own dedicated segment of shared memory since each block is scheduled to run exclusively on a particular SM. Shared memory is configured into 32 four-byte wide banks on each SM on the GPGPU, thus a 32 thread warp can access shared memory in parallel if no threads within the warp access the same bank [6]. Shared memory cannot be accessed between SMs, and therefore shared memory cannot be shared between thread blocks. The amount of shared memory is orders of magnitude smaller than global memory, thus the use of shared memory increases the complexity of CUDA implementation. For implementations which cannot utilize shared memory due to memory size constraints, or implementation complexity, the GPGPU offers two alternative types of memory to further increase performance: constant, and texture memory.

2.6.4 Constant Memory Utilization

In situations where the implementation of shared memory becomes overly complex, constant memory can be implemented in order to increase the CGMA ratio. Constant memory is readonly memory and is located in global memory, however it utilizes direct on-chip caching. Constant memory typically has a size of 64 KB for NVIDIA GPGPUs with compute capability 1.0-3.0. Constant memory has the performance of register accesses, due to caching, as long as the threads within a warp have the same memory access pattern. If all threads within a warp access access consecutive word addresses spatially, then only a single access transaction will be performed, which will increase memory performance and the implementation's CGMA ratio [9].

2.6.5 Texture Memory Utilization

In situations where the use of shared memory and/or constant memory cannot be utilized due to size constraints or implementation complexity, texture memory can be utilized to increase performance. Texture memory on the GPGPU is memory which is normally used for the graphics pipeline, however it is also available for general purpose computing. Texture memory is cached on-chip, like constant memory, and has great performance benefit when memory accesses exhibit spatial locality. Texture, like constant, memory is readonly and is highly optimized for spatial locality due to its design for graphics performance. Texture memory offers unique performance benefits that are not offered by other memory types: interpolation between values, automatic normalization, and automatic boundary handling. Texture readonly memory costs a single read from the texture cache on a cache hit, and a global memory read on a cache miss. For implementations that have a readonly memory access patterns, with high spatial locality, texture memory can be utilized to increase performance by avoiding the costs of global memory accesses.

2.6.6 Avoiding Warp Divergence

The GPGPU SIMD streaming multiprocessors (SM) have a performance drawback of warp divergence. As stated previously, a warp is the fundamental unit of parallelism on the GPGPU. Groups of threads are collected into blocks and partitioned into warps based on the architecture fixed warp size. Each block is exclusively assigned a SM, where the partitioned warps execute in lockstep level parallelism. Due to the SMs having an SIMD architecture, every thread executing in its particular warp must run the same instruction; however, in software there are normally conditional branches. If a thread in a warp executes a conditional path while another thread within that same warp does not execute the same path, then this is what is defined as warp divergence. Warp divergence causes all threads to stall for instruction level synchronization, thus “long

code paths in a conditional can cause a 2-times slowdown for each conditional within a warp and a 2^N slowdown for N nested loops" [6]. From the programmers point of view, if they know their specific GPGPU warp size, and partition their threads into blocks which guaranteed the same conditional paths, they can avoid warp divergence by ensuring that each thread within a warp executes the same instruction. This however, like shared memory utilization, increases the complexity of the parallel implementation. The code shown in Figure 2.5 is an example of warp divergence, for the thread execution is based on the parity of the thread ID. This implies that half of the threads within a warp will execute different conditional paths, thus introducing a 2 times slowdown in implementation performance. Warp divergence can decrease the level of parallelism in CUDA implementation due to the nature of SIMD. The CUDA compiler (nvcc) does perform conditional branch voting, which determines how to schedule threads based on conditional paths, however the programmer is best fit to solve the thread divergence problem, or at least minimize the warp divergence within their implementation.

```
1   if(thread_id % 2 == 0)
2       data[thread_id] = pow(2.0, 2.0); // Divergence Path #1
3   else
4       data[thread_id] = sqrt(2.0);     // Divergence Path #2
```

Figure 2.5: Warp Divergent Example Code

Chapter 3: Harris Corner Detection

3.1 Corner Detection Introduction

The Harris corner detection algorithm, developed by C. Harris and M. Stephens in 1988, detects the location of corner points within an image [10]. Corner points are used for defining features because they have “well-defined position[s] and can be robustly detected” [24]. Corner points are inherently unique and are great interest points due to their invariance to translation, rotation, illumination, and noise. Due to the intrinsic properties of corner points, the Harris corner detection algorithm has been utilized frequently for computer vision system applications, such as motion detection, image registration, video tracking, panorama stitching, 3D modeling, and object recognition.

3.2 Corner Detection Qualitative Description

A corner can be considered as the intersection of two well-defined edges. The Harris corner detection algorithm searches for corner points by looking at regions within an image which contains high gradient values in all directions. A window is iteratively scanned across the X and Y gradients of the input image, and if high changes in intensity exist in multiple directions, then a corner is inferred to exist within the current window. Figure 3.1 shows the different types of regions that can exist within an image.

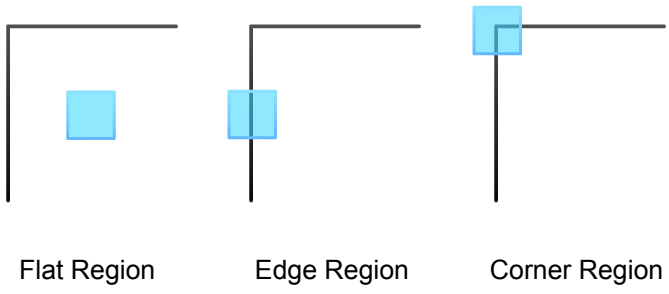


Figure 3.1: Directional Intensity Change Types

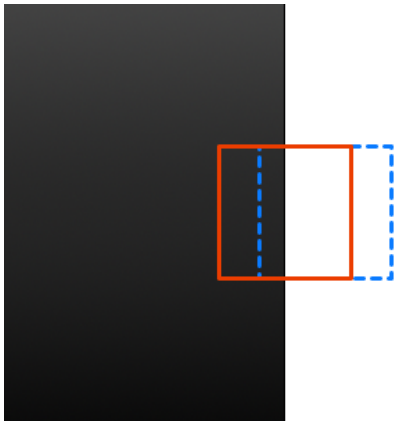
3.3 Corner Detection Mathematical Description

A corner within a region of interest (ROI) can be identified by calculating the sum of squared difference (SSD) between the ROI and shifted nearby regions. The SSD formula, shown in Equation 3.1, quantifies the difference between ROI and shifted region by summating the squared differences pixel by pixel. The function *I,* in Equation 3.1, represents the input image. The *(x,y)* coordinates specify the ROI, and the $(\Delta u, \Delta v)$ coordinate specifies the offset of the shifted region from the ROI.

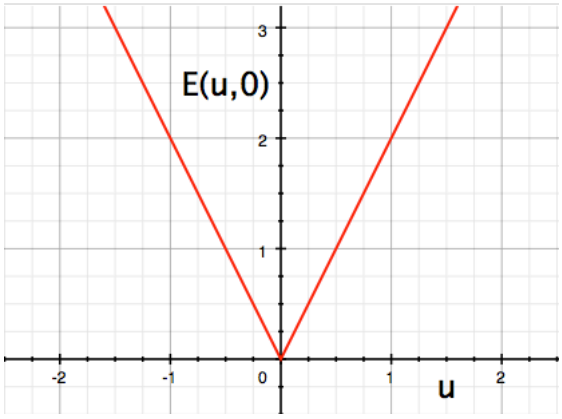
$$E(\Delta u, \Delta v) = \sum_{(x,y) \in ROI} \left\{ I(x + \Delta u, y + \Delta v) - I(x,y) \right\}^2$$

Equation 3.1: SSD Equation

Figures 3.2-3.4 (a) show the ROI (red box) containing an edge, defined by the *(x,y)* coordinates, and the shifted region (blue dashed box), defined by the $(\Delta u, \Delta v)$ offset. Consider iterating the shifted region away from the ROI in only the horizontal direction, shown in Figure 3.2 (a), thus only varying the Δu coordinate. When the Δu coordinate is at zero, the ROI and shifted region are the same region, thus resulting in a SSD of zero. As the shifted region iterates farther from the ROI in the horizontal direction the SSD increases significantly, shown in Figure 3.2 (b). This implies that the ROI and shifted region become more different as the shifted region iterates in the horizontal direction.



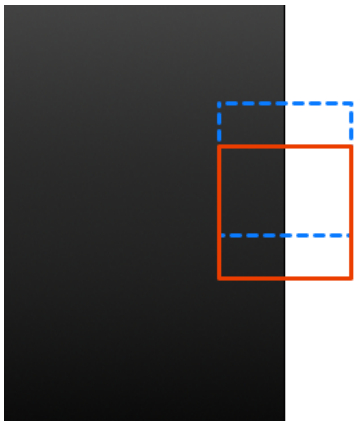
(a) Shifted Region Horizontally



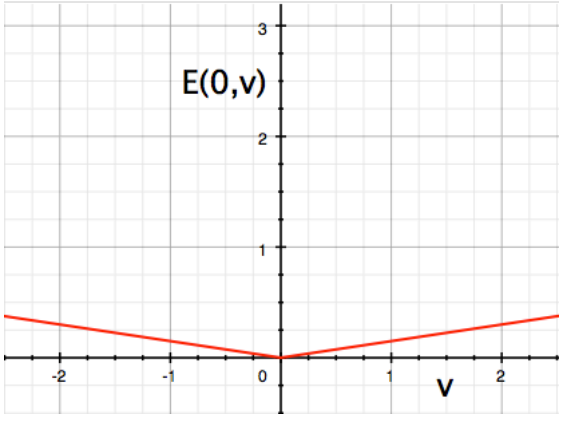
(b) SSD Increases Significantly

Figure 3.2: SSD When Shifting Region Horizontally Away From ROI

Now consider iterating the shifted region away from the ROI in only the vertical direction, shown in Figure 3.3 (a), thus only varying the Δv coordinate. As the shifted region iterates farther away from the ROI in the vertical direction, the SSD does not increase much, shown in Figure 3.3 (b). This implies that the ROI and shifted region stay similar as the shifted region iterates in the vertical direction.



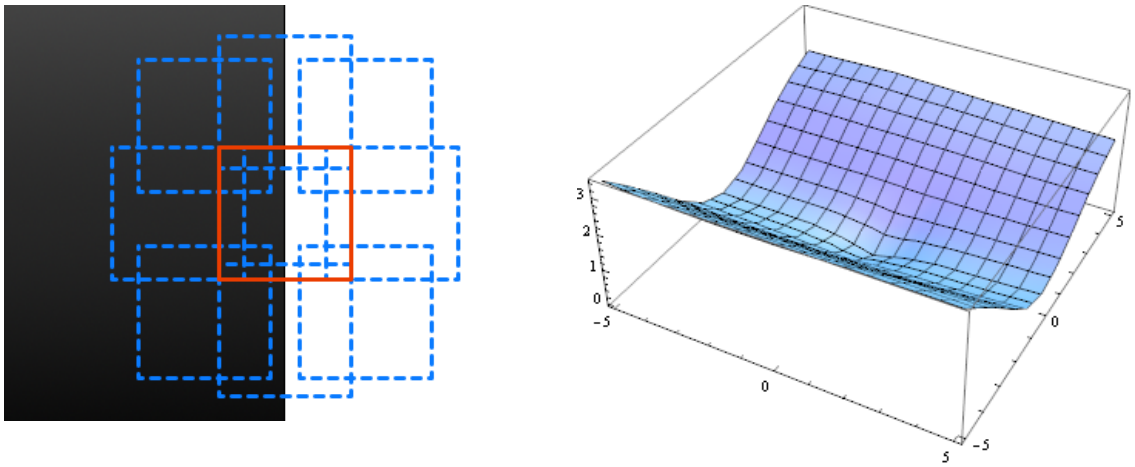
(a) Shifted Region Horizontally



(b) SSD Increases Minimally

Figure 3.3: SSD When Shifting Region Vertically Away From ROI

Now consider iterating the shifted region in all directions away from the ROI, shown in Figure 3.4 (a), thus varying both the Δu and Δv coordinates. Figure 3.4 (b) shows the SSD surface produced by iterating the shifted region in all directions. Since the SSD is only significant when iterating the shifted region way from the ROI in the horizontal direction, the SSD surface resembles a canyon shape, which implies the existence of an edge within the ROI.

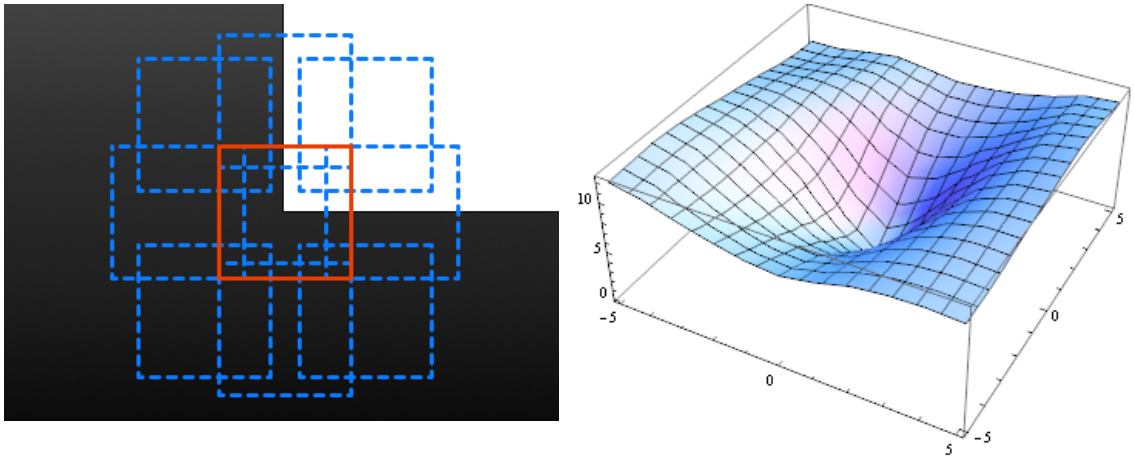


(a) Shifted Region in All Directions

(b) SSD Increases in Only One Dimension

Figure 3.4: SSD Surface for ROI Containing an Edge

If a ROI contains a corner, as shown in Figure 3.5 (a), the SSD will increase significantly regardless of the shifted window direction. The SSD for a corner existing within the ROI will have the surface shape shown in Figure 3.5 (b). The concave surface is zero-valued at the origin and increases in all directions away from the origin. A corner can be identified within a ROI based solely on the shape of the SSD surface.



(a) Shifted Region in All Directions (b) SSD Increases in All Both Dimensions

Figure 3.5: SSD Surface for ROI Containing a Corner

The shape of the SSD surface can be accurately approximated by its behavior at the origin. The Taylor series expansion can be utilized to approximate the surface behavior by expanding the SSD equation near the origin. The Taylor series states that a function's behavior at a specific point can be approximated by the infinite sum of that function's derivatives. Equation 3.2 shows the 1D Taylor series expansion about point *a*.

$$f(x) = f(a) + \frac{df}{dx}(x-a) + \frac{1}{2!}\frac{d^2y}{dx^2}(x-a)^2 \ldots$$

Equation 3.2: Taylor Series Expansion Equation

Under the assumption that the shifted window offsets Δu and Δv are minimal, the Taylor series can be utilized to accurately approximate the SSD surface. By utilizing Taylor series expansion, the pixel intensities within the shifted region can be approximated by the ROI gradients, shown in Equation 3.3. I_x is the partial derivative of

the ROI in the X (horizontal) direction, and I_y is the partial derivative of the ROI in the Y (vertical) direction.

$$I(x+\Delta u, y+\Delta v) \approx I(x,y) + I_x(x,y)\Delta u + I_y(x,y)\Delta v$$

Equation 3.3: Shifted Region Approximation Based on Taylor Series

The SSD equation can be reduced to only be dependent on the gradients of the ROI. Equation 3.4 shows the approximated SSD equation by substituting the Taylor series approximation, shown in Equation 3.3, into the SSD Equation 3.1.

$$E(\Delta u, \Delta v) \approx \sum_{(x,y)\in ROI} \left\{ I_x(x,y)\Delta u + I_y(x,y)\Delta v \right\}^2$$

Equation 3.4: SSD Approximation Equation

The SSD approximation only depends on the ROI gradients I_x and I_y, and not the ROI's pixel intensity values. The SSD approximation can be converted to matrix form by factoring non-summation dependent variables Δu and Δv (derivation shown in the Equation 3.5).

$$E(\Delta u, \Delta v) \approx \sum_{(x,y)\in ROI} \left\{ \Delta u^2 I_x{}^2(x,y) + 2\Delta u \Delta v I_x(x,y) I_y(x,y) + \Delta v^2 I_y{}^2(x,y) \right\}$$

$$\approx \sum_{(x,y)\in ROI} \left\{ \begin{bmatrix} \Delta u & \Delta v \end{bmatrix} \begin{bmatrix} I_x{}^2(x,y) & I_x(x,y)I_y(x,y) \\ I_x(x,y)I_y(x,y) & I_y{}^2(x,y) \end{bmatrix} \begin{bmatrix} \Delta u \\ \Delta v \end{bmatrix} \right\}$$

$$\approx \begin{bmatrix} \Delta u & \Delta v \end{bmatrix} \sum_{(x,y)\in ROI} \left\{ \begin{bmatrix} I_x{}^2(x,y) & I_x(x,y)I_y(x,y) \\ I_x(x,y)I_y(x,y) & I_y{}^2(x,y) \end{bmatrix} \right\} \begin{bmatrix} \Delta u \\ \Delta v \end{bmatrix}$$

$$\approx \begin{pmatrix} \Delta u & \Delta v \end{pmatrix} H \begin{pmatrix} \Delta u \\ \Delta v \end{pmatrix}$$

Equation 3.5: SSD Equation in Matrix Form

$$H = \sum_{(x,y)\in ROI} \begin{pmatrix} I_x^2(x,y) & I_x(x,y)I_y(x,y) \\ I_x(x,y)I_y(x,y) & I_y^2(x,y) \end{pmatrix}$$

Equation 3.6: Harris Matrix

The 2x2 matrix *H*—Harris matrix—is defined in Equation 3.6. The Harris matrix describes the gradient distribution within the ROI, therefore the Harris matrix can be used to classify corner features. The gradient distribution is variant to corner rotation within the ROI, therefore the eigenvalues of the Harris matrix are used to create a rotationally invariant description of the gradient distribution. The eigenvalues of the Harris matrix define the shape of the eclipse which encapsulates the horizontal and vertical gradient distribution of the ROI, shown in Figure 3.6. The eigenvalues of the Harris matrix are invariant to rotation, intensity scaling, and affine transformations, thus the eigenvalues of the Harris matrix are used as the characteristic for detecting corners. If a corner exists within a ROI, then both eigenvalues of the Harris matrix will have

significant magnitude, which implies a large gradient distribution in the horizontal and vertical directions.

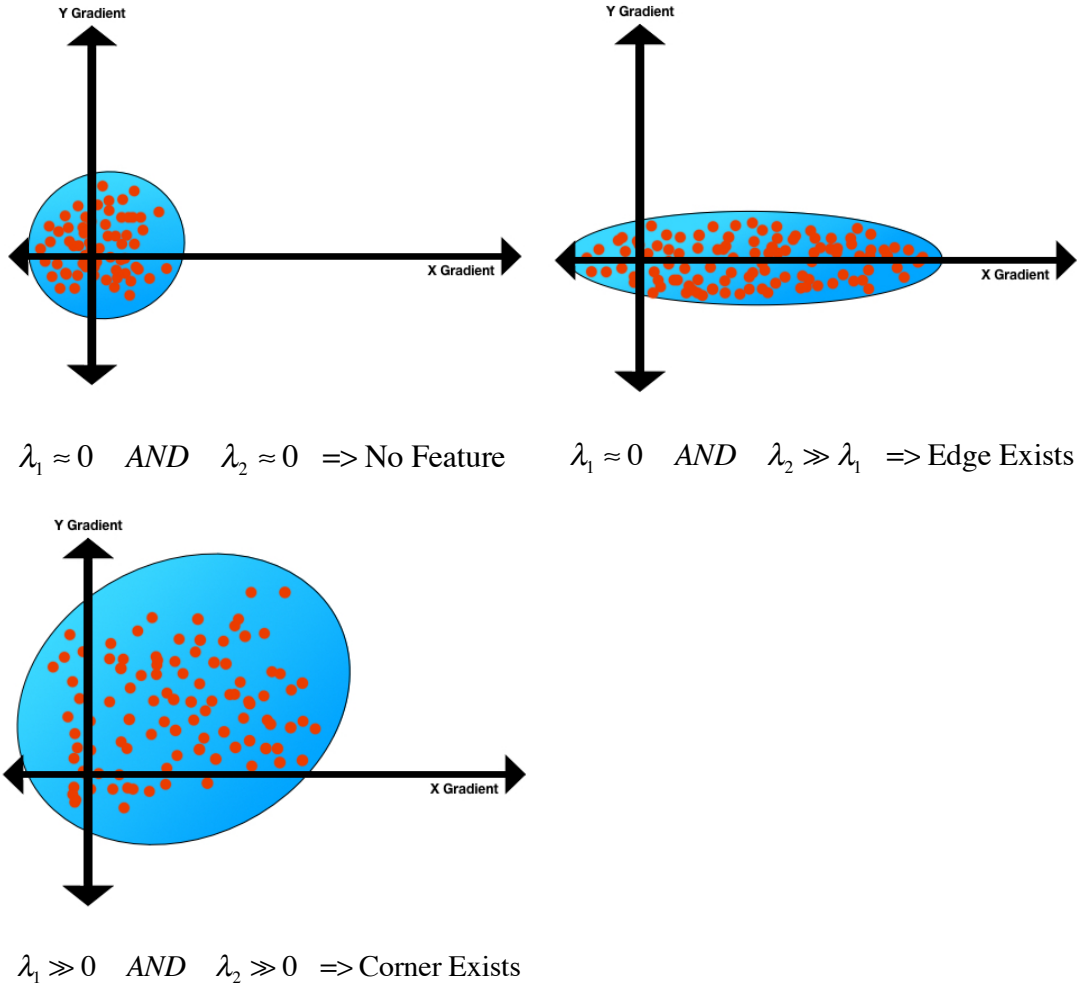


Figure 3.6: Eclipse Representation for Harris Corner Detection

Based on the eigenvalues of the Harris matrix, a corner score is assigned to identify the likelihood of a corner existing within the ROI. A corner is considered detected when the corner score is significantly greater than zero, which implies a large encompassing eclipse over the gradient distribution. The corner score equation is described in Equation 3.7.

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$$

Equation 3.7: Corner Score Equation

The *k* term is considered the sensitivity parameter of the corner detector, which is manually adjusted, however it has been empirically shown that it typically ranges from 0.04-0.06. Figure 3.7 shows the corner response of an example geometric input image. The corner points of the input image in Figure 3.7 (a) produce a significant corner score greater than zero, while the edge and flat regions produce a smaller corner score.

(a) Input Image

(b) Corner Response
Threshold > 0

Figure 3.7: Corner Response Example

The Harris corner detection algorithm can be used to produce a corner response, which is computed by determining the eigenvalues of the Harris matrix at each ROI within the image. The Harris matrix is computed for every pixel within in the image, thus it is very computationally intensive. Once a corner response is created for an image, the

corner locations can be extracted as feature locations for higher level computer vision algorithms.

3.4 Corner Detection Algorithm

1) *Denoise Input Image Using Gaussian Smoothing Filter*
2) *Compute Image Gradients* I_x *and* I_y
3) *Compute Image Gradient Products* I_x^2, I_y^2, *and* I_xI_y
4) *For Every Pixel Location*
   a) *Define ROI Around Pixel*
   a) *Compute Harris Matrix from* I_x^2, I_y^2, *and* I_xI_y *for ROI*
   b) *Compute Eigenvalues of Harris Matrix*
   c) *Assign Corner Score to Pixel*
5) *Threshold Corner Response*
6) *Perform Non-maxima Suppression on Corner Response*

3.5 Corner Detection Software Architecture

The Harris corner detection algorithm, described in Section 3.4, can be partitioned into incremental stages: image convolution, array multiplication, corner detection, and non-maxima suppression. Figure 3.8 shows the Harris corner detection software architecture and algorithm flow from input image to suppressed corner response. The input image is first convolved with a Gaussian smoothing filter in order to remove any unwanted noise from the image. The smoothed image is then convolved with gradient directional filters in order to calculated the I_x and I_y gradients. The I_x and I_y gradients are then element-by-element multiplied to calculate I_x^2, I_y^2, and I_xI_y products which are used as input for the corner detector stage. The corner detector, at every spatial location, calculates the Harris matrix and it's eigenvalues for the ROI. Based on

the eigenvalues calculated for a ROI, a corner score is determined and assigned to that spatial location. The corner response is then suppressed using non-maxima suppression to define a minimum distance between adjacent corner detections in the corner response. This section will describe each stage and its description, which will follow into optimizing the implementation using CUDA in Chapter 4.

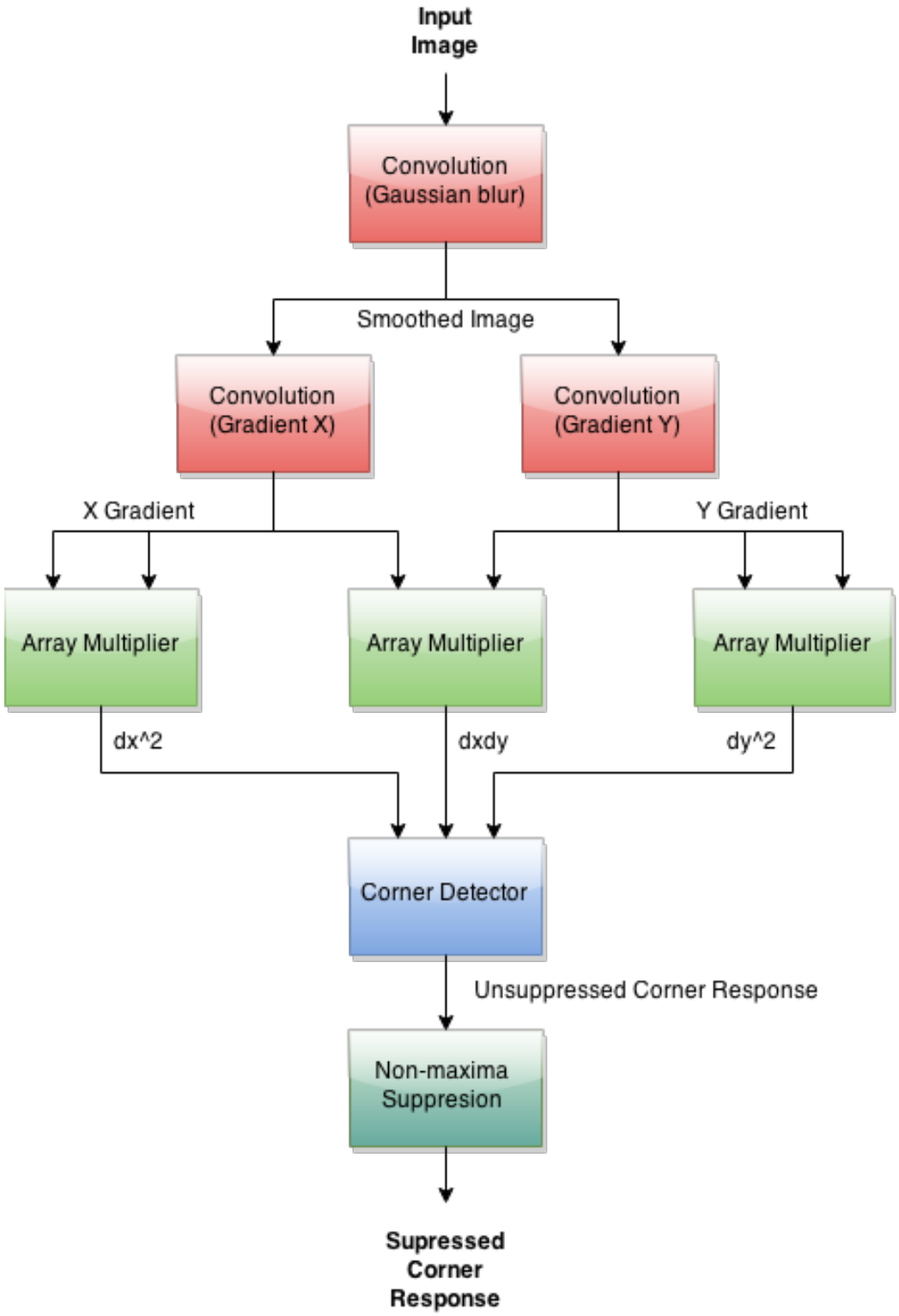


Figure 3.8: Harris Corner Detection Software Architecture

3.5.1 Image Convolution

Image convolution, also known as spatial filtering, is defined as two dimensional discrete convolution. Image convolution is the process of sweeping a filter mask, also known as convolution kernel, across every pixel in an image while performing a scalar product reduction between neighboring pixel and the filter mask coefficients [11]. Computing the value of each filtered pixel involves the centering the filter mask at the desired pixel, shown in red in Figure 3.9. Once the filter mask is centered over the pixel, the neighbors for that pixel are multiplied with the corresponding filter mask coefficients, then all values are reduced into a sum that represents the filtered value. Figure 3.9 visually describes the image convolution process for a single pixel. Based on the filter mask coefficients, several different filtering operations can be performed on an input image. For the application of Harris corner detection, only Gaussian blurring and Sobel gradient filtering will be discussed. The Harris corner detection algorithm requires three separate image convolutions for an input image: Gaussian blurring, directional gradient X, and directional gradient Y.

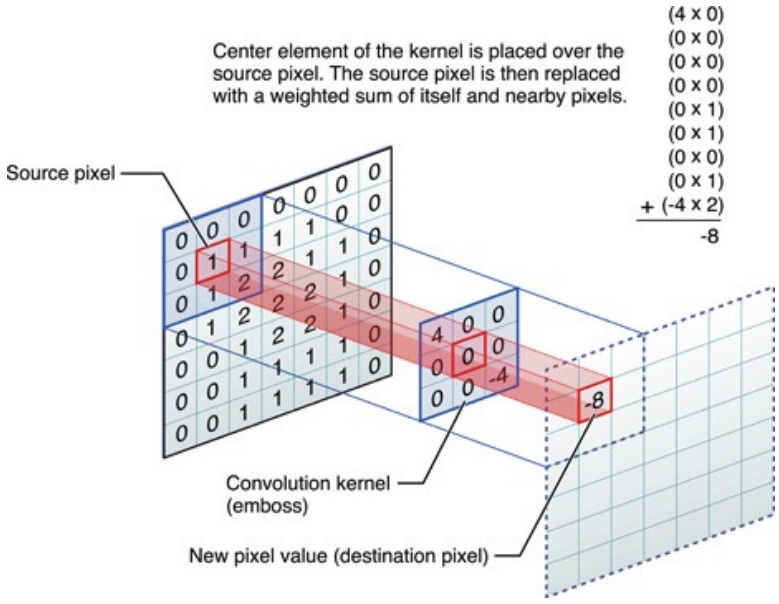


Figure 3.9: Image Convolution Visual Description [12]

3.5.1.1 Gaussian Blurring

The first stage in the Harris corner detection software architecture is to perform a Gaussian smoothing (LPF) operation on the input image in order to reduce noise: Gaussian noise is commonly found in digital images due to electrical sensor interference [13]. The Gaussian filter mask, shown in Figure 3.10, when convolved with the input image, filters out high frequency intensity changes in the image, thus providing a smoother output image. Gaussian smoothing is a prerequisite for determining image gradients in order to eliminate the false intensity changes when computing the I_x and I_y directional gradients.

$$f(x,y) = e^{-\frac{(x-x_o)^2}{2\sigma_x^2} - \frac{(y-y_o)^2}{2\sigma_y^2}}$$

Equation 3.10: 2D Gaussian Distribution Equation

$$\begin{pmatrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{pmatrix}$$

Figure 3.10: 3x3 Gaussian Convolution Filter masks(sigma = .85)

The mathematical model of the Harris corner detection algorithm does not theoretically require Gaussian filtering, however the presence of noise in digital images and video may cause false corner detection; thus, smoothing the image before running the detection algorithm is necessary in implementation.

3.5.1.2 Image Gradients

The image gradients in X and Y directions can be determined by performing image convolutions with the Sobel directional filter masks, shown in Figure 3.11. The Sobel filter mask, when convolved with the input image, effectively takes the directional derivative of the input image and produces an edge map. The I_x and I_y gradients produced by convolving the input image with the Sobel filter masks describe the pixel intensity changes within a ROI.

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

(a) Sobel X Gradient Filter (b) Sobel Y Gradient Filter

Figure 3.11: 3x3 Sobel Gradient Filter masks

3.5.2 Array Multiplier

Array multiplication, unlike matrix multiplication, is element-wise multiplication between two matrices (or images). The array multiplier stage of the Harris corner detection software architecture computes I_x^2, I_y^2, and I_xI_y products which are used as input to he corner detector stage to compute the Harris matrix at every ROI.

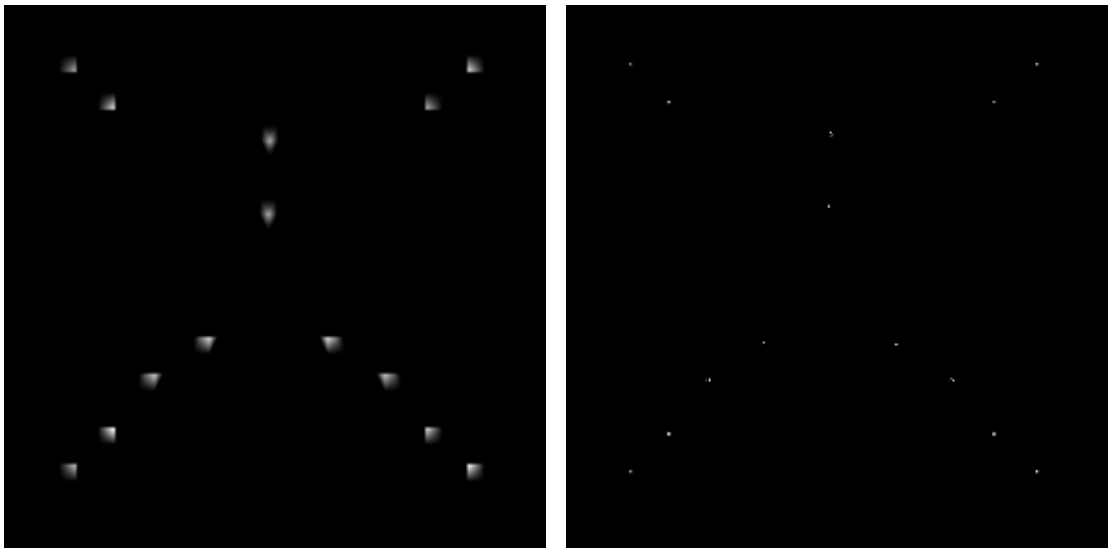
3.5.3 Corner Detector

The corner detector stage of the Harris corner detection algorithm calculates the unsuppressed corner response based on the image's gradients and sensitivity parameter. The corner detector iterates over every pixel in the image, defines a ROI around that pixel, and calculates the Harris matrix for that ROI. The elements of the Harris matrix are computed by summing the directional gradient products I_x^2, I_y^2, and I_xI_y over the ROI. The eigenvalues for the Harris matrix are then calculated using the

quadratic formula and are used to assign a corner score for every pixel in the corner response.

3.5.4 Non-maxima Suppression (NMS)

The non-maxima suppression (NMS) stage of the Harris corner detection software architecture can be considered as a non-linear filter which filters out pixel values which are non-maximums in local neighborhoods. A well-defined corner response for each pixel neighborhood is desired when identifying unique corner feature. NMS involves iterating over every pixel in the corner response. For every pixel, the neighborhood surrounding the pixel is extracted. If a pixel is not the maximum in its neighborhood, then the pixel value is set to zero, otherwise it is left unchanged. The NMS effectively defines the minimum distance between features allowed in the image based on the neighborhood size. Figure 3.12 shows an example of performing a 3x3 neighborhood NMS on the corner response image computed from Figure 3.7 (a).



(a) Corner Response (b) Suppressed Corner Response

Figure 3.12: Unsuppressed and Suppressed Corner Responses

Chapter 4: Harris Corner Detection GPGPU Implementation

4.1 Corner Detection Parallel Software Architecture

This section will discuss the CUDA implementations for each of the software architecture stages, shown in Figure 3.8, and the necessary CUDA optimizations in order to achieve high performance corner detection. Each stage in the Harris corner detection software architecture can be implemented to run in parallel using CUDA. The software architecture can be further decomposed into a parallel architecture by identifying independent stages. Figure 4.1 shows the parallel architecture, for each component can be implemented to run in parallel. Each segmented region in Figure 4.1 identifies a CUDA kernel invocation, where a CUDA kernel is the GPGPU function to execute. The dependence of a sequential implementation flow still remain, for the execution must start on the left of Figure 4.1 and incrementally finish at the right of the parallel software architecture. The following sections will describe the CUDA optimizations made for each stage in the Harris corner detection implementation in order to achieve high performance.

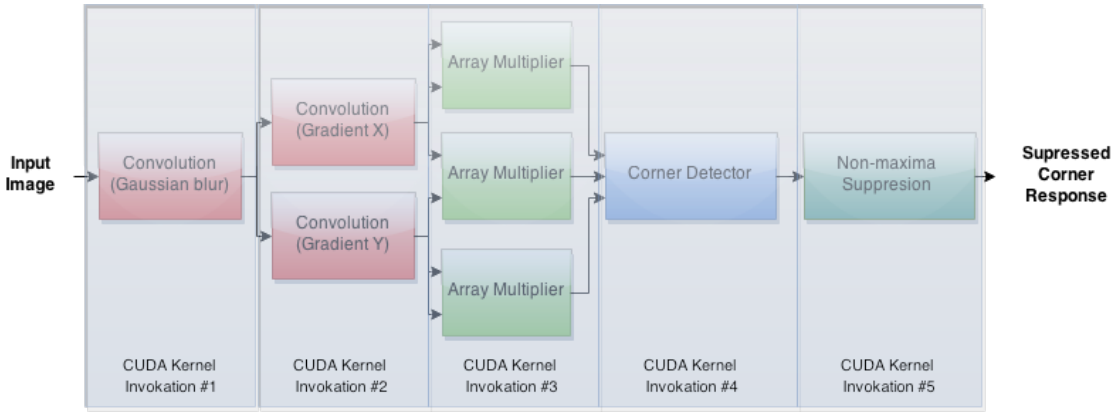


Figure 4.1: Harris Corner Detection Parallel Software Architecture

4.2 GPGPU Convolution

The Harris corner detection implementation requires three separate convolutions for each input image: Gaussian smoothing, directional gradient X, and directional gradient Y. Image convolution, or spatial filtering, is a natural fit for the CUDA software platform. The CUDA image convolution implementation can be intuitively understood by defining the CUDA thread configuration grid the same size as the input image, thus each thread corresponds to a pixel's spatial location. Each thread will then perform the neighboring reduction within the pixel's neighborhood and the convolution filter mask. The first stage of Harris corner detection implementation is the convolution with a Gaussian smoothing filter to remove noise in order to avoid false corner detection. If the input image has size MxM and the filter mask has size NxN, then the number of multiplications required to convolve the input image with the filter mask is $M^2 * N^2$ (where M >> N).

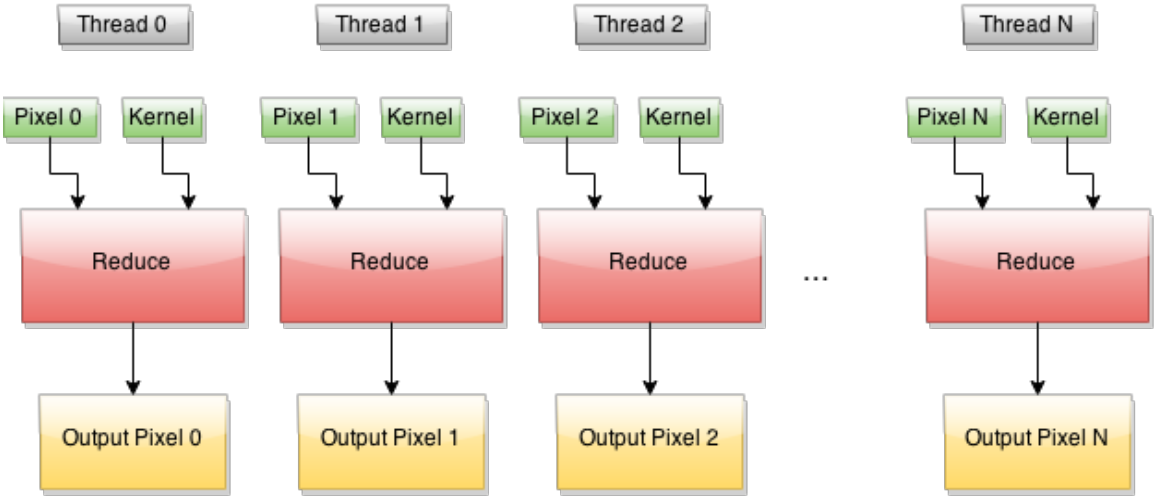


Figure 4.2: CUDA GPGPU Convolution Algorithm Flow

4.2.1 Naive Convolution GPGPU Implementation

The naive CUDA implementation of image convolution consists of loading the input image and filter mask into global memory (slowest memory) on the GPGPU, then spawning a CUDA thread for every pixel in the image. Each thread will then perform a reduction with its pixel's neighborhood and the filter mask. The CUDA image convolution implementation can be visually described by Figure 4.2.

The GPGPU has a separate memory management system than the host CPU; therefore, in order to perform work on the GPGPU, the CPU must first copy the image and filter mask to the GPGPU memory. The naive CUDA image convolution implementation does not consider data-bus overhead, discussed in Section 2.5.1, and has the memory transfer flow shown in Figure 4.3. As discussed in Chapter 2, the host CPU can only transfer memory to the GPGPU if the memory is pinned (non-paged). Copying the image and filter mask to the GPGPU from the host CPU involves first copying the memory to host pinned memory and then copying the memory to the GPGPU global memory over the PCI express bus, resulting in poor memory transfer performance.

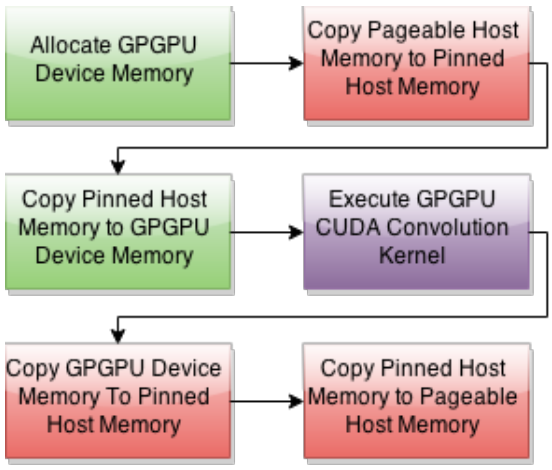


Figure 4.3: Naive CUDA Convolution Memory Transfers

```
FOR every thread
      FOR i=0 to kernel dimension
            FOR j=0 to kernel dimension
            value = value + kernel[i][j] * neighborhood[i][j]
            END FOR
      END FOR

      pixels[thread row][thread col] = value
END FOR
```

Figure 4.4: Naive Convolution CUDA Kernel Pseudo Code

That naive implementation uses the slowest memory type, global memory, on the GPGPU device for the input image and filter mask. The pseudo code in Figure 4.4 shows the CUDA kernel which is executed for every spawned thread. If the filter mask has size NxN, then each thread has 2 * N^2 global memory access (highlighted in green) for every 2 * N^2 computations, implying a CGMA ratio of 1. The naive implementation performance can by increased by optimizing the memory transfers from the host CPU to the GPGPU device, as well as increasing the CGMA ratio by utilizing faster memories on the GPGPU device for the input image and filter mask.

4.2.2 Optimized Convolution GPGPU Implementation

4.2.2.1 Separable Convolution Filter Masks

The total number of multiplications can be decreased by utilizing filter mask separability. A filter mask is considered separable if the filter mask can be represented by the convolution between a vertical and row vector, as shown in Equation 4.1. If a filter mask is separable, the image convolution can be computed by two separate convolutions.

$$I_{MxM} * F_{NxN} = (I_{MxM} * C_{Nx1}) * R_{1xN}$$

Equation 4.1: Convolution Filter Mask Separability Association

The convolution between a vertical and row vector is equivalent to the outer product of the vectors, thus a 2D filter mask can be easily separated into two 1D filter masks. Equations 4.2-4.4 shows the separability for the Gaussian and Sobel direction gradient filter masks.

$$\frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} = \frac{1}{16} \times \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \times \begin{pmatrix} 1 & 2 & 1 \end{pmatrix}$$

Equation 4.2: Gaussian Separated Filter Mask

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \times \begin{pmatrix} -1 & 0 & 1 \end{pmatrix}$$

Equation 4.3: Sobel X Gradient Separated Filter Mask

$$\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix} \times \begin{pmatrix} 1 & 2 & 1 \end{pmatrix}$$

Equation 4.4: Sobel Y Gradient Separated Filter Mask

The total number of multiplication required to perform the convolution with the input image and the two 1D filter masks is $M^2 * 2N$ (where M >> N), thus only requiring a fraction of 2 / N multiplications. Victor Podlozhnyuk, from NVIDIA, states that separable filter masks have the added benefits of “reducing the arithmetic complexity and bandwidth usage of the computation for each data point” [9]. The number of multiplications avoided by utilizing separable filter masks can be quantified as the expression $M^2 * N(N-2)$, where M represents the image’s square dimension size, and N represents the filter mask’s square dimension size. Figure 4.5 shows the number of multiplications avoided by utilizing separable filter masks over traditional convolution filters.

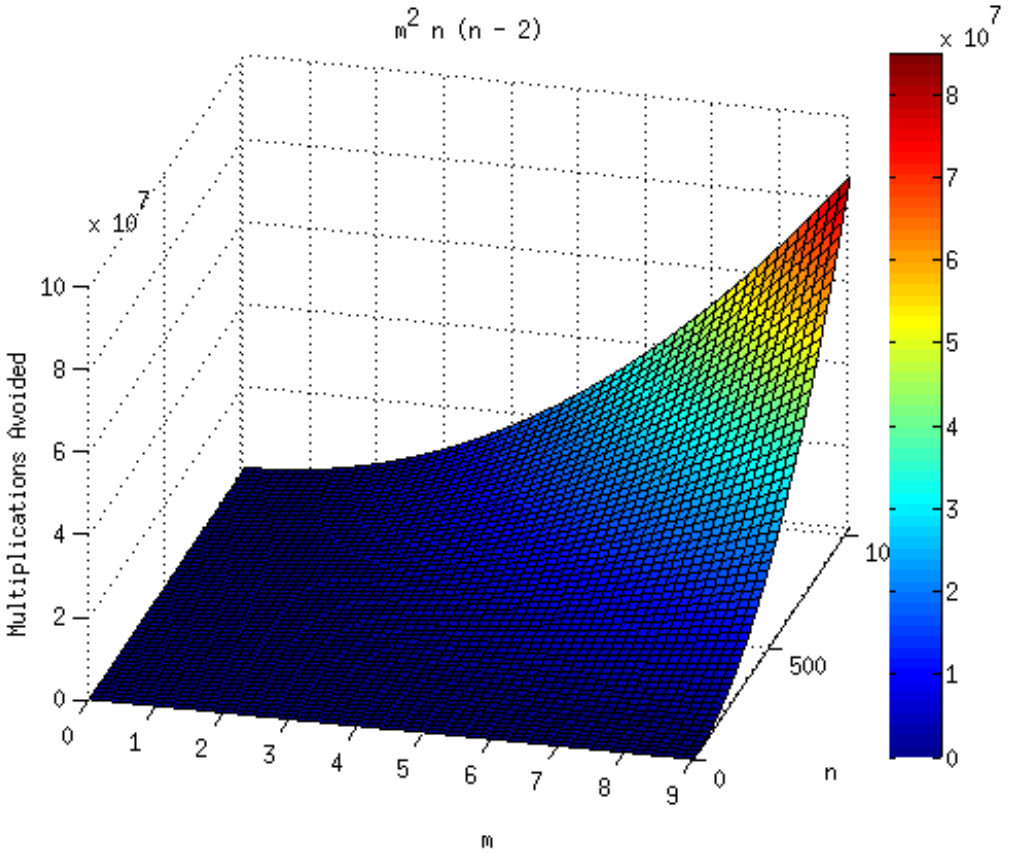


Figure 4.5: Avoided Multiplications by Utilizing Separable Convolution Filters

Implementing separable convolution increases the number convolution CUDA kernel invocations, however CUDA kernel invocations are inexpensive. By utilizing separable filter masks, the total number of multiplications is greatly reduced and performance is increased.

4.2.2.2 Async Memory Transfers

By default, all memory copies to the GPGPU from host CPU and vice-versa are synchronized to the host CPU execution. Thus, a copy operation from the host CPU to the GPGPU will block the host CPU program execution until the copy is finished. The execution timeline for separable CUDA convolution is shown in Figure 4.6.

Figure 4.6: Separable CUDA Convolution Execution Flow

The entire image is first transferred from the host CPU pinned memory to the GPGPU over the PCI express bus (Host to Device), the convolution with the 1D row filter mask is executed, then the convolution with that 1D column filter mask is executed, in sequential order. Host to GPGPU memory transfers and CUDA kernel execution occur sequentially, however it is possible to overlap the CUDA kernel execution with the memory transfers from the host to the GPGPU [9]. By exploiting separable filter masks, and the fact that memory is stored sequentially for row major images, it is possible to pipeline the memory transfers from the host to the GPGPU with 1D row convolution operations.

Since the launch of a CUDA kernel is inexpensive, there will be a CUDA kernel specialized to perform a 1D row convolution on a single row of pixels from an image with a 1D row filter mask. Once a single row from the input image is loaded onto the GPGPU

memory, the 1D convolution CUDA kernel can be executed. With an input image of size MxM, there will be total of M separate 1D convolution kernel invocations in order to convolve the 1D row filter mask with the entire input image.

The asynchronous pipeline parallelism is possible with CUDA streams. A CUDA stream “is simply a sequence of operations that are performed in order on the device” [9]. The default CUDA stream, used by CUDA’s memory transferring API, blocks host CPU execution until finished. CUDA allows for host code to create multiple streams which will run asynchronously from the host execution, thus memory transfers and CUDA kernels can by performed in parallel. By incorporating the concept of pipelining, CUDA convolution kernels can be executing on the GPGPU while the next pixel row is being transferring over to the GPGPU. Figure 4.7 shows the execution timeline for the asynchronous memory transfers running in parallel with row convolution operations. While row N of the image data is being transfer to the GPGPU, convolution of image row N-1 is being computed simultaneously.

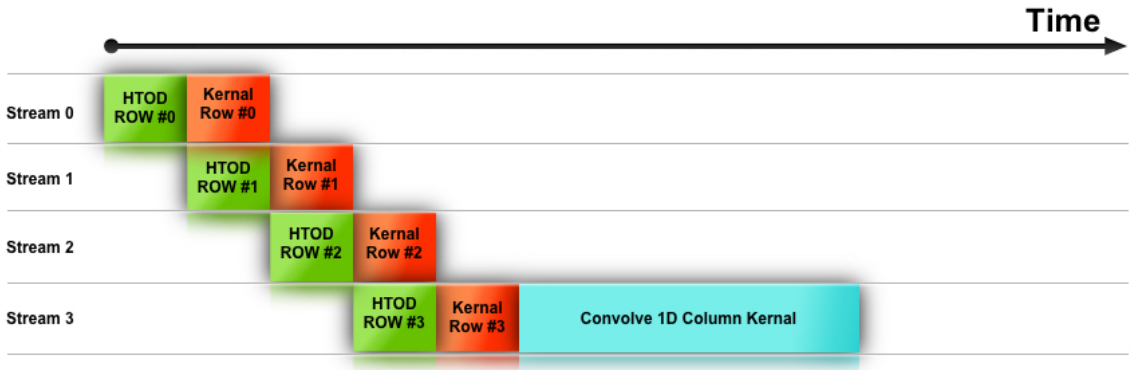


Figure 4.7: Pipelined CUDA Separable Convolution Execution Flow

Let T_{trans} represent the time to transfer the entire image to the GPGPU, T_{RC} represent the time to perform row convolution, and T_{CC} represent the total time to execute the column convolution. By utilizing asynchronous memory transfers, the entire

convolution can be approximated by $T_{CC} + (T_{trans} + T_{RC}) / 2$ units of time, thus saving approximately $(T_{trans} + T_{RC}) / 2$ time over the synchronous implementation.

4.2.2.3 Constant Memory Utilization

Since each thread within the GPGPU convolution kernel reads the same filter mask with the same access pattern, memory reads to the filter mask can be optimized by loading the filter mask into constant memory on the GPGPU. The memory reads to the filter mask are coalesced for every block, and therefore for every warp; thus memory access performance of the convolution kernel will be improved and the number of global memory accesses will be decreased due to the constant memory on-chip cache.

4.2.2.4 Shared Memory Utilization

The final bottleneck in the optimized CUDA convolution implementation is the global memory accesses to pixel values. Shared memory has a much higher bandwidth and much lower delay access time than global memory [5]. If a 1D filter mask has N elements, and the image has size MxM, then there are M*N global memory reads due to accessing the pixel data. By utilizing on-chip shared memory, redundant accesses to global memory pixel data can be avoided and the total number of global memory access for pixel data can be reduced to M.

All threads that exist within the same block have a dedicated segment of shared memory which is available for use on each streaming multiprocessor. The shared memory for a row of pixels from the image can be populated in parallel by having each thread load its appropriate pixel data into the shared memory segment before performing the 1D row convolution operation. Complications appears when concerning border pixels within the thread blocks, for threads that exist on the edge of a thread block needs to have access to shared memory segments of neighboring blocks. The solution is to add a small amount of redundancy to the shared memory blocks. By creating a shared

memory segment that is greater than the dimension size of the thread block, threads located at the border of a block can efficiently access the values that extend outside of their regular scope. Figure 4.8 shows how shared memory is overlapped between thread blocks to allow for shared memory accesses when convolving pixels that are spatially located at the edges of blocks. The first row in Figure 4.8 shows the threads configured into 4 separate thread blocks, each containing 6 threads, resulting in 24 threads total. Each thread block allocates a shared memory segment with 2 extra elements of added redundancy. Shared memory blocks now contain pixel information from the their neighboring thread blocks. This allows, for example, thread 7 in block 2, to access the data from thread 6 in block 1 from directly from its own portion of shared memory.

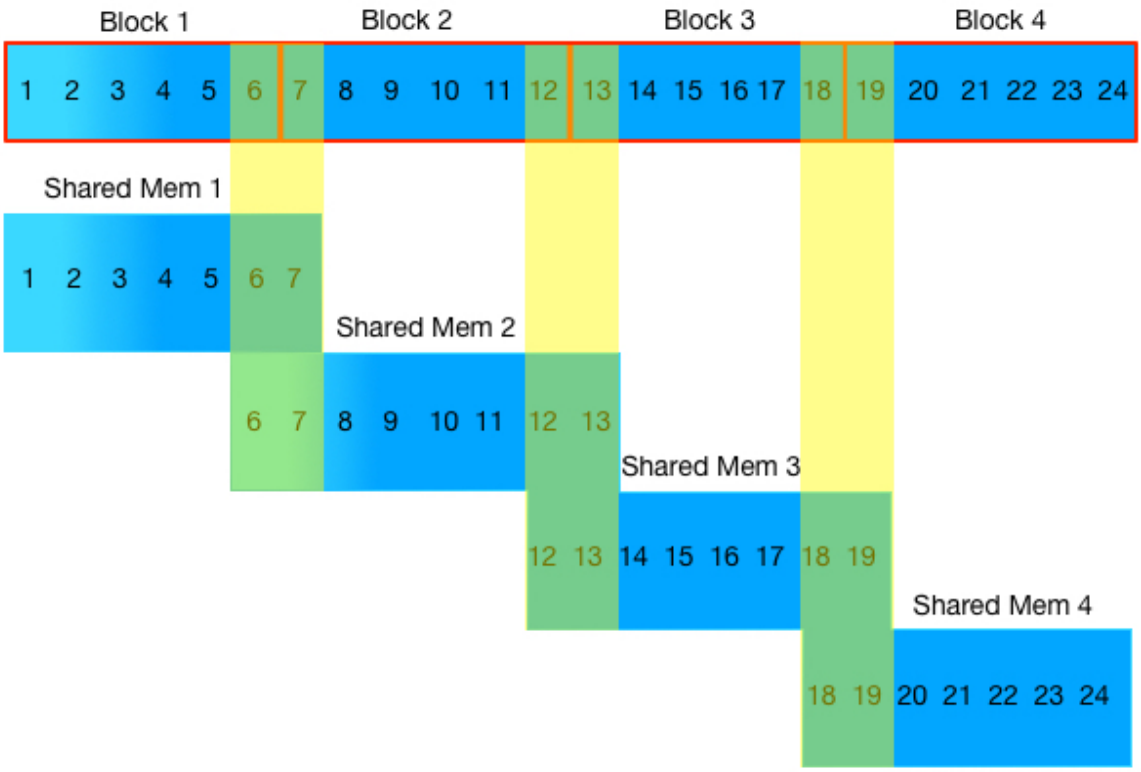


Figure 4.8: Shared Memory Overlap Between Thread Blocks

Once the shared memory is loaded for each block, including the redundant overlaps, the 1D row convolution can be performed directly from shared memory. By

operating directly from shared memory, the total global memory accesses reduces from M*N to M, where M is the image dimension size and N is the length of the 1D filter mask.

4.2.3 Convolution Performance Results

The optimized CUDA implementation of image convolution performance results were compared to the standard C, MATLAB, and naive CUDA implementations. The performance results were acquired by measuring the elapsed time to perform a single image convolution with a fixed 3x3 Gaussian filter mask over several different square image dimension sizes. The optimized CUDA implementation incorporates all optimization techniques described in Section 4.2.2: separable convolution masks, asynchronous memory transfers, constant and shared memory utilization.

Figure 4.9 shows how the image convolution process times vary with image size for the different implementations. The performance results show that the standard C and MATLAB implementations behave parabolically with image dimension size, thus realtime performance is not feasible. The optimized CUDA implementation outperformed all of the other implementations and platforms by orders of magnitude. Table 4.1 shows the speedup factors, ratio of processing times, of the optimized CUDA image convolution implementation over the other implementations.

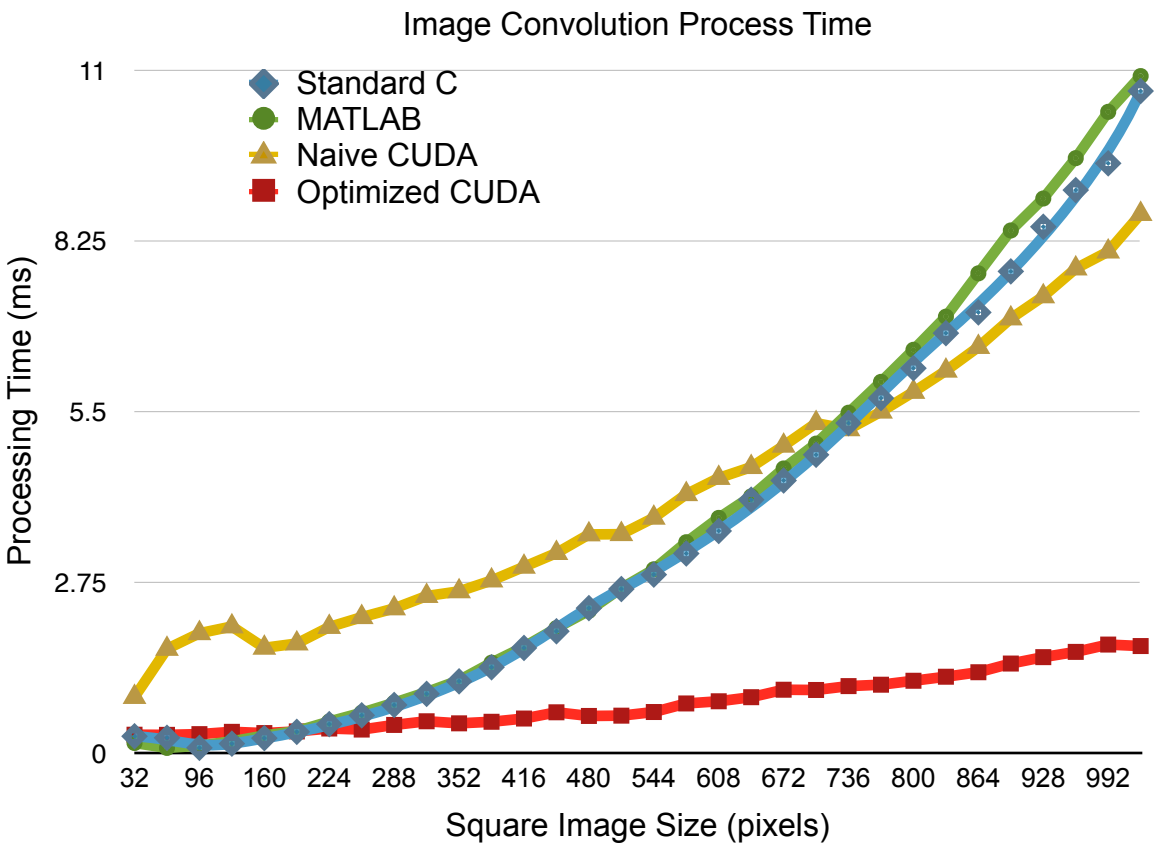


Figure 4.9: Image Convolution Performance Results

Image Size	Speedup Over C Sequential (s/s)	Speedup Over MATLAB (s/s)	Speedup Over Naive CUDA (s/s)
32x32	0.441	0.551	3.047
64x64	0.342	0.284	5.704
128x128	0.612	0.555	5.95
256x256	1.821	1.714	5.77
512x512	4.123	4.404	5.845
1024x1024	5.921	6.323	5.035

Table 4.1: Optimized CUDA Convolution Speedup

4.3 GPGPU Corner Detector

The corner detection algorithm is a non-linear filtering operation which produces a corner response based on the image gradients in the X and Y directions. A pixel's output intensity in the corner response calculation is independent of any other pixel, thus a parallel CUDA implementation is the perfect platform for increasing performance. The corner response calculation involves calculating the Harris matrix, define in Equation 3.6, for each ROI (region of interest) defined around each pixel. Each corner response pixel can be calculated in parallel by creating a CUDA thread configuration grid with the same size as the image, thus each thread will calculate its appropriate corner score output pixel independently. Figure 4.10 shows the parallel implementation for the corner response algorithm.

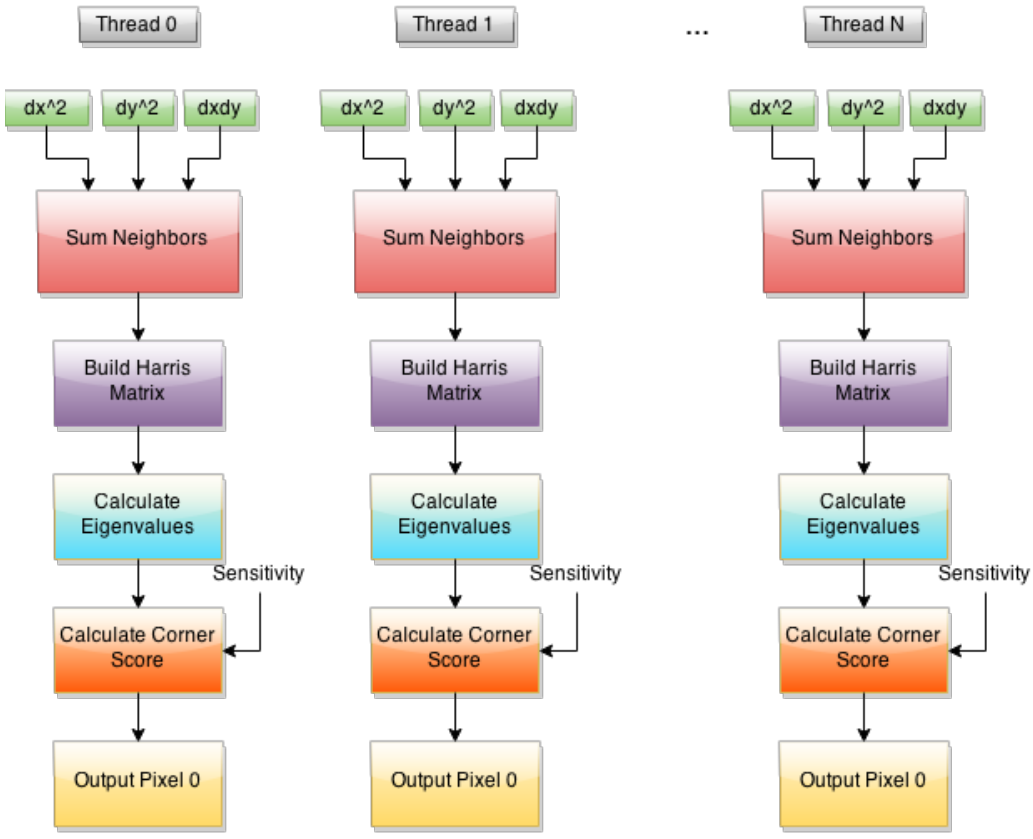


Figure 4.10: CUDA GPGPU Corner Detector Algorithm Flow

4.3.1 Naive Corner Detector GPGPU Implementation

The naive CUDA GPGPU implementation of the corner detector algorithm involves loading the three product gradient images I_x^2, I_y^2, and I_xI_y into GPGPU global memory (slowest memory). Each thread will then execute the naive CUDA kernel, which will perform a summation of the product gradient images for the input pixel's ROI.

```
FOR every thread
      a = sum neighborhood(dx2, thread row, thread col)
        b = sum neighborhood(dy2, thread row, thread col)
        c = sum neighborhood(dxdy, thread row, thread col)
        M = build matrix [a, c,
                          c, b]
        l = compute eigenvalues(M)
        r = l(0) * l(1) - sensitivity * (l1 + l2)^2

        corner response[thread row][thread col] = r

END FOR
```

Figure 4.11: Naive Corner Detector CUDA Kernel Pseudo Code

The corner detector's most computational intensive task is the summations of the image gradient products over each ROI. Figure 4.11 shows the pseudo code for the CUDA kernel to run for every pixel in the corner detector implementation in parallel. Let W define the dimension size of the ROI around the pixel. The number of global memory accesses for a ROI summation is W^2, thus the total number of global memory accesses for a thread will be $3W^2$, implying a CGMA ratio of 1. The total number of memory accesses could be decreased by utilizing shared memory, using the same technique as convolution; however, an optimization using integral images will show a constant number of global memory accesses regardless of ROI size.

4.3.2 Optimized Corner Detector GPGPU Implementation

4.3.2.1 Integral Images

An integral image can be considered the two dimensional exclusive scan of the input image. The integral image at coordinate *(x,y)* is represented by the summation of all pixel values from the point *(x,y)* to the origin *(0,0)*. Integral images optimize the ROI summation calculation, for the “summation of pixel values within the window can be calculated in 3 additions and 4 memory accesses” [2]. The summation of a neighborhood in an image can be defined as an arithmetic calculation of the integral neighborhood corner points. Figure 4.12 overlays the internal image summation areas over the original image. The summation of the neighborhood (highlighted in green) in Figure 4.12 can be computed in four arithmetic operations using the integral image, shown in Equation 4.1.

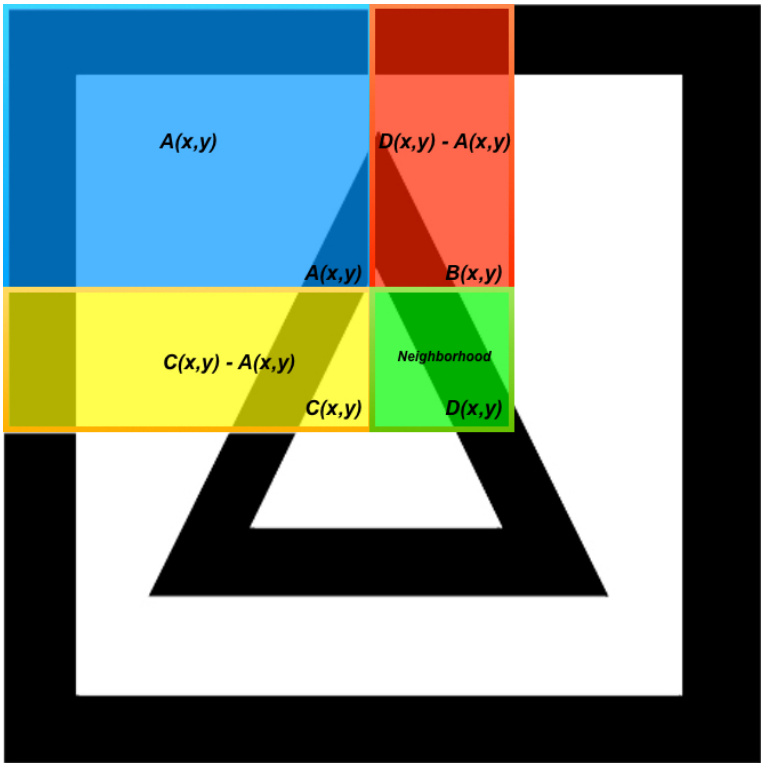


Figure 4.12: Integral Image Neighborhood Summation

$$\sum_{(x,y)\in neighborhood} I(x,y) = A(x,y) + D(x,y) - B(x,y) - C(x,y)$$

Equation 4.1: Neighborhood Summation Equation Utilizing Integral Image

Equation 4.1 represents the neighborhood summation calculation by utilizing integral images, where *I* represents the original image and *A, B, C, D* represent the corner points in the integral image.

To compute the integral image efficiently in CUDA, optimizations should be focused towards the exclusive scan operation. Exclusive scanning is defined as the 1D accumulation of values in an array, for each value computed is equal to the sum of all previous values, excluding the current value. Equation 4.2 shows the relationship between the 1D input, and the exclusive scan output.

$$[x_0, x_1, x_2, ..., x_N] -> [0, x_0, (x_0 + x_1), ..., (0 + x_0 + x_1 + ... + x_{N-1})]$$

Equation 4.2: Exclusive Scan Operation

The parallel implementation of the exclusive scan operation involves two separate phases in order to achieve O(N) work complexity: *up-sweep,* and *down-sweep.* The *up-sweep* implementation can be conceptually visualized as overlaying a balanced tree over the input data. The *up-sweep* performs a summation of the children nodes of the balanced tree and assigns each summation result to the parents. Figure 4.13 shows the visual representation of the parallel *up-sweep* phase. Each thread performs $\log_2(N)$ iteration where N is the size of the input data. At each iteration, only half of the threads are active from the previous iteration. The active threads are highlighted in red in each iteration of the *up-sweep* phase shown in Figure 4.13.

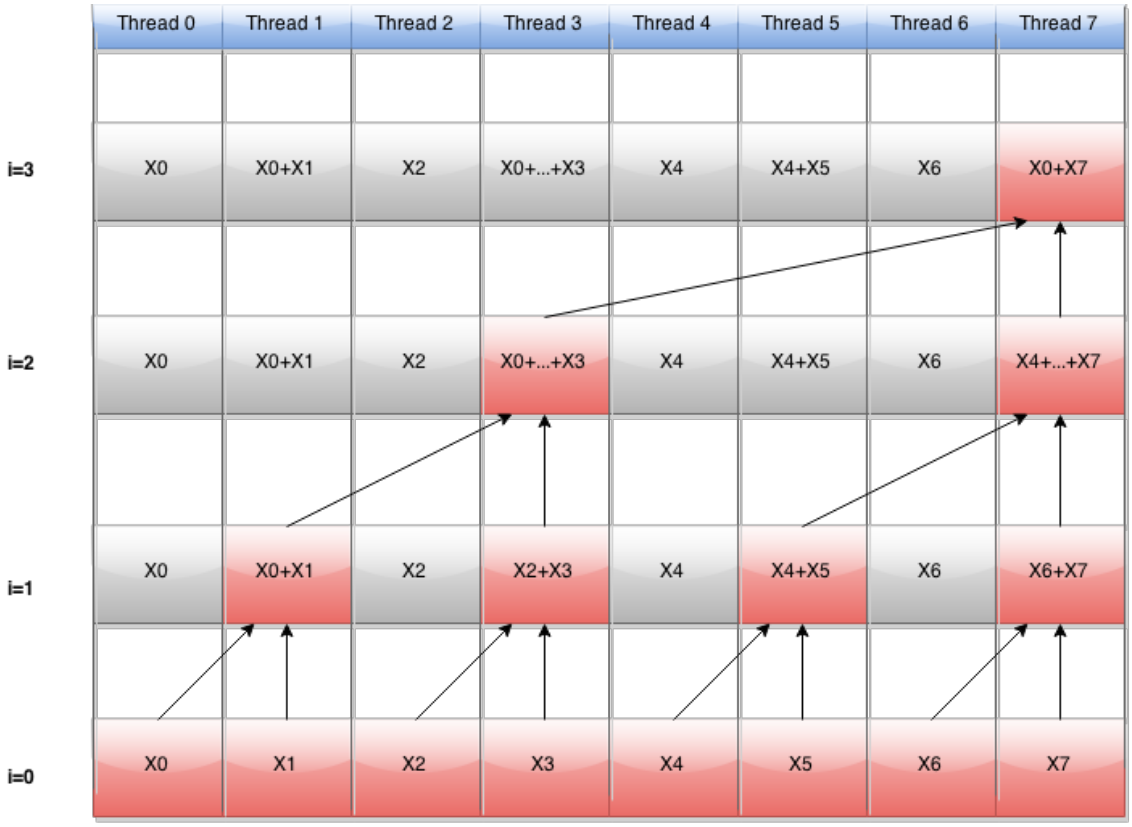


Figure 4.13: Parallel *Up-Sweep* Scan Implementation

After the *up-sweep* phase has finished on the input data, the *down-sweep* phase must be performed in order to complete the exclusive scan operation. Figure 4.14 shows the parallel implementation of the *down-sweep* phase. Before starting the *down-sweep* operation, the last element from the *up-sweep* phase is set to zero. At each iteration, twice the number of threads are active than the previous iteration. The active threads are highlighted in red in Figure 4.14. Once the *down-sweep* phase is finished, the exclusive scan operation is complete. The parallel exclusive scan operation is work efficient and has a O(N) work complexity, same as the sequential implementation.

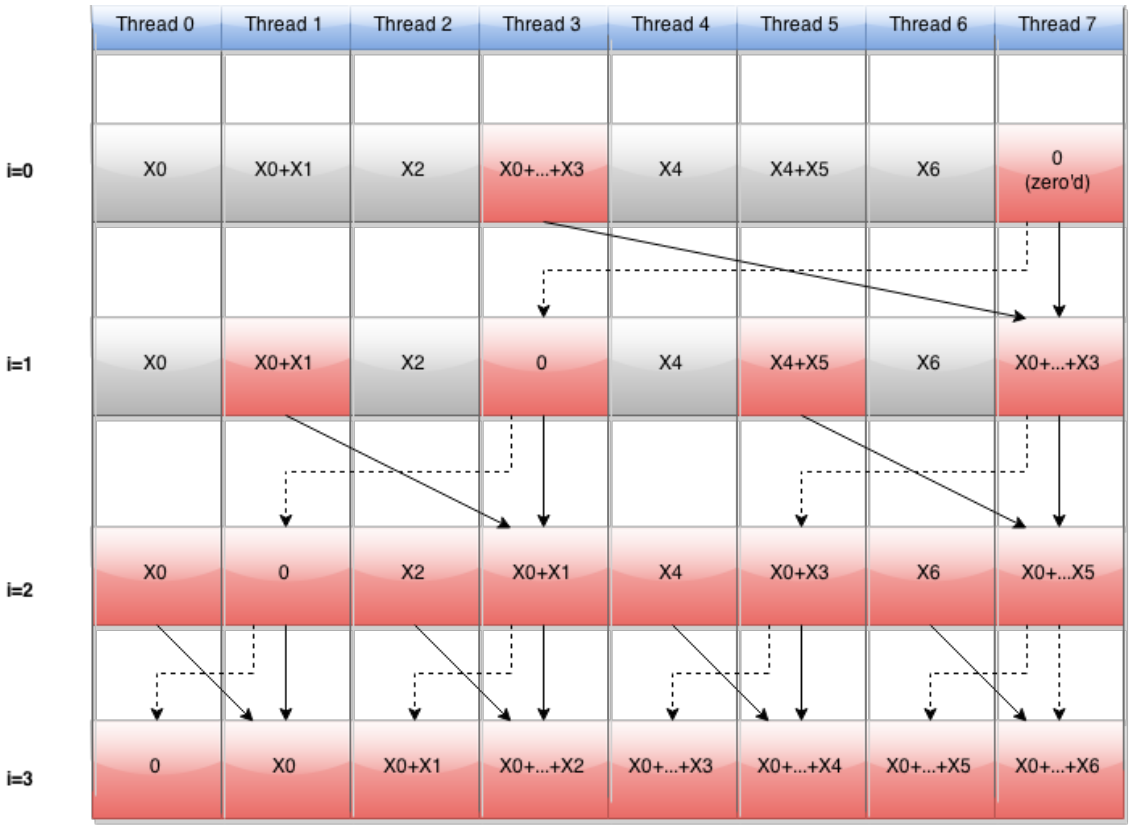


Figure 4.14: Parallel *Down-Sweep* Scan Implementation

The naive CUDA implementation of the exclusive scan operation loads the entire input array into shared memory and performs the entire scan operation within a single block of threads. Utilizing only a single thread block limits the maximum input data size to the GPGPU maximum threads per a block specification. Since a single block executes exclusively on a single SM, utilizing a single block on the GPGPU leaves the rest of the SMs idle, yielding poor thread occupancy. A higher degree of parallelism can be achieved by incorporating more thread blocks to utilize all SMs on the GPGPU; however, this increases the complexity of the implementation since the shared memory segments are not shared between thread blocks.

The optimized CUDA implementation of the exclusive scan operation can be implemented by partitioning the input data, size N, into B thread blocks. The number of thread blocks should be partitioned evenly by the number of threads per block; thus, the

number of thread blocks should be equal to *(N / T)*, where *T* is the number of threads per a block. The number of thread blocks should at least be equal to the number of streaming multiprocessors, *S,* on the GPGPU to ensure that every SM is performing work. Thus, the number of threads per block should be defined as *N / S*, which creates the same number of thread blocks as there are SMs. Table 4.2 shows an example of a thread configuration that utilizes all SMs on the GPGPU with a given data input size.

Variable	Expression	Value
Input Data Size	N *(independent)*	1024
Number of SMs	S *(independent)*	8
Threads Per Block	T = (N / S) (dependent)	32
Number of Blocks	B = (N / T) (dependent)	32

Table 4.2: Example Thread Configuration For High SM Occupancy

Each block of threads will compute its local exclusive scan operation on *T* elements of the input data. Once all of the thread blocks are finished computing the scan operation on their portions of the data, the last element of every block will be extracted to form an auxiliary array [14]. The same exclusive scan operation will then be performed on the the auxiliary array, and once finished, the elements of the auxiliary array will be summated back into the segmented scan arrays to complete the exclusive scan operation. The CUDA exclusive scan implementation spanning over multiple thread blocks is visually represented in Figure 4.15.

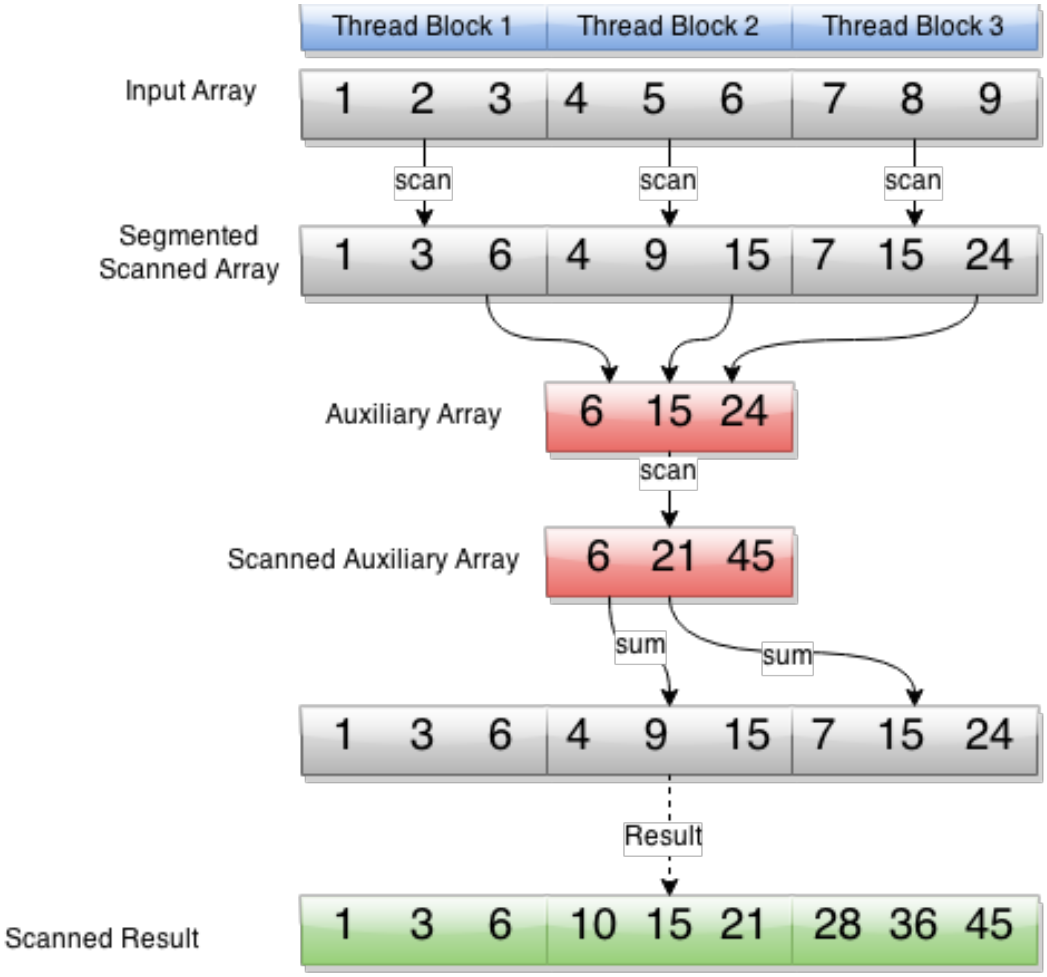


Figure 4.15: Parallel Scan Involving Multiple Thread Blocks

The integral image is computed by first scanning the rows of the input image, then performing the scan on the columns. In order to utilize spatial locality of the GPGPU L1 and L2 caches, as well as code reuse, the result of the exclusive scan on the image's rows will be transposed, then the result will be exclusively scanned again, effectively operating on the columns. Once the second exclusive scan operation is completed, the image is transposed again in order to obtain the final integral image result. Row major images are addressed in memory sequentially, thus the first pixel of row N is addressed sequentially in memory after the last pixel in row N-1. Transposing the image in order to compute the exclusive scan on the columns will effectively improve the cache hit-ratio

and improve the overall performance of the GPGPU integral image implementation. The integral image architecture is showing in Figure 4.16.

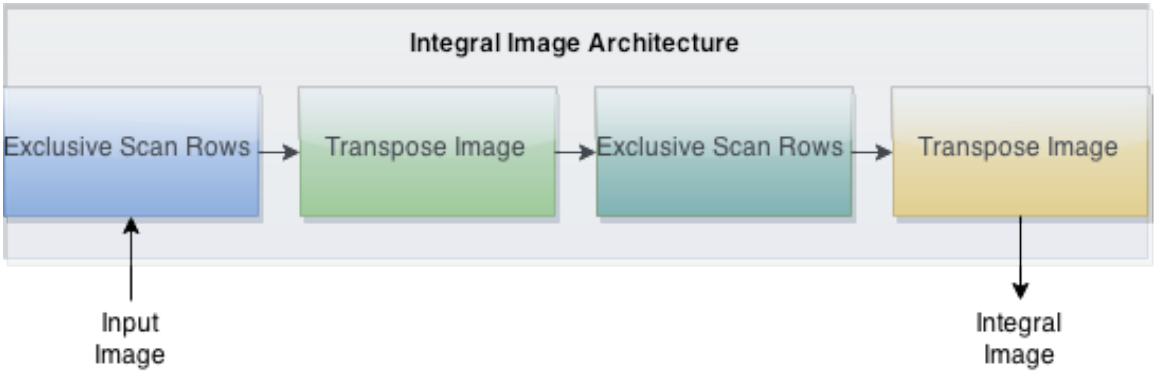


Figure 4.16: Integral Image Algorithm Architecture

By utilizing integral images, only four global memory access are necessary to compute the summation of a neighborhood regardless of the neighborhood dimension. By creating integral images for I_x^2, I_y^2, and I_xI_y gradient products, the total number of global memory accesses reduces from $3W^2$ to a constant of 12 (4 global memory accesses per ROI summation). Figure 4.17 shows the comparison of integral image processing times over several different platforms: standard C, MATLAB, and optimized CUDA. The optimized CUDA implementation showed great performance over the standard C and MATLAB implementations due to the optimized parallel *up-sweep* and *down-sweep* implementations.

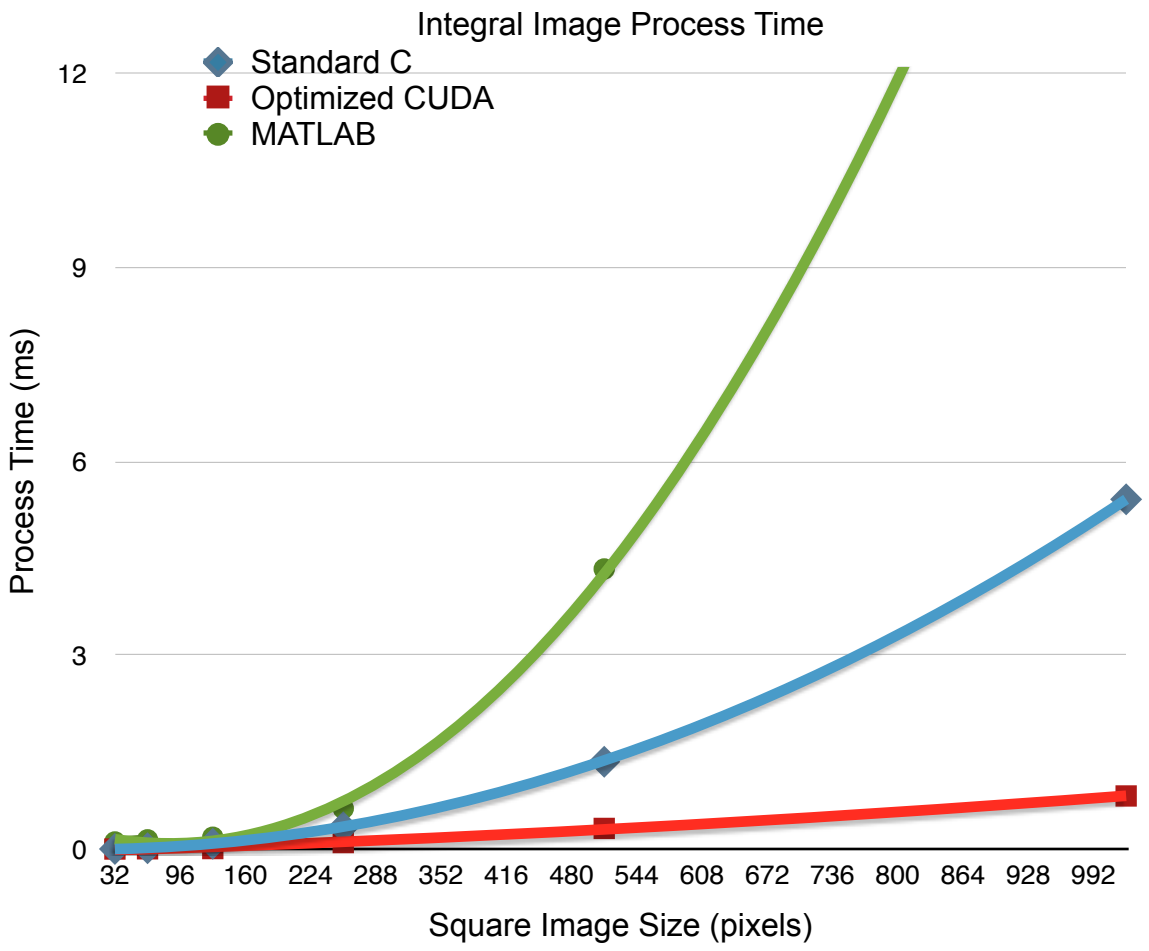


Figure 4.17: CPU vs GPGPU Integral Image Process Time

4.3.3 Corner Detector Performance Results

The corner detector performance results, shown in Figure 4.18, were compared over several different platform implementations: standard C, naive CUDA, and optimized CUDA. The optimized CUDA implementation utilized all performance optimizations discussed in Section 4.3.2. Figure 4.18 shows that the standard C implementation processing time has a parabolic relationship with the image dimension size, thus it is not suitable for realtime Harris corner detection. The CUDA implementations showed significant speedup over the standard C implementation. All implementations show equivalent detection accuracy, for the CUDA optimizations do not compromise detection precision. The optimized CUDA implementation showed the greatest speedup, with

corner detection processing time not exceeding 2.5 ms for all image dimension sizes up to 1024x1024 pixels.

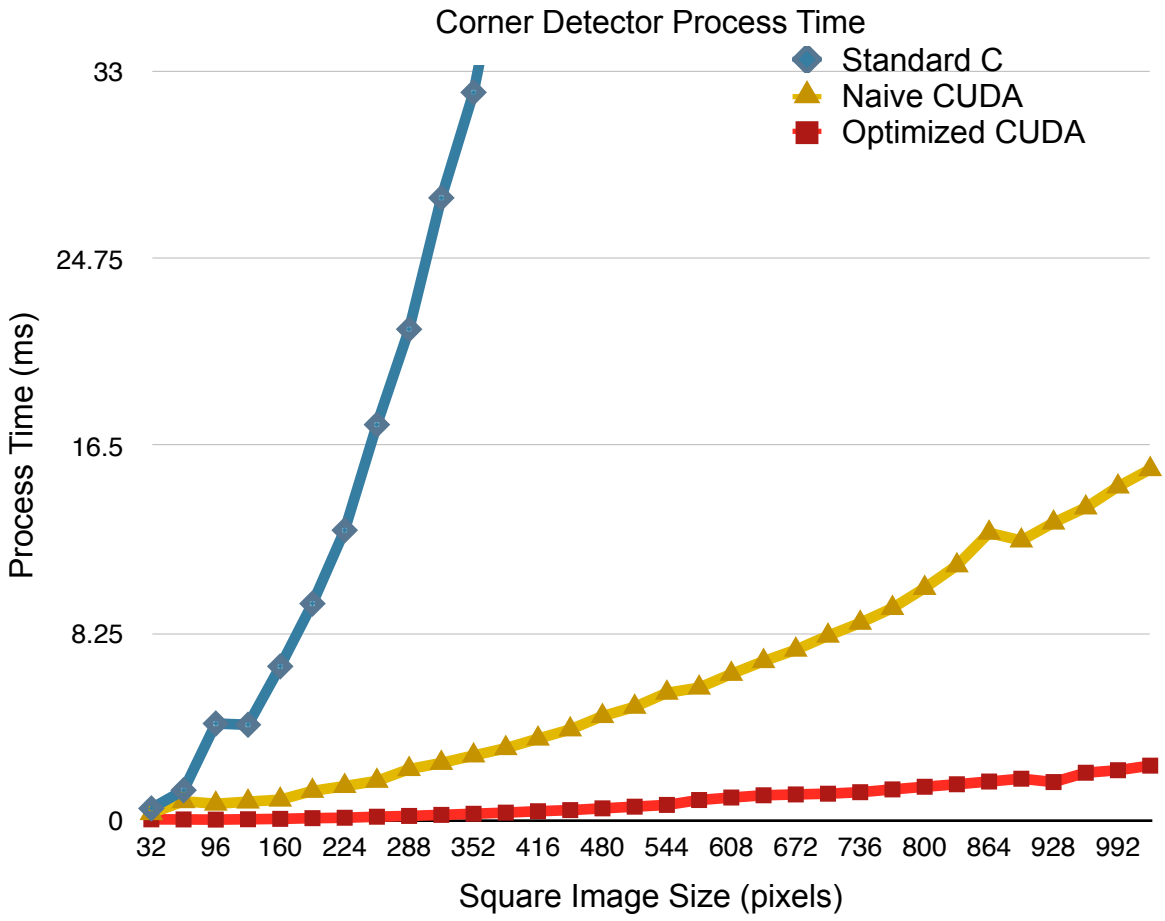


Figure 4.18: Corner Detection Performance Results

4.4 GPGPU Non-maxima Suppression (NMS)

Non-maxima suppression (NMS) is a non-linear filter, which operates by filtering non-maxima values within a local neighborhood. NMS effectively attempts to distribute the corner feature locations, represented by the corner response, more evenly by defining a minimum distance between corner feature points. Since each local neighborhood is processed with zero dependence on any other neighborhood in NMS, each neighborhood can be processed in parallel on the GPGPU architecture.

4.4.1 Naive NMS GPGPU Implementation

The naive GPGPU NMS involves spawning a CUDA thread for every pixel in the corner response. The entire corner response is first loaded into global memory (slowest memory). Each thread will determine if their pixel is the maximum value within its local neighborhood. If the pixel is not the maximum, then the output pixel value is set to zero, otherwise the pixel value is left unchanged. Each thread iterates over its own local neighborhood in raster scan order until a neighboring value greater than the pixel is found. Figure 4.19 shows the naive CUDA kernel implementation.

```
FOR every thread
       FOR neighbor in pixel neighborhood
              IF neighbor > pixel
                     pixel = 0
                     break
              END IF
       END FOR

       result[thread row][thread col] = pixel
END FOR
```

Figure 4.19: Naive Non-maxima Suppression CUDA Kernel Pseudo Code

The naive implementation requires W^2M^2 global memory reads as a worst case scenario, where W represents the dimension size of the neighborhood, and M represents the image dimension size. Each thread suffers its worst case performance if the pixel being considered is a local neighborhood maximum, thus the algorithmic complexity for a single thread is $O(W^2)$.

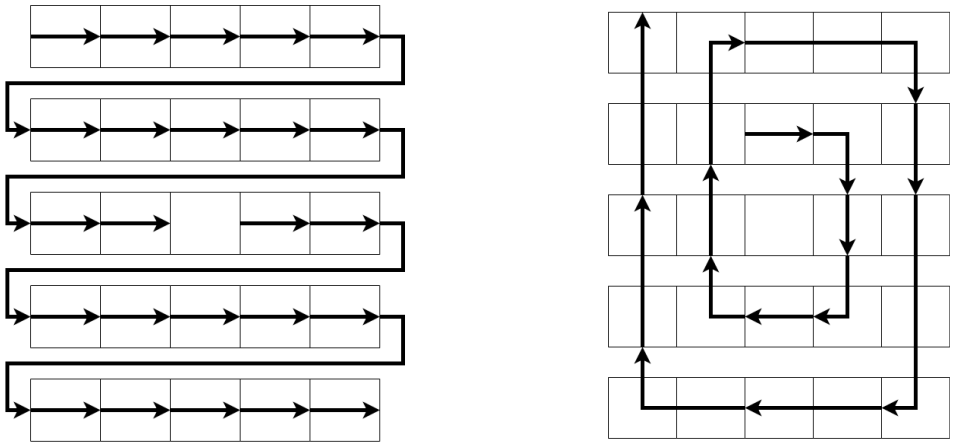
4.4.2 Optimized NMS GPGPU Implementation

4.4.2.1 Spiral Scanning

Every thread has a best case scenario when there is only a single comparison within the local neighborhood before breaking its execution. This implies that the first

neighbor visited has a greater value than the pixel, which further implies that the pixel is not a local maximum. The worst case performance is invoked when the pixel is a local maximum, thus (W^2-1) comparisons are necessary. Förester and Gülch presented the idea that the average number of comparisons can be reduced by iterating the neighborhood pixels in a spiral scan order, rather than a raster scan order [15]. A corner response has zero-valued intensity for the majority of the corner scores; therefore, only a smaller percentage of the corner response pixels are neighborhood maximums. A local maximum pixel within a W^2 sized neighborhood is guaranteed to be the local maximum within its $(W-1)^2$, $(W-2)^2$, …, $(3)^2$ sized neighborhoods. By visiting smaller sized local neighborhoods in spiral order first, partial neighborhood local maximums can be determined before moving onto larger sized neighborhoods [15].

A spiral scan order can be implemented to determine local maximums in smaller neighborhoods before scanning neighbors that are farther away from the pixel. Figure 4.20 shows the difference between the raster and spiral scan order.



(a) Raster Scan Order (b) Spiral Scan Order

Figure 4.20: Iterative Neighborhood Scanning Orders

The spiral scan order will determine non-maximum pixels within the 3^2 sized neighborhoods first, which will break on average since the corner response pixels are on average non-local maximums. By failing the first small neighborhood comparison, computation can be stopped and larger neighborhood comparisons are not necessary; therefore, the number of comparisons for a W^2 size neighborhood shows roughly the same number of comparisons for a 3^2 sized neighborhood on average. By utilizing a spiral scan order for neighborhood iteration, the number of comparisons can be significantly decreased in the NMS implementation. Spiral scanning effectively reduces the number of global reads, therefore it increases the CGMA ratio of the CUDA NMS implementation.

4.4.2.2 Corner Segmentation

The NMS algorithm effectively erodes low magnitude corner response by determine local neighborhood maximums; however, the majority of the corner response has zero-valued intensity. A zero intensity value in the corner response has a zero probability that it is a local maximum, thus performing NMS on that value is not necessary. The NMS algorithm can be optimized by only suppressing pixels with a non-zero corner score and ignoring all other pixels. This effectively minimizes the number of threads running on the GPGPU and allows for more resources per thread. Rather than suppressing the corner response directly, the non-zero corner score pixels can be segmented from its zero-valued background.

The segmentation is done by applying a threshold filter to segment all non-zero corner scores away from the background. The non-zero corner scores are then extracted from the corner response into an array. Figure 4.21 shows the segmentation of the corner response from its zero-valued background. The segmented information is compiled into a 1D array, therefore the spatial coordinates from the corner response must be extracted along with the corner scores.

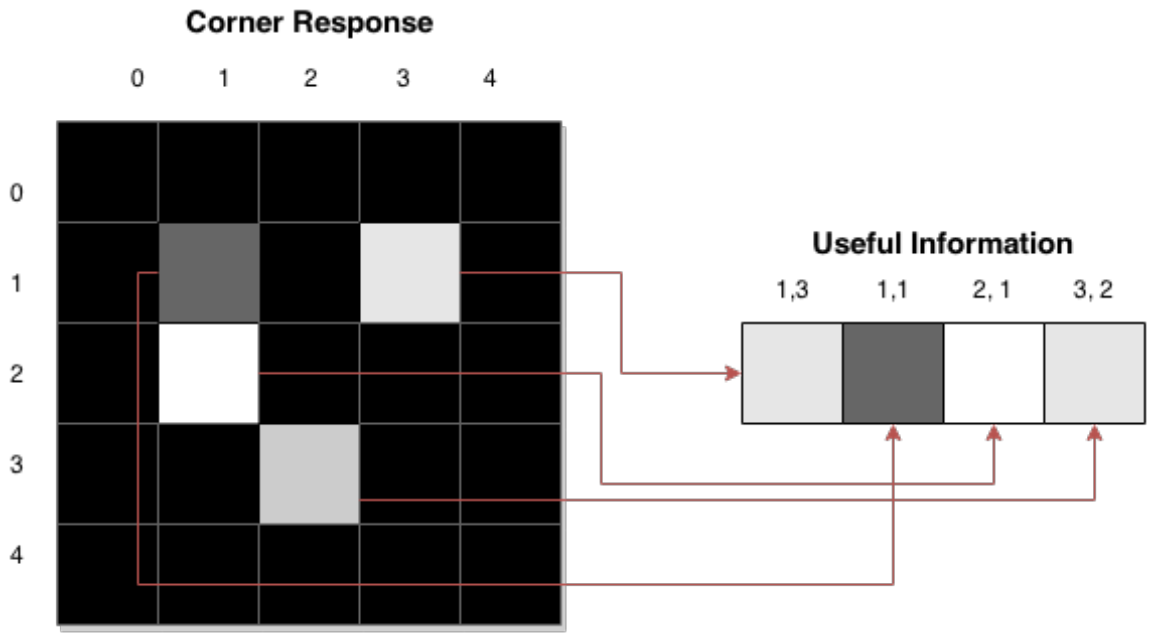


Figure 4.21: Corner Response Segmentation

By utilizing corner response segmentation, each non-zero corner score within the segmentation information has a much higher probability that it's a neighborhood maximum. Rather than spawning a thread to perform NMS for every corner score in the corner response, a thread only needs to be spawned for every non-zero corner score in the corner response segmentation. This results in a significantly less number of threads running on the GPGPU, thus increasing performance of the CUDA NMS implementation.

4.4.2.3 Texture Memory Utilization

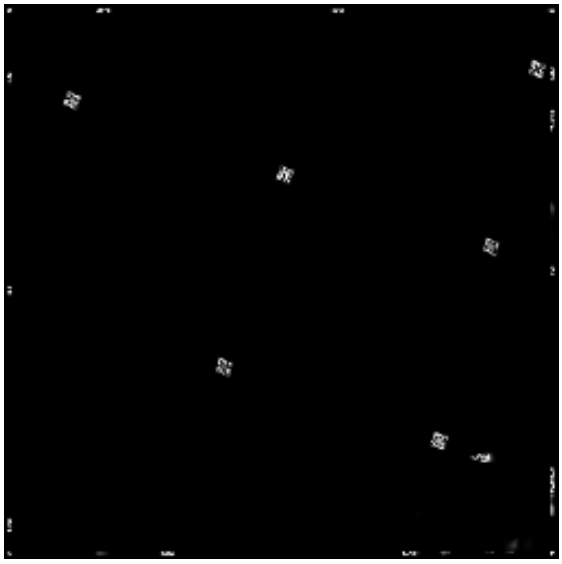
Each NMS result pixel is computed by performing (W^2-1) global memory reads, worst case, around the each corner score pixel. Due to the corner segmentation, explained in Section 4.4.2.2, thread IDs no longer correlate to spatial locations of the corner response. This increases the complexity of shared memory usage; however, since the memory reads of each thread have spatial locality, texture memory can be

utilized to increase memory access performance and increase the CGMA ratio. By loading the segmented corner response into texture memory, threads will experience a higher cache hit-ratio due to the on-chip texture cache, and therefore will have an increase in memory access performance.

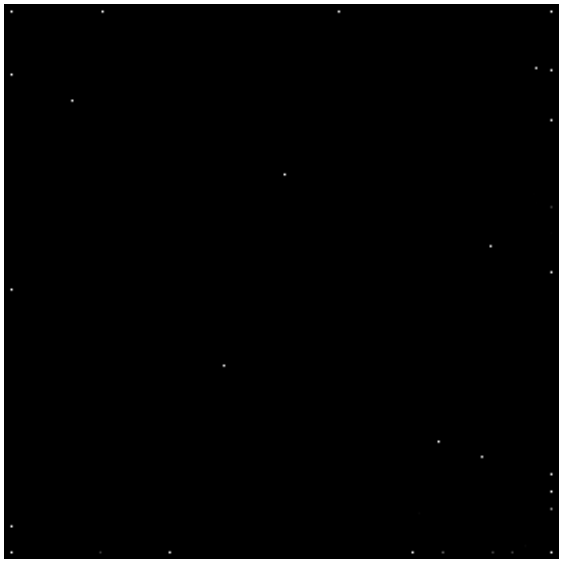
4.4.3 NMS Performance Results

The NMS performance results were obtained by performing NMS on several different platforms: standard C, MATLAB, naive CUDA, and optimized CUDA. The optimized CUDA implementation of NMS takes into account all optimizations discussed in Section 4.4.2: spiral scanning, corner segmentation, texture memory utilization. Figure 4.22 shows the input image used for benchmarking the NMS performance. The performance NMS was measured by computing the suppressed corner response of the input over several different resolutions and NMS neighborhood dimension sizes. The input image was down sampled from 1024x1024 to 32x32 logarithmically, and the neighborhood dimension size was iterated from 3 to 7 linearly. Figure 4.23-4.25 shows the performance results of the NMS implementations while varying the size of the NMS neighborhood. As the NMS neighborhood dimension size increases, the processing times for the standard C, MATLAB, and naive CUDA implementations increase; however, the optimized CUDA implementation sustains almost constant performance due to spiral neighborhood scanning. The standard C and MATLAB implementations show that they are not optimal for realtime Harris corner detection due to the rate of increase in processing time with image dimension size. The CUDA implementations, naive and optimized, show great performance, however the optimized CUDA implementation showed the best performance with a computation time under 2 ms for all neighborhood dimension sizes.

(a) Input Image



(b) Corner Response of (a)
Sensitivity = .04
Threshold = 10^6



(c) NMS of (b)
Neighborhood Dimension = 7

Figure 4.22: NMS Performance Test Input Image

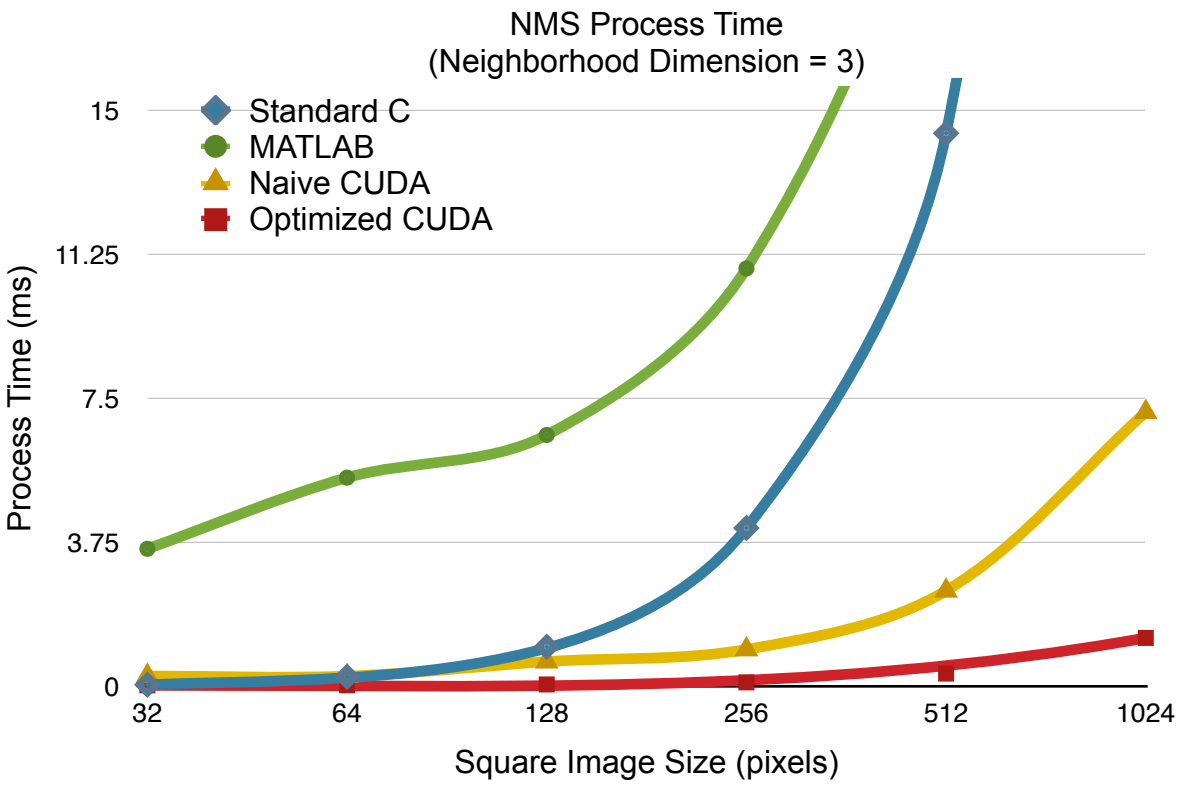


Figure 4.23: NMS Process Time With Neighborhood Dimension of 3

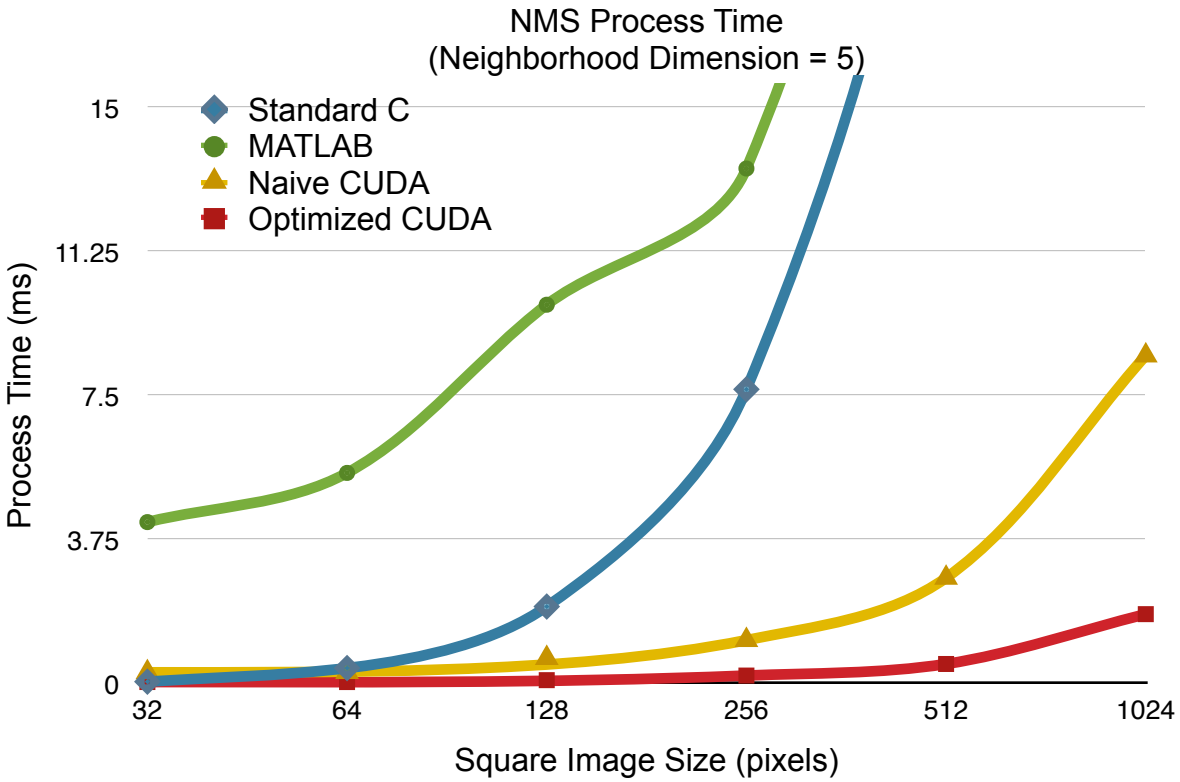


Figure 4.24: NMS Process Time With Neighborhood Dimension of 5

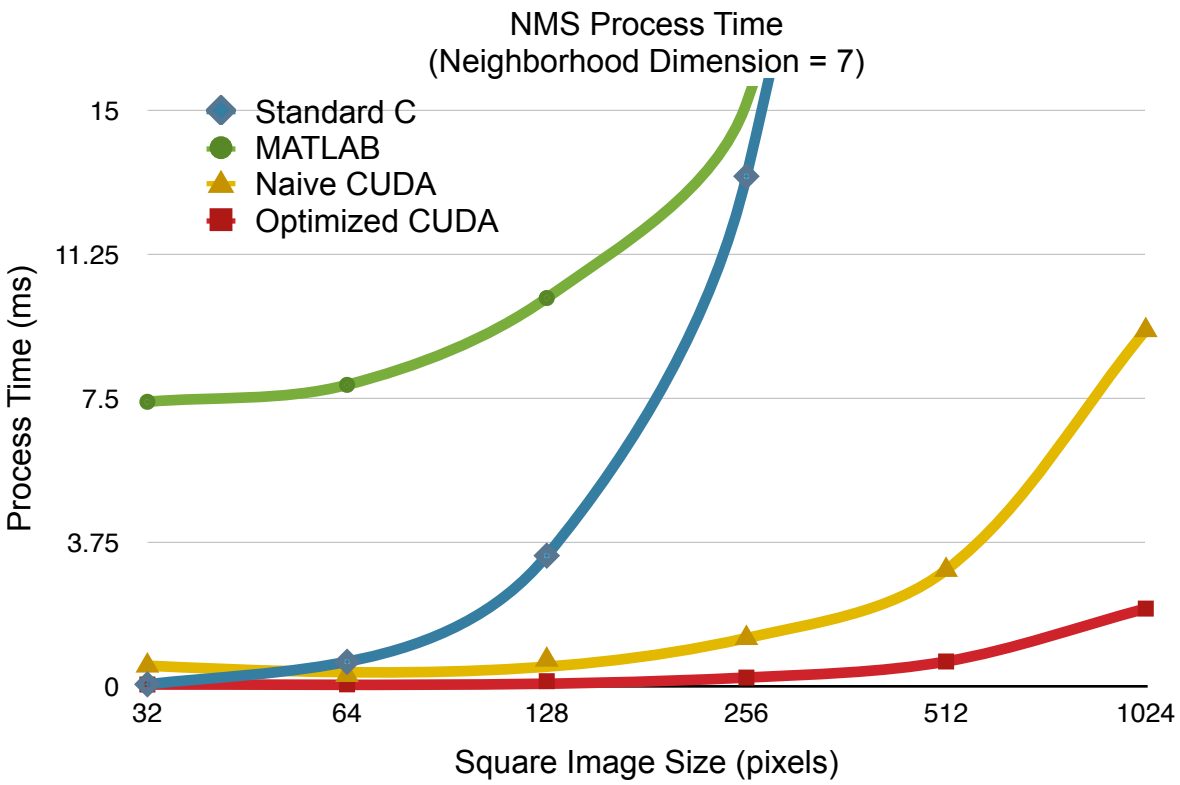


Figure 4.25: NMS Process Time With Neighborhood Dimension of 7

4.5 GPGPU Harris Corner Detection Performance Results

The performance results of the Harris corner detection implementations were compared for several different platforms: standard C, MATLAB, OpenCV, naive CUDA, and optimized CUDA. OpenCV is a cross-platform software library which aims for realtime image processing and computer vision performance [16]. OpenCV was considered as a benchmark to compare it's cutting edge Harris corner detection implementation, which is commonly used in industry, to the optimized CUDA implementation developed in this thesis. The performance results were measured by executing the Harris corner detection over a series of image resolutions (32x32 - 1024x1024), using the input parameters shown in Table 4.3. The NMS threshold was manually selected for each input image to yield the best results.

Parameter Type	Parameter Value
Gaussian Kernel Size	3x3
Sobel Kernel Size	3x3
Corner Detector Sensitive	0.04
NMS Neighborhood Size	5x5

Table 4.3: Harris Corner Detection Parameters

The test images used for the performance measurements are shown in Figures 4.26-4.28. Each image has a starting resolution of 1024x1024 pixels, which were all down sampled in increments of 32x32 pixels in order to compare the performance results for smaller image resolutions. The optimized CUDA implementation incorporates all optimizations discussed in Chapter 4. The CUDA optimizations made to the Harris corner detection implementation are summarized by Table 4.4. The optimized CUDA performance results are sensitive to the type of input data due to the spiral neighborhood iteration for NMS, discussed in Section 4.2.2.1; therefore, the performance results found for all platforms were averaged for all images to rightfully compare the implementations.

Original Image
1024x1024

Processed
NMS Threshold = 10^{7}

Figure 4.26: Harris Corner Detection Performed on Image of an F18

Original Image
1024x1024

Processed
NMS Threshold = 10^{10}

Figure 4.27: Harris Corner Detection Performed on Image of the Eiffel Tower

Original Image
1024x1024

Processed
NMS Threshold = $20x10^{8}$

Figure 4.28: Harris Corner Detection Performed on Image of Mt.Whitney CA

Stage	Optimization	Description	Section
Convolution	Separable Filters	Reduce Multiplications / Global Memory Reads	4.2.2.1
	Constant Memory Utilization	Store Filter in Constant Memory to Increase CGMA Ratio	4.2.2.3
	Asynchronous Memory Transfers	Pipeline Convolution and Memory Transfers	4.2.2.2
	Shared Memory Utilization	Store Image in Constant Memory to Increase CGMA Ratio	4.2.2.4
Corner Detector	Integral images	Reduce Neighborhood Summation to 4 Arithmetic Operations	4.3.2.1
Non-Maxima Suppression	Spiral Scanning	Reduce Number of Neighborhood Comparisons to Increase CGMA Ratio	4.4.2.1
	Corner Segmentation	Increase GPGPU Workload Efficenency	4.4.2.2
	Texture Memory Utilizations	Increase CGMA Ratio by Exploiting Spatial Locality	4.4.2.3

Table 4.4: GPGPU Harris Corner Detection Optimization Summary

Figure 4.29 shows the processing time for the Harris corner detection implementation running on platforms standard C, MATLAB, OpenCV, naive CUDA, and optimized CUDA. The naive and optimized CUDA performance measurements include

memory bus transfer time from host CPU memory to GPGPU memory and vice-versa. The standard C and MATLAB implementations show processing times greater than 220 ms for image resolution 1024x1024 pixels, which proves that their implementations for realtime Harris corner detection are not feasible. Both the OpenCV and naive CUDA implementations show similar processing times of 50 ms for image resolution 1024x1024 pixels, which is a significant improvement over the standard C and MATLAB implementations; however, their processing times yield undesirable results for realtime performance. To rightfully compare the processing times against OpenCV, naive CUDA, and optimized CUDA, the processing times were re-plotted with different time scale in Figure 4.30 to show the performance improvement of the optimized CUDA implementation. The optimized CUDA implementation showed a processing time of 11 ms for image resolution 1024x1024 pixels, thus it was deemed best fit for realtime Harris corner detection over the other implementations. Figure 4.31 shows the speedup characteristics of the optimized CUDA implementation over the other platforms: standard C, MATLAB, OpenCV, and naive CUDA. The optimized CUDA Harris corner detection implementation had an average speedup of 14.9 over standard C, 33.8 over MATLAB, 3.73 over OpenCV, and 6.8 over the naive CUDA implementation. Table 4.5 shows the feasible processing FPS (frames per seconds) for the optimized CUDA Harris corner detection over several different image resolutions. By utilizing the optimized CUDA implementation, realtime corner detection is feasible with a CUDA software solution.

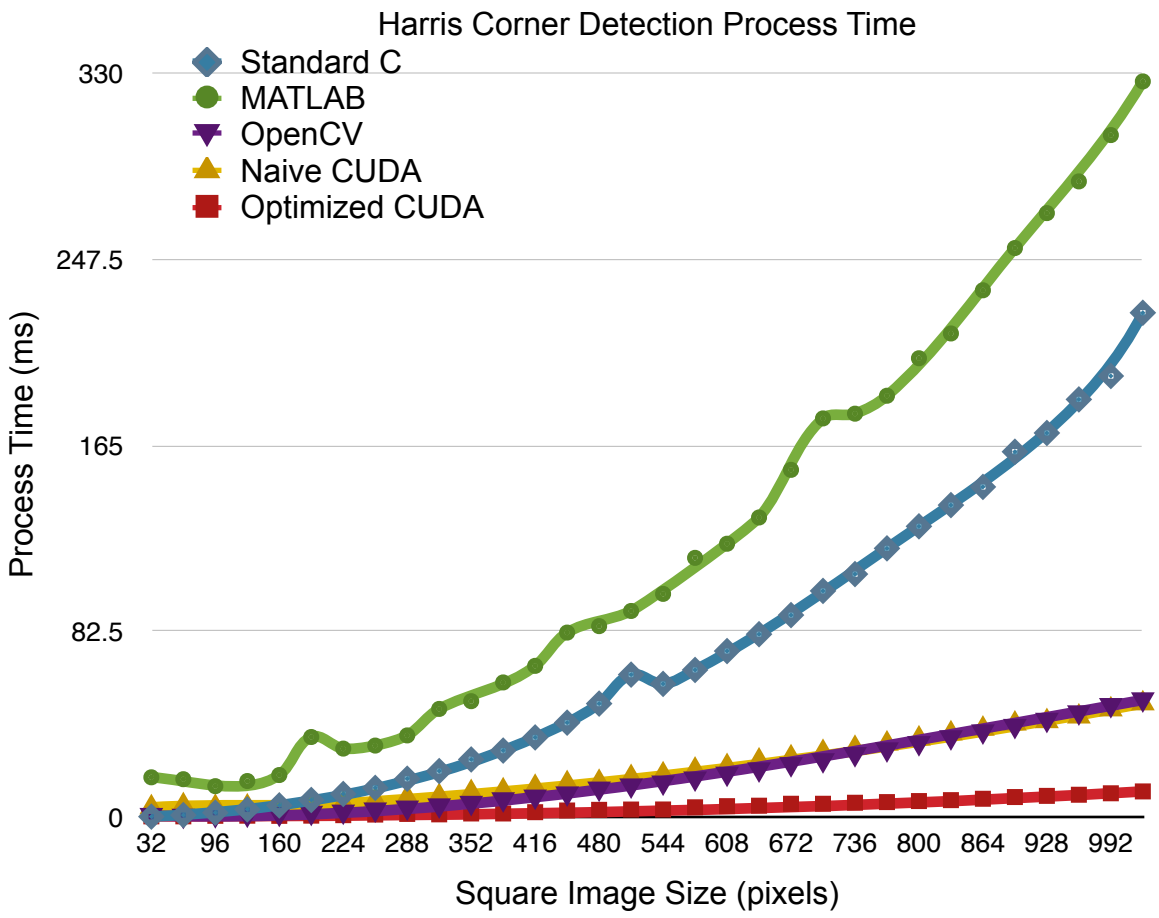


Figure 4.29: Harris Corner Detection Process Time For All Platforms

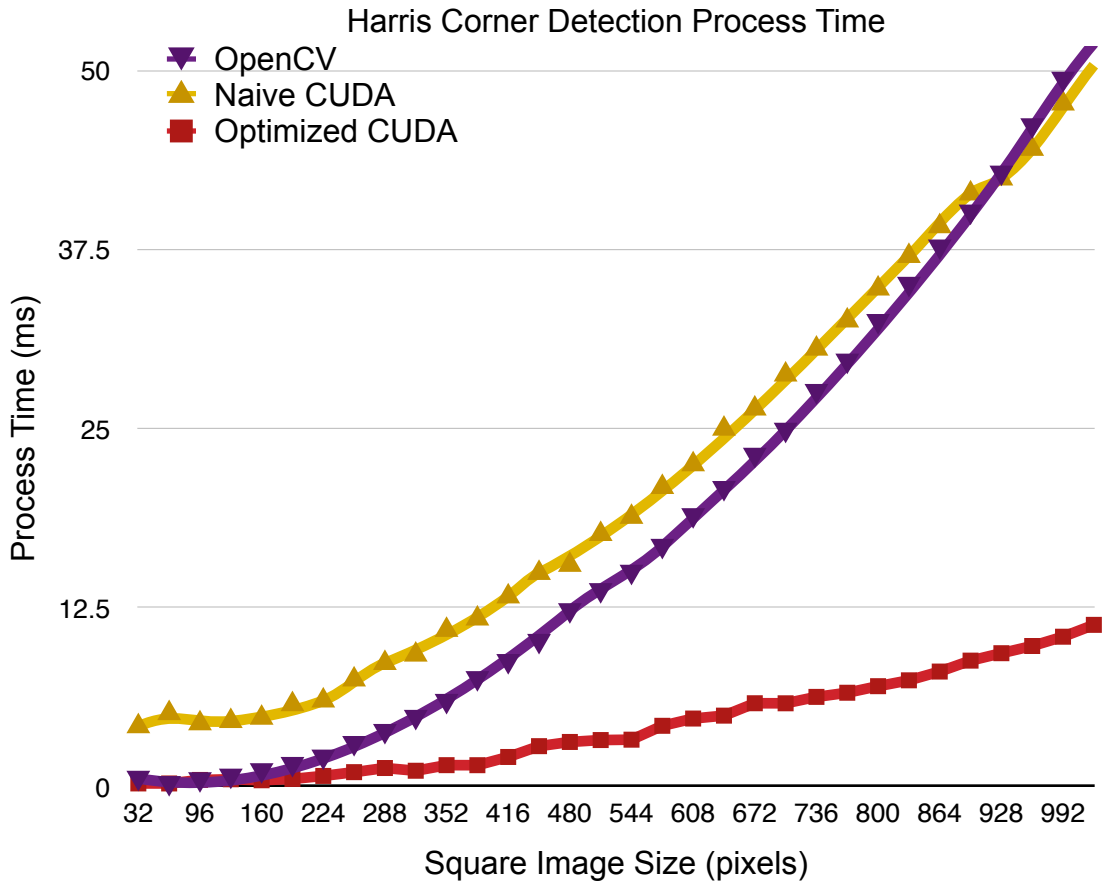


Figure 4.30: Harris Process Time For High Performance Platforms

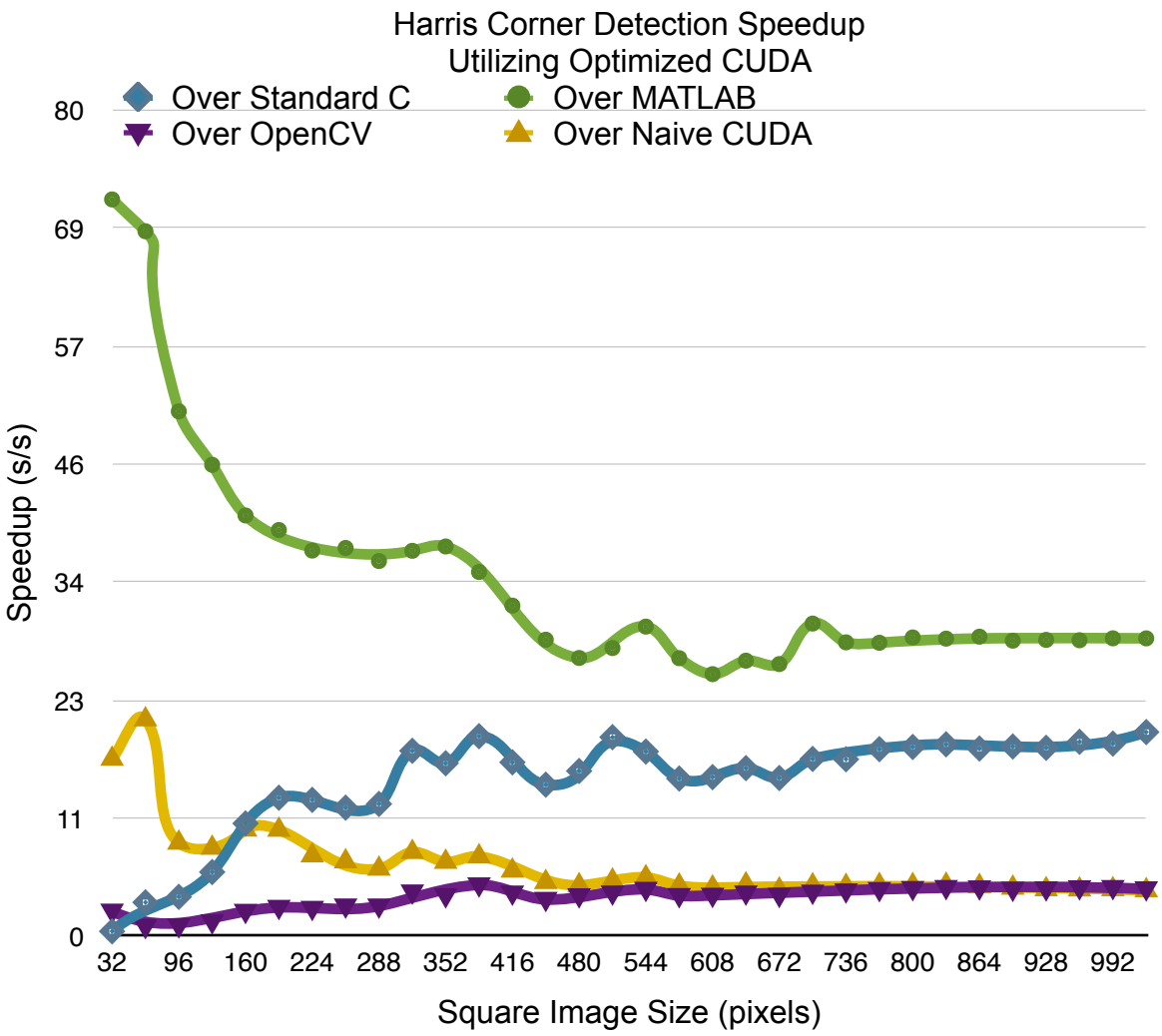


Figure 4.31: Optimized CUDA Harris Corner Detection Speedup

Image Resolution	Optimized CUDA Processing Time (ms)	Optimized CUDA Feasible Processing FPS (1/s)
32x32	0.246242	4061
64x64	0.247053	4047
128x128	0.536684	1863
512x512	3.27323	305
1024x1024	11.3103	88

Table 4.5: Optimized CUDA Harris Corner Detection Feasible FPS

Chapter 5: Feature Matching Application

5.1 Feature Matching Introduction

In a feature matching computer vision system, once features have been detected and extracted, they are matched to other features to determine correspondence. Feature matching is used extensively in computer vision systems for several applications: motion detection, image registration, video tracking, panorama stitching, 3D modeling, and object recognition. Feature matching is computationally intensive and can be broken into three main processes: feature detection, feature description, and feature matching. Feature detection is the process of locating feature points within an image. Feature description is the process of describing the located features uniquely. Feature matching is the process of determining the correspondence of features between feature sets.

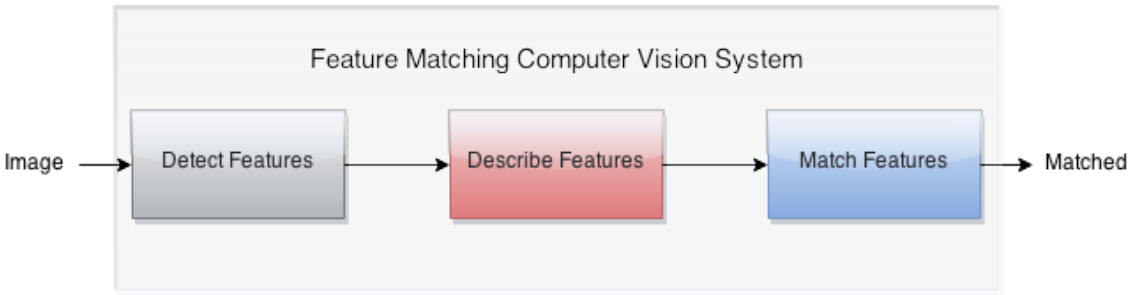


Figure 5.1: Feature Matching Computer Vision System

5.2 Feature Matching Implementation

The optimized CUDA implementation of the Harris corner detection algorithm, discussed in Chapter 4, can be utilized to provide significant performance benefit to the feature matching computer vision system shown in Figure 5.1. There are several different types of feature detection, descriptions, and matching techniques, show in Table 5.1. The algorithms used for each stage in the feature matching implementation are highlighted in green in Table 5.1.

Stage	Type
Feature Detection	HARRIS STAR SIFT SURF ORB BRISK MSER GFTT DENSE SIMPLE BLOB
Feature Description	SURF SIFT BRIEF BRISK ORB FREAK
Feature Matching	FLANN BASED (K-NN) BRUTE FORCE

Table 5.1: Feature Matching Stage Types

5.2.1 SURF Overview

The feature description algorithm selected for the feature matching system was SURF (Speeded Up Robust Features) [17]. SURF is used to describe the features due its high computational performance. The developers of the SURF descriptor—Herber Bay, Tinne Tuytelaars, and Luc Van Gool— state that the “SURF descriptor outperforms the other descriptors in a systematic and significant way” [17]. The SURF descriptor divides the located feature region into 4x4 square subregions. For each subregion, the Haar wavelet responses are determined in the X and Y directions and weighted by a Gaussian filter to reduce noise [18]. A vector is then formed by summing the Haar responses in the X and Y directions, shown in Equation 5.1, to describe the feature. SURF provides a robust way for describing features uniquely which are insensitive to noise, thus SURF is a good candidate for the feature matching computer vision system.

$$v = (\sum dx, \sum dy, \sum |dx|, \sum |dy|)$$

Equation 5.1: SURF Feature Descriptor Vector

5.2.2 FLANN (K-NN) Overview

FLANN (Fast Library for Approximate Nearest Neighbors) was chosen for feature matching due to its performance optimizations over linear searching [19]. FL ANN is a software library which implements the K-Nearest Neighbor (K-NN) classification algorithm. K-NN classification algorithm classifies a feature point to a cluster based on the nearest neighbors. The algorithm involves a majority voting scheme which classifies a feature point based on the closest neighbors around that feature point [20]. Figure 5.2 shows an example of K-NN classification, where the circle is the feature to be classified. The squares and triangles represent feature clusters which were populated into the feature space from predefined training data. The K-NN algorithm's goal is to determine which cluster the circle belongs. The K-NN algorithm looks at the nearest K neighbors of the circle and classifies the feature to the most common feature found in the local neighborhood. Table 5.2 shows how the circle is classified in Figure 5.2 based on the size of K (number of neighbors). The number of neighbors considered is always an odd number in order to avoid equal cluster classification.

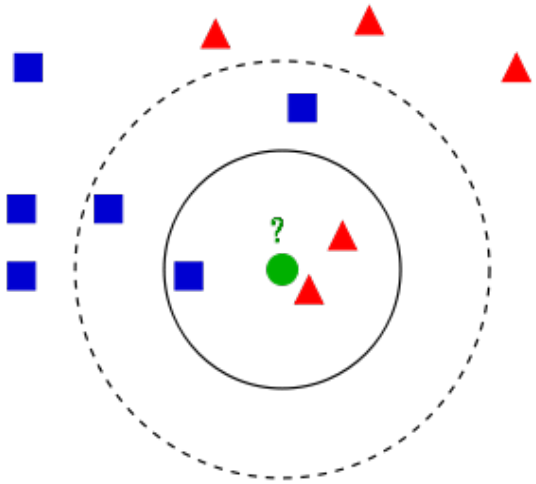


Figure 5.2: K-NN Classification Example [20]

K (Number of Neighbors)	Triangles In Neighborhood	Squares In Neighborhood	Circle Classifcation
1	1	0	Triangle
3	2	1	Triangle
5	2	3	Square
11	5	6	Square

Table 5.2: Circle Classification of Figure 5.2

5.3 Feature Matching Performance Results

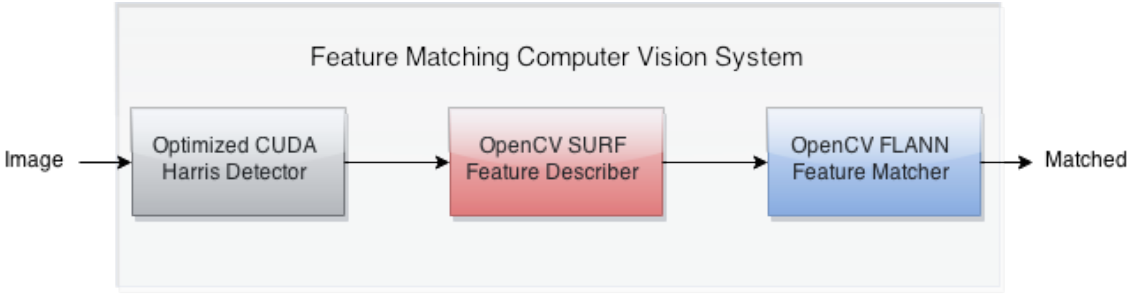
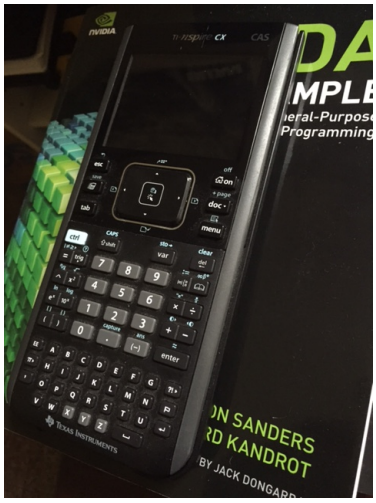


Figure 5.3: Feature Matching Stage Implementations

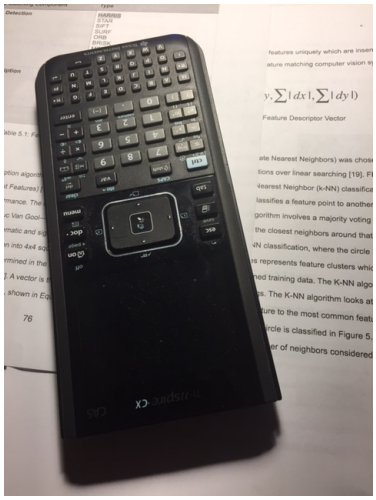
The feature describer and matcher stages selected for the feature matching computer vision system are shown in Figure 5.3. The describer and matcher stages were implement in OpenCV. The feature matching system was executed by first providing a training image to establish a feature set basis, then providing scene images where the features from the training image were matched to. Figure 5.4 (a) shows the training image used, and Figure 5.4 (b,c,d) shows the scene images used for measuring the performance of the feature matching computer vision system.

(a) Training Image



(b) Scene 1

(c) Scene 2



(d) Scene 3

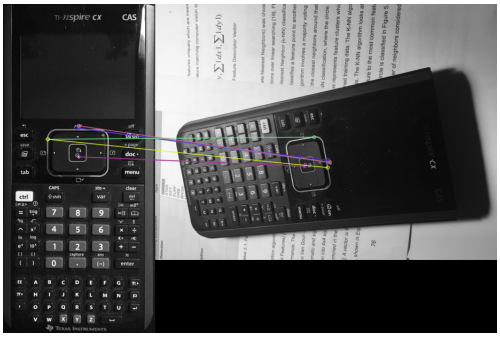
Figure 5.4: Training Image and Scene Images

The feature matching performance results were analyzed utilizing several Harris corner detection implementations discussed in Chapter 4: standard C, MATLAB, OpenCV, naive CUDA, and optimized CUDA. The performance results were analyzed by feature matching a training image to multiple scene images, shown in Figure 5.4. The scene images were sub-sampled from image dimensions 1024x1024 to 32x32

logarithmically to measure performance processing time over various image resolutions. Figure 5.5 shows the output of the feature matching computer vision system utilizing the top 10% best feature matches for each scene image.

(a) Matching to Scene 1

(a) Matching to Scene 2



(a) Matching to Scene 3

Figure 5.5: Feature Matching System Result

Figure 5.6 shows the processing times of the feature matching system with the corner detection implemented on different platforms: standard C, MATLAB, OpenCV, naive CUDA, and optimized CUDA. The feature description and matching algorithms, SURF and FLANN, were implemented sequentially in OpenCV on the CPU, with the exception of the MATLAB implementation The feature matching system utilizing the optimized CUDA Harris corner detection showed the best performance when compared

against the other platforms. Figure 5.7 shows the performance speedup of utilizing the optimized CUDA Harris corner detection implementation over other implementations in the feature matching computer vision system. The feature matching system utilizing the standard C implementation showed very poor performance for image dimensions above 256x256 for processing time increased parabolically with image dimension size. The feature matching system utilizing OpenCV or the naive CUDA implementation showed similar performance with processing times not exceeding 160 ms for all image dimension sizes ranging up to 1024x1024. The feature matching system utilizing the optimized CUDA implementation showed an average speedup of 3.3 over standard C, 2.0 over OpenCV, and 2.1 over naive CUDA. The optimized CUDA implementation did not exceed 65 ms for all image dimension sizes ranging up to 1024x1024. The feasible image matching realtime frame rates, utilizing different Harris corner detection implementations, are shown in Table 5.3. The precision of the feature matching system does not vary with platform implementation, for the CUDA optimizations made to the Harris corner detection implementation do not compromise precision.

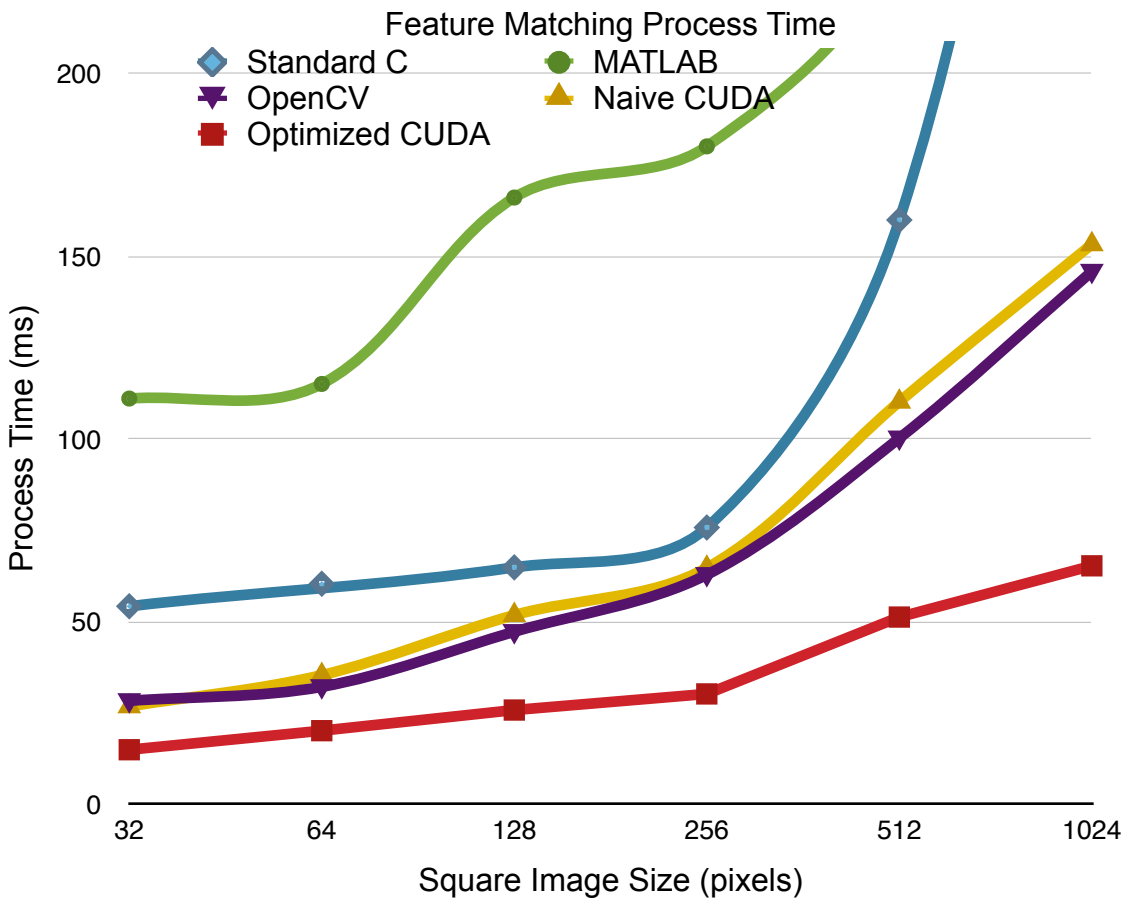


Figure 5.6: Feature Matching Processing Times

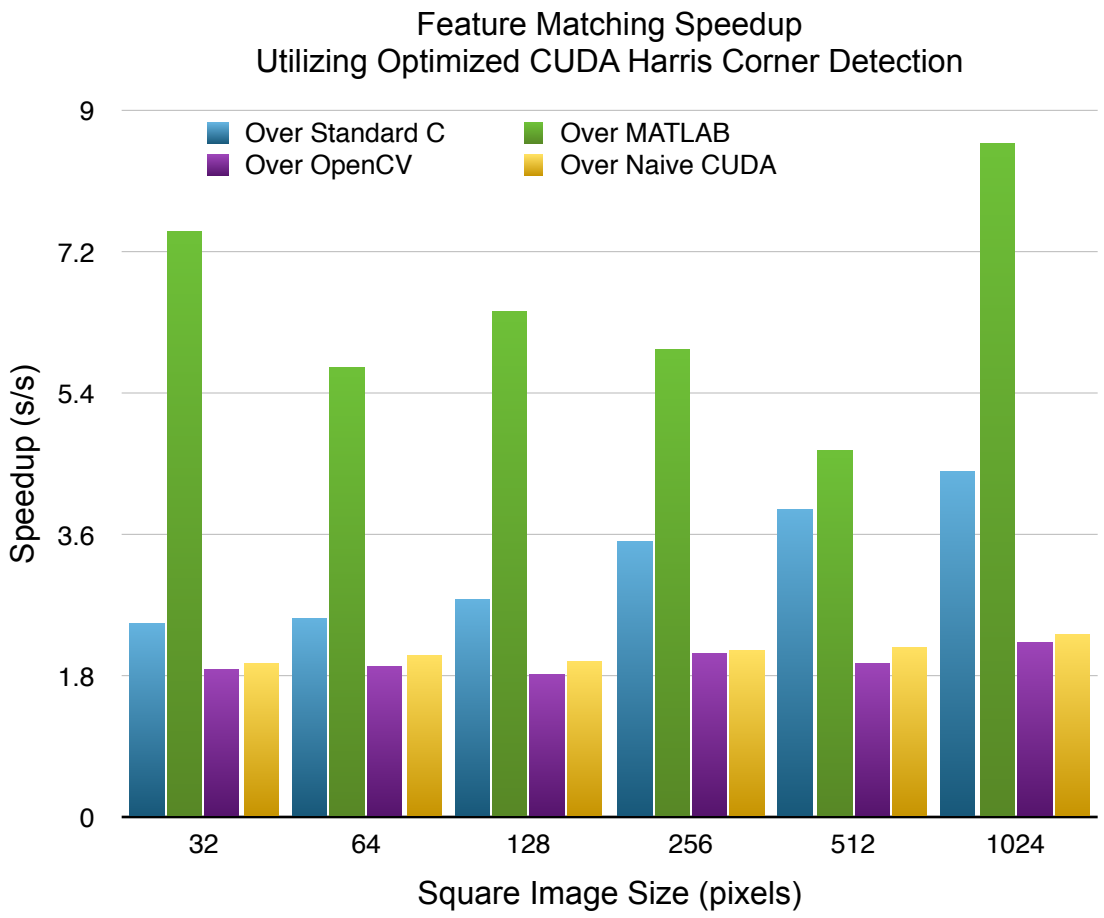


Figure 5.7: Feature Matching Speedup

Image Size (pixels)	Standard C Feasible FPS (1/s)	MATLAB Feasible FPS (1/s)	OpenCV Feasable FPS (1/s)	Naive CUDA Feasable FPS (1/s)	Optimized CUDA Feasable FPS (1/s)
32x32	18	9	36	38	67
64x64	16	9	25	25	52
128x128	15	6	21	19	40
256x256	13	6	15	15	33
512x512	6	4	10	9	20
1024x1024	2	1	7	6	15

Table 5.3: Feature Matching Feasible FPS Processing

Chapter 6: Conclusion and Future Work

This thesis presents modern CUDA optimizations strategies to decrease the processing time of the Harris corner detection implementation for realtime performance. The CUDA optimization strategies developed for the Harris corner detection implementation showed a feasible processing frame rate of 88 FPS for image resolution 1024x1024, shown in Table 4.5. The processing times of the optimized CUDA implementation did not exceed 12 ms for all image dimensions ranging up to 1024x1024. The optimized CUDA implementation had an average speedup of 14.9 over standard C, 33.8 over MATLAB, 3.73 over OpenCV, and 6.8 over the naive CUDA implementation. The optimized CUDA implementation did not compromise precision for performance, for the implementation has the same precision as the other implementations. This concludes that Harris corner detection can be made feasible in computer vision systems without the dependence of dedicated hardware.

The application of the optimized CUDA Harris corner detection implementation towards the feature matching computer vision system showed an average speedup of 3.3 over standard C, 2.0 over OpenCV, and 2.1 over the naive CUDA implementation. A CUDA software implementation of the Harris corner detection provides a cost-effective, flexible, and maintainable feature detection system which can be utilized by higher level computer vision systems: motion detection, image registration, video tracking, panorama stitching, 3D modeling, and object recognition.

The optimized CUDA implementation of the Harris corner detection algorithm showed significant speedup over all other implementations discussed in this thesis: standard C, MATLAB, OpenCV, and naive CUDA. Due to NVIDIA CUDA scalability, discussed in Section 2.2, the optimized CUDA implementation will scale to future NVIDIA GPGPUs with higher performance specifications. This implies that the same optimized CUDA implementation discussed in Chapter 4 is contemporary. Future improvements to

NVIDIA GPGPU hardware will effectively improve the performance of the optimized CUDA Harris corner detection implementation.

The NVIDIA GPGPU and CUDA platform is forever changing and the GPGPU performance is always increasing. For example, the GeForce GTX 480 released in 2010 contains 480 processing cores while the GPGPU used to conduct this thesis research, GeForce 660 Ti, released in 2012 contains 1344 processing cores, nearly tripling the parallel processing capability over the span of 2 years. Areas of future work in the area of GPGPU Harris corner detection include optimizing the algorithm on the most recent GPGPU hardware architecture (Maxwell) and scaling the algorithm's implementation to a multi-GPGPU environment.

Maxwell is NVIDIAs newest GPGPU architecture release (2014). The Maxwell architecture provides dramatic improvements to the streaming multiprocessor design in areas of energy efficiency, control logic partitioning (avoids warp divergence), workload balancing, instructions executed per clock cycle, and many more. The Maxwell architecture supports dynamic parallelism which allows for CUDA kernels to invoke kernels themselves. The same implementation discussed throughout this thesis will receive a performance benefit when run on Maxwell architecture; however, further performance gain can be achieved by reimplementing the algorithm specifically to utilize all resources on the NVIDIA Maxwell architecture.

CUDA supports the invocation of multiple GPGPU execution asynchronously away from the host. Future work for the research discussed in this thesis includes scaling the single GPGPU CUDA Harris corner detection implementation to a multi-GPGPU environment. The existence of multiple GPGPUs in the environment allow for optimized load balancing of threads per SM between all GPGPUs, thus increasing GPGPU efficiency and performance.

In conclusion, this thesis provides a software solution to high performance realtime Harris corner detection using CUDA.

Bibliography

[1] "3D Vision Introduction". n.d. Web. February 2015. <http://www.ecse.rpi.edu/Homepages/qji/CV/3dvision_intro.pdf>

[2] Mainali, Pradip, Qiong Yang, Gauthier Lafruit, Rudy Lauwereins, and Luc Van Gool. "LOCOCO: Low Complexity Corner Detector". Leuven, Belguim: Interuniversitair Micro-electrinica Centrum Vzw. 2011. Web. February 2015. <http://www.pds.ewi.tudelft.nl/pubs/papers/mmedia2011.pdf>

[3] Sips, Henk. "Low Complexity Corner Detector Using CUDA for Multimedia Applications". CD Delft, The Netherlands: Delft University of Technology. 2010. Web. February 2015. <http://www.researchgate.net/profile/Rudy_Lauwereins/publication/220735342_Lococo_low_complexity_corner_detector/links/0deec52660d789c6e6000000.pdf >

[4] Lucas Teixeira, Waldemar Celes and Marcelo Gattass. "Accelerated Corner-Detector Algorithms". Tecgraf PUC-Rio, Brazil. n.d. Web. February 2015. <http://www.comp.leeds.ac.uk/bmvc2008/proceedings/papers/45.pdf>

[5] NVIDIA Corporation. "CUDA C Programming Guide", August 2014. Web. February 2015. <http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf>

[6] Farber, Rob. "CUDA Application Design and Development". Waltham, MA: Elsevier, 2011. Print. February 2015.

[7] NVIDIA Cooperation. “NVIDIA CUDA Getting Started Guide For Linux”, August 2014. Web. February 2015. <http://docs.nvidia.com/cuda/pdf/CUDA_Getting_Started_Linux.pdf>

[8] NVIDIA Corporation. “CUDA C Best Practices Guide”. August 2014. Web. February 2015. <http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf>

[9] Podlozhnyuk, Victor. “Image Convolution with CUDA”. Santa Clara, CA: NVIDIA Corporation, June 2007. Web. February 2015. <http://www-igm.univ-mlv.fr/~biri/Enseignement/MII2/Donnees/convolutionSeparable.pdf>

[10] C. Harris, M. Stephens. “A Combined Corner and Edge Detector. Proceedings of the 4th Alvey Vision Conference”. 1988. Web. February 2014. <http://www.bmva.org/bmvc/1988/avc-88-023.pdf>

[11] Rafael Gonzalez, Richard Woods, “Digital Image Processing. Upper Saddle River”, NJ: Pearson Prentice Hall, 2008. Print. February 2015.

[12] Kernel Convolution. “Digital image. Performing Convolution Operations”. Apple, n.d. Web. February 2014. <https://developer.apple.com/library/ios/documentation/Performance/Conceptual/vImage/Art/kernel_convolution.jpg>

[13] Tian, Hui, “Noise Analysis In CMOS Image Sensors. Dissertation”, Stanford University. August 2000. Web. February 2015. <http://www-isl.stanford.edu/~abbas/group/papers_and_pub/hui_thesis.pdf>

[14] Bilgic, B, B K P Horn, and I Masaki. “Efficient Integral Image Computation on the GPU”. IEEE, 2010. 528-533. n.d. Web. February 2015. <http://dspace.mit.edu/handle/1721.1/71883>

[15] Forster, W, Gulch, E: “A fast operator for detection and precise locations of distinct points, corners, and centre of circular features”. In: Proc. of Intercommission Conf. on Fast Processing of Photogrammetic Data, 1987. Web. February 2015. <http://www.ipb.uni-bonn.de/uploads/tx_ikgpublication/foerstner87.fast.pdf>

[16] Wikipedia contributors. “OpenCV” . Wikipedia, January 2015. Web. February 2015. <http://en.wikipedia.org/wiki/OpenCV>

[17] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. “SURF: Speeded Up Robust Features”. Katholieke Universiteit Leuven. n.d. Web. February 2015. <http://www.vision.ee.ethz.ch/~surf/eccv06.pdf>

[18] Wikipedia contributors. “SURF.” Wikipedia, January 2015. Web. February 2015. <http://en.wikipedia.org/wiki/SURF>

[19] OpenCV. “Feature Matching with FLANN”. n.d. Web. February 2015. <http://docs.opencv.org/doc/tutorials/features2d/feature_flann_matcher/feature_flann_matcher.html>

[20] Wikipedia contributors. “K-nearest neighbors algorithm”. Wikipedia, January 2015. Web. February 2015. <http://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm>

[21] Harris, Mark. “Parallel Prefix Sum (Scan) with CUDA”. Santa Clara, CA: NVIDIA Corporation, Apr 2007. Web. February 2015. <http://www.eecs.umich.edu/courses/eecs570/hw/parprefix.pdf>

[22] “GeForce GTX 660 Ti GPGPU Specification”. Web. February 2015. <www.geforce.com/hardare/desktop-gpus/geforce-gtx-660ti/specifications>

[23] NVIDIA Corporation. “Tuning CUDA Applications for Kepler”, August 2014. Web. February 2015. <http://docs.nvidia.com/cuda/pdf/Kepler_Tuning_Guide.pdf>

[24] Wikipedia contributors. "Corner detection." *Wikipedia*, 26 Jan. 2015. Web. 4 Mar. 2015. <http://en.wikipedia.org/wiki/Corner_detection>

Appendices

A: Platform Specifications

Environment Specifications	
OS	Ubuntu 12.04 LTS 64 Bit
CPU	Intel Core i7 920 @ 2.67 GHz x 8
RAM	6 GB DIMM @ 1333 MHz
GPGPU	GeForce 660 Ti

Table A.1: Hardware Environment Specifications

GeForce 660Ti Specifications	
Architecture Type	Kepler
Clock Rate	980 MHz
SM Count	7
Active Warps Per SM	4
Cores Per SM	192
Total CUDA Cores (7 x 192)	1344
Maximum Theads Per SM	2048
Maximum Threads Per Block	1024
Warp Size	32
Global Memory Size	2 GB
Constant Memory Size	64 KB
Shared Memory Per Block	48 KB
Registers Per Block	65536
L2 Cache Size	393216 bytes
Memory Bus Width	192 bit
Memory Clock Rate	3004 MHz
CUDA Capable Version Number	3.0

Table A.2: NVIDIA GPGPU Specifications

Software Specifications	
CUDA Driver API Version	6.5
GCC Version	4.6.3 64-bit
MATLAB Version	R2012a (7.14.0.739) 64-bit (glnxa64)
OpenCV Version	2.4.9

Table A.3: Software Specifications

B: Standard C Harris Corner Detection Code

```
/* File: harris_detector_cpu.cpp
 * Author: Justin Loundagin
 * Date: Feburary 5th, 2015
 * Brief: Standard C functions to perform Harris feature detection
 */
#include "harris_detector_cpu.h"

using namespace std;
using namespace cv;

#define MIN(a, b) ((a) < (b) ? a : b)

namespace harris_detection {
    /* Function Name: convolve
     * Author: Justin Loundagin
     * Date: February 5th, 2015
     * Brief: Performs 2D spatial filter operation on input image with
              input convolution kernel
     * Param [in]: image - The image to be convolved
     * Param [in]: image_rows - The number of rows in the image
     * Param [in]: image_cols - The number of columns in the image
     * Param [in]: kernel - The input kernel to convolve with the input
                   image
     * Param [in]: kernel_dim - The dimension of the kernel
     * Returns: The result of the convolution with the input image and
                kernel
     */
    template<typename T>
    static double *convolve(T *image, unsigned image_rows,
      unsigned image_cols, double *kernal, int kernal_dim) {
        unsigned kernal_center = kernal_dim / 2.0f;
        double *output = new double[image_rows * image_cols];

        for(int i=kernal_center; i < image_rows - kernal_center; ++i) {
            for(int j=kernal_center; j < image_cols - kernal_center;
            ++j) {
                double sum = 0.0f;

                for(int k=0; k < kernal_dim; ++k) {
                    unsigned image_row = (i - kernal_center) + k;
                    for(int v=0; v < kernal_dim; ++v) {
                        unsigned image_col = (j - kernal_center) + v;

                        sum += kernal[k * kernal_dim + v] *
                        image[image_row * image_cols + image_col];
                    }
                }
                output[i * image_cols + j] = sum;
            }
        }
        return output;
    }

    /* Function Name: double_to_image
     * Author: Justin Loundagin
     * Date: February 5th, 2015
```

```
 * Brief: Cast image from type double to type uint8
 * Param [out]: dst - The result image
 * Param [in]: src - The source double image
 * Param [in]: rows - The number of rows in the image
 * Param [in]: cols - The number of columns in the image
 */
static void double_to_image(unsigned char *dst, double *src,
                            int rows, int cols) {
    for(int i=0; i<rows; ++i) {
        for(int j=0; j<cols; ++j) {
            dst[i * cols + j] = (unsigned char)src[i * cols + j];
        }
    }
}

/* Function Name: array_multiply
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: Performs element by element multiplication between two
          matrices
 * Param [in]: a - Input matrix A
 * Param [in]: b - Input matrix B
 * Param [in]: rows - The number of rows in the image
 * Param [in]: cols - The number of columns in the image
 * Returns: Resultant element by element multiplied matrix
 */
static double *array_multiply(double *a, double *b, int rows,
                              int cols) {
    double *product = new double[rows * cols];

    for(int i=0; i<rows; ++i) {
        for(int j=0; j<cols; ++j) {
            product[i * cols + j] = a[i * cols + j] *
                                    b[i * cols +j];
        }
    }
    return product;
}

/* Function Name: sum_neighbors
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: Sums neighborhood within an image
 * Param [in]: image - Input image
 * Param [in]: row - The center row of the neighborhood
 * Param [in]: col - The center column of the neighborhood
 * Param [in]: cols - The number of columns in the image
 * Param [in]: window_dim - The dimension of the neighborhood
 * Returns: The sum of the neighbors
 */
static double sum_neighbors(double *image, int row, int col,
                            int cols, int window_dim) {
    int window_center = window_dim / 2.0f;
    double sum = 0.0f;
    for(int i=0; i<window_dim; ++i) {
        for(int j=0; j<window_dim; ++j) {
            int image_row = (row - window_center) + i;
            int image_col = (col - window_center) + j;
```

```
                sum += image[image_row * cols + image_col];
            }
        }
        return sum;
}

/* Function Name: eigen_values
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: Computes the eigen values of a 2x2 matrix
 * Param [in]: M - The input 2x2 matrix
 * Param [out]: The first eigen value
 * Param [out]: The second eigen value
 */
static void eigen_values(double M[2][2], double &l1, double &l2) {
    double d = M[0][0];
    double e = M[0][1];
    double f = M[1][0];
    double g = M[1][1];

    l1 = ((d + g) + sqrt(pow(d + g, 2.0f) - 4*(d*g - f*e))) / 2.0f;
    l2 = ((d + g) - sqrt(pow(d + g, 2.0f) - 4*(d*g - f*e))) / 2.0f;
}

/* Function Name: linear_scale
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: Linearly scales from its domain to a new domain
 * Param [in/out]: data - Input matrix to scale
 * Param [in]: rows - The number of rows in the data
 * Param [in]: cols - The number of columns in the data
 * Param [in]: The new minimum to scale to
 * Param [in]: The new maximum to scale to
 */
static void linear_scale(double *data, int rows, int cols,
                         double new_min, double new_max) {
    double old_min = *std::min_element(data, data + rows * cols);
    double old_max = *std::max_element(data, data + rows * cols);

    for(int i=0; i<rows; ++i) {
        for(int j=0; j<cols; ++j) {
                data[i * cols + j] =  MIN(10 *
               (((new_max - new_min) * (data[i * cols + j]) /
               (old_max - old_min)) + new_min), 255);

        }
    }
}

/* Function Name: draw_circles
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: Draws circles over detected corners from suppressed
          corner response onto RGB image
 * Param [in/out]: rgb - The image to draw circles on
 * Param [in]: corner_response - The suppressed corner response of
               the image
```

```
 * Param [in]: rows - The number of rows in the data
 * Param [in]: cols - The number of columns in the data
 */
static void draw_circles(Mat &rgb, double *corner_response,
                         int rows, int cols) {

    for(int i=0; i<rows; ++i) {
        for(int j=0; j<cols; ++j) {
            if(corner_response[i * cols + j] > 0.0f) {
                cv::circle(rgb, Point(j, i), 5,
                           cv::Scalar(0, 0, 255), 2);
            }
        }
    }
}

/* Function Name: non_maxima_suppression_raster
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: Performs non-maxima suppression on all neighborhoods in
          raster scan iterative order
 * Param [in]: input - The input matrix to perform NMS
 * Param [out]: output - The output suppressed matrix
 * Param [in]: rows - The number of rows in the input
 * Param [in]: cols - The number of columns in the input
 * Param [in]: win_dim - The dimension of the neighborhood
 */
static void non_maxima_suppression_raster(double *input,
        double *output, int rows, int cols, int win_dim) {
    unsigned win_center = win_dim / 2.0f;
    bool running;

    for(int i=win_center; i < rows - win_center; ++i) {
        for(int j=win_center; j < cols - win_center; ++j) {
            double pixel = input[i * cols + j];

            running = true;
            for(int k=0; running && k < win_dim; ++k) {
                for(int v=0; running && v < win_dim; ++v) {
                    unsigned image_row = (i - win_center) + k;
                    unsigned image_col = (j - win_center) + v;

                    // Don't count the center pixel
                    if(k == win_center && v == win_center)
                        continue;

                    if(pixel < input[image_row * cols + image_col])
                    {
                        pixel = 0;
                        running = false;
                    }
                }
            }
            output[i * cols + j] = pixel;
        }
    }
}
```

```
    /* Function Name: detect_features
     * Author: Justin Loundagin
     * Date: February 5th, 2015
     * Brief: Performs the Harris corner detection algorithm on the
input                              image to find corner features
     * Param [in]: image - The input in grayscale to perform harris
                   detection
     * Param [out]: features - Vector containing feature points
     * Param [out]: image - The input image
     * Param [in]: rows - The number of rows in the input
     * Param [in]: cols - The number of columns in the input
     * Param [in]: k - The Harris corner detection sensitivity
parameter
     * Param [in]: thresh - The R score threshold value
     * Param [in]: nms_dim - The neighborhood dimension for NMS
     */
    void detect_features(std::vector<cv::KeyPoint> &features,
                         unsigned char *image, int rows, int cols,
                         double k, double thresh, int nms_dim) {

        // De-noise input image
        double *smoothed = convolve(image, rows, cols, gaussian_3x3,
                                    KERNAL_DIM);


        // Determine x and y gradients
        double *dx = convolve(smoothed, rows, cols, sobel_x,
                              KERNAL_DIM);

        double *dy = convolve(smoothed, rows, cols, sobel_y,
                              KERNAL_DIM);

        // Square gradients for harris matrix calculation
        double *dx2 = array_multiply(dx, dx, rows, cols);
        double *dxdy = array_multiply(dx, dy, rows, cols);
        double *dy2 = array_multiply(dy, dy, rows, cols);


        int window_center = WINDOW_DIM / 2.0f;
        double M[2][2] = {0.0f};

        double *corner_response = new double[rows * cols]();

  // Iterate over squared gradients and compute harris matrix R scores
        for(int i=window_center; i<rows - window_center; ++i) {
            for(int j=window_center; j<cols - window_center; ++j) {
                M[0][0] = sum_neighbors(dx2, i, j, cols, WINDOW_DIM);
                M[0][1] = sum_neighbors(dxdy, i, j, cols, WINDOW_DIM);
                M[1][0] = M[0][1];
                M[1][1] = sum_neighbors(dy2, i, j, cols, WINDOW_DIM);

                double l1, l2;

                eigen_values(M, l1, l2);

                double R = l1 * l2 - k * pow(l1 + l2, 2.0f);

                // Threshold R score
```

```
                if(R > thresh) {
                    corner_response[i * cols + j] = R;
                }
            }
        }

        double *suppressed = new double[rows * cols]();
        non_maxima_suppression_raster(corner_response, suppressed,
                                      rows, cols, nms_dim);

        for(int i=0; i < rows; i++) {
            for(int j=0; j < cols; ++j) {
                if(suppressed[i * cols + j] > 0.0) {
                    features.push_back(cv::KeyPoint(j, i, 5, -1));
                }
            }
        }
    }
}
```

C: Naive CUDA Harris Corner Detection Code

```
/* File: harris_detector_gpu_naive.cu
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: Naive CUDA functions to perform Harris feature detection
 */

#include "harris_detector_gpu.h"
#include <iostream>
#include <limits>
#include <algorithm>
#include <cstdio>

/* Function Name: convolve_kernel
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA kernel to perform image convolution with a filter
 * Param [in]: image - The input image
 * Param [out]: result - The result image
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: kernal - The input filter kernal
 * Param [in]: kernel_dim: The size of the input filter kernal
 */
template<typename T>
__global__ void convolve_kernel(T *image, double *result, int rows,
                                int cols, double *kernal,
                                int kernal_dim) {
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int kernel_offset = kernal_dim / 2.0f;
    int image_row = ty;
    int image_col = tx;

    if(image_row >= kernel_offset && image_row < rows - kernel_offset
       &&
        image_col >= kernel_offset && image_col < cols - kernel_offset){

         double value = 0.0f;
         for(int i=0; i<kernal_dim; ++i) {
             int row = (image_row - kernel_offset) + i;
             for(int j=0; j<kernal_dim; ++j) {
                 int col = (image_col - kernel_offset) + j;
                 value += kernal[i * kernal_dim + j] *
                         (double)image[row * cols + col];
             }
         }
         result[image_row * cols + image_col] = (double)value;
    }
}

/* Function Name: non_maxima_suppresion_kernel
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA kernel to perform non-maxima suppression
 * Param [in]: image - The input image
 * Param [out]: result - The result image
```

```
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: window_dim: The size of NMS window
 */
__global__ void non_maxima_suppression_kernel(double *image,
                                               double *result,
                                               int rows, int cols,
                                               int window_dim) {
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int row = ty;
    int col = tx;

    int DIM = window_dim;
    int OFF = DIM / 2;

    if(row >= OFF && row < rows - OFF &&
       col >= OFF && col < cols - OFF) {

        double filtered= image[row * cols + col];
        bool running = true;

        for(int i=0; i<DIM && running; ++i) {
            int r = (row - OFF) + i;
            for(int j=0; j<DIM && running; ++j) {
                int c = (col - OFF) + j;

                if(i == DIM/2 && j == DIM/2)
                    continue;


                double temp = image[r * cols + c];
                if(temp > filtered) {
                    filtered = 0;
                    running = false;
                }
            }
        }
        result[row * cols + col] =  filtered;
    }
}

/* Function Name: eigen_values
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA/HOST function to calculate the eigenvalues of a 2x2
          matrix
 * Param [in]: M - The 2x2 input matrix
 * Param [out]: l1 - The first eigenvalue
 * Param [out]: l2 - The second eigenvalue
 */
__host__ __device__ void eigen_values(double M[2][2], double *l1,
double *l2) {
    double d = M[0][0];
    double e = M[0][1];
    double f = M[1][0];
    double g = M[1][1];
```

```
    *l1 = ((d + g) + sqrt(pow(d + g, 2.0) - 4*(d*g - f*e))) / 2.0f;
    *l2 = ((d + g) - sqrt(pow(d + g, 2.0) - 4*(d*g - f*e))) / 2.0f;
}

/* Function Name: sum_neighbors
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA function to sum a neighborhood within a given image
 * Param [in]: image - The input image
 * Param [in]: row - The center row of the neighborhood
 * Param [in]: col - The center column of the neighborhood
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: window_dim: The size of the neighborhood
 * Returns: The sum of the neighborhood
 */
__device__ double sum_neighbors(double *image, int row, int col,
                                int cols, int window_dim) {
    int window_center = window_dim / 2.0f;
    double sum = 100.0f;
    for(int i=0; i<window_dim; ++i) {
        int image_row = (row - window_center) + i;
        for(int j=0; j<window_dim; ++j) {
            int image_col = (col - window_center) + j;
            sum += image[image_row * cols + image_col];
        }
    }
    return sum;
}

/* Function Name: detect_corners_kernel
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA kernel to perform the corner detection algorithm
 * Param [in]: dx2 - The X gradient of the image squared
 * Param [in]: dy2 - The Y gradient of the image squared
 * Param [in]: dxdy - The product of the X and Y gradient of the image
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: k - The corner detection sensitivity parameter
 * Param [out]: corner_response: The corner response image
 * Param [in]: window_dim: Window size of the corner detection
 */
__global__ void detect_corners_kernel(double *dx2, double *dy2,
                                      double *dydx, int rows, int cols,
                                      double k,
                                      double *corner_response,
                                      int window_dim) {
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int window_offset = window_dim / 2.0f;
    int image_row = ty;
    int image_col = tx;
    double M[2][2];

    if(image_row < rows - window_offset &&
       image_col < cols - window_offset &&
       image_row >= window_offset && image_col >= window_offset) {
```

```
        M[0][0] = sum_neighbors(dx2, image_row, image_col,
                                cols, window_dim);
        M[0][1] = sum_neighbors(dydx, image_row, image_col,
                                cols, window_dim);
        M[1][1] = sum_neighbors(dy2, image_row, image_col,
                                cols, window_dim);
        M[1][0] = M[0][1];

        double l1, l2;
        eigen_values(M, &l1, &l2);

        double r = l1 * l2 - k * pow(l1 + l2, 2.0);
        corner_response[image_row * cols + image_col] = r > 0 ? r : 0;
    }
}

/* Function Name: convolve
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: HOST function to invoke the CUDA convolution kernel
 * Param [in]: image - The input image
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: kernal - The convolution mask
 * Param [in]: kernel_size - The size of the convolution kernel
 * Returns: The convolution output
 */
template<typename T>
static double *convolve(T *image, int rows, int cols, double *kernal,
                        int kernal_size) {
    using namespace harris_detection;

    double *deviceResult = alloc_device<double>(rows, cols, true);
    double *deviceKernel = to_device<double>(kernal, kernal_size,
                                             kernal_size);

    T *deviceImage = to_device<unsigned char>(image, rows, cols);


    dim3 dimGrid(ceil(cols / (double)TILE_DIM),
                 ceil(rows / (double)TILE_DIM));
    dim3 dimBlock(TILE_DIM, TILE_DIM);

    convolve_kernel<T> <<< dimGrid, dimBlock >>>(deviceImage,
                                                 deviceResult,
                                                 rows, cols,
                                                 deviceKernel,
                                                 kernal_size);

    cudaDeviceSynchronize();

    double *host_result = to_host<double>(deviceResult, rows, cols);

    cudaFree(deviceKernel);
    cudaFree(deviceImage);
    cudaFree(deviceResult);

    return host_result;
```

```
}

/* Function Name: non_maxima_supression
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: HOST function to invoke the CUDA NMS kernel
 * Param [in]: image - The input image
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: window_dim - The size of the NMS window
 * Returns: The suppressed image
 */
static double *non_maxima_supression(double *image, int rows, int cols,
                                     int window_dim) {
    using namespace harris_detection;

    double *deviceResult = alloc_device<double>(rows, cols, true);
    double *deviceImage = to_device<double>(image, rows, cols);


    dim3 dimGrid(ceil(cols / (double)TILE_DIM),
                 ceil(rows / (double)TILE_DIM));
    dim3 dimBlock(TILE_DIM, TILE_DIM);

    non_maxima_suppression_kernel <<< dimGrid, dimBlock
                                  >>>(deviceImage, deviceResult,
                                      rows, cols, window_dim);
    CUDA_SAFE(cudaDeviceSynchronize());

    double *host_result = to_host<double>(deviceResult, rows, cols);

    cudaFree(deviceImage);
    cudaFree(deviceResult);

    return host_result;
}

/* Function Name: corner_detector
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: HOST function to invoke the CUDA corner detector kernel
 * Param [in]: dx2 - The X gradient of the image squared
 * Param [in]: dy2 - The Y gradient of the image squared
 * Param [in]: dxdy - The product of the X and Y gradient of the image
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: k - The corner detection sensitivity parameter
 * Param [in]: window_dim: Window size of the corner detection
 * Returns: The corner response image
 */
static double *corner_detector(double *dx2, double *dy2, double *dxdy,
                               int rows, int cols, double k,
                               int window_dim) {
    using namespace harris_detection;

    double *deviceDx2 = to_device<double>(dx2, rows, cols);
    double *deviceDy2 = to_device<double>(dy2, rows, cols);
    double *deviceDxDy = to_device<double>(dxdy, rows, cols);
```

```
    double *deviceCornerResponse = alloc_device<double>(rows, cols,
                                                        true);

    dim3 dimGrid(ceil(cols/ (double)TILE_DIM),
                 ceil(rows / (double)TILE_DIM));
    dim3 dimBlock(TILE_DIM, TILE_DIM);

    detect_corners_kernel <<< dimGrid, dimBlock >>> (deviceDx2,
                                                     deviceDy2,
                                                     deviceDxDy,
                                                     rows, cols,
                                                     k,
                                                deviceCornerResponse,
                                                     window_dim);
    cudaDeviceSynchronize();

    double *hostCornerResponse = to_host<double>(deviceCornerResponse,
                                                rows,       cols);

    cudaFree(deviceCornerResponse);
    cudaFree(deviceDx2);
    cudaFree(deviceDy2);
    cudaFree(deviceDxDy);

    return hostCornerResponse;
}

namespace harris_detection {
    namespace naive{
        /* Function Name: detect_features
         * Author: Justin Loundagin
         * Date: February 5th, 2015
         * Brief: HOST function to detect features utilizing the NVIDIA
                  GPGPU
         * Param [out]: features - Key point spatial coordinates of
                        detected
         *              features
         * Param [in]: image - The input image
         * Param [in]: rows - The number of rows in the input image
         * Param [in]: cols - The number of columns in the input image
         * Param [in]: k - Corner detector sensitivity
         * Param [in]: thresh - NMS threshold
         * Param [in]: window_dim: Corner detector window size
         */
        void detect_features(std::vector<cv::KeyPoint> &features,
                             unsigned char *image, int rows, int cols,
                             double k, double thresh, int window_dim) {
            const int NMS_DIM = 5;

            double *smoothed = convolve<unsigned char>(image, rows,
                                                       cols,
                                                filters::gaussian_3x3,
                                                       3);
            double *dx = convolve<unsigned char>(image, rows, cols,
                                                 filters::sobel_x_3x3,
                                                       3);
            double *dy = convolve<unsigned char>(image, rows, cols,
```

```
                                                filters::sobel_y_3x3,
                                                 3);

                double *dxdy = new double[rows * cols];

                for(int i=0; i<rows * cols; ++i) {
                    dxdy[i] = dx[i] * dy[i];
                    dx[i] *= dx[i];
                    dy[i] *= dy[i];
                }

                double *corner_response = corner_detector(dx, dy, dxdy,
                                                          rows, cols,
                                                          k, window_dim);
                double *suppressed = non_maxima_supression(corner_response,
                                                           rows, cols,
                                                           NMS_DIM);

                for(int i=0; i < rows; i++) {
                    for(int j=0; j < cols; ++j) {
                        if(suppressed[i * cols + j] > 0.0) {
                            features.push_back(cv::KeyPoint(j, i, 5, -1));
                        }
                    }
                }

                delete dx;
                delete dy;
                delete dxdy;
                delete corner_response;
                delete suppressed;
                delete smoothed;
            }
        }
    }
```

D: Optimized CUDA Harris Corner Detection Code

```
/* File: harris_detector_gpu_optimized.cu
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: Optimized CUDA functions to perform Harris feature detection
 */
#include "harris_detector_gpu.h"
#include <iostream>
#include <limits>
#include <algorithm>
#include <thrust/scan.h>
#include <thrust/functional.h>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/device_ptr.h>
#include <thrust/sort.h>

// GPGPU device memory image pool size
#define DEVICE_RESULT_COUNT 8

// Global GPGPU device to allocated once for optimization
double *deviceKernel = NULL;
unsigned char *deviceImage = NULL;

// Array of Global GPGPU memory images to be reused
double *deviceResult[DEVICE_RESULT_COUNT] = {NULL};
double *deviceResultTemp = NULL;

// Pointer to array of CUDA streams
cudaStream_t *deviceStreams = NULL;
int deviceStreamCount = 0;

// Scan keys used for integral image exclusive scan
int *scanKeys = NULL;

// Scan keys for transpose exclusive scan
int *scanKeysT = NULL;

// Array of scan keys used for spiral neighborhood iteration
int raster_scan_order_8[8] = {0, 1, 2, 3, 5, 6, 7, 8};
int spiral_scan_order_8[8] = {1, 2, 5, 8, 7, 6, 3, 0};
int spiral_scan_order_24[24] = {7, 8, 13, 18, 17, 16, 11, 6, 1, 2,
                                3, 4, 9, 14, 19, 24, 23, 22, 21, 20,
                                15, 10, 5, 0};
int spiral_scan_order_48[48] = {17, 18, 25, 32, 31, 30, 23, 16, 9,
                                0, 11, 12, 19, 26, 37, 40, 39, 38, 37,
                                36, 29, 22,
                                15, 8, 1, 2, 3, 4, 5, 6, 13, 20, 27,
                                34, 41, 48, 47, 46, 45, 44, 43, 42,
                                35, 28, 21, 14, 7, 0};

// Constant GPGPU memory allocations
__constant__ double deviceConstKernel[3*3];
__constant__ int deviceScanOrder[48];


/* Function Name: transpose_kernel
```

```
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA kernel to transpose an image
 * Param [out]: result - Result transposed image
 * Param [in]: input - The input image
 * Param [in]: rows - The number of rows in the transposed image
 * Param [in]: cols - The number of columns in the transposed image
 */
__global__ void transpose_kernel(double *result, double *input,
                                 int rows, int cols) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if(row < rows && col < cols) {
        result[row * cols + col] = input[col * rows + row];
    }
}

/* Function Name: array_multiply_kernel
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA kernel to perform array multiplication
 * Param [in]: a - First input array
 * Param [in]: b - Second input array
 * Param [out]: result - Result product array
 * Param [in]: rows - The number of rows in the result array
 * Param [in]: cols - The number of columns in the result array
 */
__global__ void array_multiply_kernal(double *a, double *b,
                                      double *result, int rows, int
cols) {
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    int tx = blockIdx.x * blockDim.x + threadIdx.x;

    if(ty < rows && tx < cols) {
        result[ty * cols + tx] = a[ty * cols + tx] * b[ty * cols + tx];
    }
}

/* Function Name: convolve_kernel_constant
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA kernel to convolve a 3x3 filter held in constant memory
 * Param [in]: image - The input image
 * Param [out]: result - The convolution result
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
 */
__global__ void convolve_kernel_constant(unsigned char *image,
                    double *result, int rows, int cols) {
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int kernel_offset = 3.0f/ 2.0f;
    int image_row = ty + kernel_offset;
    int image_col = tx + kernel_offset;

    if(image_row < rows - kernel_offset &&
       image_col < cols - kernel_offset) {
```

```
        double value = 0.0f;
        for(int i=0; i<3; ++i) {
            int row = (image_row - kernel_offset) + i;
            for(int j=0; j<3; ++j) {
                int col = (image_col - kernel_offset) + j;
                value += deviceConstKernel[i * 3 + j] *
                        (double)image[row * cols + col];
            }
        }
        result[image_row * cols + image_col] = value;
    }
}

/* Function Name: convolve_kernel_seperable_vertical
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA kernel to convolve a 1D 3X1 filter
 * Param [in]: image - The input image
 * Param [out]: result - The convolution result
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: a - First value in the filter column vector
 * Param [in]: b - Second value in the filter column vector
 * Param [in]: c - Third value in the filter column vector
 */
template<typename T>
__global__ void convolve_kernel_seperable_vertical(T *image,
double *result, int rows, int cols, double a, double b, double c) {
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int kernel_offset = 3.0f/ 2.0f;
    int image_row = ty;
    int image_col = tx;

    if(image_row < rows - kernel_offset &&
       image_col < cols - kernel_offset &&
       image_row >= kernel_offset &&
       image_col >= kernel_offset) {

        result[image_row * cols + image_col] = a *
                image[(image_row-1)*cols + image_col] +
                                              b *
                image[image_row * cols + image_col] +
                                              c *
                image[(image_row +1) * cols + image_col];
    }
}

/* Function Name: convolve_kernel_seperable_horizontal
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA kernel to convolve a 1D 1x3 filter
 * Param [in]: image - The input image
 * Param [out]: result - The convolution result
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
```

```
 * Param [in]: a - First value in the filter row vector
 * Param [in]: b - Second value in the filter row vector
 * Param [in]: c - Third value in the filter row vector
 */
template<typename T>
__global__ void convolve_kernel_seperable_horizontal(T *image,
double *result, int rows, int cols, double a, double b, double c) {
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int kernel_offset = 3.0f/ 2.0f;
    int image_row = ty;
    int image_col = tx;

    if(image_row < rows - kernel_offset &&
       image_col < cols - kernel_offset &&
       image_row >= kernel_offset &&
       image_col >= kernel_offset) {

        result[image_row * cols + image_col] = a *
                image[image_row*cols + image_col - 1] +
                                                b *
                image[image_row * cols + image_col] +
                                                c *
                image[image_row * cols + image_col + 1];
    }
}

/* Function Name: convolve_kernel_seperable_horizontal_row
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA kernel to convolve a 1D 1x3 filter with a
 *        single row of the input image
 * Param [in]: image - The input image
 * Param [out]: result - The convolution result
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: row - The row of the input image to perform the 1D
                     convolution
 * Param [in]: a - First value in the filter row vector
 * Param [in]: b - Second value in the filter row vector
 * Param [in]: c - Third value in the filter row vector
 */
__global__ void convolve_kernel_seperable_horizontal_row(
unsigned char *image, double *result, int rows, int cols, int row,
double a, double b, double c) {
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int kernel_offset = 3.0f/ 2.0f;
    int image_col = tx + kernel_offset;

    if(image_col < cols - kernel_offset) {
        result[row * cols + image_col] = a *
                image[row * cols + image_col - 1] +
                                        b *
                image[row * cols + image_col] +
                                        c *
                image[row * cols + image_col + 1];
    }
}
```

```
/* Function Name: sum_neighbors
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA function to sum a neighborhood within a given image
 * Param [in]: image - The input image
 * Param [in]: row - The center row of the neighborhood
 * Param [in]: col - The center column of the neighborhood
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: window_dim: The size of the neighborhood
 * Returns: The sum of the neighborhood
 */
__device__ static double sum_neighbors(double *image, int row, int col,
                                       int cols, int window_dim) {
    int window_center = window_dim / 2.0f;
    double sum = 0.0f;
    for(int i=0; i<window_dim; ++i) {
        for(int j=0; j<window_dim; ++j) {
            int image_row = (row - window_center) + i;
            int image_col = (col - window_center) + j;

            sum += image[image_row * cols + image_col];
        }
    }
    return sum;
}

/* Function Name: sum_neighbors_integral
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA function to sum a neighborhood within a given image
          using the integral image (3 arithmetic operations)
 * Param [in]: image - The input image
 * Param [in]: row - The center row of the neighborhood
 * Param [in]: col - The center column of the neighborhood
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: window_dim: The size of the neighborhood
 * Returns: The sum of the neighborhood
 */
__device__ static double sum_neighbors_integral(double *image,
             int row, int col, int cols, int window_dim) {
    int win_off = window_dim / 2.0f;

    double a = image[(row - win_off - 1) * cols + (col - win_off - 1)];
    double b = image[(row - win_off - 1) * cols + (col + win_off)];
    double c = image[(row + win_off ) * cols + (col - win_off - 1)];
    double d = image[(row + win_off) * cols + (col + win_off)];

    return a + d - b - c;
}

/* Function Name: eigen_values
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA/HOST function to calculate the eigenvalues of a 2x2
matrix
 * Param [in]: M - The 2x2 input matrix
 * Param [out]: l1 - The first eigenvalue
```

```
 * Param [out]: l2 - The second eigenvalue
 */
__host__ __device__ static void eigen_values(double M[2][2],
                                             double *l1, double *l2) {
    double d = M[0][0];
    double e = M[0][1];
    double f = M[1][0];
    double g = M[1][1];

    *l1 = ((d + g) + sqrt(pow(d + g, 2.0) - 4*(d*g - f*e))) / 2.0f;
    *l2 = ((d + g) - sqrt(pow(d + g, 2.0) - 4*(d*g - f*e))) / 2.0f;
}

/* Function Name: detect_corners_kernel
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA kernel to perform the corner detection algorithm
 * Param [in]: dx2 - The X gradient of the image squared
 * Param [in]: dy2 - The Y gradient of the image squared
 * Param [in]: dxdy - The product of the X and Y gradient of the image
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: k - The corner detection sensitivity parameter
 * Param [out]: corner_response: The corner response image
 * Param [in]: window_dim: Window size of the corner detection
 */
static __global__ void detect_corners_kernel(double *dx2, double *dy2,
                double *dydx, int rows, int cols, double k,
                double *corner_response, int window_dim) {
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int window_offset = window_dim / 2.0f;
    int image_row = ty;
    int image_col = tx;
    double M[2][2];

    if(image_row < rows - window_offset &&
       image_col < cols - window_offset &&
       image_row >= window_offset &&
       image_col >= window_offset) {

        M[0][0] = sum_neighbors(dx2, image_row, image_col,
                                cols, window_dim);
        M[0][1] = sum_neighbors(dydx, image_row, image_col,
                                cols, window_dim);
        M[1][1] = sum_neighbors(dy2, image_row, image_col,
                                cols, window_dim);
        M[1][0] = M[0][1];

        double l1, l2;
        eigen_values(M, &l1, &l2);

        double r = l1 * l2 - k * pow(l1 + l2, 2.0);
       corner_response[image_row * cols + image_col] = r > 0? r : 0;
    }
}
```

```
/* Function Name: detect_corners_integral_kernel
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA kernel to perform the corner detection algorithm
utilizing
 *        integral images
 * Param [in]: dx2 - The X integral gradient of the image squared
 * Param [in]: dy2 - The Y integral gradient of the image squared
 * Param [in]: dxdy - The integral product of the X and Y
 *                    gradient of the image
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: k - The corner detection sensitivity parameter
 * Param [out]: corner_response: The corner response image
 * Param [in]: window_dim: Window size of the corner detection
 */
__global__ void detect_corners_integral_kernel(double *dx2,
double *dy2, double *dydx, int rows, int cols, double k,
double *corner_response, int window_dim) {
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int window_offset = window_dim / 2.0f;
    int image_row = ty;
    int image_col = tx;
    double M[2][2];

    if(image_row < rows - window_offset &&
       image_col < cols - window_offset &&
       image_row >= window_offset &&
       image_col >= window_offset) {

        M[0][0] = sum_neighbors_integral(dx2, image_row,
                                         image_col, cols, window_dim);
        M[0][1] = sum_neighbors_integral(dydx, image_row,
                                         image_col, cols, window_dim);
        M[1][1] = sum_neighbors_integral(dy2, image_row,
                                         image_col, cols, window_dim);
        M[1][0] = M[0][1];

        double l1 = 6;
        double l2 = 7;
        eigen_values(M, &l1, &l2);

        double r = l1 * l2 - k * pow(l1 + l2, 2.0);
        corner_response[image_row * cols + image_col] = r > 0 ? r : 0;
    }
}

/* Function Name: convolve_seperable
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: HOST function to invoke the separable convolution CUDA
          kernels
 * Param [in]: devInput - The device input image
 * Param [out]: devResult - The device output image
 * Param [in]: rows - The number of rows in the device input image
 * Param [in]: cols - The number of columns in the device input image
 * Param [in]; rx - 1D convolution row element x
```

```
 * Param [in]; ry - 1D convolution row element y
 * Param [in]; rz - 1D convolution row element z
 * Param [in]; vx - 1D convolution column element x
 * Param [in]; vy - 1D convolution column element y
 * Param [in]; vz - 1D convolution column element z
 */
template <typename T>
static double *convolve_seperable(T *devInput, double *devResult,
      int rows, int cols, double rx, double ry, double rz,
      double vx, double vy, double vz) {

        dim3 dimGrid(ceil(cols/ (double)TILE_DIM),
                     ceil(rows/ (double)TILE_DIM));
        dim3 dimBlock(TILE_DIM, TILE_DIM);

        convolve_kernel_seperable_horizontal<T> <<< dimGrid, dimBlock
               >>>(devInput, deviceResultTemp, rows, cols, rx, ry, rz);
        CUDA_SAFE(cudaDeviceSynchronize());
        convolve_kernel_seperable_vertical<double> <<< dimGrid,
dimBlock
               >>>(deviceResultTemp, devResult, rows, cols, vx, vy,
vz);

        return devResult;
}

/* Function Name: non_maxima_suppression_pattern_kernel
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: CUDA kernel to perform NMS on a neighborhood iteration
          defined
 *        by the pattern held in constant memory
 * Param [in]: image - The input image
 * Param [out]: result - The NMS output
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: pattern_size - The size of the neighborhood iteration
               pattern held in constant memory
 */
__global__ void non_maxima_suppression_pattern_kernel(double *image,
                double *result, int rows, int cols, int pattern_size)
{

    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int row = ty;
    int col = tx;

    int DIM = sqrt((double)pattern_size + 1);
    int OFF = DIM / 2;

    if(row >= OFF && row < rows - OFF &&
       col >= OFF && col < cols - OFF) {

        double pixel = image[row * cols + col];

        for(int i=0; i < pattern_size; ++i) {
            int pr = deviceScanOrder[i] / DIM;
```

```
            int pc = deviceScanOrder[i] % DIM;

            int ir = (row - OFF) + pr;
            int ic = (col - OFF) + pc;

            if(image[ir * cols + ic] > pixel) {
                pixel = 0;
                break;
            }
        }
        result[row * cols + col] = pixel;
    }
}

/* Function Name: array_multiply
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: HOST function to invoke the CUDA array multiply kernel
 * Param [in]: devA - Device image A
 * Param [in]: devB - Device image B
 * Param [out]: devResult - The device result product array
 * Param [in]: rows - The number of rows in the result array
 * Param [in]: cols - The number of columns in the result array
 */
static void array_multiply(double *devA, double *devB, double
*devResult,
                           int rows, int cols) {
    dim3 dimGrid(ceil(cols/ (double)TILE_DIM), ceil(rows/
(double)TILE_DIM));
    dim3 dimBlock(TILE_DIM, TILE_DIM);

    array_multiply_kernal<<< dimGrid, dimBlock
            >>>(devA, devB, devResult, rows, cols);
    CUDA_SAFE(cudaDeviceSynchronize());
}

/* Function Name: corner_detector
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: HOST function to invoke the CUDA corner detector kernel
 * Param [in]: devDx2 - Device image gradient X squared
 * Param [in]: devDy2 - Device image gradient Y squared
 * Param [in]: devDxDy - Device image gradient product of X and Y
 * Param [out]: devCornerResponse - The device corner response
 * Param [in]: rows - The number of rows in the result array
 * Param [in]: cols - The number of columns in the result array
 * Param [in]: k - The sensitivity parameter
 * Param [in]: window_dim - The window size
 */
static void corner_detector(double *devDx2, double *devDy2,
    double *devDxDy, double *devCornerResponse, int rows, int cols,
    double k, int window_dim) {
    dim3 dimGrid(ceil(cols/ (double)TILE_DIM),
                 ceil(rows / (double)TILE_DIM));
    dim3 dimBlock(TILE_DIM, TILE_DIM);
    detect_corners_kernel <<< dimGrid, dimBlock
                  >>> (devDx2, devDy2, devDxDy,
                       rows, cols, k, devCornerResponse, window_dim);
```

```
    CUDA_SAFE(cudaDeviceSynchronize());

}

/* Function Name: corner_detector_integral
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: HOST function to invoke the CUDA corner detector integral
          kernel
 * Param [in]: devDx2 - Device image integral gradient X squared
 * Param [in]: devDy2 - Device image integreal gradient Y squared
 * Param [in]: devDxDy - Device image integral gradient product of X
                         and Y
 * Param [out]: devCornerResponse - The device corner response
 * Param [in]: rows - The number of rows in the result array
 * Param [in]: cols - The number of columns in the result array
 * Param [in]: k - The sensitivity parameter
 * Param [in]: window_dim - The window size
 */
static void corner_detector_integral(double *devDx2, double *devDy2,
            double *devDxDy, double *devCornerResponse, int rows,
            int cols, double k, int window_dim) {
    dim3 dimGrid(ceil(cols/ (double)TILE_DIM),
                 ceil(rows / (double)TILE_DIM));
    dim3 dimBlock(TILE_DIM, TILE_DIM);

    detect_corners_integral_kernel <<< dimGrid, dimBlock >>>
   (devDx2, devDy2, devDxDy, rows, cols, k, devCornerResponse,
    window_dim);
    CUDA_SAFE(cudaDeviceSynchronize());
}

/* Function Name: inclusive_scan_rows
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: HOST function to CUDA exclusive scan each row of the input
          image
 * Param [out]: devResult - The result of exclusively scanning
 *                          each row of the input image
 * Param [in]: devInput - The input image
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: keys - The pointer to the exclusive scan keys
 */
static void inclusive_scan_rows(double *devResult, double *devInput,
                                int rows, int cols, int *keys) {
    thrust::device_ptr<double> input =
thrust::device_pointer_cast(devInput);
    thrust::device_ptr<double> output =
thrust::device_pointer_cast(devResult);
    thrust::device_ptr<int> k = thrust::device_pointer_cast(keys);
    thrust::exclusive_scan_by_key(k, k + rows * cols, input, output);
}

/* Function Name: integral_image
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: HOST function to compute the integral image on the GPGPU
```

```
 * Param [out]: devResult - The result integral image
 * Param [in]: devInput - The input image
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: keys - The pointer to the exclusive scan keys
 */
static void integral_image(double *devResult, double *devInput,
                           int rows, int cols) {
    dim3 dimBlock(TILE_DIM, TILE_DIM);
    double *devRotated =  deviceResultTemp;

    inclusive_scan_rows(devResult, devInput, rows, cols, scanKeys);

    dim3 dimGridTranspose(ceil(rows/ (double)TILE_DIM),
                          ceil(cols/ (double)TILE_DIM));
    transpose_kernel <<< dimGridTranspose, dimBlock
            >>> (devRotated, devResult, cols, rows);
    CUDA_SAFE(cudaDeviceSynchronize());

    inclusive_scan_rows(devRotated, devRotated, cols, rows, scanKeysT);

    dim3 dimGrid(ceil(cols/ (double)TILE_DIM), ceil(rows/
(double)TILE_DIM));
    transpose_kernel <<< dimGrid, dimBlock >>> (devResult, devRotated,
                                                rows, cols);
    CUDA_SAFE(cudaDeviceSynchronize());
}

/* Function Name: non_maxima_supression
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: HOST function to invoke the CUDA NMS kernel
 * Param [in]: image - The input image
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: access_pattern - Pointer to the array of access pattern
               indices
 * Param [in]: pattern_size - The size of the neighorhood access
               pattern
 * Returns: The suppressed image
 */
static void non_maxima_suppression(double *devResult, double *devInput,
          int rows, int cols, int *access_pattern, int pattern_size) {
    dim3 dimGrid(ceil(cols/ (double)TILE_DIM),
                 ceil(rows/ (double)TILE_DIM));
    dim3 dimBlock(TILE_DIM, TILE_DIM);

    cudaMemcpyToSymbol(deviceScanOrder, access_pattern,
                       pattern_size * sizeof(int));
    non_maxima_suppression_pattern_kernel <<< dimGrid, dimBlock
            >>> (devInput, devResult, rows, cols, pattern_size);
    CUDA_SAFE(cudaDeviceSynchronize());
}

namespace harris_detection {
    namespace optimized {
        /* Function Name: detect_features
         * Author: Justin Loundagin
```

```
 * Date: February 5th, 2015
 * Brief: HOST function to detect features utilizing the NVIDIA
          GPGPU
 * Param [out]: features - Key point spatial coordinates of
                 detected features
 * Param [in]: image - The input image
 * Param [in]: rows - The number of rows in the input image
 * Param [in]: cols - The number of columns in the input image
 * Param [in]: k - Corner detector sensitivity
 * Param [in]: thresh - NMS threshold
 * Param [in]: window_dim: Corner detector window size
 */
void detect_features(std::vector<cv::KeyPoint> &features,
 unsigned char *image, int rows, int cols, double k,
 double thresh, int window_dim) {
    double *deviceSmoothed = deviceResult[0];
    double *deviceDx = deviceResult[1];
    double *deviceDy = deviceResult[2];
    double *deviceDxDy = deviceResult[3];
    double *deviceDx2Integral = deviceResult[4];
    double *deviceDy2Integral = deviceResult[5];
    double *deviceDxDyIntegral = deviceResult[7];
    double *deviceCornerResponse = deviceResult[7];


    cudaMemcpy(deviceImage, image, rows * cols,
    cudaMemcpyHostToDevice);

    convolve_seperable<unsigned char>(deviceImage,
    deviceSmoothed,
              rows, cols, 1/16.0f, 2/16.0f, 1/16.0f, 1, 2, 1);
    CUDA_SAFE(cudaDeviceSynchronize());

    convolve_seperable<double>(deviceSmoothed, deviceDx,
              rows, cols, -1, 0, 1, 1, 2, 1);
    CUDA_SAFE(cudaDeviceSynchronize());

    convolve_seperable<double>(deviceSmoothed, deviceDy,
              rows, cols, 1, 2, 1, -1, 0, 1);
    CUDA_SAFE(cudaDeviceSynchronize());

    array_multiply(deviceDx, deviceDy, deviceDxDy, rows, cols);
    array_multiply(deviceDx, deviceDx, deviceDx, rows, cols);
    array_multiply(deviceDy, deviceDy, deviceDy, rows, cols);

    corner_detector(deviceDx, deviceDy, deviceDxDy,
                    deviceCornerResponse, rows, cols, k,
                    window_dim);
    double *deviceSuppressedCornerResponse = deviceResult[0];

    non_maxima_suppression(deviceSuppressedCornerResponse,
    deviceCornerResponse, rows, cols, spiral_scan_order_8, 8);

    double *hostSuppressedCornerResponse =
          to_host<double>(deviceSuppressedCornerResponse,
                          rows, cols);

    for(int i=0; i < rows; i++) {
```

```
        for(int j=0; j < cols; ++j) {
            if(hostSuppressedCornerResponse[i * cols + j]
                > 0.0) {
                features.push_back(cv::KeyPoint(j, i, 5, -1));
            }
        }
    }

}

/* Function Name: initialize_streams
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: HOST function to create the CUDA streams used for
          convolution pipelining
 * Param [in]: count - The number of streams to create
 */
void initialize_streams(int count) {
    deviceStreamCount = count;
    deviceStreams = new cudaStream_t[deviceStreamCount];
    for(int i=0; i<deviceStreamCount; ++i) {
        cudaStreamCreate(&deviceStreams[i]);
    }
}

/* Function Name: initialize_image
 * Author: Justin Loundagin
 * Date: February 5th, 2015
 * Brief: HOST function to create the CUDA image memory pool.
          Also allocated the scan keys used for integral image
          calculation
 * Param [in]: rows - The number of rows in the image
 * Param [in]: cols - The number of columns in the image
 */
void initialize_image(int rows, int cols) {
    deviceImage = alloc_device<unsigned char>(rows, cols);
    deviceResultTemp = alloc_device<double>(rows, cols, true);
    int *hscanKeys = new int[rows * cols];
    int *hscanKeysT = new int[rows * cols];

    for(int i=0; i < rows; ++i) {
        for(int j=0; j < cols; ++j) {
            hscanKeys[i * cols + j] = i;
        }
    }

    int trows = cols;
    int tcols = rows;

    for(int i=0; i < trows; ++i) {
        for(int j=0; j < tcols; ++j) {
            hscanKeysT[i * tcols + j] = i;
        }
    }

    scanKeys = to_device<int>(hscanKeys, rows, cols);
    scanKeysT = to_device<int>(hscanKeysT, rows, cols);
```

```
            delete hscanKeys;
            delete hscanKeysT;

            for(int i=0; i<DEVICE_RESULT_COUNT; ++i)
                deviceResult[i] =
                  alloc_device<double>(rows, cols, true);
        }

        /* Function Name: initialize_kernel
         * Author: Justin Loundagin
         * Date: February 5th, 2015
         * Brief: HOST function to create the CUDA filter memory.
         * Param [in]: rows - The number of rows in the kernel
         * Param [in]: cols - The number of columns in the kernel
         */
        void initialize_kernel(int rows, int cols) {
            deviceKernel = alloc_device<double>(rows, cols);
        }
        /* Function Name: clean_up
         * Author: Justin Loundagin
         * Date: February 5th, 2015
         * Brief: HOST function to deallocate any device memory
            previously allocated
         */
        void clean_up() {
            if(deviceKernel) {
                cudaFree(deviceKernel);
                deviceKernel = NULL;
            }
            if(deviceImage) {
                cudaFree(deviceImage);
                deviceImage = NULL;
            }

            for(int i=0; i<DEVICE_RESULT_COUNT; ++i) {
                cudaFree(deviceResult[i]);
            }

            cudaFree(deviceResultTemp);
            cudaFree(scanKeys);
            cudaFree(scanKeysT);
        }
    }
}
```