2016

Dynamically parallel CAMSHIFT: GPU accelerated object tracking in digital video

Matthew J. Perry
Eastern Washington University

Dynamically Parallel CAMSHIFT:

GPU accelerated object tracking in digital video

A Thesis

Presented To

Eastern Washington University

Cheney, Washington

In Partial Fulfillment of the Requirements

for the Degree

Masters of Science in Computer Science

By

Matthew J. Perry

Summer 2016

THESIS OF MATTHEW JAMES PERRY APPROVED BY

_____ DATE _____
YUN TIAN, PHD. GRADUATE STUDY COMMITEE CHAIR

_____ DATE _____
CAROL TAYLOR, PHD. GRADUATE STUDY COMMITTEE MEMBER

_____ DATE _____
ESTEBAN RODRIGUEZ-MAREK, M.S. GRADUATE STUDY COMMITTEE MEMBER

MASTER'S THESIS

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Eastern Washington University, I agree that the JFK Library shall make copies freely available for inspection. I further agree that copying of this project in whole or in part is allowable only for scholarly purposes. It is understood, however, that any copying or publication of this thesis for commercial purposes, or for financial gain, shall not be allowed without my written permission.

Signature _____

Date _____

Abstract

The CAMSHIFT algorithm is widely used for tracking dynamically sized and positioned objects in real-time applications. In spite of its extensive study on the platform of sequential CPU, its research on massively parallel Graphical Processing Unit (GPU) platform is quite limited. In this work, we designed and implemented two different parallel algorithms for CAMSHIFT using CUDA. The first design performs calculations on the GPU, but requires iterative data transfers back to the host CPU for condition checking, which bottlenecks the entire program. In the second design, we propose an enhanced parallel reduction-based CAMSHIFT using dynamic parallelism to reduce overhead of data transfers between the CPU and GPU. Test results for a 400 by 400 search window show that the second design is up to five times faster than the first design and nine times faster than a pure CPU implementation. We also investigate the deployment of dynamic parallelism for multiple object tracking using CAMSHIFT.

Acknowledgement

A special thank you to Yun Tian. His guidance and knowledge about digital image processing and the CUDA programming model were indispensable to this project.

Contents

1 Introduction

"Continuously Adjusting MeanShift" (CAMSHIFT) is a well-established computer vision algorithm for tracking objects in digital video. It is based on the MeanShift algorithm, iteratively moving the center of its search window towards the peak density of a probability distribution image. The probability distribution used in CAMSHIFT is made from the hue color model profile of its tracking target. Unlike MeanShift, CAMSHIFT can adjust its search window size and orientation to follow the dynamic probability distributions of moving targets between video frames. This is possible based on the statistical moments under the search window within a probability distribution image. CAMSHIFT can contribute as a layer in a diverse spectrum of applications, for example: estimating the distance a tennis player covers throughout an entire match, controlling the rotation of surveillance cameras towards a probable target, or providing an interactive user-interface for a video game. However, CAMSHIFT, in its most basic form, will fail to reliably track an object under certain conditions within a video sequence. There has been much research contributing different potential solutions to make the algorithm more robust under these conditions. These efforts proposed ways to both improve the accuracy of the basic algorithm and add extended features to help recover lost targets. Many of the applications where CAMSHIFT can contribute demand real-time performance. Improving the performance of the algorithm's basic computation becomes increasingly important as more features are added to improve its tracking accuracy and reliability. Some past research has looked into general purpose programming on the graphics processing unit (GPGPU) to parallelize some of the CAMSHIFT computational workload. Parallel reduction is a commonly applied GPGPU algorithm to accelerate the computation of the statistical moments under the search window. Past GPGPU implementations were bounded by the iterative nature of the CAMSHIFT algorithm to rely on the CPU for processing between parallel reductions.

This paper explores newer GPGPU techniques to improve the runtime performance of the basic CAMSHIFT algorithm. Two designs of CAMSHIFT using the CUDA programming model were developed to test the findings in this study. The main contribution of this paper to the CAMSHIFT algorithm is an optimized parallel reduction routine with dynamic parallelism, a newer feature provided by the CUDA runtime, to help minimize CPU bound processing. The optimization of video frame preprocessing with CUDA and an extended use of dynamic parallelism for multiple object

tracking were also implemented and discussed in this paper. The goal of this project was to develop a fast enough basic tracking routine where more robust features may be added later and still achieve a runtime well under real-time performances demands. An equivalent sequential CPU version written in C++, and OpenCV for video frame preprocessing, was developed for comparable test results.

The paper is organized in the following manner. Section 2 describes the important concepts and components of digital video processing and the CUDA programming model that will be used throughout the rest of the discussion. Section 3 describes the origins of the CAMSHIFT algorithm. Section 4 examines past research that has gone into extending the CAMSHIFT algorithm to improve its performance and other related works. Section 5 explains the methodology carried out in implementing the three designs of CAMSHIFT developed in this project. Section 6 describes how the two versions were tested and compared. Section 7 reflects on the results found in section 6 and the tracking performance under different test conditions. Section 8 looks at the necessary improvements made apparent from the examination in section 7 and the future direction of the project. The paper ends with concluding remarks in section 9.

2 Problem Space

2.1 Image Processing Concepts

2.1.1 Digital Color Models

A digital color model describes how color can be represented as a mathematical abstraction that computers can display, reproduce, and persist.

The "Red, Green and Blue" (RGB) color model is an additive approach to representing digital color. Its typical use is in displayable digital imagery on monitor screens. The intensities of the three RGB color values combine to produce a distinct digital color. The fullest intensity of each color produces the color white, while zero intensity in all three color values produces black. An equal intensity value for all three colors will produce a gray color. Any other combination of intensity values produces a hue, a pure color without any white or black [3]. The saturation of the color is based on the difference between the minimum and maximum intensities [4]. Figure 2.1 represents how the gamut of producible colors can be visualized as a RGB cube. Each primary color is a dimension of the color cube and its diagonal transitioning from the influence of black to white.

However, it is not always intuitive to produce expected colors by directly manipulating the color intensities of this color model without a visual reference.

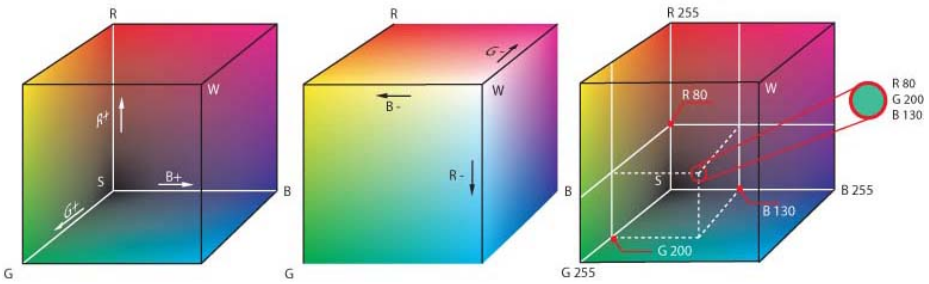


Figure 2.1: RGB color model cube [5]

The "Hue, Saturation, Value" (HSV) color model was invented by Alvy Ray Smith in 1978 to express digital color more intuitively, "mimicking the way an artist mixes paints on his palette" [7]. The HSV model has the shape of a hexacone with a polar coordinate system, as shown in Figure 2.2. The model separates the pure hue from the influence of white and black mixed into the color. The top axis represents hue as a degree around the color wheel. The horizontal axis corresponds to saturation, the degree of white present in the color space. Saturation measures the intensity of a color represented as a percentage ranging from 0% to 100%. No saturation results in a pure white and full saturation represents the pure color. Value represents the brightness of a color defined by the percentage of black mixed in. Its degree decreases down the vertical axis towards the tip of the hexacone is zero value is pure black.

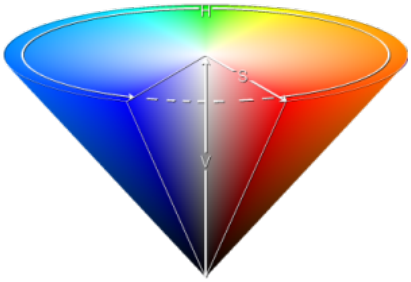


Figure 2.2: HSV Hexacone [6]

The hue of an object's appearance is especially useful for object tracking. It is less prone to changes in illumination than saturation and value. Its single valued representation is also simpler to process than using three values in the RGB model.

2.1.2 Digital Image Model

A digital image model represents the translation of a physical light signal captured by a camera into a digital color model stored on a computer. A digital image is typically stored in a frame buffer in a computer's memory. A frame buffer is a two-dimensional array storing the pixel values of a

given digital color model as a tuple.

2.1.3 Digital Video Model

A digital video model is a file format combining a series of digital images. Rendering each image in the series to the display monitor at a constant sample rate gives the appearance of animation. Typically, sampling at a rate of 25 to 30 frames per second represents an accurate "real-time" depiction of the captured events. The *.mov* video file format was used for the input and output video files in this project. It is a video container for the mpeg-4 audio and video file compression format. The OpenCV library was used to read the input video file into memory and write an output video file out to memory after processing.

2.1.4 OpenCV

OpenCV is an open-source computer vision API allowing the application programmer to read, write, and edit digital images and video. OpenCV stores RGB video in reverse "Blue, Green, Red" (BGR) order as image *Mat* objects. Each primary color intensity is represented by an 8-bit value ranging from 0 to 255. The *Mat* object exposes certain features about the image to the application programmer. This includes the number of columns and rows in the matrix, as well as many methods for image manipulation, e.g. copying images or converting their color models. The *Mat* coordinate system starts in the top-left corner with the x-coordinate along the horizontal axis and the y-coordinate along the vertical axis, as shown in Figure 2.3. Each *Mat* index is accessible through the method *at(x,y)*. However, faster access is possible through the underlying *data* array storing its pixel tuples. The *data* array is one-dimensional and stores each value within a pixel tuple in separate contiguous indexes as *unsigned char* primitives. One benefit of using the slower *at(x,y)* method is that it abstracts away the maintenance of offsetting the starting indexes by the size of the tuples.

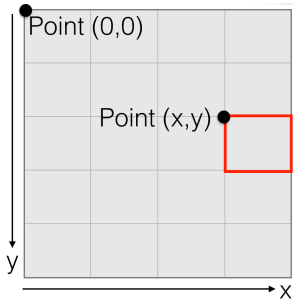


Figure 2.3: OpenCV Mat coordinate system

The *cvtColor* function converts the color model of a *Mat* another color model. The *cvtColor* conversion from BGR to HSV uses the equations found in Figure 2.4.

$$V = max(B, G, R) \quad S = \begin{cases} \frac{V - min(B,G,R)}{V} & \text{if } V \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad H = \begin{cases} \frac{240 + 60(R-G)}{V - min(B,G,R))} & \text{if } V = B \\ \frac{120 + 60(B-R)}{(V - min(B,G,R))} & \text{if } V = G \\ \frac{60(G-B)}{(V - min(B,G,R))} & \text{if } V = R \end{cases}$$

Figure 2.4: Value, Saturation, Hue computations [8]

The full 360 degrees of hue cannot be represented by the 8-bit storage used in OpenCV. The hue values are halved to fit this representation with a range of 0 to 179. Saturation and value are normalized in OpenCV to ranges of 0 to 255 to fit in this representation as well. The OpenCV API includes many digital processing algorithms, as well, including a CAMSHIFT implementation. The OpenCV CAMSHIFT uses an elliptically shaped tracking window. It was, therefore, not a direct comparison with the rectangular search window described in the methodology of this project.

2.1.5 Image Histogram

A histogram is an estimation of the probability distribution of numerical data within a sample set. As Figure 2.5 shows, histograms are visually represented as bar graphs. The horizontal axis represents the range of possible values within the sample set, and the vertical axis represents the frequency within a range found in the sample set. The bins are equally sized, non-overlapping, and consecutive value ranges. Wider ranges of bin values "smooth" the frequency of the distribution, providing a rougher estimate.

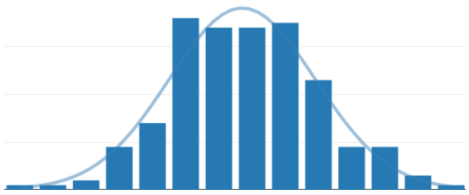


Figure 2.5: Histogram as a bar graph

In a computer program, a histogram is stored as an array of memory with each index representing an individual bin. To sequentially construct a histogram, each input value comprising the distribution is examined and a bin value is incremented each time its index is hashed by the input value divided by the bin width. After all of the input data is processed, each bin value is divided by the input size. The final bin result is a percentage of how many input values fell under the bin range in the initial distribution. The percentage is represented by a floating-point value between 0 and 1. The total of all the bins always equals 1. Hashing future distribution values in the histogram serves as a look-up of the probability that the values belong to the initial distribution.

2.1.6 Histogram Back-Projection

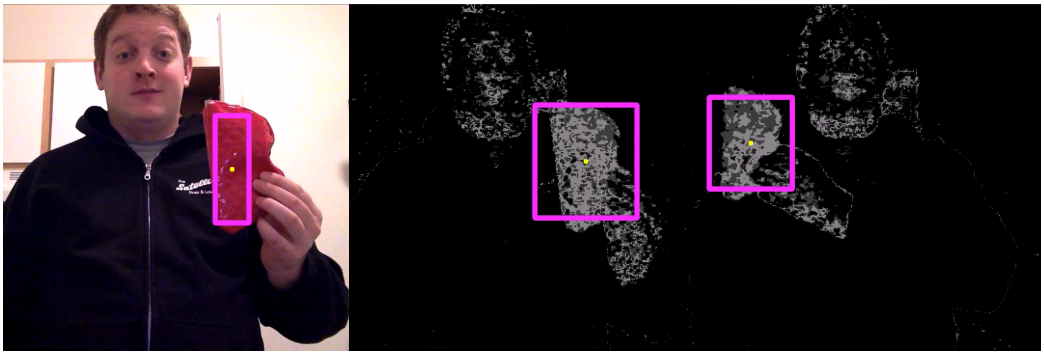


Figure 2.6: Grayscale Normalized Representation of Histogram Back Projection

Histogram back-projection is the process of replacing the pixel values of a digital image with the corresponding bin values of a histogram. The back-projected image represents the probability distribution found within the image. Figure 2.6 gives a normalized visual perspective of what a histogram back-projections looks like in the grayscale color model. Normalization of the back-projection consists of multiplying each value by 255 to bring it into the grayscale color model [9]. The probability distribution image contains peaks of probability densities. Figure 2.7 shows a top-down three-dimensional representation of the modes of probability densities found in the histogram back-projected video sequence also shown in Figure 2.6.

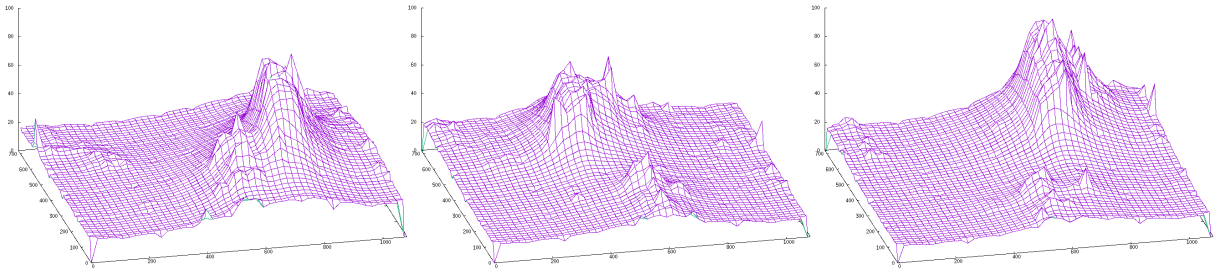


Figure 2.7: 3D Mesh Graphs of the Modes of Probability Distribution Images

2.2 Graphics Processing Unit

The Graphical Processing Unit (GPU) is a specialized computer processor. It was originally designed to offload processing and manipulating frame buffer memory in real-time 3-D graphical digital images. The Arithmetic Logical Units (ALU) have slower clock cycles than those found on a typical general-purpose CPU. What the GPU loses in per ALU clock cycle it makes up for with its massively parallel design. Its number of processing cores outnumber those found on a typical

general-purpose CPU and enough brute force of concurrent ALU processing to outperform the CPU for certain kinds of processing tasks. The possible uses of GPU hardware expanded greatly with the advent of floating-point support and programmable shaders. An entire family of algorithms whose input data can be broken up into separate parts can now take advantage of the general purpose parallel computation on the GPU.

2.3 Maxwell Architecture

The server used in this project had a commodity level motherboard equipped with four Nvidia GeForce GTX 970 graphic cards. The 900 series of Nvidia graphics cards are designed with the Maxwell architecture. The ability for the massively parallel computation of Nvidia graphics cards lies in the streaming multiprocessor (SM). Each SM receives only one instruction at a time and runs in parallel with the other SMs. The Maxwell SM has a quadrant-based design comprised of four independent 32-core processing blocks. Each processing block has its own scheduler capable of dispatching two instructions per clock cycle and its own local memory. Each core can execute a sequential thread in parallel with other cores.[10] A GeForce GTX 970 graphics card is equipped with thirteen SMs, totaling 1664 cores.

2.4 CUDA Programming Model

CUDA is a GPGPU platform and API created by Nvidia for writing parallel programs that run natively only on Nvidia graphics cards. Resources located in the memory of the CPU and GPU are labeled with the keywords *host* and *device*, respectively. CUDA applications run parallel code on the GPU from a host function by launching a *global* kernel. A kernel is a special parallel function whose instructions are written from the perspective of a single thread's sequential execution. The *global* keyword labels a function that may be called from either another host or device function. The CUDA programming model use a Same Instruction Multiple Threads (SIMT) parallel pattern, the sequential instructions of a kernel are executed in parallel by multiple threads. The launch of a global kernel must be configured explicitly by the application programmer to inform the GPU of the dimensionality of parallelism. The kernel is organized as a grid of thread blocks. Grids and blocks can have up to three-dimensions. All threads of the same block have the same block index within the grid. Each thread within a block has its own index.[2] Within a one-dimensional grid and thread block to identify its order within the grid based on the following calculations:

1. threadID = threadIdx.x

2. absoluteThreadID = blockIdx.x * blockDim.x + threadIdx.x

The *threadID* is the number assigned to the thread within its thread block. The *absoluteThreadID* is the index within the entire grid based on the block size, the block index in the grid, and its *threadID* within a particular block. The GTX 970 can have a maximum block size of 1024 threads. It is a recommended best practice to write kernel code that may be executed independently by each thread block for two reasons. First, the order of block execution is not knowable before runtime. Second, there is no synchronization natively supported between different blocks [12]. Even within a block, the order of thread execution is not guaranteed before runtime. Fortunately, the threads within a block can be explicitly synchronized with the *__syncthreads()* API function which serves as a thread barrier. All threads reaching the barrier stall their further execution until all other threads within the block reach the barrier before continuing. Threads within a block are further organized into sub-groups called warps. A warp is comprised of 32 threads and serves as the unit of execution for the scheduler. A single instruction is executed simultaneously by all threads within a warp. Warps cannot be split between different thread blocks. However, any logical divergence between threads of a warp results in sequential execution of the divergent threads. This is an automatic resolution policy handled by the Nvidia hardware. Memory reads are simultaneously executed by half-warp groupings of 16 threads. An entire warp can read from memory in one clock cycle because there can be two memory reads per clock cycle.

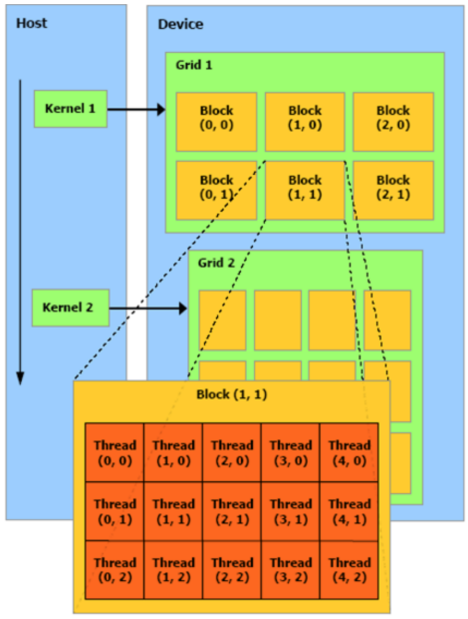


Figure 2.8: CUDA Kernel Grid [11]

2.5 CUDA Memory Types

Whereas the typical CPU is equipped with many registers and cache layers for hiding the memory latency of reading data from disk, the CUDA programming model tends to hide such latency through massive parallelism [12]. However, there are a variety of device memory types available to the CUDA application programmer, each with its own best use case and performance considerations. Utilizing the correct memory types for the appropriate situation can help the performance of a CUDA application.

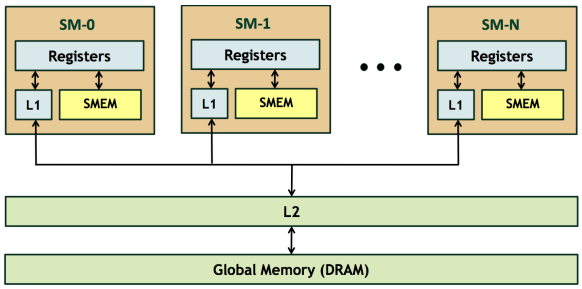
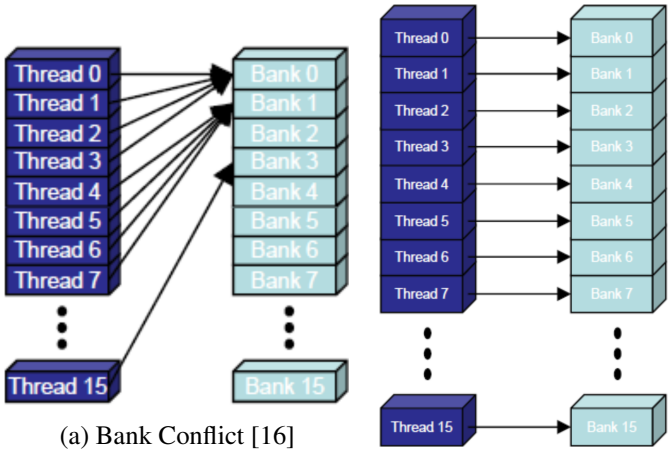


Figure 2.9: Device Memory Layout [15]

2.5.1 32-bit Registers

32-bit registers are the fastest and most abundant type of device memory physically located in a SM. A single register can store a floating-point precision operand. Pairs of registers are capable of storing double-precision 64-bit arithmetic operands. The hardware is optimized for 32-bit floating-point precision without the overhead of maintaining register pairs to store double-precision operands. All data primitives and pointers stored in the other device memory types must first be read into registers before threads can operate on them. The maximum number of registers used per thread can be adjusted by setting a compilation flag. However, when all registers are fully occupied, memory will spill over to cache layers with slower access. The GTX 970 card has 256 kb of registry memory per SM and 65,536 total registers available per block.

2.5.2 Shared Memory



(a) Bank Conflict [16]

(b) Linear Addressing [16]

Figure 2.10: Shared Memory Access Patterns [16]

Shared memory is another fast memory type physically located in a SM. It is used for the storage and exchange of data between the threads within a thread block. Shared memory cannot be exchanged between different thread blocks. It can be statically allocated within a kernel function as multiple arrays. However, the application programmer must know the appropriate allocation size before runtime. Dynamic allocation is possible, but comes with the limitation of allowing only one shared memory pointer per block. There are 49,152 bytes of shared memory available per thread block and 96 kilobytes total per SM. The shared memory access is broken into 32-bit sections called *words*.

Concurrent access to shared memory is made possible by arranging *words* into memory banks. The Maxwell architecture organizes shared memory into 32 banks of *words*. Successive *words* are assigned to successive banks, wrapping around to the first bank after every 32nd *word*. The bandwidth of each bank is one *word* per clock cycle. Performance is fastest when each warp makes only one memory request from each bank at a time [17]. The same value from the same bank is broadcasted without conflict when read by an entire warp [16]. Similarly, when some threads of a warp read the same *word* from the same bank, the value is multicasted to those threads without conflict [16]. On the other hand, when different *words* are requested from within the same bank by different threads within a warp, a bank conflict occurs. Bank conflicts are resolved by hardware through an enforced sequential series of memory requests. Avoiding bank conflicts in application

code improves run-time performance. Different shared memory access patterns can be used to ensure that bank conflicts do not occur. The simplest conflict free access pattern is linear addressing, as shown in figure 2.10b. Warp threads access consecutive indexes of shared memory in linear addressing conflict free, because there are 32 threads in a warp and 32 banks of shared memory. There is no inter-block shared memory conflicts. When one block stalls from bank conflicts, another can take its place for execution until the memory is accessed.

2.5.3 Texture Memory

Texture memory is a read-only memory type cached on chip. There is only 48 kilobytes of texture cache in a Maxwell SM and has been combined with the L1 cache. It was originally designed for traditional graphics applications, but is frequently used in GPGPU applications as well. Texture cache is efficient for neighboring threads to access memory indexes with close spatial locality within two-dimensional arrays [18]. The disadvantage of texture memory is it has read-only restrictions. Updating texture memory requires slow memory copy to the entire texture.

2.5.4 Constant Memory

Constant memory is another form of read-only memory. It resides off-chip, but each SM contains a small cache optimized for constant memory reads. Reading from constant cache is as fast as reading from a register for a half-warp, if only the same memory address is requested. Otherwise, the different requests are issued sequentially to the different threads in the half-warp. [21]

2.5.5 Global Memory

Global memory gets its name from its scope within an application. It allows the exchange of data between *device* and *host*. It is also useful for persistent storage throughout the lifetime of the application [20]. There is over four gigabytes of global memory per GTX 970 card. Global memory resides off-chip in device DRAM and is about 100 times slower than shared memory [17].

There are two ways to declare global memory. It can be declared statically with global scope using the CUDA *__device__* keyword or allocated dynamically from the host using the *cudaMalloc()* API call and assigned to a regular host pointer variable. However, *host* code cannot directly access the data pointed to via a *host* pointer reference. The global *device* memory must be transferred back to *host* allocated memory. This memory transfer is quite slow, depending on the size of memory,

and should be limited as much as possible within a CUDA application.

2.5.6 Occupancy

The use of registers, shared memory, and different types of cache located physically in the SM can affect occupancy. Occupancy is the number of active warps that a SM can keep ready for execution in its cores. Dividing the actual number of warps per SM by the maximum number of warps a SM can hold provides the occupancy number of an application. An application tends to perform faster with a higher occupancy number because more active warps execute more work in parallel and the scheduler has less overhead of swapping in and out different warps. [19]

2.5.7 Local Memory and Register Spilling

Another consideration about the overuse of memory types located in the SM is memory spilling. [15] When a SM runs out memory resources, the L2 cache layer is used to store data outside of the SM. The memory is coined "local" because it is stored in the privately accessible registers of an individual thread. The older Nvidia Fermi architecture spilled memory into the L1 cache layer and only used the L2 cache if evicted from the L1 cache. L2 cache is located outside of the SM. In many cases, it is better to recompute a value than store it in a local variable, because this helps avoid memory spilling and takes advantage of the highly parallelized CUDA programming model.

2.6 Reduction Kernels

A reduction is a simple parallel algorithm used to compute the summation of a large linear array. It takes a divide and conquer approach. The tree pattern of this division of labor is shown in 2.11. The complete reduction in CUDA uses a two-phase recursive kernel:

1. The input array is divided into subsections across multiple thread blocks

2. A first-phase kernel reduces each subsection into an intermediate total by the different thread blocks

3. A second-phase kernel reduces the intermediate totals into a final total summation of the entire input array

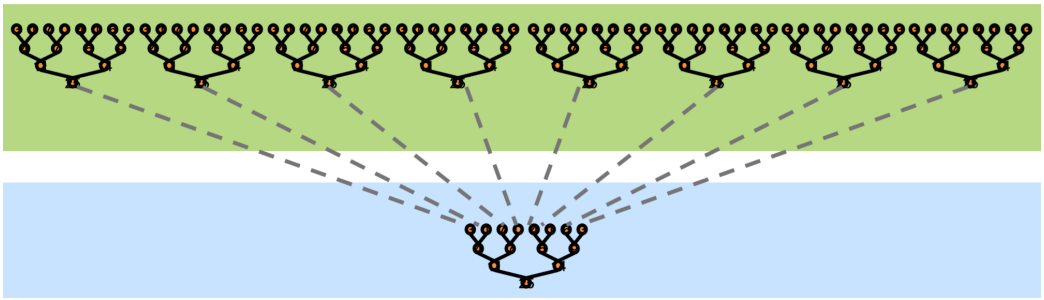


Figure 2.11: Two Kernel Reduction Overview [29]

A simple reduction of a single input array can use the same reduction kernel recursively so long as the block size is a power of two. The global input memory is first read into the shared memory of each thread block. Each thread reads in a corresponding input value from the global memory index matching the thread's *absoluteTheadID* and loads the input value to the shared memory index matching its *threadID*, the thread indexes discussed in section 2.4. If an *absoluteTheadID* is larger than the length of the entire input array, then its corresponding shared memory index is padded with a value of zero. The reduction proceeds in stages of combining higher shared memory index values into the lower indexes until the total is consolidated within the first index of the array. The final result within a block is written back to another global memory array the size of the number of blocks at the index of the block ID. The two most common reduction patterns, interleaved and sequential addressing, are shown in Figure 2.12.

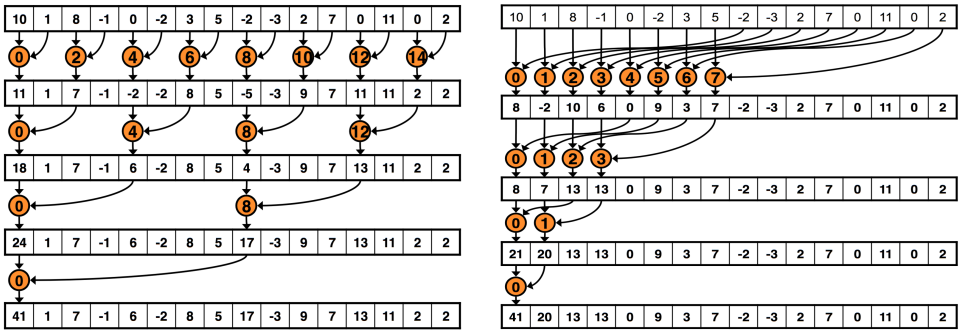


Figure 2.12: Interleaved vs Sequential Block Reduction [29]

Interleaving addressing is slower than a sequential addressing pattern [29]. One reason it can be slower is the use of the modulus operator to calculate which threads are the lower index of the stride. This creates highly divergent warps which get sequential executed, and the use of modulus operator itself is relatively slow. Optimizing the interleaved reduction pattern with a strided index avoids

divergent warp branching. However, this causes a new problem of shared memory bank conflicts within the thread warps.

Sequential addressing is a conflict-free access pattern for shared memory, similar to linear addressing access discussed in section 2.5.2. There are *log(N)* stages per block of the reduction for *N* defined as the block's input size rounded to its nearest higher power of two. There are ways to further optimize a sequential reduction kernel. Using an equal number of threads as the number of global memory indexes to read the input into shared memory is wasteful. Half of the threads are only used for the memory loading and end their execution after this step. If the block size equals a power of two, then the block size can be halved. Each thread would add two global input indexes strided by the initial difference in the reduction stages and load the result into one index in shared memory. The initial stage of the reduction can then be skipped and the remaining threads have at least another round of the reduction. [29] The threads with an *absoluteTheadID* greater than the input size only need to load the lower input value. This mimicks an add-on-load with a padded value of zero. Another optimization is unrolling the for-loop controlling the reduction once the stride decreases to the size of a warp [29]. This eliminates the need for thread synchronization during the final steps by the last warp, because the instructions are executed simultaneously within the warp.

3 Background

3.1 MeanShift Algorithm

The CAMSHIFT algorithm is based on the MeanShift algorithm. The MeanShift algorithm is used to find the mode or peak in a static probability distribution. It iteratively climbs the gradient of a probability distribution until it reaches a gradient of zero, indicating the zenith of the distribution. It is a non-parametric technique similar to kernel density estimations, however it operates on discrete, rather than continuous, distributions. The algorithm proceeds in the following manner:

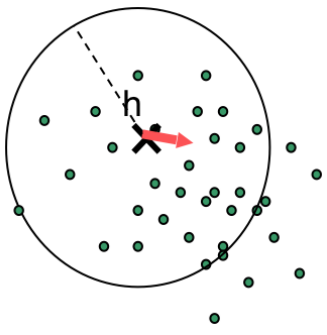


Figure 3.1: Mean Shift Search Window

1. An initial search window size and location is chosen

2. Compute the mean location within the search window

3. The search window is then re-centered at the mean location

Step 2 is defined by computing a mean shift vector from the center of the window. The mean shift vector is a directed average distance calculated based on the average of all vectors connecting each point in the distribution to the center of the search window. Steps 2 and 3 are iteratively repeated until the difference between current and previous search window centroids is equal to or less than a predefined minimum threshold. The mode of a distribution is reached upon convergence [1].

3.1.1 Limitations of MeanShift

The MeanShift algorithm fails as an object tracking algorithm for video sequences, because it was designed for static probability distributions. The mean shift vector does not reveal any information about distributions with changing sample sizes. However, objects in a video sequence can change sizes between video frames. The distribution of pixels representing an object will change with its size between video frames. Tracking an object with a fixed search window size can fail in different situations. If the search window is too large compared to the object's pixel distribution, then extraneous background pixels will be introduced into the distribution. Noise is introduced by extraneous background pixels that hash into higher probability bin values of the object's color histogram. If the search window is too small, then the algorithm might steer towards a false local peak of the probability distribution because the entire distribution isn't considered in the computation. Furthermore, too small of a search window will not accurately cover a larger object, producing inaccurate results.

3.2 CAMSHIFT Algorithm

The CAMSHIFT algorithm was developed by Gary Bradski in 1998 as an extension of the MeanShift algorithm [22]. Unlike MeanShift, CAMSHIFT continuously adjusts its search window size to track the changing sizes of objects across video frames. The basic form of the CAMSHIFT algorithm proceeds in the following manner:

1. An initial search window size and location is chosen

2. The pixels under the search window are back-projected by a histogram of the object's hue values

3. The statistical moments of the back-projected values under the search window are calculated
4. The new centroid of the search window is based on the zeroth and 1st-order statistical moments
5. The size and orientation of the search window are optionally calculated based on the zeroth and 2nd-order statistical moments
6. The center of the search window is moved to the new centroid and its orientation is optionally adjusted
7. The distance between previous and new centroids are compared for convergence

Steps 2 through 7 are repeated until convergence is reached. Convergence is defined as a distance between centroids under a pre-defined threshold. The histogram is pre-defined and saved for repeated use. The hue component is used to create the histogram because it is less susceptible to the effects of lighting changes than a combination of RGB values which include the saturation level [22]. The zeroth moment is defined as:

$$M_{00} = \sum_{x} \sum_{y} P(I(x, y)) \tag{3.1}$$

where:

I : hue value of the video frame at a particular x and y-coordinate

P : histogram bin value using the hue as a hash key value

Bradski described the zeroth-order moment of the distribution as its "area" irrespective of any directionality found under the search window [22]. The first-order moments for the x and y-coordinates are defined as:

$$M_{10} = \sum_{x} \sum_{y} xP(I(x, y)); \quad M_{01} = \sum_{x} \sum_{y} yP(I(x, y)); \tag{3.2}$$

The first-order moments are the "area" with respect to the position along the coordinate axises. The x and y-coordinates of the new centroid for the search window are derived from the following

equations:

$$x_c = \frac{M_{10}}{M_{00}}; \quad y_c = \frac{M_{01}}{M_{00}}; \tag{3.3}$$

The 2nd-order moments for the x and y-coordinates are defined as:

$$M_{11} = \sum_x \sum_y xyP(I(x,y)); \quad M_{20} = \sum_x \sum_y x^2 P(I(x,y)); \quad M_{02} = \sum_x \sum_y y^2 P(I(x,y)); \tag{3.4}$$

Calculating a rectangular shaped window size is computable based on the zeroth moment. If the zeroth-order moment is a lower value, then the search window should decrease in size. This assumes extraneous pixels under the search window with low probability values were involved in the zeroth moment calculation. Conversely, if the zeroth-order moment has a higher value, then the search window should increase in size. This assumes the search window was filled with higher probability values of matching the object's color profile and has not expanded to extremities of the object yet. The width of a rectangular shaped search window is defined as:

$$width = 2 * \sqrt{\frac{M_{00}}{max}} \tag{3.5}$$

The height of the search window is arbitrarily sized by multiplying the width by a constant factor. The constant factor is determined through experiment beforehand. Certain assumptions about the dimensions of the object need to be made to tune the ratio of height and width. Bradski's original goal for CAMSHIFT was to track human faces. He set the ratio at 1.2 to produce an elongated search window proportional to a human face. Bradski's implementation used a histogram normalized to store 8-bit values between 0 to 255. The maximum value *max* in equation 3.5 was set to 255. The maximum value can be altogether omitted from equation 3.5 if the histogram is not normalized and keeps its possible range between 0 and 1, because the maximum bin value is 1.

The search window can also be elliptically shaped and re-sized using the following equations

[9]:

$$length = \sqrt{\frac{(a+c) + \sqrt{b^2 + (a-c)^2}}{2}}; \quad width = \sqrt{\frac{(a+c) - \sqrt{b^2 + (a-c)^2}}{2}}; \tag{3.6}$$

where:

$$a = \frac{M_{20}}{M_{00}}; \quad b = 2(\frac{M_{11}}{M_{00}} - x_c y_c); \quad c = \frac{M_{02}}{M_{00}} - y_c^2; \tag{3.7}$$

3.2.1 CAMSHIFT limitations

The basic CAMSHIFT algorithm has several deficiencies because it follows the highest density in the probability distribution image based *only* on hue. Extreme illumination levels can affect its tracking accuracy because hue is ambiguously defined. This can be visualized by the constriction of the radius in color wheel representing hue. Low lighting in a digital image corresponds to lower levels of brightness (V). The diameter of the HSV hexacone narrows as brightness declines, constricting the radius of the hue color wheel. Similarly, different hues become more indistinguishable as saturation decreases. This occurs in digital images with more lighting and white present.

The presence of too much background noise obscures which pixels in the probability distribution image belong to the object. Background noise is defined by pixels not belonging to the object that have a similar histogram probability to the object. When background noise dominates the statistical moment computation the new centroid will be centered around a point in the background rather than the target object. The search window will grow to the size of the background noise as a result. This becomes a difficult situation for the tracking to lock back onto the object. Figure 5.1 illustrates the ill-effects caused by too much background noise.

A similar problem occurs from extreme object occlusion. Occlusion occurs when one object moves in front of an object and blocks the camera's vantage point. If the occlusion blocks only a portion of the object, then the search window will shrink to the still visible area until the occlusion ends and recover back to the size of the entire object. However, if the occlusion is large enough to completely block out the object from the video frame, then tracking might become lost. The search can leave the region where the object is and become susceptible to background noise. The algorithm performs better against noisy occlusion, absorbing the occlusion and maintaining its position [22].

4 Related Works

4.1 Previous CAMSHIFT Extensions

4.1.1 Original CAMSHIFT Extension

Bradski used thresholding to extend the basic algorithm to help reduce the effects of the deficiencies described in section 3.2.1. The corresponding levels of saturation and brightness were used to selectively choose which hue values were permissible to include in the statistical moments computation. Whenever low brightness or saturation values corresponded to a hue value, the hue was ignored. The threshold values were set as constant values. If the overall video scene was very dim, then CAMSHIFT could not track without camera adjustments for more brightness. This is because too few pixels would remain after thresholding. An upper bound threshold was also used for high levels of brightness. This was used because whiter colors can appear similar to flesh colored hues, affecting facial tracking in his implementation. Constant threshold levels might work for the lighting levels of a particular scene. However, if the lighting levels change throughout the series of video frames, then the threshold levels might not be appropriate under different illumination.

4.1.2 SURF Method Extension

Li et al. [9] proposed using adaptive adjustments to the threshold values for brightness and saturation. The thresholds were dynamically adjusted by considering the texture information of each video frame in addition to the HSV levels in individual pixels. The texture information is the spatial arrangement of colors within an image, including the overall lighting levels within the video frame. Using an integral image, corresponding to the original video frame, was used for quick look-ups of the levels under different sub-regions. Each pixel value in an integral image stores the summation of original values from the neighboring pixels above and to the left of the pixel [25]. Calculating the area of any sized rectangle in an integral image is constant time. The threshold levels were adjusted during the initialization period when the search window is first set in a given video frame. The adjustment ended "when the ratio of back projection value between the whole image and the search window reaches the minimum" [9]. When a video frame had overall higher level of lighting illumination, pixels with high saturation values were ignored. Pixels with lower saturation values were ignored when lower illumination levels were detected. The adjustment of the thresholds was kept between 30 and 120, because too much noise occurs at the extremes beyond

these values. However, because thresholding reduces the number of useable pixels, Li et al. found that the threshold adjustment needs to stop when a minimum of thirty percent of useable hue values remain within the search window to maintain enough of the object's color profile.

An additional extension was added to detect and recover lost objects. Detecting a lost object was defined by the bhattacharrya distance [23] of the original probability distribution and the one found under the search window. The bhattacharrya distance is a value signifying the likeness between probability distributions. If the bhattacharrya distance was less than 0.8, then it was considered failed tracking. A bhattacharrya distance of 1 signifies an exactly equal comparison, which is not likely to ever be encountered in practice. In the event of an object being lost, the Speeded Up Robust Features (SURF) [24] method was used to identify the lost target in the overall image.

The SURF algorithm detects, describes and matches features in an image. It relies on the integral image for faster computation of the key points of interest during the detection phase. A Gaussian pyramid is created to find interest points at different scales. This is created by applying a Hessian matrix, comprised of box filters approximating second order Gaussian filters, to individual pixels of the integral image. The image is repeatedly smoothed using increasing scale sizes to form the layers of the pyramid. The Hessian determinant is used to identify extremum points of a smoothed region of interest [9]. The highest and lowest extremum points compared to its 26 adjacent points is marked as an interest point. The interest point descriptor relies on the dominant orientations of all the interest points determined by scanning the Haar wavelet response of the x and y-coordinates of interest points from different angles. A descriptor matching the descriptor of the original target marks a region for rediscovery.

4.2 Previous GPU implementations

4.2.1 Previous OpenGL Version

The previous two implementations used a static single histogram throughout the life of their applications to model the probability distribution of the target object. A single histogram might fail to accurately model the different colored sides of three-dimensional objects. Tracking and recovery can fail if the side of the object used to create the histogram is not exposed to the camera's vantage point.

Exner et al. [26] used multiple reference histograms to model a composite probability distri-

bution of the different sides of three-dimensional objects. Each reference histogram of the current search window with a different enough distribution was integrated into a normalized accumulated reference histogram. The individual reference histograms were saved for object identification during search window drift caused by object occlusion. Histogram intersection summated the minimum bin values between the target histogram and each reference histogram. The histogram pair with the maximum intersection value affirmed an object's identification. An overall maximum intersection value below a threshold indicated a lost object.

Their lost object re-detection strategy used a recursive hierarchical search window subdivision scheme. Starting with the entire frame, the search window was split into four regions. If all of the child windows uniformly converged within their parent, then the object was re-detected at the point of convergence. Otherwise, the recursive division into sub-quadrants was further applied to the sub-regions. If the zeroth-moment was sufficiently small or all four quadrants diverged, then recursion ended in a parent quadrant.

Exner et al. utilized the GPU to accelerate building the multiple histograms and calculating the statistical moments by using the OpenGL API. The video frames and histograms were loaded into vector buffer objects (VBO) in OpenGL texture memory. To save computation, only every *ith* pixel was directly rendered with vertex texture coordinates. A geometry shader generated the neighboring vertexes with the same histogram value. Each histogram bin was filled by the additive alpha blending of the alpha channels in the mapped vectors. The alpha blending is a sequential process per bin, but the vertex shaders ran each process in parallel. They implemented tracking for different color models. Mapping triple values pairs of RGB, HSV, or YUV color models to two-dimensional textures required tiling the textured 2D color planes. Calculating moments followed a similar approach to building a histogram. In parallel, shaders used additive blending to summate the alpha channels of the neighboring pixels of a mapped regional pixel. The resulting five statistical moments were stored in the RGBRG channels of two File Buffer Object (FBO) pixels. Their implementation required the CPU to compute the new centroid and determine if convergence occurred, requiring the transfer of calculated statistical moments between device VRAM and host RAM every iteration.

4.2.2 Previous CUDA Version

Studies have shown that the CUDA implemented versions of GPGPU parallel matrix problems can outperform the OpenGL equivalent [27]. It has the advantage of more memory types for different use cases, like texture memory for faster cached access than global memory. Inter-block communication is an advantage for CUDA over the OpenGL fragment shader which can only write to itself. CUDA is a native proprietary programming model specifically for the Nvidia family of GPU, whereas OpenGL is an extra virtualized layer between application and hardware. Jo et al. [28] successfully implemented a CUDA CAMSHIFT program with real-time performance for rotating a PTZ security camera along the track of a targeted object. The video frame was stored as texture memory, with an image mask as the search window. A sequentially patterned parallel reduction was adapted to calculate the statistical moments under the masked texture. The results from the reduction had processed with the CPU after the reduction kernels finished to calculate the new centroid and search window size. Iterations of kernel launches for the two-phase parallel reduction and memory transfers the statistical moments back to the *host* to update the search window would continue until *host* calculated convergence of centroids was reached.

4.2.3 Reliance of Related Works

Extending the basic CAMSHIFT algorithm has been shown to improve its accuracy and reliability. However, the previous extensions required overhead that slows down the runtime performance of the overall routine. The SURF method requires the construction of an integral image per video frame to analyze the texture information of the pixel groupings. Exner et al. [26] required the construction of a new reference histogram whenever a probability distribution image differed from the original reference histogram. The use of GPGPU reductions can adapt the CAMSHIFT statistical moment calculations into a faster version than the equivalent with *host* CPU processing. A faster CAMSHIFT can hide the overhead of extensions that provide robust performance. CUDA has an API allowing native access to optimized memory types for different appropriate use-cases. The proper use of CUDA can outperform an OpenGL equivalent. The following section describes how the sequential reduction pattern can be used with dynamic parallelism in CUDA to improved the runtime and design of a GPGPU parallel CAMSHIFT.

5 Methodology

Three versions of CAMSHIFT were implemented for comparisons in this project: one entirely reliant on CPU processing and two CUDA versions. Many iterations went into the development of the CUDA implementations; the presented CUDA versions offer a look at two different approaches to tracking multiple objects in parallel. The first version, developed earlier, attempted to track all objects across the blocks of the same grid. The more mature version, separated the blocks tracking a given object into its own grid with the use of dynamic parallelism. All versions shared the same pre-processing steps using OpenCV and C++ to build the hue histogram and remove potential background noise. The project required the hardware from a remote server. Real-time tracking with a live video feed was not tested, given this limitation. The tracking was performed on previously recorded video files and pre-detemined initial search window locations.

5.1 Video Frame Pre-processing

5.1.1 Image Histogram Construction

The project used a single static histogram throughout the lifetime of the application. There would not have been significant performance gain by parallelizing its construction, because the construction was carried out only once for the entire lifespan of the application. It stored floating-point values with a non-normalized discrete range between 0 and 1 based on the probability of the hue values found in the initial search window. The histogram array had a length of 60 index bins. The width of each bin held three consecutive hue values to cover the range of OpenCV 8-bit hue values described in 2.1.4. The initial search window was constructed from reading in its top-left and bottom-right corners from a file. These coordinates were predefined to overlap the target in the first video frame. The first video frame was converted from BGR to HSV using the OpenCV *cvtColor()* described in [8]. The hue channel was extracted into a separate *unsigned char* array using OpenCV. The indexes in the hue array belonging to the search window were traversed, incrementing a bin whenever its hash value was encountered. The hash value was derived from a hue value divided by the bin width. Each total bin count was divided by the size of the search window to arrive at the final percentage value. The histogram was doubled in size for each additionally tracked object. Accessing the histogram bins for a given object was then achieved by offsetting the bin index by the product of the object ID and the original histogram length.

A feature was added to write a histogram out to file after its construction. Reading the histogram in from file into its array on a new execution of the application allowed the same object to be tracked from different videos having to know its initial position or construct the histogram again.

5.1.2 Background Noise Removal

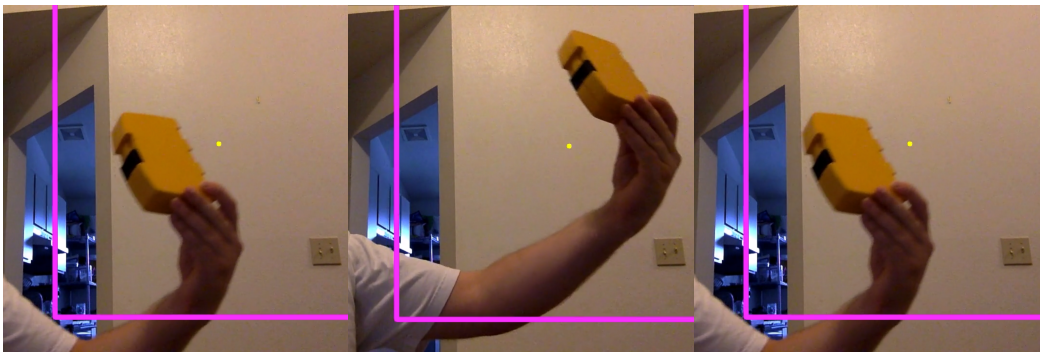


Figure 5.1: Effects of Background Noise

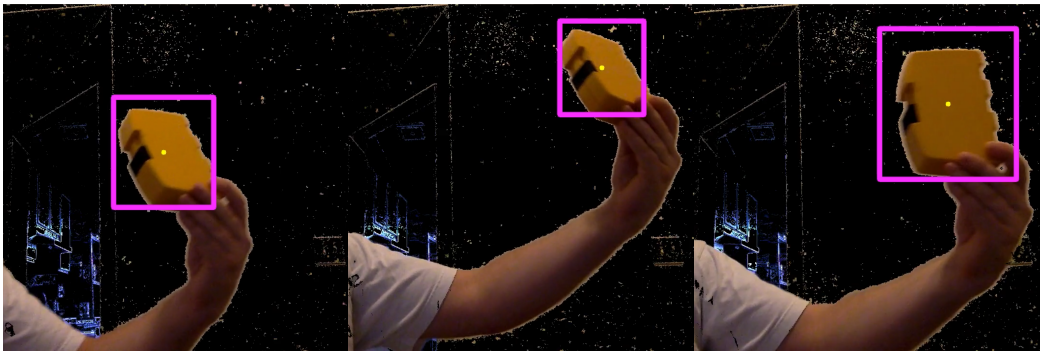


Figure 5.2: Background Noise Removed with Background Subtraction

An optional background subtraction feature was added as a menu option for video sequences that experiments showed had too much background noise. It was carried out before the CAMSHIFT processing in each video frame. Background subtraction compares a background model to the pixels in each frame and creates a mask of pixels that do not match the background model. The pixels in the mask that match the background are filled with zeros. The mask is then applied to the current frame creating a new image with only colored pixels not belonging to the background model. The background model was the first frame of the video sequences. These steps are carried out in OpenCV with the following:

1. Ptr<BackgroundSubtractor >subtractor->apply(frame, mask, 0);

2. cvtColor(mask, mask, COLOR_GRAY2BGR);

3. frame.copyTo(foreground_image, mask);

The *foreground_image Mat* was then used for further CAMSHIFT processing with the background pixels replaced by zeros.

5.2 CPU version

A custom implementation of a CPU version of CAMSHIFT was a fair comparison to the GPU version rather than using the standard OpenCV CAMSHIFT implementation for two reasons. First, the OpenCV version uses an elliptically shaped search window rather than the rectangular shaped search window used in this project. Secondly, a custom implementation allowed a one-to-one comparison by processing the same data in equivalent ways.

The video frames needed to be first converted from a BGR *Mat* to the corresponding hue value array. This conversion was carried out for the CPU version with the following steps in OpenCV:

1. *cvtColor* the BGR *Mat* to a HSV *Mat*

2. *split* to extract the HSV *Mat* into separate *Mat* channels

3. *memcpy* the *uchar* hue *data* in the hue channel *Mat* into an allocated array

The search window was defined by its top-left and bottom-right corner points and initialized by the position read in during the histogram construction. Calculating the moments was controlled by a doubly-nested for-loop covering only the search window. The outer-loop started from the top-left x-coordinate to the bottom-right x-coordinate. The inner-loop started from the top-left y-coordinate to the bottom-right y-coordinate. The points had coordinates defined by their placement within the overall video frame. Translating the two-dimensional x and y-coordinates to the one-dimensional index was defined as:

$$absolute\_index = (frame\_width * x) + y \tag{5.1}$$

Each hue value was histogram back-projected by looking up the hashed histogram bin value corresponding to the hue value. The hue value was used to lookup the histogram value. Each back-projected histogram value was used in the statistical moment calculations in equations 3.1 and 3.2.

5.2.1 Floating-Point Considerations

Computers adhering to the IEE-754 Floating-Point Standard internally represent a floating-point value with three underlying integers: sign (*S*), exponent (*E*), and mantissa (*M*). The value is then computed as follows:

$$value = (-1)^S * 1.M * (2^{E-bias}) \tag{5.2}$$

The exponent determines the range of representable numbers, whereas the mantissa determines the possible precision. During floating-point arithmetic, the mantissa of a smaller value is bit-shifted right until its exponent matches the exponent of the larger value. Inaccuracy occurs from precision rounding after an arithmetic operation between floating-point numbers, if the resulting mantissa requires too many bits to be represented exactly. The order in which a series of floating-point values is added together will affect the accuracy of its summation. In many cases, applications first sort values before adding them together limit the rounding error caused by bit-shifting the mantissa of the smaller value.[2] Otherwise, slightly different results should be expected between a sequential and parallel version of a summating a float-point series. The different versions of this project occasionally had some minor discrepancies between the search window positions. Experiments led to the conclusion to use double floating-point precision for the storing the zeroth and first-order moments of the CPU version. It was particularly susceptible to inaccuracy because of its iterative addition of increasingly large values storing the statistical moment totals with much smaller histogram values. The GPU version did not suffer in the same manner because the work was broken into parallel parts. Floating-point precision is preferable over double precision in CUDA applications because the hardware is optimized for 32-bit precision.

5.3 Shared Features in CUDA Versions

Both versions of the CUDA CAMSHIFT presented in following sections performed a sequential reduction pattern to compute the statistical moments of CAMSHIFT. The reduction was performed twice, the first on the sub-regions of the search windows assigned to each thread block, and the second on these intermediate block totals. The input values were stored in three statically allocated arrays in shared memory representing the zeroth and first-order statistical moments in the algorithm.

A different thread in each block read the mapped indexes of the search window into shared memory from a linear representation of the video frame buffer in global memory. The pre-processing construction of the hue histogram was the same as the CPU version for both presented GPU versions. The constructed histogram in the host memory was transferred into constant memory for access from the device reduction kernels. The histogram back-projection occurred by hashing the read-in hue value from global memory into the constant memory histogram and loading the hashed result into shared memory.

5.4 Non-Dynamic Parallel CUDA Version

The non-dynamic parallel (NDP) design was different than the mature dynamic parallel design in a number of ways. The BGR to HSV conversion of the video frame pre-processing in the early design used the same OpenCV library calls as the CPU design. The dynamic parallel design parallelized this conversion in a kernel used to load the video frame in global memory. The NDP version required the CPU to coordinate the reduction kernels and guide the convergence logic in CAMSHIFT. This workflow required memory transfer overhead between the host and device. The total blocks assigned to each target object were stored in an array for look-up to offset and demarcate which blocks belonged to an object's reduction. The block size totals array had to compute in the host and transferred into device global memory. The final reduction values after the second kernel had to be transferred back to the host for the computation of the new centroid coordinates and the check for the convergence of previous and current centroids.

The strategy of the NDP version was to use the same kernels to track multiple objects. The blocks associated with the first object would be calculated first. The number of blocks was calculated by dividing the search window size by the block size of 1024 and rounding the float-point result to the nearest higher integer. The last block resulting from the rounding-up would have buffer threads that load zeros into the shared memory input and then do nothing. The block total was stored in the index hashed by the object's ID and added to a overall total variable. Each subsequent object's block total was calculated and stored in the same way. The first reduction kernel was configured as a one-dimensional grid of the total number of blocks summated in the above process, each block with 1024 threads. An object ID per block was set, demarcating which object the block was contributing computation for, by comparing the block ID to the array storing different objects' block

totals. If the block ID was greater than the block total stored in one index and less than the block total in the next index, then its block belong to the object ID matching the first index. Mapping a reduction kernel thread to a video frame global memory input index required first subtracting from its *absoluteThreadID* the total number of threads from the blocks before the blocks associated with its target to have a relative ID to its search window.

The second reduction kernel of the NDP version was configured to a grid size of one block per object tracked. Each block was configured to the size of the total blocks used in the first reduction kernel. The second kernel did not use a sequential reduction pattern, because the block size was set to a non-power of two. Instead, it simply loaded shared memory with each thread associated with each block in the first kernel. Only one thread was used to add all of the intermediate block results together that were in the range of its object. The thread then calculated the new centroids and window dimensions and stored these values in global memory. A test showed that there was not a significant difference in runtime between a linear summation by one thread in the final reduction step versus rounding block size to a higher power of two and doing a sequential reduction pattern. This is because the input size was under 1024 and within only one block. The NDP version was limited to how many search window blocks in the first reduction could be represented by less than 1024 threads in the second final reduction stage. Checking for the centroid convergence from the host required transferring the new centroids and windows dimensions from the device global memory and scanning an array of boolean flags set by the final reduction kernels if convergence had occurred. The runtime of any object in this design was limited by the computation of the largest search window. Much of the overhead in the NDP version was eliminated by using dynamic parallelism in the second design. The extra memory latency between *host* and *device* memory transferring at the end of each iteration in the CAMSHIFT algorithm was no longer needed. Nor was it required to offset and consider the computation of other objects within the same kernel.

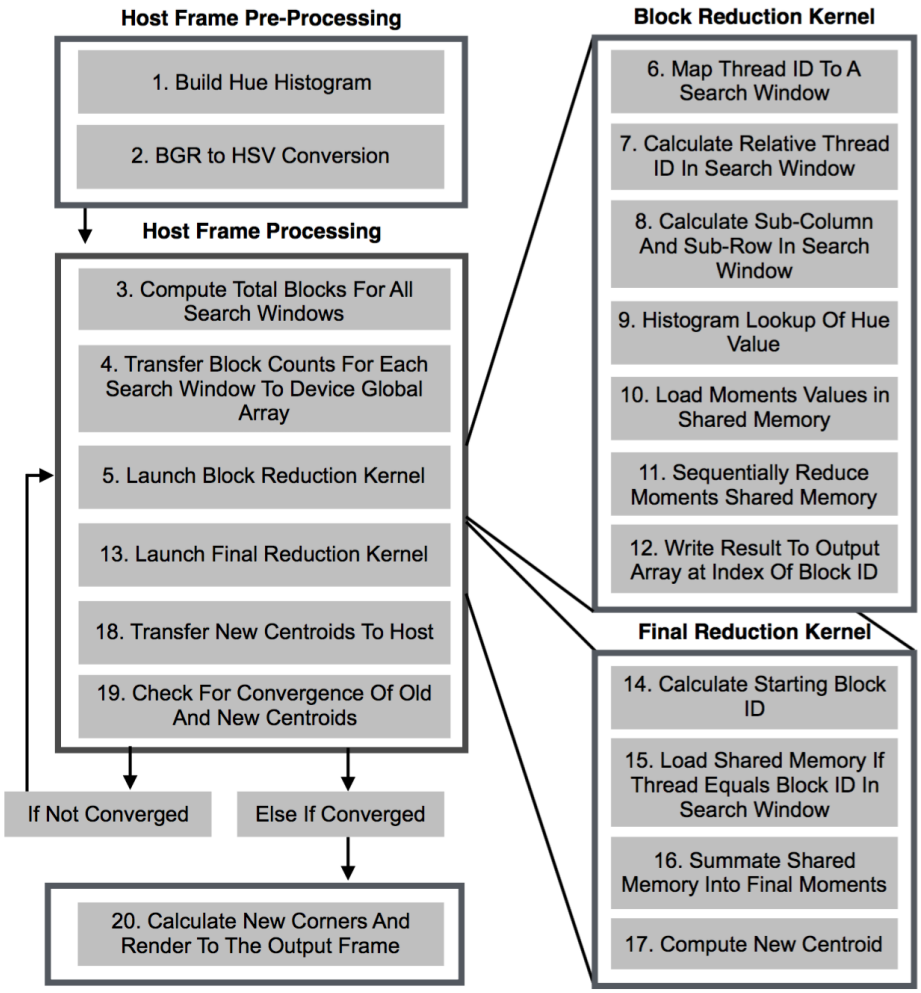


Figure 5.3: Non-Dynamic Parallel GPU Overview

5.5 Dynamic Parallelism

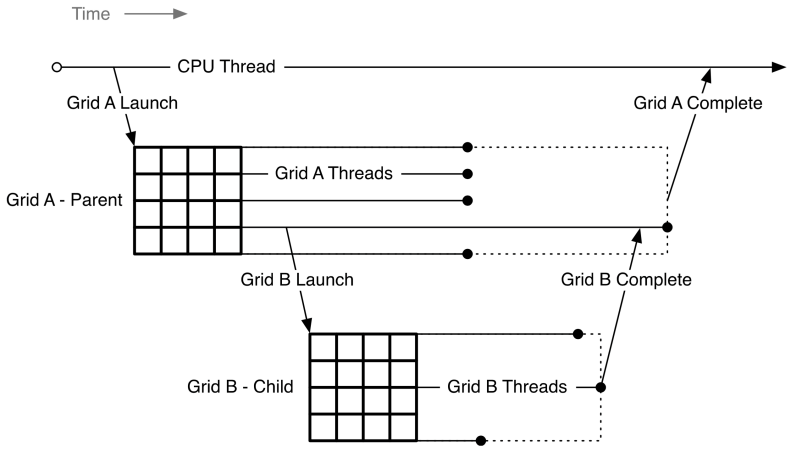


Figure 5.4: Dynamic Parallelism Parent and Child grid execution [13]

Dynamic Parallelism (DP) in CUDA enables a kernel to spawn and synchronize nested kernels. Applications with iterative stages of kernel launches can shift the control to the parent kernel away from the CPU's responsibilities. The parent kernel can collect the output produced by its child kernels and control the next launch of children accordingly. CUDA DP also supports recursive kernel launches from within a parent kernel. Large data transfers is a bottleneck in GPGPU applications. The lack of CPU reliance improves application runtime where repeated data transfers between device and host would otherwise be required. The GPU schedulers and load balancers can increase the parallelism by dynamically responding to parent kernel's decisions and changing work loads. Figure 5.4 shows the runtime synchronization of nested grids. Even if the parent threads do not explicitly synchronize their child grids, an implicit synchronization between the parent and child is guaranteed by the CUDA runtime [13]. However, to ensure all child grids have finished work before the parent can accurately read their output, child grids must be explicitly synchronized with *cudaDeviceSynchronize()* called by the application programmer. There is also some overhead in DP kernel launching. The parameters of the kernel launch must be parsed, implicitly calling *cudaGetParameterBuffer* and *cudaLaunchDevice*. The device runtime manager must be setup and enqueued before managing and dispatching the child kernels. [14]

5.6 Dynamic Parallel CUDA Version

The kernels in the dynamic parallel (DP) version of CAMSHIFT could be written from the perspective of only tracking one object. The *host* control only had to launch a parent kernel and retrieve the final new centroid coordinates and search window dimensions after the processing for each frame. Figure 5.5 shows the overview of the logic flow controlled by the parent kernel.

5.6.1 CUDA Histogram

After the histogram was constructed as described in section 5.1.1, it was copied into *device* constant memory for use in the kernels. The constant memory histogram was statically sized as the maximum number of trackable objects multiplied by the 60 histogram bins per object. The maximum number of trackable objects was set to only three as a proof of concept. Any more objects would probably become too crowded and cause too much object occlusion to accurately track all of the objects.

Constant memory was chosen for the histogram's storage because the histogram was a read-only

one-dimensional linear array, constructed once, and re-used through the application lifetime. The rationale behind increasing the histogram bin width to three was to improve the runtime performance at the expense of tracking accuracy. A smaller range of histogram bins would increase the likelihood of a constant cache hit by the reads of the threads within a warp.

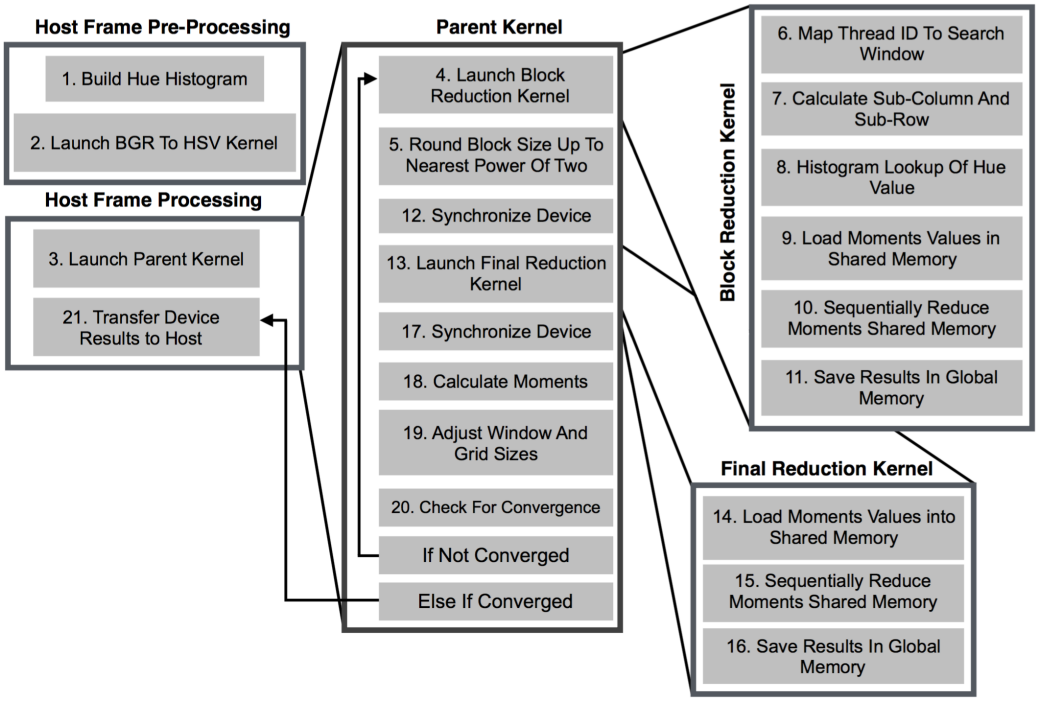


Figure 5.5: Dynamic Parallel GPU Overview

5.6.2 CUDA BGR to HSV Conversion

The conversion of the BGR video frame to HSV was parallelized using a CUDA kernel as an optimized preprocessing step before loading the video frame hue values into *global* memory. The hue array was statically allocated as *device* global memory for the following reasons. The video frame had a fixed size throughout the life of the application. The array needed to be updated with each subsequent video frame. Using global memory allowed each index to be updated concurrently by the threads of the conversion kernel. The conversion kernel proceeded in the following steps:

1. *cudaMemcpy* the BGR *data* array directly from *Mat* into a host reference pointer to device global memory

2. Launch kernel with a block size the third of the BGR array

3. Each CUDA thread read in a BGR pixel triplet based on its absolute thread ID within the grid

4. Each CUDA thread converts the BGR pixel triplet into HSV values with the equations in figure 2.4

5. Each CUDA thread write the hue value into a statically pre-allocated global memory array based on its absolute thread ID within the grid

The extracted BGR array from step 1 had each BGR triplet stored in three contiguous indexes and required offsetting the access in step 2 by a multiple of three. Experiments revealed a rounding difference between the actual OpenCV API conversion function and implementing its underlying equations separately. This resulted in occasional off-by-one value discrepancies, but the difference did not affect the positioning of the two versions by more than a few pixels. The increased histogram bin width helped decreased the impact of the conversion discrepancy by hiding some inaccuracy within the bin widths.

5.6.3 Dynamically Parallel Reduction

Dynamic parallelism minimized the memory transfer between host and device. A parent kernel was configured with a grid size of one block per object and a block size of one thread. The block ID of the parent kernel set the object ID for offsetting access to the histogram. The parent kernel blocks controlled their convergence logic with only information concerning their own respective target object. The use of dynamic parallelism allowed the parent kernel blocks to launch their own reduction kernels using their own scheduling resources. These reduction kernels likewise had no need for any knowledge of blocks reducing other search windows statistical moments. The first child block reduction grid was launched from the parent *device* thread and synchronized before the second child reduction was launched from within the parent thread to further reduce the final statistical moments. Explicit device synchronization was required between reduction kernel launches by the parent control thread to ensure the first kernel finished its computation first. The parent grid calculated the new centroid and search window dimensions based off of the second child reduction values, and updated the variables used in the first reduction kernel representing the search window dimensions and positions. If the new centroid converged with the previous centroid, then the parent thread terminated. The *host* would then copy the memory from the *global* memory and update its variables storing the new centroid and top-left and bottom-right corners of the search window for display. However, if there was no convergence, then the parent thread calculated the new grid size for the

first kernel and repeated the two-step reduction kernel launches again. The only per-frame memory transfer between host and device was the host-to-device image hue array and the device-to-host new centroid and height and width after convergence. The use of statically sized global memory allowed for the re-use of memory between iterations without any need to re-allocate memory.

5.6.4 Sequential Reduction of Statistical Moments

The sequential reduction pattern described in section 2.6 was adapted in this project to calculate the statistical moments of the search window. The problem space did not allow for the recursive use of a single reduction kernel for the two levels of reduction. Two different reduction kernels were required, each with its own unique tasks. The thread blocks in the first kernel had the following tasks:

1. Map absolute thread IDs to the absolute index in the hue array
2. Calculate the x and y-coordinates of the original Mat object corresponding to the absolute index
3. Load the shared memory with histogram back-projected values
4. Reduce the shared memory to intermediate totals
5. Write the intermediate totals out to global memory

The block size of the first kernel was set to the maximum allowed value of 1024 threads. The number of blocks was calculated by rounding up to the nearest integer value the window size divided by the block size. Each thread block had three statically allocated shared memory arrays of floating-point precision, each the same size as the thread block, representing the zeroth and first-order moments respectively. The precision errors described in section 5.2.1 did not result from using floating-point precision in the reductions, because the totals were broken into intermediate totals by the first reduction kernel. Combining the intermediate totals in the second reduction avoided the accumulative error experienced in the CPU version with floating-point values. The absolute thread ID, defined by equation 2, corresponded to a relative position within the search window, conceived as a one-dimensional array. Translating this relative index value to its absolute frame index within the entire hue array was computed using the following equation:

$$absolute\_frame\_index = (W * topY) + topX + (W * subY) + subX \tag{5.3}$$

where:

$topX$: x-coordinate of the start of the search window in the video frame Mat object

$topY$: y-coordinate of the start of the search window in the video frame Mat object

W : the width of video frame

$subX$: x-coordinate relative to within the search window

$subY$: y-coordinate relative to within the search window

The subX and subY coordinates were calculated from the thread ID using the following equations:

$$\begin{aligned} subX &= i/w \\ subY &= i \bmod w \end{aligned} \tag{5.4}$$

where:

i : the absolute thread ID within the grid

w : search window width

Any thread with an *absoluteThreadID* larger than the size of the search window filled its shared memory indexes with zeros. Otherwise, the absolute frame index was used to read in a hue value from global memory. The hue value served as the hash key to an index in the constant memory histogram. Each thread used its *threadID* within the block as the index to load the retrieved histogram value into shared memory. The zeroth moment shared memory array was loaded with only the histogram value. The first-order x and y moment arrays were loaded with the histogram value multiplied by the *subX* and *subY* values respectively. The first reduction kernel proceeded with the sequential addressing pattern described in section 2.6 for each shared memory array in the same way after loaded. The intermediate values were written out to another statically allocated global memory array by the first thread in the block. The zeroth-order moment intermediate value was written to the index of the block ID, the first-order moment with respect to x was written to the block ID offset

by the number of blocks, and the first-order moment with respect to y was written to the block ID offset by twice the number of blocks.

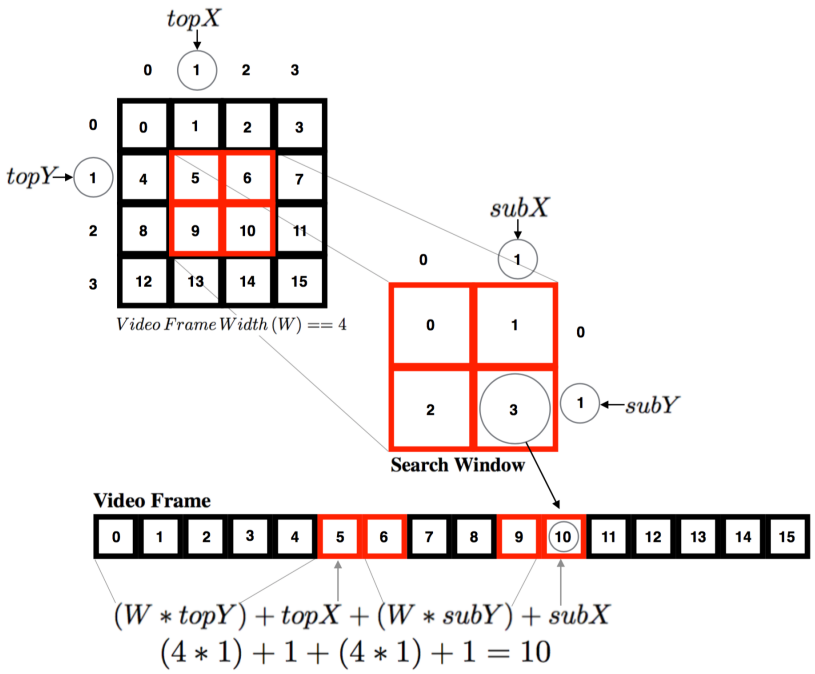


Figure 5.6: Example Mapping Thread ID of 3 to Search Window Index in Video Frame

The second reduction kernel had the simple task of further reducing the intermediate values from the first reduction. The grid size in the first reduction was guaranteed to be less than 1024 because the dimensions of a video frame was only 1024 by 720. This allowed the second reduction kernel to be launched in a single block. The block size was set as the number of blocks from the first kernel was rounded up to the nearest power of two, in order to maintain the same reduction pattern as the first kernel. The original number of blocks from the first kernel was kept as the new length of the input data for the second reduction. Any thread with a block *threadID* less than this length filled its shared memory indexes with a buffer value of zero. Otherwise, the threads read in the first reduction output values to three shared memory arrays and proceeded as described in section 2.6. The first three indexes in the same output global array were used to write the final moments results out to global memory.

5.6.5 Multiple Object Tracking

The initial attempt at extending the GPU version to track multiple objects with the NDP version kept the calculations of different objects statistical moments in the blocks of the same reduction

kernels. This approach was overly complicated and inefficient. It required the use of offsets to differentiate which blocks corresponded to a given object. The number of blocks per object were used as the offsets and had to be stored in global memory for access across thread blocks. The runtime of the reductions for a smaller search window was limited by the completion of the largest search window reduction, because all block offsets needed to be updated in the parent thread before the next iteration. Checking for the convergence of multiple objects required setting a boolean flag for each object upon its convergence. The condition for a completely processed video frame required the parent thread to scan an array storing all of the convergence flags and finding all true values.

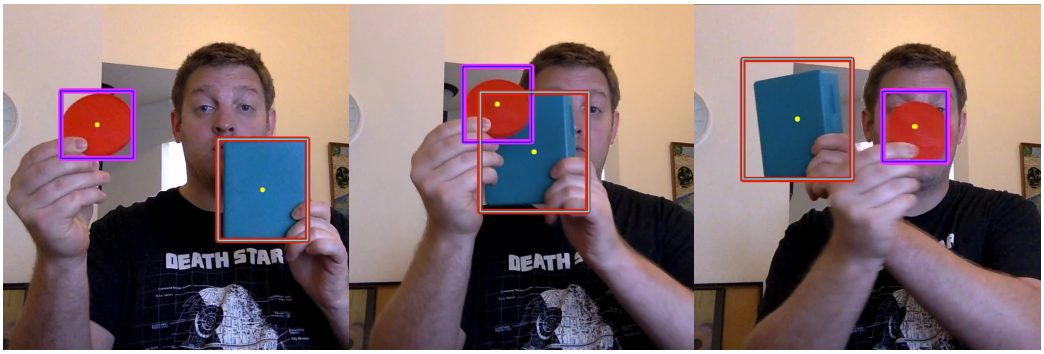


Figure 5.7: Extension for Multi-Object Tracking

The extension to tracking multiple objects in this project was greatly simplified and accelerated by utilizing dynamic parallelism. Each object was tracked separately by its own thread in the parent kernel launched from the host. The parent control kernel and both child reduction kernels were kept simplified because all blocks of their reductions belonged to the same object. The thread ID of each parent thread identified the object ID used by the parent thread and all of its nested threads for offsetting access to any shared data structures. The histogram array and moment output arrays were extended by the multiple of objects tracked to offset their storage. The object ID was used to perform pointer arithmetic to point threads to the start of their object's offset section of the arrays. Each parent thread could operate in complete isolation from each another until its own convergence. Only the host thread had to synchronize the parent grid launch to ensure reading the accurate final centroids and search window dimensions.

5.6.6 Lost Object Recovery

An extension to this project was attempted to help recover from the failed tracking that results from the deficiencies in the algorithm described in section 3.2.1. The approach was to expand the search window to the size of the entire video frame when a lost object was detected. Lost object detection was defined by an convergence iteration count of over twenty times. An obvious problem with this definition is that it will not recognize when a search window has locked onto any background noise or similar object. However, it is quite effective in the event of a drifting search window that has lost the object because of occlusion by a dissimilar object. The window can drift if the background hue values are different enough from its histogram distribution to result in no convergence. Another problem arises during the recovery of the lost tracking. Only expanding the search window to the entire frame leaves the convergence algorithm vulnerable to mistaking background noise as the object for rediscovery. Coupling the recovery algorithm with preprocessed background subtraction helps mitigate this risk.

6 Results

The test results collected in this project were run on a server with an Intel i7 CPU with 6 cores and a clock speed of 3.50 GHz. The server had 4 Nvidia GTX 970 GPUs with 1.2 GHz clock speed. Reading the video frames into memory and writing the processed video frames to an output video file shared the same OpenCV library calls for all three versions of the implemented CAMSHIFT. The collected test results were comprised of the two computational tasks that differed between the CPU version and the improved dynamically parallel design on the GPU. The first task was the pre-processing stage of BGR-to-HSV video frame conversion before the actual CAMSHIFT statistical moment computation. The test results of this stage for the dynamically parallel GPU version included both the host-to-device memory transfer of the BGR video frame and the runtime of the BGR-to-HSV conversion kernel. The earlier non-dynamically parallel GPU version shared the same OpenCV BGR-to-HSV video frame conversion as the CPU version. Therefore, the memory transfer of the prepared video frame was not required for this conversion and was not included in its timing. The second computational task timed and compared was the CAMSHIFT statistical moment computation. The memory transfer for each video frame was included in the results for the non-dynamically parallel GPU version, because it occurred after the first task. However, this memory

transfer was not considered in the dynamically parallel GPU version because the conversion hue array was already stored in global memory at the conclusion of its BGR-to-HSV conversion kernel. Test results were gathered for both single and two object multi-tracking. Statically sized search windows were used to show the average runtimes at set search window sizes.

6.1 Single Object Tracking Results

All three designs tested could track a search window the size of the entire frame, because the frame size could be divided into less than the NDP design limit of 1024 block size in the second block reduction kernel.

	RGB TO HSV Setup	Convergence Computation	Total
200 x 200 search window			
CPU	2.94 ms	1.95 ms	4.89 ms
NDP GPU	2.94 ms	0.38 ms	3.32 ms
Speed-Up	0	5.13	1.47
400 x 400 search window			
CPU	3.32 ms	3.74 ms	7.06 ms
NDP GPU	3.32 ms	0.61 ms	3.93 ms
Speed-Up	0	6.13	1.8
1080 x 720 search window			
CPU	2.96 ms	19.34 ms	22.3 ms
NDP GPU	2.96 ms	1.01 ms	3.97 ms
Speed-Up	0	19.15	5.62

Figure 6.1: CPU and Non-Dynamic Parallel GPU Single Object Tracking Design Comparison

6.1.1 CPU and Non-Dynamic Parallel GPU design comparison

The runtime of the per-frame computation of the statistical moments and convergence is the only way to compare the differences between the NDP and the CPU versions. This is because both versions shared the same pre-processing for the input preparations. The speed-up for the NDP version over the CPU version was modest for small to medium sized search windows. Figure 6.1 shows the more drastic improvement provided by the parallelized version for single object tracking once the search window is expanded to the entire video frame.

6.1.2 CPU and Dynamic Parallel GPU design comparison

The DP design showed five times speed-up in the pre-processing color model conversion over the other designs. This step pre-loads the global converged hue video frame at the conclusion of the

conversion kernel, hiding the latency of each video frame memory transfer to device. The speed-up increased compared to the CPU design without the initial per-frame memory transfer latency.

	RGB TO HSV Setup	Convergence Computation	Total
200 x 200 search window			
CPU	2.94 ms	1.95 ms	4.89 ms
DP GPU	0.58 ms	0.19 ms	0.77 ms
Speed-Up	5.0	10.2	6.35
400 x 400 search window			
CPU	3.32 ms	3.74 ms	7.06 ms
DP GPU	0.57 ms	0.19 ms	0.76 ms
Speed-Up	5.82	19.34	9.29
1080 x 720 search window			
CPU	2.96 ms	19.34 ms	22.3 ms
DP GPU	0.57 ms	0.4 ms	0.97 ms
Speed-Up	5.2	48.4	22.97

Figure 6.2: CPU and Dynamic Parallel GPU Single Object Tracking Design Comparison

6.2 Non-Dynamic Parallel and Dynamic Parallel GPU Design Comparison

	RGB TO HSV Setup	Convergence Computation	Total
200 x 200 search window			
NDP GPU	2.94 ms	0.38 ms	3.32 ms
DP GPU	0.58 ms	0.19 ms	0.77 ms
Speed-Up	5.0	2	4.3
400 x 400 search window			
NDP GPU	3.32 ms	0.61 ms	3.93 ms
DP GPU	0.57 ms	0.19 ms	0.76 ms
Speed-Up	5.82	3.2	5.2
1080 x 720 search window			
NDP GPU	2.96 ms	1.01 ms	3.97 ms
DP GPU	0.57 ms	0.4 ms	0.97 ms
Speed-Up	5.2	2.5	4.1

Figure 6.3: Non-Dynamic Parallel and Dynamic Parallel GPU Single Object Tracking Design Comparison

The use of a dynamically parallel design shows a five time total speed-up over the earlier design. The comparison of the convergence computation is admittedly unfair between these designs because of the included extra initial memory transfer in the earlier design. However, the total speed-up considers all memory transfer latency between the two designs. The DP design showed performance

gains without the iterative and intermediate device-to-host memory transfer of results. The dynamically parallel design was also worth implementing over the earlier design because of the simpler logic within the reduction kernels.

6.3 Multiple Object Tracking Results

Only the earlier NDP design could not track multiple objects because of its problem with block size overflow. Regardless, most trackable objects fit within a smaller 400 by 400 search window. The full-frame 1080 by 720 search window feature was only tested for gaining understanding of its time-cost for a possible future extension of object re-detection. The results for tracking two objects are presented in the following sections.

6.3.1 Non-dynamic parallelism GPU versus CPU Designs

The CPU design tracked multiple objects consecutively by calculating the per-frame final centroids of the different object sequentially. The NDP GPU design attempted to parallelize multiple object tracking. Its design was limited by slowing the faster search window computation to the time cost of the slower search window during every iteration of the per-frame computation, due to the sharing of kernels between the blocks of both search windows. As expected, the CPU design roughly doubled its time cost during the convergence computation. The CPU design became unusable for a real-time application when tracking two search windows both sized the entire frame.

	RGB TO HSV Setup	Convergence Computation	Total
200 x 200 search window			
CPU	3 ms	2.4 ms	5.8 ms
NDP GPU	3 ms	0.38 ms	3.38 ms
Speed-Up	0	6.3	1.7
400 x 400 search window			
CPU	3 ms	9 ms	12 ms
NDP GPU	3 ms	0.79 ms	3.79 ms
Speed-Up	0	11.4	3.2
1080 x 720 search window			
CPU	3 ms	37.2 ms	40.2 ms
NDP GPU	N/A	N/A	N/A
Speed-Up	N/A	N/A	N/A

Figure 6.4: CPU and Non-Dynamic Parallel GPU Version Multiple Object Tracking Comparison

6.3.2 Dynamic parallel GPU versus CPU Designs

	RGB TO HSV Setup	Convergence Computation	Total
200 x 200 search window			
CPU	3 ms	2.4 ms	5.8 ms
DP GPU	0.6 ms	0.24 ms	0.84 ms
Speed-Up	5	10	6.9
400 x 400 search window			
CPU	3 ms	9 ms	12 ms
DP GPU	0.6 ms	0.34 ms	0.94 ms
Speed-Up	5	26.5	12.8
1080 x 720 search window			
CPU	3 ms	37.2 ms	40.2 ms
DP GPU	0.6 ms	0.8 ms	1.4 ms
Speed-Up	5	46.5	28.7

Figure 6.5: CPU and Dynamic Parallel GPU Version Multiple Object Tracking Comparison

The timing results showed the DP GPU design actually doubled for the convergence computation. This might be explainable in two ways. One explanation is the added overhead of maintaining and parsing the scheduling information by the parent grid. There are three layers of kernel launching recursion. The second explanation might be the decreased occupancy for branches of the recursion due to more active blocks. However, the slowdown results in a time that is still much below the requirements of real-time performance. The comparison of the DP GPU design and the CPU design, shown in figure 6.5, an improved speed-up over single object tracking still occurred.

6.4 Non-Dynamic Parallel GPU versus Dynamic Parallel GPU Designs

The increased target of two objects did not increase the speed-up for the DP GPU design over the NDP GPU design. However, the DP GPU design was the better design because it continued to show a speed-up and had no block size overflow problem limiting the number of search windows. The design could theoretically compute over as many search windows as maintainable in device memory.

	RGB TO HSV Setup	Convergence Computation	Total
200 x 200 search window			
NDP GPU	3 ms	0.38 ms	3.38 ms
DP GPU	0.6 ms	0.24 ms	0.84 ms
Speed-Up	5	1.6	4
400 x 400 search window			
NDP GPU	3 ms	0.79 ms	3.79 ms
DP GPU	0.6 ms	0.34 ms	0.94 ms
Speed-Up	5	2.3	4
1080 x 720 search window			
NDP GPU	N/A	N/A	N/A
DP GPU	0.6 ms	0.8 ms	1.4 ms
Speed-Up	N/A	N/A	N/A

Figure 6.6: Non-Dynamic Parallel GPU and Dynamically Parallel GPU Version Multiple Object Tracking Comparison

7 Observations and Discussion

The goal of this paper was to increase the computational speed of the basic CAMSHIFT computation. Without the use of thresholding the saturation and brightness levels or any other robust extensions, its performance quality remained vulnerable to its traditional deficiencies. However, object tracking in this project still performed well under specific conditions. Objects with a small probability distribution range were accurately tracked without background noise. Even when occluded by another object of a different histogram distribution, these objects could be recovered consistently due to their high probability histogram values. When the object was occluded by a dissimilar obstruction, the search window would shrink to the size of the nearest, if any, uncovered portion of the object.

Sometimes the window would not recover to a large enough size if it became too small, however. An explanation might be the use of histogram ranges between 0 to 1 were too small to recover the size of the window after occlusion. The problem also occurred when the object just moved far enough away from the camera. The past implementations in the related works always normalized the histogram values to contain 8-bit integers.The non-normalized percentage values were kept in this project, however, to keep the summations of the statistical moments low enough to fit in the optimized 32-bit registers of the SM. It is unclear why smaller histogram values affects it. It might simply be the over-importance each pixel takes in the smaller sample size. A heuristic was added to

increase the search window size large enough to recover tracking if the window shrank too small. If the search window's total size, regardless of dimensionality, fell below 20, then it was resized to have a width of 200. Experiments derived this heuristic based on tweaking the thresholds until it performed well.

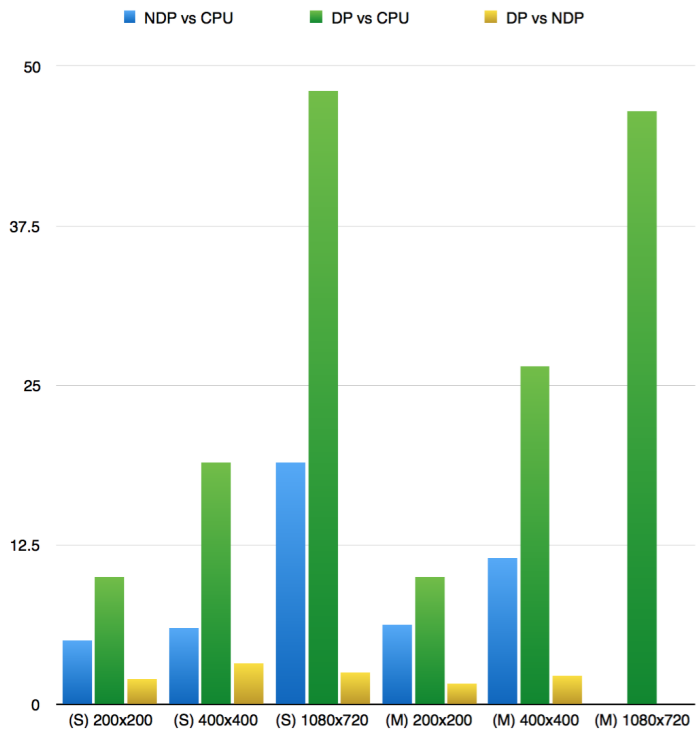


Figure 7.1: Statistical Moment Computation Speed-Up Overview

Both versions were particularly prone to background noise interference. Not using thresholding based on the lighting was a major contributor to this problem. But imprecise hue definition during histogram back-projection also played a role in making the background seem more noisy than it should. First, the OpenCV use of 8-bit storage for the BGR primary color values cut the range of translated hue values in half from 360 degrees to 180. This alone cut the range of hue definition in half. Second, the increased histogram bin width to three hue values per bin further reduced the number of distinguishable hue probabilities.

Background noise removal with the OpenCV background subtraction was a very effective pre-processing technique to compensate the increased background noise susceptibility. Figure 5.2 shows how effective of a measure it could be. However, experiments proved that background subtraction can be a limited technique in most situations. If the camera is not stationary, then the background

model will not be comparable with the background in each subsequent video frame. It did not work well for video sequences that built the histogram of the object based on the initial search window in the first frame either. The background model treated the pixel values at the initial position of the object as a part of background. Tracking the object in this video sequence often fails, because the first few video frames black out the object if it does not change positions quickly enough. The search window often would drift too far away for the object to be recovered. The added feature to read the histogram from a file instead was added partly to fix this issue. The first frame would not have to include the object for histogram construction and the background model would not include the object. However, the object detection routine needs improvement to ensure this process will consistently work.

Another performance issue arose from the hue definition ambiguity. Tracking objects with more complex hue compositions did not perform well without background noise removal. There are several factors that could account for this. A complex hue composition would correlate to a more evenly distributed histogram. A wider range of background hue values would have a more significant histogram value. Object occlusion would also have a greater chance of confusing the algorithm because of a wider range of coinciding significant histogram bins in the different objects' color models.

The equations for calculating the window size presented some issues as well. Objects with wider rather than taller dimensions tended to have windows that were too large, because the height of the search window was defined as a simple ratio of the calculated width. Likewise, when the object was much taller than its width, using a constant ratio to calculate the height resulted in a window too short to cover its entirety. Rotating the object would shrink the search window, as well, because a significant portion of the object would angle outside of search window coverage.

8 Future Work

The next focus for this project will be directed towards improving its tracking accuracy. The first step towards this goal is increasing the distinguishable range of different hues. Another important step is adding extensions to the basic routine to make the tracking more robust. A combination of the SURF extension with the use of different multiple histograms for three-dimensional object profiling would be an interesting experiment. The use of atomic operations to construct histograms

with a CUDA kernel is a well-established technique [30][31]. Atomic operations can allow threads to safely increment the histogram bin counters without conflict by enforcing sequential operations between the conflicting updates. The use of dynamic parallelism to launch the CAMSHIFT reduction kernels in parallel with reference histogram construction kernels should also be investigated.

Other features of the CUDA API should be explored for a even faster CAMSHIFT routine. Better runtime performance might be achieved using asynchronous memory transfers to hide latency.

This project was limited in its application by its reliance on a remote server. An extension to real-time object tracking is an important step towards integrating the routine into other applications. A different way the project could be extended to real-time object tracking is porting it to the iOS framework using the Metal API. Metal is a relatively new API that exposes native low-level calls to the GPU within Apple hardware devices for custom shaders and general purpose progamming.

9 Conclusion

This paper presented a computer vision system for tracking multiple object positions and sizes across the span of digital video frames. The system utilizes the CUDA programming model to distribute the requisite work across the many cores of the Nvidia graphics card streaming multiprocessors. The basis of this system is the well-established CAMSHIFT algorithm. It calculates the local peak of the probability distribution image comprised histogram values of an object's color profile. The conversion from the RGB color model, stored in regular video frames, to the isolated color found in the hue channel of the HSV color model was shown to perform faster via CUDA parallelization than in an OpenCV equivalent. The CAMSHIFT algorithm was translated into the well-established two-phase sequential addressing reduction parallel algorithm to show significant speed-up over an equivalent version using only CPU processing. The help of dynamic parallelism allowed the CAMSHIFT computation to avoid any CPU intervention until convergence despite the iterative nature of the algorithm. The good run-time speeds of this implementation were fast enough to allow adding image preprocessing techniques for more robust performance while remaining under the time requirements for real-time performance. Background subtraction proved to be an effective noise removal procedure to allow tracking in the presence of similarly colored backgrounds under certain conditions. Thresholding based on different color model channels would be an easy and efficient extension to avoid the effects on hue values of the presence of intense illumination levels.

Vita

Author: Matthew J. Perry

Place of Birth: Saint Louis Park, Minnesota

Undergraduate Schools Attended: University of Montana,

Eastern Washington University

Degrees Awarded: Bachelor of Arts in Philosophy, 2009, University of Montana

Professional Experience: Junior Software Engineer, Commerce Architects, Spokane, Washington, 2016

References

[1] Y. Cheng, "Mean shift, mode seeking, and clustering," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 17, no. 8, pp. 790–799, 1995.

[2] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Newnes, 2012, no. 151-170.

[3] [Online]. Available: https://en.wikipedia.org/wiki/Hue#cite_note-1

[4] [Online]. Available: https://en.wikipedia.org/wiki/RGB_color_model

[5] [Online]. Available: https://commons.wikimedia.org/wiki/File:RGB_farbwuerfel.jpg

[6] (2006). [Online]. Available: http://david.navi.cx/blog/wp-content/uploads/2006/06/HSV_cone.png

[7] A. R. Smith, "Color gamut transform pairs," *ACM Siggraph Computer Graphics*, vol. 12, no. 3, pp. 12–19, 1978.

[8] [Online]. Available: http://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html

[9] J. Li, J. Zhang, Z. Zhou, W. Guo, B. Wang, and Q. Zhao, "Object tracking using improved camshift with surf method," in *Open-Source Software for Scientific Computation (OSSC), 2011 International Workshop on*. IEEE, 2011, pp. 136–141.

[10] M. Harris, "Maxwell: The most advanced cuda gpu ever made." [Online]. Available: https://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/

[11] [Online]. Available: http://geco.mines.edu/tesla/cuda_tutorial_mio/pic/Picture1.png

[12] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008, pp. 73–82.

[13] [Online]. Available: http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf

[14] J. Wang and S. Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured gpu applications," in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 51–60.

[15] N. Paulius Micikevicius, "Local memory and register spilling." [Online]. Available: http://on-demand.gputechconf.com/gtc-express/2011/presentations/register_spilling.pdf

[16] N. Gupta, "Bank conflicts in shared memory in cuda." [Online]. Available: http://cuda-programming.blogspot.com/2013/02/bank-conflicts-in-shared-memory-in-cuda.html

[17] "Advanced cuda webinar: Memory optimizations." [Online]. Available: http://on-demand.gputechconf.com/gtc-express/2011/presentations/NVIDIA_GPU_Computing_Webinars_CUDA_Memory_Optimization.pdf

[18] N. Gupta, "Texture memory in cuda, what is texture memory in cuda programming." [Online]. Available: http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html

[19] N. C. Dr. Justin Luitjens, Dr. Steven Rennich, "Cuda warps and occupancy: Gpu computing webinar 7/12/2011." [Online]. Available: http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf

[20] M. Harris, "How to access global memory efficiently in cuda c/c++ kernels." [Online]. Available: https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/

[21] N. Gupta, "What is "constant memory" in cuda, constant memory in cuda." [Online]. Available: http://cuda-programming.blogspot.com/2013/01/what-is-constant-memory-in-cuda.html

[22] G. R. Bradski, "Computer vision face tracking for use in a perceptual user interface," 1998.

[23] A. Bhattachayya, "On a measure of divergence between two statistical population defined by their population distributions," *Bulletin Calcutta Mathematical Society*, vol. 35, pp. 99–109, 1943.

[24] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *Computer vision–ECCV 2006*. Springer, 2006, pp. 404–417.

[25] Badgerati, "Computer vision - the integral image." [Online]. Available: https://computersciencesource.wordpress.com/2010/09/03/computer-vision-the-integral-image/

[26] D. Exner, E. Bruns, D. Kurz, A. Grundhöfer, and O. Bimber, "Fast and robust camshift tracking," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*. IEEE, 2010, pp. 9–16.

[27] R. Amorim, G. Haase, M. Liebmann, and R. W. d. Santos, "Comparing cuda and opengl implementations for a jacobi iteration," in *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*. IEEE, 2009, pp. 22–32.

[28] J. H. Jo and S. G. Lee, "Cuda based camshift algorithm for object tracking systems," 2013.

[29] M. Harris *et al.*, "Optimizing parallel reduction in cuda," *NVIDIA Developer Technology*, vol. 2, no. 4, 2007.

[30] V. Podlozhnyuk, "Histogram calculation in cuda," *NVIDIA Corporation, White Paper*, 2007.

[31] N. Sakharnykh, "Gpu pro tip: Fast histograms using shared atomics on maxwell." [Online]. Available: https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/