GPU PARALLEL PROCESSING FOR FAST ROBOTIC PERCEPTION

A THESIS
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF ENGINEER

Paul Baumstarck
December 2009
Approved for the department.

Andrew Ng, Adviser

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this thesis in electronic format. An original signed hard copy of the signature page is on file in University Archives.
Abstract

In this thesis we detail how we built GPU-executed versions of two modern computer vision applications: one a visual servoing algorithm (to control a robotic arm with visual feedback) and the other a sliding-window object detector (to recognize instances of an object class in an image). Starting from slow reference solutions for both algorithms, our goal was to achieve as close to real-time speeds as possible so that these state-of-the-art algorithms could be incorporated onto a mobile robotics platform. We used general-purpose computing on graphics processing units (GPGPU) to accelerate the programs as this strategy has already been used to achieve large speed-ups (of factors of ten to a hundred) on other complex problems. We furthermore developed our applications with CUDA (Compute Unified Device Architecture), which is a programming interface developed by NVIDIA Corporation to perform GPGPU on their graphics hardware. In this thesis we first give a detailed introduction to CUDA, and then we describe in-depth how we built the GPGPU versions of the two computer vision algorithms. In both cases our goals of real-time execution were achieved, seeing end-to-end speed-ups of around fiftyfold.
# Table of Contents

Abstract iv

1 Introduction 1

2 Introduction to CUDA 4

2.1 Kernels ........................................... 5

2.1.1 Two-Dimensional Thread Blocks ...................... 6

2.1.2 Constraints on Kernels ................................. 7

2.2 Two-Tiered Parallelism ................................. 8

2.3 Memory in CUDA ................................... 11

2.3.1 Kernel Memory Hierarchy ......................... 12

2.3.2 Efficient Memory Access .......................... 14

2.4 Taxonomy of Function Classes ....................... 15

2.4.1 One-to-One Functions ............................. 16

2.4.2 Endogenous Functions .............................. 17

2.4.3 Exogenous Functions ................................ 19

2.5 Common Blocking and Gridding Strategies .......... 23

2.5.1 Linear Operations ............................... 23

2.5.2 2-D Operations ................................. 23

2.5.3 2-D Exogenous Operations .......................... 25

3 Visual Servoing 27
4.3.1 Efficient Calculation of NCCC ......................................................... 77
4.3.2 Loop Unrolling and Templating ....................................................... 80
4.4 Integral Images .................................................................................... 85
  4.4.1 Integral Image Precision Errors ......................................................... 87
  4.4.2 Block-Wise Integral Images ............................................................... 89
  4.4.3 Row-wise Integral Images ................................................................. 92
  4.4.4 Integral Image Benchmarking ........................................................... 94
4.5 Conversion of Max-Pooling ................................................................. 95
4.6 Conversion of Classifier Evaluation ....................................................... 98
  4.6.1 Individual Decision Tree Evaluation ................................................ 98
  4.6.2 Decision Tree Ensemble Evaluation ............................................... 100
  4.6.3 Ensemble Evaluation ..................................................................... 103
4.7 Benchmark Results ............................................................................... 103
4.8 Conclusion .......................................................................................... 105

5 Conclusion .......................................................................................... 107

List of References ...................................................................................... 110
List of Tables

4.1 Preliminary SVL benchmarks for small patch dictionaries. . . . . . . . . . . . . . . 73
4.2 Average runtimes over all patches for convolutions with different styles of templating. 83
4.3 Benchmarks of different 7-by-7–window max computations. . . . . . . . . . . . . . . 97
4.4 Benchmarks of Table 4.3 redone using the \texttt{fmaxf} function. . . . . . . . . . . . . 98
List of Illustrations

2.1 Using a single block of threads running an internal serial loop to process the full input data range. The present focus of the block is highlighted in orange, with its position moving to others as the threads loop. ................................................................. 9

2.2 Diagrams of the values that threadIdx.{x,y} and blockIdx.{x,y} take for every thread over a 64-by-64 input with a 16-by-16 block and a 4-by-4 grid. The boundaries of the 16-by-16 blocks are shown in white. ................................................... 10

2.3 CUDA memory hierarchy. Local memory is the fastest for a thread to use, and is accessible only by itself. Shared memory is accessible by all threads within a block, and is the fastest way to share data between threads, though it does not persist beyond the lifetime of the block. Global memory is accessible by all threads and carries the highest latency. ................................................................. 13

2.4 Intermediate representation of input–output dependence graph (IODG) for a 16-by-16 element-wise product. Inputs are shown as black nodes, outputs as red nodes, and data dependence as edges. When converted to a full IODG, the black nodes disappear (as we conflate all input and output nodes), as do all of the edges which would have been self-loops. This creates the edge-less graph of a one-to-one function. .............. 17

2.5 IODG for 4-by-4–block DCT over a 16-by-16–pixel input, an endogenous function. (a) shows the 3-D intermediate view with input nodes in black, output nodes in red, and data dependence edges in blue. (b) shows the 2-D IODG with only output nodes and edges. The 4-by-4 subsets of $G$ that satisfy the conditions of endogeny are evident. 19

2.6 IODG for convolution of a 16-by-16 input with a 4-by-4 template, anchored at the upper left pixel. There exists no strict subsets of $G$ that are self-contained, illustrating the problem of exogenous functions. ................................................................. 21
2.7 Modified IODG for convolution of a 16-by-16 input with a 4-by-4 template, showing invalid as a green node. ................................. 22

2.8 Example of thread overrun with a 2-D input when the data size is not a multiple of the block size. ................................. 24

2.9 Two approaches to computing exogenous output as shown with convolution. (a) shows the case of reading in data only over the range of the block \( b_w \)-by-\( b_h \), and computing a restricted output range (green) based on the template size (blue). (b) shows reading in enough data \( (b_w + t_w - 1')\)-by-\( (b_h + t_h - 1') \) to compute an output for every thread. 26

3.1 A typical scene containing the robot arm. .................................................. 29

3.2 (a) shows a typical synthetic image of the arm at a certain pose defined by some configuration parameters \( \theta' \), and (b) shows a red–cyan color composite of the synthetic image with the scene of Figure 3.1. The visual servoing algorithm localizes the arm’s true location by finding the synthetic pose that best matches the given scene. The pose considered here is a poor match. ................................. 30

3.3 Blurred edge maps of scene and synthetic pose images. ................................. 31

3.4 A red–cyan color composite of the soft edge maps from the scene and the pose images. (a) shows the full images while (b) shows the detail on the robot arm. ................................. 32

3.5 Close-up of the synthetic robot arm model. .................................................. 32

3.6 Three-kernel Sobel implementation. The left-side figures correspond to computing \( H_- \), and the right-side to \( H_+ \). First operation: column-wise convolutions that produce two temporary results from the input. Second: row-wise convolutions on each of the temporaries. Third: combining both with a quadrature add to achieve the soft edge map. ................................. 38

3.7 Graphical depiction of the parallelized summation algorithm. The original range of input \( x \) is shown in green. In the first iteration, \( n/2 \) threads (with their temporary data shown in cyan) “fold” the data range by adding the first and last halves of it together. At successive iterations, the upper half of the threads exits (turn grey) and the remaining half adds the “abandoned” temporaries to its own sums. In the final iteration, the full sum of the input range is achieved in the first element (orange). 42
3.8 Row-wise strategy for applying the parallelized summation algorithm to sum up a \( w \)-by-\( h \) image (in green). The first call to the kernel (“1”) is gridded 1-by-\( h \) and sums up each row, outputting \( h \) elements in another linear stretch. The second call to the same kernel (“2”) needs only one block to sum up the row of intermediate results to achieve the result (orange).

3.9 Percent time spent in each segment of the CUDA pipeline at this stage of implementation.

3.10 Overlaid poses using color channel multiplexing. (a) shows the image used in computation while (b) gives the same against a black background for better visibility of the multiple poses.

3.11 Motivation for merging the multiple, finely shaded triangles of the synthetic arm model into simpler equivalent shapes so as to boost rendering speed. The model of (a) requires rendering 50 interpolated-color triangles, but the model of (b) requires drawing only one flat-color quadrilateral—a much faster task for OpenGL. The model of (b) should produce nearly the same score as (a) after being put through the visual servoing pipeline, so this detail reduction should not effect results.

3.12 Greedy agglomerative algorithm illustration. (a) shows the beginning set \( Tris \). (b) shows turning one triangle into the first entry of \( Polys \). (c) shows selecting all neighboring triangles of the current polygon, with the shared points highlighted. (d) shows selecting triangles that are normal-wise aligned with the polygon: the green triangles are passed and the gray triangle is not. (e) shows the updated polygon. While triangles are still being added to the current polygon, the algorithm loops from (e) to (c). Else it loops back to (b) and converts the next free triangle to a polygon, proceeding until all triangles have been consumed.

3.13 Triangle fan–style rendering. (a) shows a polygon properly drawn when the vertices are provided in the correct order, creating a “fan” of contiguous triangles. (b) shows the same fan with the order of the 0th and 3rd points flipped. The shape becomes distorted.
3.14 Specialized merging of triangles into polygons. (a) continues the polygon-growing example of Figure 3.12. In (b) we detect two new triangles to merge. In (c) we show that we must add the green point in order to incorporate the upper triangle, but we must remove the orange point to incorporate the lower triangle. (d) shows the new shape and its vertices. Some of these points will be valid to use as the “anchor point” and some will not.

3.15 Sample results of the greedy agglomerative procedure. (a) shows the original model while (b) shows one of the reduced-detail models. While the smooth, specular regions of (a) contrast with the matte, patchy regions of (b), the essential outlines and color scheme of the reduced-detail model are unperturbed.

3.16 Change in the element-wise product results per pose with reduced-detail models. The $x$ axis shows the pose number while the $y$ axis plots the score (the axis bounds are 720–810). The top blue line in the plot represents the results from the original model, while the lower lines represent those from the reduced-detail versions.

3.17 End-to-end time spent in OpenGL rendering with the original and the reduced-detail models. The bottommost bar shows baseline of $\approx7$ s, with all other models taking significantly less time.

3.18 Exaggerated examples of two-by-two tiled multi-pose processing. (a) shows a non-color-multiplexed image with four poses per frame, and (b) shows a color-multiplexed image with sixteen poses per frame. For greater detail the arms are shown larger than they actually appear in the real images (and also with black backgrounds).

3.19 End-to-end processing time for the CUDA visual servoing pipeline with all optimization variants. The left side figure shows times with the original arm model; the right side with the reduced-detail model. The columns correspond, in order, to the full-scale, no-CMUX version; the full-scale CMUX version; the half-scale, no-CMUX version; and the half-scale CMUX version (illustrated by the glyphs beneath).

4.1 Example of sliding-window object detection. A window is moved over the entire image frame, extracting sub-windows that are presented to the classifier.

4.2 An example sub-window being presented to the classifier (a) and an example image patch (b). For illustration’s sake, the patch was drawn from the test image (location shown with a green rectangle).
4.3 Normalized cross-correlation coefficient result and max-pooling result. (a) shows the NCCC response image; (b) the response after max-pooling; and (c) is a red–cyan color composite of (b) with the negative mug image, showing the strong response around the area from where the original patch was extracted.

4.4 Schematic of an example decision tree. Each decision node switches on a value from the feature vector, \( x \). Leaf nodes return probabilities of the classification result.

4.5 Axis-aligned–linearly separable data being classified by a decision tree. The decision tree can achieve near-perfect classification with only one decision node.

4.6 Non–axis-aligned–linearly separable data being classified by a decision tree. Each sub-figure shows a different model, with the depth of the tree model increasing by one from left to right and from top to bottom. The tree is obviously making successive approximations to the boundary, but, by depth 6, its error is still over ten times worse than in Figure 4.5.

4.7 An image with generalized “channels.” (a) is an intensity (grayscale) image; (b) is a soft edge map; and (c) is a depth image.

4.8 Benchmarking results of variously templated versions of the normalized cross-correlation coefficient calculation kernel of Section 4.3.1. All runtimes are given in milliseconds, and all color scales are identical.

4.9 Illustration of integral images. The heat map shows the integral image (non-decreasing in value from left to right and top to bottom). To calculate the sum of the original image over region IV, the four pixels shown are added as (4) – (1) – (3) + (2).

4.10 Errors when computing integral images with single-precision floating-point numbers. The top row pertains to integral images and the bottom row to squared integral images. The left column shows ground truth results with double-precision. The middle column shows the absolute error versus the single-precision–based integral images. The right column is a thresholded version of the middle column, showing correct pixel values in blue with incorrect values in red.

4.11 Errors in the patch feature response image (NCCC result of (4.1)) resulting from the use of single-precision numbers for the integral images. (a) shows the ground truth feature values; (b) the values from using single-precision integral images; (c) the absolute error between the two images.
4.12 Sample block-wise integral image showing for integrated blocks. To compute the sum of the dotted-outlined image region requires computing a convoluted sum between the nine labeled pixels. 

4.13 How selecting the block size in block-wise integral images can affect whether error is carried over into the result. This uses the integral error images of Figure 4.10. 

4.14 Example row-wise integral image over a non-random 640-by-480–pixel input with a block size of 256. 

4.15 Benchmarking of NCCC functions, (a) without integral images, and (b) with. (c) shows the relative speed-up from (a) to (b). While the smallest patch sizes can be processed faster without integral images, they help significantly on the larger patch sizes. 

4.16 Scheme for evaluating the decision trees on all feature vectors. Each thread of a block computes results for a certain tree, each block consumes $k$ feature vectors. The grid size $g$ is chosen so that $gk \geq m$. 

4.17 Runtime for decision tree trial versus number of per-block iterations, $k$. 

4.18 Speed-up over the reference CPU solution for the final version of the CUDA sliding-window object detector. The results were obtained on two different GPUs with four different patch dictionaries (for four different objects).
Chapter 1

Introduction

In this thesis we discuss how we applied general-purpose computing on graphics processing units (GPGPU) to significantly speed-up software implementations of several state-of-the-art computer vision algorithms. Our goal was to achieve speeds as close to real-time as possible so as to facilitate these algorithms’ being incorporated onto a robotics platform, as this would greatly enhance the perceptive power of the robot. The two problems we discuss in detail are visual servoing and sliding-window object detection. Working from reference solutions that were optimized for CPU execution, we developed new GPGPU versions that significantly out-performed the originals.

GPGPU is the use of a computer’s GPU to perform generalized computation instead of the strict rendering work for which it was designed. The advantage of running general applications on the GPU is to exploit its unique hardware design that is highly parallelized compared to a CPU, focused, as it is, on processing data at high throughputs rather than caching and controlling it [1]. Modern GPUs have the ability to create hundreds of threads with low overhead and execute them all in parallel. Meanwhile modern consumer CPUs yet typically have only two, four, or eight processing cores, and even those cannot focus on one task exclusively as they must periodically service other system processes. Furthermore, the raw performance of GPUs is increasing faster than that of CPUs, both in terms of computational capabilities and memory bandwidth [2].

While speeding up the execution of a program may be regarded as a matter of convenience, it can be what brings high-level theories into the domain of real-world application. In the field of computer vision, some of the current hard problems include object detection, scene segmentation [3], and 3-D reconstruction [4]. Yet the algorithms that achieve the best performance in these arenas typically
take minutes to process a single image, built as they are on increasingly complex constructions (such as Markov Random Fields [5], [6]). While such runtimes are not too debilitating in the research realm, they are precisely what keep such state-of-the-art algorithms from being used on a modern-day robotics platform. While computer vision is attempting to recreate the major abilities of human sight, robotics is also attempting to replicate the major abilities of human motion, navigability, and perception. Fusing human-level vision onto a human-capable robot is a natural step, and a coveted desideratum. Yet vision algorithms that take minutes to run would be of little practical use to a robot hoping to achieve real-time perception. This was the case with the reference visual servoing and object detection algorithms we were provided with: they both performed a crucial perceptive task for a robot, yet they did so at speeds over an order of magnitude away from being real-time.

We chose to use GPGPU to optimize these reference solutions not only because of the GPU’s raw advantages over the CPU, but because the computer vision and image processing tasks we were to convert (convolution, edge detection, etc.) have already proven amenable to acceleration in this realm [7]. This is particularly because these operations are easily parallelizable at the per-pixel level, so the GPU’s hundreds of threads and brute-force arithmetic power make it much faster at performing them than the CPU. While a function having such characteristics is a good sign that large speed-ups are possible with GPGPU, we will show that achieving the largest performance gain relies heavily on optimizing with respect to the specific application being converted, and the strengths of the specific GPGPU application programming interface (API) being used.

For building our GPGPU applications, we chose to use NVIDIA Corporation’s CUDA API. CUDA is a powerful tool because it exposes nearly full, C-style programming while providing a simple yet powerful structure for controlling a function’s parallel execution [1]. CUDA strikes a balance between giving the programmer low-level control over what code is written versus providing high-level mechanisms to handle the parallel scheduling, which is the crux of the GPU’s speed-up.

It is because of this power that CUDA has been applied to speed-up many computational intensive tasks with great success, and in a wide number of fields outside of computer vision, including molecular dynamics, medical imaging, and fluid dynamics [8]. CUDA conversions have achieved speed-ups of nearly a hundredfold on simple procedures (such as convolution) and of tenfold even with complex algorithms. NVIDIA itself has published code samples of how to use CUDA to speed up separable convolution [9], the Discrete Cosine Transform [10], FFT-based 2-D convolution [11], and image denoising [12]. In the research realm, CUDA has been applied to the maxflow/mincut problem [13], the Canny edge algorithm [7], and the singular value decomposition [14]. In machine
learning, CUDA has been used to dramatically speed-up the training of support vector machines [15] and of deep belief networks [16]. Perhaps the most public example of CUDA’s abilities comes in how it now provides almost half of the computational muscle of Folding@Home, the most powerful distributed computing project in the world with a sustained output of 5 PetaFLOPS—over three times that of the world’s fastest supercomputer [17].

In the first section of this thesis we provide a detailed introduction to the CUDA API. We also examine concepts and programming strategies that will find important application when we discuss how we converted the reference solutions to use GPGPU. The next two sections cover each of our conversion efforts for the two computer vision problems: visual servoing and sliding-window object detection.
Chapter 2

Introduction to CUDA

In this chapter we provide an introduction to the CUDA programming environment. We first present a walkthrough of simple code examples, explicating CUDA functionality and terminology as it appears. This is standard tutorial material that appears in the CUDA documentation itself (most notably in the Programming Guide [1]), but it is presented here for the sake of completeness. In later sections, we will develop a novel taxonomy of function classes, which will be useful to describe how amenable certain functions are to CUDA implementation, and dictate the best form for their conversion. We finish by offering further code samples that will facilitate discussion of the actual CUDA functions used in the chapters on converting the computer vision algorithms.

The GPU was originally designed for rendering graphics to the screen, and traditionally the programmer could access the GPU’s resources only in the indirect manner of forming a request for some rendered image [18]. Graphics libraries such as OpenGL provide the user with a large set of high-level constructs for making such requests [19], but the user is essentially limited to specifying the makeup of a 3-D world and asking the GPU to render a specific viewpoint of it. While the user has great power to define that 3-D world, they must provide the GPU with a geometric description of it as input, and the end result of their request must be a rendered image. They have little ability to control what instructions are actually executed on the GPU.

Using the GPU to instead do general-purpose programming is made possible by a new application programming interface (API) that provides more-direct, low-level access to the GPU. Instead of being restrained to inputting the description of a 3-D world and receiving back a 2-D image, GPGPU allows the user to treat the GPU essentially as a CPU. They provide input data and instructions, execute
the instructions to modify a persistent machine state, and can examine the resulting data [2].

However, the design of a GPGPU API must strike a balance between how “CPU-like” it makes the interface to the GPU versus how GPU-specific the interface is—i.e., how much of the GPU’s power it allows to be tapped. On the one hand, if the API exposes the GPU in exactly the same manner as the CPU, then code can easily be cross-compiled on the CPU or GPU without any alteration. But this will not expose the true power of the GPU—its massively parallel structure—, and so the API will not be useful for achieving large speed-ups. On the other hand, if the API exposes the exact Assembly-level functionality of the GPU, one will be able to fully harness its power, but it will be exceptionally more difficult to do so.

CUDA strikes a balance between these two extremes. It is built on the C programming language, so code can be written on the CPU using the same general style as for a CPU. CUDA adds only a few constructs and keywords that control how to execute the code on the GPU. These constructs are high-level and leave most of the detailed scheduling to the GPU; yet they expose a powerful and flexible interface to tap the GPU’s parallelism.

2.1 Kernels

To motivate our introduction to CUDA, we present the simple problem of computing the element-wise product of two 128-element vectors. On a CPU, this is easily accomplishable with a loop as shown below:

```c
for ( int i=0; i<128; ++i )
C[i] = A[i] * B[i];
```

While there is nothing wrong with this serial solution, it is obvious that each of the 128 additions could be executed in parallel, as none of the operations conflict with each other (that is, none of the individual operations alter data that any of the others depend upon as inputs).

The way to parallelize this element-wise product in CUDA is to write a “kernel.” This is a special type of function that will be executed in parallel by multiple threads on the GPU [1]. First we must specify the body of the function, and then we must specify how many threads we want to execute it (i.e., how many copies of the function we want run in parallel). A kernel that performs this element-wise product is given below:

```c
__global__ void element_wise_product(float *A, float *B, float *C) {
    const int i = threadIdx.x;
```
(2.2) introduces a great deal of CUDA syntax. First, functions that are to be executed on the GPU must have their definitions preceded by the `__global__` keyword (there are other possible keywords—`__device__` being another—but which can only be used under specific circumstances). A kernel is also called with a new `<<<...>>>`-based syntax that mimics C-style templatization (e.g., `foo<int>(bar);`). The triple bracket allows the user to specify the layout of threads that will execute the kernel. The two arguments it contains specify the “grid dimension” and the “block dimension,” respectively (this two-tiered structure will be discussed in detail in Section 2.2). For the moment, only the block dimension (given as 128) is important, as this specifies the number of threads that will execute this function call simultaneously. Threads that run simultaneously are said to constitute a “block.” There is a maximum number of threads that can be put in a single block, which is GPU-dependent (typically 256 or 512 [1]). And there is an additional constraint (in some versions of CUDA) that the number of threads requested must be a multiple of 16. The number of threads here (128) satisfies both of these constraints.

Since the function `element.wise.product` is identified as a kernel by the `__global__` keyword, it is granted access to several important CUDA variables. The one used here is the `threadIdx` variable, which is a three-element vector with members `x`, `y`, and `z`. Because the multiple threads executing this function will all be running the same code, this variable lets them distinguish “which” thread they are and so differentiate their work. Each thread will have a different value for `threadIdx` that identifies its location in the block of threads. In (2.2), by specifying 128 as the block size, the GPU will allocate a one-dimensional stretch of 128 threads. These threads will have individual `threadIdx.x` values ranging from 0 to 127, while their `.y` and `.z` values will all be 0. Each thread uses this variable to determine which index of the result it is to compute (line 3 of (2.2)), and so the entire sum is computed in parallel.

### 2.1.1 Two-Dimensional Thread Blocks

We specified a one-dimensional thread block above, but, since `threadIdx` is a three-element vector, we can specify up to three-dimensional thread blocks. While a one-dimensional thread block is ideal
for accessing linear memory, 2-D blocks greatly facilitate the task of operating on matrices. 3-D blocks further generalize to fields and multi-channel images.

Below we give a simple example of using a 2-D block. Consider the case of computing the element-wise product not between two 128-element vectors but between two 16-by-16-element matrices. A kernel to accomplish this is given below:

```c
__global__ void element_wise_product(float A[16][16], float B[16][16],
float C[16][16]) {
  const int i = threadIdx.x, j = threadIdx.y;
  C[i][j] = A[i][j] * B[i][j];
}

int main() {
  dim3 blockDim(16, 16);
  element_wise_product<<<1, blockDim>>>(A, B, C);
}
```

The great change here from (2.2) is that, instead of specifying the block dimension (inside of the triple brackets) with an integer—implying 1-D allocation—, here we specify a variable `blockDim` of the CUDA-provided type `dim3`. The declaration of `blockDim` on line 7 specifies first the $x$ dimension and the $y$ dimension sizes of the block. Inside the kernel, we simply access the $x$ and $y$ members of `threadIdx` to find the two-dimensional coordinates of the thread. Thereafter the computation is as direct as that of the vector product above in (2.2).

### 2.1.2 Constraints on Kernels

While we have discussed some of the special variables visible only inside of kernels, these functions also operate under some constraints relative to plain C functions. First, they must all be of return type `void`. This should be clear as, with multiple parallel-wise-executing copies of the function being launched by one call from the CPU, how would the CPU interpret the return type `int` from every thread? A kernel must do all of its output through modifying the values of variables passed in as pointers (or through the modification of global pointers).

Kernels also cannot declare static variables as these involve state persisting beyond allocation of the block of threads. They also do not support recursion, as this would involve pushing arbitrary number of frames onto the stack. Additionally, kernels can only do stack allocation, so there is no heap for doing dynamic allocation. There are other restrictions that kernels operate under (such as not supporting variable arguments), but we have already given the most crucial ones from an
algorithmic perspective.

Those restrictions aside, kernels are otherwise the same as C functions, and so can contain general
math, pointer arithmetic, and all other operations necessary to implement complex algorithms.
Compared to learning an entirely new programming language to do GPGPU, learning to write C
inside of these novel constraints presents a much faster learning curve.

2.2 Two-Tiered Parallelism

Two-dimensional thread blocks make it far easier to write kernels that work on matrices and images,
but we mentioned above that NVIDIA GPUs have a limit on how many threads can be executed in
a single block. While this limit is currently increasing [1], it is typically bounded by 512. This makes
it impossible to compute the element-wise product between two 64-by-64 matrices, for example, with
a simple kernel as we have seen so far, as this would require over four thousand threads.

One way around this is to admit the limits on parallelism and introduce a serial element into
the kernel. For example, say our maximum thread count is 256, limiting us to a 2-D block size of
just 16-by-16. To perform a 64-by-64 matrix element-wise product, then, we would have to write
a double loop inside of our kernel, having the block of threads compute one 16-by-16 sector of the
result at a time before jumping to the next until the entire result is computed (this logic is illustrated
in Figure 2.1). A great portion of the computation is still done in parallel (256 elements of the result
computed at a time), but a 4-by-4 serial loop (16 iterations) is required to compute the full result.

While this internal loop is a simple solution here, in general it is not scalable. If we hard-code
the block size and loop dimensions into the kernels, then we are being short-sighted. If we later
acquire a GPU with a maximum thread count of 1024, then we could use a block size of 32-by-32
threads, and we would need only perform a 2-by-2 serial loop inside of the kernel. Using the old
16-by-16-thread block would be four times slower than necessary on the new GPU. To avoid this,
we could write our kernels such that we set the block dimension dynamically depending on what
GPU architecture we have, thus maximizing its benefits. But this adds complexity and brittleness
to the development cycle.

CUDA itself provides a better way to surpass the maximum thread count with the “grid dimen-
sion.” After we specify the size of a single block, we specify with the grid how many times we want
to replicate (or tile) that block in up to two dimensions. The grid dimension is the first number
Figure 2.1: Using a single block of threads running an internal serial loop to process the full input data range. The present focus of the block is highlighted in orange, with its position moving to others as the threads loop.

provided to the triple bracket syntax first shown in (2.2). Like the block dimension, it is also specified in the form of a `dim3` variable (though the third variable is here vestigial as grids are limited to being only one- or two-dimensional).

Applying this to the 64-by-64 element-wise product above, instead of putting an explicit serial loop inside of our kernel, we use the grid to define the dimensions of the loop needed to complete the computation. For a block size of 16-by-16, the grid size would be 4-by-4 (the same size as the serial loop we previously needed). For a block of 4-by-64, we need a grid of size 16-by-1. We give an example kernel below for the 16-by-16 block case:

```c
__global__ void big_product(float A[64][64], B[64][64], C[64][64]) {
    const int i = blockDim.x * blockIdx.x + threadIdx.x;
    const int j = blockDim.y * blockIdx.y + threadIdx.y;
    C[i][j] = A[i][j] * B[i][j];
}

int main() {
    dim3 blockDim(16, 16);
    dim3 gridDim(4, 4);
    kernel_function<<<gridDim, blockDim>>>(A, B);
}
```

The new CUDA variables here are `blockDim` and `blockIdx`. `blockIdx` functions analogously to `threadIdx`, only it tells the thread the index of its block inside of the grid (instead of its location
inside of its block). With a 4-by-4 grid, all of the blockIdx.x and blockIdx.y members will take on values from 0 to 3. This is fully illustrated in Figure 2.2, which shows the values that threadIdx and blockIdx take for each thread that will be executed over this 64-by-64 range. The variable blockDim, meanwhile, simply contains the block size (here 16-by-16). With these three variables we can compute each thread’s two-dimensional index inside of the full computation. There is a fourth variable, gridDim, not used in this example, which also stores the dimensionality of the grid (here 4-by-4), which is useful in some cases.

Figure 2.2: Diagrams of the values that threadIdx.\{x,y\} and blockIdx.\{x,y\} take for every thread over a 64-by-64 input with a 16-by-16 block and a 4-by-4 grid. The boundaries of the 16-by-16 blocks are shown in white.

The first advantage of using grids over serial loops is that we do not need to put serial code inside of our kernels. The greatest advantage, however, is that, unlike using a serial loop, the different blocks of a grid need not be executed serially. In general, the GPU schedules the different blocks to run as quickly as possible. Multiple blocks can actually be run concurrently so long as the maximum thread count between them does not violate the GPU maximum.

For example, on a GPU with a maximum thread count of 256, running the above example with a
16-by-16 block would mean that only one block could be executed at a time, so 16 parallel executions of the block would still be needed to cover the full 64-by-64 data range. Using grids here might be no faster than writing a loop inside of the kernel. But, on a GPU with a maximum thread count of 1024, up to four 16-by-16 blocks could be executed simultaneously, meaning only 4 serial executions would be needed. This speed-up manifests itself as soon as we compile our code and run it on a newer GPU, which comes gratis from the programmer’s perspective (requiring no updating of the code as would be the case with the explicit serial loop).

One contingency in grid-based programming is that, since the different blocks of a grid can be executed in serial or in parallel, in general the programmer is given no guarantees on the order in which the blocks will be scheduled. The GPU will not iterate through blocks in any deterministic fashion (say, starting at the upper-left block and proceeding in increasing linear index). Thus, when programming with grids, the kernels must make no assumption on which parts of the result may have been computed before it or which may be computed later. The only guarantee each kernel is given is that all of the threads in its block will execute in parallel, but every block could be either the first or the last executed.

Blocks and grids define the two-tiered structure of parallelism that CUDA exposes. The lower “block” level specifies the minimum number of threads that will execute in parallel at a time, while the higher “grid” level specifies how many copies of the inner computation must be replicated to cover the full result. But this replication of blocks can be done in serial, in parallel, or in a mixture of both. It is through this powerful yet accessible structure that CUDA allows the programmer to write parallelizable computations of arbitrary size.

2.3 Memory in CUDA

Kernels can only access memory that is explicitly allocated on the GPU. CPU-side (or “host-side”) data cannot be accessed by kernels unless space is allocated for it on the GPU and it is manually transferred over. Similarly, CPU-side applications cannot access data on the GPU unless they explicitly allocate space for it and have it transferred back.

In this section we provide an overview of the types of memory that are accessible to a kernel, with discussion of their relative strengths and weaknesses. We highlight the implications that these different classes of memory have for kernels and for algorithm design, and describe the most efficient “styles” of memory access for each.
2.3.1 Kernel Memory Hierarchy

There are three types of memory accessible from with kernels, differentiated by their levels of latency and their inter-thread accessibility. The first, most general type is “global memory,” which—as its name implies—is read–write accessible by all threads. But, because of its flexibility, global memory is also the slowest, taking hundreds of GPU clock cycles for a single transaction to resolve [1]. The pointers passed to a kernel through its arguments will all refer to global memory. In fact, all the example kernels given above used global memory to do their input and output.

The next, fastest type of memory for a thread to access is its “local memory,” which contains all of its stack-allocated variables and temporaries. It is accessible only to the thread that created it, and not to any other threads, including those within the same block.

The last type of memory is a middle ground global and local memory. This is called “shared memory” because it is read–write accessible to all of the threads within a block, but it does not persist past the lifetime of the block. It has much lower latency attached to it than global memory, and, as all threads within a block can see it, it is the best way to share data between threads in case the algorithm requires such inter-thread cooperation (as can often the case, as we will discuss in Section 2.4). A graphical illustration of CUDA’s memory hierarchy is given in Figure 2.3.

While the roles of local and global memory are clear, shared memory proves critical in kernels where the threads are reading many of the same locations from global memory [1] (such as when they are doing convolution and the image lies in global memory). Instead of having all threads read global memory whenever they need a value, often the fastest solution will be to have all of the threads cache the global memory into shared memory, and thereafter work from the shared version of it.

An example of using shared memory is given below. Say, for a 16-by-16 matrix, we want to compute, for each pixel, how many of the pixels in the same row have a value greater than the current pixel’s value. A simple kernel to do this—that works directly from global memory—could take this form:

```c
__global__ void greater_than_sum(float *A, int *answer) {
    const int i = threadIdx.y, j = threadIdx.x;
    float myValue = A[16*i + j]; // Read in our value to compare
    int result = 0; // Initialize temporary result to 0
    for (int ix=0; ix<16; ++ix) {
        if (A[16*i + ix] > myValue) {
            ++result;
        }
    }
}
```
Figure 2.3: CUDA memory hierarchy. Local memory is the fastest for a thread to use, and is accessible only by itself. Shared memory is accessible by all threads within a block, and is the fastest way to share data between threads, though it does not persist beyond the lifetime of the block. Global memory is accessible by all threads and carries the highest latency.

But every thread accessing global memory in the inner loop with $A[16i + ix]$ at line 6 is terribly inefficient. The better option is to first have all of the threads read in one value of the matrix $A$ (as was done in line 3), save this to shared memory, and then read from shared memory inside of the loop:

```c
__global__ void greater_than_sum(float *A, int *answer) {
  __shared__ float data[256];
  const int i = threadIdx.y, j = threadIdx.j;
  float myValue = A[16*i + j];
  data[16*i + j] = myValue; // Save our result to shared memory
  __syncthreads(); // synchronization barrier
  int result = 0;
  for (int ix=0; ix<16; ++ix) {
    answer[16*i + j] = result; // Save result back to global memory
  }
}
```

(2.5)
if ( data[16*i + ix] > myValue ) { // Read from shared memory
    ++result;
}
}

answer[16*i + j] = result;

(2.6)

Shared memory declarations are identified by the `__shared__` keyword. The specific command on line 2 allocates 256 floats for all of the threads, not for each.

Now that we are using shared memory, the kernel contains two stages. The first is where each thread loads its own value of `A` (line 4) and saves it into shared memory (line 5). The second stage (beginning on line 8) is where the result is computed using the shared data. A new CUDA command must lie between these stages: `syncthreads()`. This is a “synchronization barrier” that ensures that all the threads within the block will converge at this point before any proceed beyond. This is essential because, even though the threads are all executing in parallel, they are not necessarily executing at the same speed. Since each thread is reading in data that other threads will use later on, we must be sure that all the threads have finished saving their data to shared memory before any of them attempt to use it. This is the effect of the synchronization barrier.

### 2.3.2 Efficient Memory Access

Because of the high overhead of accessing global memory, CUDA does a great deal behind the scenes to make reads to global memory efficient. In particular, it attempts to “coalesce” multiple reads to global memory from multiple threads into as few instructions as possible. As an example, CUDA can coalesce simultaneous reads by 16 threads into only a single instruction so long as the threads access a 128-byte segment of memory as four-byte words [1] (such as single-precision floating-point numbers). Reads are also coalesced for one-byte words when 32 bytes of memory are addressed at a time (such as `char`s), or 64 bytes at a time for 2-byte words (such as C `short`s).

For the CUDA programmer, this means that the most efficient accesses to global memory will be with all the threads reading and writing sequential locations. All the kernels given above made use of this strategy, whether they were modifying global memory or reading it into shared memory. This is because we used the `threadIdx` variable in all of those functions to structure our reads and writes, and the sequential nature of this variable is conducive to these linear access patterns.
An important application of this fact can be seen in convolution. If an image is stored in row-major order, then a row convolution algorithm will find each row laid out sequentially in memory, and reads to a row will naturally be coalesced. But, for convolving the columns of an image, each successive pixel along a column is separated by a great distance in memory, and so will not benefit from coalesced reads and writes. Typically the way to ameliorate this is to do column convolution with blocks that have widths of at least 16 threads, and so are processing at least 16 columns at a time [9]. In this way, the reads to each row of the area of computation will be coalesced. The trade-off is that, with the block at least 16 threads wide, the height of the block becomes limited by the maximum thread count, and so column-wise operations cannot be done over so many rows as their row-wise counterparts are stretched over columns.

Shared memory, while faster than global memory, also has its own quirks. While we typically only read from and write to global memory once at the beginning and end of a kernel, respectively, shared memory is specifically used when the threads will have complex, interleaving access patterns. If multiple threads attempt to access the same “bank” in shared memory—defined as a unique 4-byte segment starting at a location that is 0 modulo 4 [1]—, the accesses will have to be serialized instead of proceeding in parallel. One can avoid bank conflicts when accessing 4-byte words (such as floats) as long as each thread in a group of 16 (defined as a “sub-warp”) accesses a unique address within the 64-byte range. The more threads that attempt to access the same address simultaneously, the more serialization that must occur, and the greater the slowdown. Thus the programmer must know to avoid bank conflicts when designing algorithms that use shared memory.

2.4 Taxonomy of Function Classes

Having discussed CUDA’s mechanics and its block-grid parallelism hierarchy, we now know the recipe for converting a function into CUDA. However, when approaching how to convert a specific function, merely knowing the mechanics of CUDA can provide little intuition on how to proceed. In this section, we define a taxonomy of function classes that addresses this problem. Over certain types of functions that one may wish to convert, this taxonomy categorizes those in terms of their input–output data interdependence, and generalizes that to the form of the kernel function necessary to implement them, and what degrees of freedom the user has in that conversion. This is a novel taxonomy and not one drawn from the CUDA documentation.

This taxonomy will apply to a subset of functions, specifically those that produce outputs that
are the same size as their inputs (e.g., as with the element-wise products adduced above). The field of image processing contains a large corpus of such functions, including edge detection, the Fourier transform, convolution, etc. These types of functions are often parallelized as typically each of their output values (or pixels) can be computed independently of the other outputs, which hints at the obvious CUDA parallelizing strategy of having each thread compute one value of the output, and then tiling enough threads (with blocks and grids) to cover the entire output range.

Within this “same-size” class of functions, how the output values depend spatially upon the input data directly affects how we can lay out blocks and grids to most efficiently compute the answer. It is over this range that the following taxonomy applies. In particular, we will define three different types of functions: “one-to-one,” “endogenous,” and “exogenous.” These classes roughly correspond to increasing complexity in terms of CUDA conversion, though this is complexity in the asymptotic sense of big O notation, and an “endogenous” function is not always guaranteed to run faster than an “exogenous” function.

2.4.1 One-to-One Functions

We will call a kernel “one-to-one” if, to compute one value of the output, we need only one value from the input, and specifically the value that lies at the same corresponding location as the output. In the case of multiple inputs (such as with a sum), we require that the output need only one value from each of the inputs, all of which must lie at the same location as the output. An example one-to-one function is an element-wise product between two vectors.

To establish this more rigorously, we first define what will be called the “input–output dependence graph” (IODG). Imagine the locations of all the output values of a computation as nodes in a graph, \( G \). If an output value at position \( g \) depends upon an input pixel at position \( g' \), then we add an edge between \( g \) and \( g' \) in the grid—though only if the nodes are distinct (i.e., we allow no self loops, since corresponding input and output locations are represented by a single node). This defines the IODG, \( (G, E) \), where \( E \) the set of edges between the nodes of the form \( (g, g') \).

If a function is one-to-one, then the output value at position/node \( g \) depends only on input pixels also at position/node \( g \). Since these are all self-loops, and we allow no self-loops in the grid, this means there are no edges in the IODG for one-to-one functions:

\[
E = \emptyset
\] (2.7)
We show an intermediate form of the IODG for a one-to-one function in Figure 2.4. For clarity we show both input and output nodes to start, as black and red, respectively. Data dependence for one-to-one functions is depicted, with edges only between spatially corresponding inputs and outputs. To convert this to a true IODG, we throw away input nodes, re-route the edges from black nodes to red nodes, and discard self-loops. In this case, the transformation to an IODG leaves only the output nodes and no edges.

![Figure 2.4: Intermediate representation of input–output dependence graph (IODG) for a 16-by-16 element-wise product. Inputs are shown as black nodes, outputs as red nodes, and data dependence as edges. When converted to a full IODG, the black nodes disappear (as we conflate all input and output nodes), as do all of the edges which would have been self-loops. This creates the edge-less graph of a one-to-one function.](image)

This class of functions is the most conducive to CUDA implementation. The strict spatial alignment between outputs and their corresponding inputs means that no data needs to be shared between threads during computation. We can operate directly on global memory without having to cache anything in shared memory, omitting the loading and synchronization stages discussed in Section 2.3.1. We can also pick an arbitrary block size for performing the operation, particularly one that maximizes coalesced memory reads, and that minimizes over-running the edges of the input (such as when the data size is not divisible by any feasible block size).

### 2.4.2 Endogenous Functions

Moving up in complexity from one-to-one functions, we shall define a function as “endogenous” if output values depend on inputs that are not strictly spatially aligned with them—so the function is
not “one-to-one”—but if there exists a block size such that all of the output data within that block’s footprint can be computed using only the input data covered by that same footprint. In regards to the input–output dependence graph, we are now allowing there to be edges between nodes (as output values will depend upon inputs that may not be spatially aligned with them), but we require that there be a subset of nodes (which can be spanned by the thread block) such that, for all of the edges touched by those nodes, they connect only to other nodes also in the same subset. In other words, there must be a self-contained subset of nodes in the graph. Formally:

\[ \mathcal{E} \neq \emptyset \land \exists \mathcal{G}_{\text{block}} \subseteq \mathcal{G} : \mathcal{G}_{\text{block}} \supseteq \bigcup_{(g_1,g_2) \in \mathcal{E}_{\text{block}}} \{g_1,g_2\} \]  

(2.8)

where \( \mathcal{E}_{\text{block}} = \bigcup_{g \in \mathcal{G}_{\text{block}}} \{e \in \mathcal{E} : e \ni g\} \)

\( \mathcal{E}_{\text{block}} \) is the subset of edges touched by any nodes in \( \mathcal{G}_{\text{block}} \). The term \( \bigcup_{(g_1,g_2) \in \mathcal{E}_{\text{block}}} \{g_1,g_2\} \) calculates the set of nodes touched by all of the edges in \( \mathcal{E}_{\text{block}} \), which we will define as \( \mathcal{G}'_{\text{block}} \). The condition that \( \mathcal{E} \) not be empty prohibits one-to-one functions from being considered endogenous (for clarity).

An example of an endogenous function is the Discrete Cosine Transform. In the DCT, the image is partitioned into \( d \times d \)–pixel blocks (where \( d \) is a parameter of the algorithm) and each is subjected to the operation \( D^T \cdot \text{block} \cdot D \), where \( D \) is a \( d \times d \) transform matrix. The output image is formed by concatenating these individual blocks back together. Within one of the DCT blocks, one output pixel depends upon all of the input pixels within the same block (whose values are entangled by the two matrix products). But it is endogenous because each \( d \times d \)–pixel output block does not depend on any values outside of the block.

Figure 2.5 shows the IODG for a 4-by-4 DCT over a 16-by-16–pixel image. Clearly the separable 4-by-4 blocks in Figure 2.5b represent multiple subsets \( \mathcal{G}_{\text{block}} \) whose subtended edges connect to no nodes outside of \( \mathcal{G}_{\text{block}} \). This is advantageous for CUDA conversion as, if we chose a thread block size of \( 4n \)-by-\( 4n \) where \( n \in \mathbb{N}^+ \) then the output values of our kernel will depend only upon the strictly overlapped input values. However, if we set our thread block size to 6-by-6, we would need to read in data from outside of the block (or “footprint”) to compute the block’s output pixels, complicating our kernel.

Endogenous functions are not as conducive to CUDA conversion as one-to-one functions, but they still possess certain advantages. Because they have edges in their dependence graphs, typically
Figure 2.5: IODG for 4-by-4–block DCT over a 16-by-16–pixel input, an endogenous function. (a) shows the 3-D intermediate view with input nodes in black, output nodes in red, and data dependence edges in blue. (b) shows the 2-D IODG with only output nodes and edges. The 4-by-4 subsets of $G$ that satisfy the conditions of endogeny are evident.

we must use shared memory and two-stage kernels to get optimized performance. But, because there exist these convenient, self-contained subsets in the dependence graph, there are also optimal thread block sizes such that each thread will only need to read in one value (from each input) to shared memory. If it is possible to use one of those optimal block sizes, then we can still read in memory in a one-to-one fashion, the only difference being that we inter it into shared memory (and pause with a synchronization barrier) rather than use it immediately as with one-to-one functions.

2.4.3 Exogenous Functions

“Exogenous” functions are the opposite class of functions versus “endogenous,” such that, to compute a certain amount of output data always requires reading in more input data that lies outside of the block footprint. An example exogenous function is convolution. As long as the template used to convolve the image is not of singleton size, then to compute one output value relies on reading in all of the input values subtended by overlaying the template with that output pixel. If we try to compute all of those extra overlaid outputs simultaneously and so achieve endogeny, we must enlarge our input footprint again to cover the extra input values. This proceeds indefinitely, such that there
are no self-contained subsets of the output locations that do not require reading in outside input locations to compute.

Exogenous functions are some of the most difficult to implement in CUDA as, because they involve input–output data dependence (like endogenous function), we must cache the input data into shared memory and use two-stage kernels. And, because there are no self-contained subsets of the nodes, we must always read in more data than what we are going to output. This means some threads will have to read in multiple data points, which is more complicated, slower, and more difficult to manage than the one-to-one read-in fashion of endogenous functions.

When searching for a formal definition of exogeny, we could simply invert the second condition of (2.8), changing the first $\exists$ to a $\nexists$. Though there is an edge case that vitiates this definition. When working on small enough inputs, it is actually possible for an exogenous function like convolution to satisfy the conditions of (2.8). Consider convolving a 16-by-16–pixel image with a 4-by-4–pixel template (the IODG for this is shown in Figure 2.6). The entire input range can be read in with one block (and so place all of $G$ in $G_{block}$), so it is impossible for the edges $E_{block}$ to subtend a graph $G'_{block}$ larger than $G_{block}$. Thus (2.8) is satisfied. However, if we were to formally recognize this as a class of endogeny, it could only be done so as a degenerate case. The endogeny/exogeny dichotomy is meant to capture the idea of ineluctably mismatched input–output data interdependence. This class of exogenous functions circumvents that simply by consuming all of the input data in one pass.

We can restore the divide between endogeny and exogeny in several ways, however. First, we could change the conditions of testing a function for exogeny to consider applying it to an unbounded input while we only have a finite number of threads for our block. This restricts $G_{block}$ to finite sizes, which erases the edge case given above: a finite number of threads can never subsume an infinite amount of input data and so allow the degenerate case of endogeny.

But there is another way to restore the divide that provides more insight into the difference between endogenous and exogenous functions. Consider the edge case above with a small-sized convolution. Even when we were able to read in all of the input data, at the edges of the image the convolution template will still spill off of the image. The algorithm thus needs some way to handle these technically undefined locations. The typical solution is to assume a default value for invalid locations that does not alter the end result of the computation. In convolution, this default value is 0 since multiplying the off-image template pixels with zero adds nothing to the convolution sums. For a max operation, this default value would be $-\infty$. Another way to achieve this effect is to say the algorithm “stops” at edges of the image and does not consider invalid locations, but this
is equivalent to assuming the default value as before and optimizing away the extra computation.

The key idea is that exogenous functions, even when they consume all of the input data, must still know what value to give to undefined locations. So they are still considering input data from a set larger than the set of output locations. This default value corresponds to a hitherto unconsidered node in our input–output dependence graph. Before we envisioned only nodes corresponding to locations in the computation, but now we must add a *sui generis* node that represents the default value of all off-image locations. Even when exogenous operations are run over bounded inputs, they must invariably make use of this default value.

The IODG of Figure 2.6 for convolution is redrawn in Figure 2.7 with this new perspective. A green node, $g_{invalid}$, is added to represent the default value for non-existent locations, and green edges are drawn between it and the nodes that reference off-image values (at the bottom and right of the grid). Now there are clearly no non-edge-breaking subsets of $\mathcal{G}$ (the red nodes).

Now we can soundly restate our definitions of the function classes. We shall not place $g_{invalid}$ in $\mathcal{G}$ (since that corresponds strictly to real input–output nodes), but we will define a new variable $\hat{\mathcal{G}} = \mathcal{G} \cup \{g_{invalid}\}$. With this we define a new graph, $(\hat{\mathcal{G}}, \hat{\mathcal{E}})$, where $\hat{\mathcal{E}}$ is the union of $\mathcal{E}$ with all of the edges between connecting to $g_{invalid}$. The new definitions of one-to-one-ness, endogeny, and
Figure 2.7: Modified IODG for convolution of a 16-by-16 input with a 4-by-4 template, showing $g_{invalid}$ as a green node.

exogeny become:

One-to-one: $\hat{\mathcal{E}} = \emptyset$ \hspace{1cm} (2.9)

Endogenous: $\hat{\mathcal{E}} \neq \emptyset \land \exists G_{block} \subseteq G : G'_{block} \subseteq G_{block} \land g_{invalid} \notin G'_{block}$ \hspace{1cm} (2.10)

Exogenous: $\hat{\mathcal{E}} \neq \emptyset \land \nexists G_{block} \subseteq G : G'_{block} \subseteq G_{block} \land g_{invalid} \notin G'_{block}$ \hspace{1cm} (2.11)

where $G'_{block} = \bigcup_{(g_1, g_2) \in \mathcal{E}'_{block}} \{g_1, g_2\}$ \hspace{1cm} (2.12)

and $\mathcal{E}'_{block} = \bigcup_{g \in G_{block}} \{e \in \hat{\mathcal{E}} : e \ni g\}$

As $G'_{block}$ is taken as the nodes subtended by edges in $\mathcal{E}_{block}$ (which is drawn from $\hat{\mathcal{E}}$), $G'_{block}$ can indeed contain $g_{invalid}$. Endogenous functions, however, will never need to consult such a default location as they consist of completely spatially self-contained operations. Exogenous functions, however, will not be able to construct a self-contained set $G_{block}$ from $G$—even by consuming all of $G$—because their algorithms eventually force them to appeal to $g_{invalid}$, which lies outside of $G$. 


2.5 Common Blocking and Gridding Strategies

We have now discussed a great deal of CUDA infrastructure, examining the major facets of its performance, and even considered the implications of general algorithmic behavior on kernel design. Now we apply this to discuss common strategies for organizing blocks and grids over data. These patterns are exemplified by NVIDIA’s own CUDA SDK code samples, and we present these low-level, tutorial-worthy ideas here that they need not be addressed when we discuss the actual kernels written for the computer vision algorithm conversions.

2.5.1 Linear Operations

Processing a linear array is the simplest case of parallelism in CUDA. With a one-to-one function, we typically pick the thread limit of the GPU as our block size, \( d_{\text{block}} \), and tile that sufficiently with the grid to cover the whole input. If the array has \( n \) elements, for example, then \( d_{\text{grid}} = \lceil n/d_{\text{block}} \rceil \).

The only issue is when the array size \( n \) is not an exact multiple of the block size \( d_{\text{block}} \), or of any feasible block size. This will be a problem for the block that operates on the tail end of the array, as there will be threads that will lie off of the valid range of data still running computation, possibly corrupting other memory. The most direct solution is to pass the array size \( n \) to the kernel and have each thread check to see if it lies in the bounds of the data before proceeding:

1. \_\_\_global\_\_ void foo(float *A, float *B, int n) {
2.  const int ix = threadIdx.x + blockIdx.x * blockDim.x;
3.  if ( ix < n ) {
5.  }
6. }

(2.13)

The bounds-checking could also have been written as \( \text{if ( } \text{ix} \geqslant n \text{ ) } \text{return} \);, forcing an early return of the current thread while the rest of the block keeps executing. This is allowed in CUDA, even if early thread termination is not how the GPU explicitly implements this syntax.

2.5.2 2-D Operations

In matrix and image processing, one-to-one operations are the most straightforward to map to CUDA implementations. The element-wise product is an example of such an operation from the matrix domain, and binary thresholding is an example from image processing. As discussed before with
one-to-one functions, one is at liberty to structure their blocks and grids however they wish to cover the whole image. Of course one should use a block $x$ dimension size that ensures coalesced reads and writes to global memory (so a dimension that is ideally a multiple of 16).

In image processing applications, for algorithmic simplicity we commonly choose blocks of size 16-by-16. This makes the tiling over the image easy to visualize, and the size of the grid is easy to compute: $d_{grid} = \lceil \frac{image.size}{d_{block}} \rceil$, where the variables are all two-dimensional vectors. But, as before with linear indices, we must specifically check for when threads are overrunning the bounds of the image. This case is illustrated in Figure 2.8. Checking for overrun is especially crucial in image processing since images are stored in row-major order: overrunning the end of the row will start editing the next row. This bounds checking can be accomplished as the 2-D analogue of the checking done above with linear arrays—with passing the image dimensions to the kernel and checking as the first step of the function:

```c
__global__ void foo(float *img, int img_w, int img_h) {
    const int x = threadIdx.x + blockIdx.x * blockDim.x;
    const int y = threadIdx.y + blockIdx.y * blockDim.y;
    if (x >= img_w || y >= img_h)
        return;
    // Function continues ...
}
```

(2.14)

Figure 2.8: Example of thread overrun with a 2-D input when the data size is not a multiple of the block size.
2.5.3 2-D Exogenous Operations

Two-dimensional exogenous operations present more of a challenge in terms of grid layout and algorithm design. A classic exogenous operation is convolution (or template matching). Say we have an image $img$ of size $w$-by-$h$ pixels, and a template $t$ of size $t_w$-by-$t_h$ pixels. With a block size of $b_w$-by-$b_h$, processing each block then involves reading in a $(b_w + t_w - 1)$-by-$(b_h + t_h - 1)$-pixel sub-window of $img$. This involves data that lies outside of the block, and we must read in $(b_w + t_w - 1)(b_h + t_h - 1)$ input points to produce only $b_w b_h$ values of output.

There are two ways to handle this imbalance. The first is to have each thread of the block read in only one input value, then have a reduced number of threads compute and output results. In this case, the bottom $t_h - 1$ rows and rightmost $t_w - 1$ columns will do no work beside reading in data. An advantage of this approach is that the kernel is easy to write: each thread reads in one value to shared memory, those not contributing to the output exit early, and the remainder calculate one result accessing shared memory as a one-to-one analogue for global memory.

A disadvantage of this approach is that it reduces the amount of work each block does. In the above example, a $b_w$-by-$b_h$-pixel block will produce only $(b_w - t_w + 1)$-by-$(b_h - t_h + 1)$ output pixels. This means that, for a 32-by-32-thread block and a 16-by-16-pixel template, only a 17-by-17-thread sub-block will produce output, or only 28% of the threads. Also, because we are allocating threads to compute output values, the next block to the right of this one will have to start at an offset of $(b_w - t_w + 1, 0)$ instead of $(b_w, 0)$—as it would with all pixels producing one output. This makes it more difficult to compute the grid and more complicated to determine a thread’s index inside of the block (since the block size does not say how many pixels over each block is shifted, meaning the blockDim variable becomes misleading).

The second approach is to have all threads compute one output value, and this requires some threads to read in multiple data points before computation begins. The advantage of this is that computing the covering grid and the thread indices is now much simpler, as blockDim and blockIdx relate directly to the thread’s coordinates relative to the input and output. The disadvantage, of course, is the complexity and latency of having multiple threads read in multiple values from global memory sequentially.

Both of these approaches are diagrammed in Figure 2.9. The second approach is typically favored because the overhead of allocating a new block of threads is greater than that of having extant threads make a few more reads to global memory. Also, with the first approach, the different blocks will be stepping through memory at a stride dependent upon the template size (or whatever other
constraint is imposed by the algorithm). This will be less likely to be reading in aligned segments of memory conducive to coalescing.

Figure 2.9: Two approaches to computing exogenous output as shown with convolution. (a) shows the case of reading in data only over the range of the block $b_w \times b_h$, and computing a restricted output range (green) based on the template size (blue). (b) shows reading in enough data ($'b_w + t_w - 1'$-by-$'b_h + t_h - 1'$) to compute an output for every thread.

This discussion introduces the concept of the “input data apron,” or simply the “apron,” [9] which is all the data that must be read in by a block in order to compute its output. In the case where the apron includes values that lie off of the block, we will use the cardinal directions (north, south, east, and west) and the intermediate directions (north-west, south-east, south-west, and north-east) to refer to in which directions it lies. In Figure 2.9a, the apron is completely overlaid by the block, so no extra data is read in. In Figure 2.9b, extra data lies to the east, south, and south-east.
Chapter 3

Visual Servoing

One of the first widespread application of robotic limbs was in the manufacturing industry. The limbs were controlled through a precisely known model of the robot’s kinematics and, where necessary, contriving the environment to suit the limitations of the robot [20]. These limbs were thus controlled in an “open-loop” fashion which used little feedback from the observed world. But this left inaccessible the realms where the robot’s position was unknown, or where the positions of the objects the robot was meant to interact with were unknown, or where the robot’s kinematics were inaccurately known [21]. This includes applications where the robot moves outside of a strictly controlled environment (such as the assembly line) and is given more general tasks to perform instead of performing one well-known action repetitively. To function in these contexts, robotic control had to advance to “closed-loop” systems that used feedback from the observed world via sensor integration. One method of achieving closed-loop control is via visual feedback, which allows non-destructive observation of the environment, and needs only cameras, which are cheap hardware. This technique is known as “visual servoing,” and has proven a powerful technique for solving difficult perceptive problems [22].

Closed-loop control becomes even more essential when a robotic limb is mounted on a mobile platform. Originally, the accuracy of the control system depended only upon the knowledge of the direct and inverse kinematics of the robot, and of its position and orientation (its “extrinsics” or “extrinsic calibration”). But a mobile platform is subject not only to these same sources of error but also to the jitter and disturbance of the underlying platform. A robot operating in this context is very likely to accumulate sufficient levels of error in its extrinsic calibration over time to vitiate
whatever task it has been set to perform, making closed-loop control essential.

The specific application of visual servoing we were concerned with here was of the kind described above. A robotic arm is mounted on a mobile, autonomous platform. The arm has a grip end-effector which we wish to apply to many tasks, including pushing elevator buttons, opening doors [23], and grasping novel objects [24], [25]. The robot was equipped with a camera system, so a visual servoing solution was implemented whereby the robot could localize its arm within the camera’s field of view. However, this solution (despite being heavily optimized itself) took half a minute to process a single frame. If this visual servoing subsystem were included on the robot, the maximum “perceptive frame rate” the robot could achieve (describing its general speed of interaction with its environment) would be only 2 frames per minute. At this speed, it would take an exceedingly long time to perform even the simplest tasks. Because this reference solution had already been optimized for CPU execution, we imagined the best way to achieve the 30-times speed-up necessary for real-time execution would be via GPGPU.

This work was done in collaboration with Quoc Le, who developed both the original visual servoing algorithm being examined, and its CPU implementation.

3.1 Outline of the Visual Servoing Algorithm

A typical pose in which one might find the arm (as seen from the robot’s camera) is given in Figure 3.1. The motivation for this scene is that the arm is attempting to pick up the stapler (specifically the one closest to the arm in the frame). The robot has depth sensors and object detectors that let it identify the 3-D location of the object of interest, and then, using the arm’s kinematics and extrinsics, the end-effector is ordered to a location over the stapler and told to pick it up.

However, because of the jitter involved with the mobile platform, the extrinsics are not precisely known, and there is error between the arm’s expected and actual positions. Say the arm’s configuration is given by the vector $\theta$. Using the best guess of the arm’s extrinsics, the robot might calculate that the end-effector is at position $p$ in the world, but in reality it is at position $\hat{p}$. This cannot be corrected for only once, as, whenever the platform moves (which is often), the arm will drift into a new uncertain extrinsic position. Rather we must continually calculate new, updated extrinsics, or find a new configuration, $\hat{\theta}$, such that the new calculated position of the robot arm coincides with its observed position, $\hat{p}$. 
The latter approach is implemented with a visual servoing algorithm. Taking the robot’s given configuration \( \theta \) as a starting point, it searches around \( \theta \) in a small radius, and takes the true (or “corrected”) configuration, \( \hat{\theta} \), as that configuration at which the arm’s expected location in the frame best matches its observed location. This comparison is made between the observed camera images (such as in Figure 3.1), which represent \( \hat{p} \), and synthetic images generated by rendering a high-detail computer model of the arm at some search configuration parameters, \( \theta' \) (and so simulating what the arm would look like at position \( p' \)).

An example of this search process is shown in Figure 3.2, where we are considering the scene of Figure 3.1. Figure 3.2a shows a synthetic image of the arm rendered at some configuration \( \theta' \) that is examined during the search. In Figure 3.2b we show an overlay of the synthetic pose and of the original scene as a red–cyan color composite. The algorithm assigns a cost to each pose (or each configuration, \( \theta' \)) depending upon how similar the synthetic and reference images are (according to some metric). More details on the exact metric employed and all the various processing steps follow in the next section.

### 3.2 Reference Solution

The reference program we were provided with was written in C++ and followed a simple program flow. It took in a reference image of the scene such as Figure 3.1, and an approximate, estimated configuration, \( \theta \), (a six-dimensional vector) for the robot arm. To find the corrected arm configuration, the program searched over a small range for each of the arm parameters, and so examined a
hyper-lattice of points in this six-dimensional space. For each configuration, the program generated a synthetic image of the arm such as that in Figure 3.2a (drawn in OpenGL) and “scored” it with the reference image to achieve a similarity measure.

Ideally this measure should have been easy to compute but also robust to differences in lighting between the two images. In the reference solution, this similarity measure was built on comparing the soft edge maps from both images (a “soft” edge map is one where edges in the image are returned as non-binary values that indicate their relative strengths [26]). Edge maps can be computed using simple mathematical methods, and they are robust to many types of image distortion [27]. Sample edge maps for both of the images above are given below in Figure 3.3. These were computed using the Sobel technique for edge detection, which will be discussed in detail later.

To further increase the robustness of the edge map comparison, the reference solution blurred the images with a 9-by-9-pixel separable Gaussian filter before and after computing the edge map. This blurring had the effect of de-focusing the strong edges—thus diluting their response—while not much changing the response of weaker edges. This equalized the strengths of the edge responses, producing a more even comparison.

To assign a score to each pose, the program summed up the element-wise product between the reference image, $R$, and the synthetic pose image, $S$: 

Figure 3.2: (a) shows a typical synthetic image of the arm at a certain pose defined by some configuration parameters $\theta'$, and (b) shows a red–cyan color composite of the synthetic image with the scene of Figure 3.1. The visual servoing algorithm localizes the arm’s true location by finding the synthetic pose that best matches the given scene. The pose considered here is a poor match.
The motivation for this score was that a pose where the synthetic arm overlapped well with its observed position in the reference image would have a high value for this product. While technically any overlap between edges from the two images would contribute to a high score, the complex form and unique shape of the arm assured that only images where the reference and posited arm positions were greatly aligned would receive the highest scores. The true pose was taken as the prospective pose that returned the highest score.

This scoring is illustrated in the color composite of Figure 3.4. The edges from both the scene image and the synthetic arm image have relatively the same strengths, but because the pose being examined was not the true pose, the edges are only feebly aligned, the element-wise product will be mostly sparse, and the image will receive a low score. The infrequency of the lines suggests that it would be difficult for a spuriously high score to be achieved by aligning the synthetic arm over random image features.

### 3.2.1 Synthetic Arm Model

The arm model used was high-detail and captured using a laser scanner. A close-up of this model is given in Figure 3.5. The synthetic pose of the arm given in Figure 3.2a was created with this model, and one can see the it compares favorably with how the actual arm looks in the scene of Figure 3.1.
Figure 3.4: A red–cyan color composite of the soft edge maps from the scene and the pose images. (a) shows the full images while (b) shows the detail on the robot arm.

The synthetic images were converted to black-and-white for comparison with the reference, but we show the full-color model below for completeness.

Figure 3.5: Close-up of the synthetic robot arm model.

In detail, the model contained approximately 27,000 individually colored points, and the mesh was created with 36,000 triangles, with each defined as three indices into the list of points. This made it easy to determine whether two triangles shared a vertex or an edge, as this required examining only which indices they had in common, and there was no need to search over points in $\mathbb{R}^3$. 
3.2.2 Sobel Soft Edge Map

The edge map algorithm used in the reference solution is the Sobel method. Of all edge detection algorithms, this is one of the simplest (both computationally and conceptually [27], [7]). The Sobel detector starts by convolving the image with two separable 3-by-3 filters. One is tuned to detect horizontal edges and is of this form:

\[
\begin{bmatrix}
-1 & -2 & -1 \\
0 & 0 & 0 \\
1 & 2 & 1
\end{bmatrix}
\ast
\begin{bmatrix}
1 & 2 & 1 \\
0 \\
1
\end{bmatrix}
\]

(3.2)

These are simple matched filters that take the difference between non-successive rows and columns, returning large results where their values differ. This detects changes in image intensity, which is highly indicative of an edge [27].

These kernels can also be understood as chained smoothing and differentiation operators. In particular, the kernel of (3.2) can also be composed as:

\[
\begin{bmatrix}
1 & 2 & 1 \\
1 & 2 & 1
\end{bmatrix}
\ast
\begin{bmatrix}
-1 \\
1
\end{bmatrix}
\]

(3.4)

The vertical edge detection kernel (3.3) can also be interpreted as the transpose of the matrices in (3.4), showing that it, too, is a chained smoothing–differentiation operation.
Call the results of convolving an image, \( I \), with these two kernels—(3.2) and (3.3)—\( H_- \) and \( H_\| \), respectively. The final soft edge map is simply the sum of these added in quadrature, or:

\[
Sobel = \sqrt{H_-^2 + H_\|^2}
\]  

(3.5)

Examples of the soft edge maps used in the detection process are shown in Figure 3.3 (using the arm scene and synthetic poses of Figure 3.1 and Figure 3.2). Despite consisting of filters designed only to detect vertical or horizontal features, the lax specificity of those filters makes the quadrature sum of them good at detecting edges at general orientations. 45°-oriented edges, in particular, appear clearly, although at admittedly smaller magnitudes than the horizontal or vertical edges.

3.2.3 Benchmark

The benchmark solution was highly optimized for CPU execution. It used streamlined OpenGL calls to render the arm at its synthetic poses, then used OpenCV calls (which made use of Intel Performance Primitives) to perform the image processing steps. In the reference configuration, the program was set up to examine some 750 different arm poses. The reference scene image and all synthetic images were 640-by-480 pixels in size. The program was able to complete this loop in approximately 30 seconds on a modern system, or at a rate of being able to examine 25 poses per second. This was indeed a fast result, but, ideally, to use visual servoing in a live robotics application, the entire loop should have run in under a second to give the robot fluid feedback.

We also benchmarked the individual program components to see where the majority of processing time was being spent: whether in the OpenGL rendering of synthetic poses, or in the image processing to compute the score versus the reference image. Of the original 30 s runtime, the OpenGL phase took 7 seconds while the image processing took 23 seconds. Clearly, if the entire loop is to run in under a second, some optimizations would have to be made to the OpenGL side as well. But, since the image processing took three times longer to begin with, we chose that as our first avenue for optimization (also because the image processing was most directly amenable to GPU acceleration via CUDA, whereas OpenGL is already run on the GPU and so is not “susceptible” to plain CUDA conversion).
3.3 CUDA Pipeline

Now we discuss the proposed GPGPU pipeline where all the steps of the image processing were to be converted to CUDA. We would still use OpenGL to render our synthetic images of the arm model, but the post-processing of them was to be placed entirely on the GPU.

Thankfully, the reference solution already “starts” on the GPU as that is where OpenGL creates the synthetic pose image as a buffer object. The reference solution had to copy these images off of the GPU so as to be modified by the CPU application, but CUDA enables direct OpenGL-to-CUDA memory transfer for moving OpenGL-rendered objects to memory that is accessible by CUDA kernels [1]. Since this is intra-GPU memory transfer, it is also faster than the GPU-to-CPU memory transfer necessary in the reference solution [29].

After copying the synthetic image to CUDA’s memory space, the image processing chain is exactly as in the reference solution:

1. Convert image to grayscale
2. Blur using a Gaussian filter
3. Perform soft edge detection
4. Blur again
5. Compute element-wise product with processed reference image
6. Sum up resulting product image

The reference scene we are comparing the poses against will have been moved to the GPU previously, as well as given the same blur–edge–blur treatment each synthetic image will be subjected to.

While this pipeline shows the result image being summed up on the GPU, in a preliminary version we simply transferred the result image back to the CPU and summed it up there. We chose this solution first for simplicity as a whole-image sum is not an operation that is trivially parallelizable. But we were forced to replace this with a GPU-side sum when we saw how inefficient memory transfer between the CPU and the GPU was. On the NVIDIA 8800 GT we were benchmarking with, the maximum GPU-to-CPU memory transfer bandwidth was 1 MB/ms. Thus a 640-by-480 single-precision floating-point image (which constituted 1.23 MB of data) would require over a millisecond to transfer. Yet typical CUDA kernels have execution times of only a few hundred microseconds.
Even if computing the whole-image sum turned out to be relatively costly on the GPU, it would still likely be much more efficient than a CPU-side sum simply by dint of reducing the end amount of memory transfer to 4 bytes for each pose (for the summed up result) versus 1,228,000 bytes.

### 3.4 Conversion of Functions to CUDA

In this section we describe the conversion of the major parts of the reference solution to CUDA kernels.

#### 3.4.1 Sobel Edge Map

To review, Sobel edge detection involves two convolutions with 3-by-3 matrices (though each is decomposable as two one-dimensional convolutions) to create $H_-$ and $H_+$, followed by computing the quadrature sum between the two temporary results.

The quadrature add step was the simplest to convert, as it was obviously a “one-to-one” function (as defined in the taxonomy of Section 2.4). A sample kernel to perform this is given below:

```c
__global__ void add_quadrature(float *input1, float *input2, float *output, int img_w, int img_h)
{
    const int x = blockDim.x * blockIdx.x + threadIdx.x;
    const int y = blockDim.y * blockIdx.y + threadIdx.y;
    const int ix = y * img_w + x;

    if (x < img_w && y < img_h) {
        const float ff1 = input1[ix] * input1[ix];
        const float ff2 = input2[ix] * input2[ix];
        output[ix] = powf(ff1 + ff2, 0.5f);
    }
}
```

The convolutions, however, are exogenous functions, where we need to store input data in shared memory—and in amounts strictly greater than how much output we are computing. One benefit, however, is that each convolution involves only six non-zero elements. Thus they can be implemented manually without requiring the convolution kernels to be explicitly stored and looped over as is done generally with convolutions.
Given that, there were two major ways to compute the 2-D convolutions. The first was as actual 3-by-3 convolutions, and the second was as chained 1-D convolutions. The first option was the most complicated from an apronning perspective: for a square thread block of size \( d \times d \), we would have to read in a \( (d+2) \times (d+2) \) pixel apron which trails off of the block footprint in all directions—and then only by one pixel at a time. The fact that we would be reading in one extra column to the east in west was the most debilitating, as these reads to global memory would be un-coalesced. Though the advantage of this 3-by-3 approach is that one kernel call would be sufficient to compute both intermediate convolution results. The quadrature add could then be performed immediately, and so the entire result could be computed with only one kernel call.

The other option of multiple 1-D convolutions is attractive because of the simplified apronning. Assuming a \( d \times d \) thread block, we can compute the column-wise convolutions by reading in only two additional rows of data: the one above and below the block. These reads would be made to linear memory and so be coalesced. The extra column reads in the row-wise convolutions would not be coalesced, of course, but their access patterns would be simpler to implement than with the full 8-direction apronning of the 2-D convolution option.

A disadvantage of this latter approach, however, is that it would take three kernels and four calls to compute the full Sobel result. The first kernel would read in the input image, do one direction of the convolutions (either row-wise or column-wise) with both vectors, and output two temporary results. The next two calls would be to load in these temporary results and perform the complementary convolution operation. A final kernel would be required to perform the quadrature add—unless this were composed with the last convolution, or the two final convolutions were conflated into one call. This proposed program flow is diagrammed in Figure 3.6.

As if often the case in CUDA conversion, while intuition can give us a sense of what will be the fastest approach, often the only sure way to tell which method is fastest is the \textit{a posteriori} approach of implementing and benchmarking several options. This we did with these two principle options—the one-kernel and the three-kernel option. The breakdown of the runtimes is given below. All times are as recorded by the CUDA Profiler, which outputs a log file of kernel execution times after a CUDA program exits [30].

1. One-function option: 197 \( \mu \)s
2. Three-function option: 402 \( \mu \)s total
   - Column-wise convolution \(- 130 \) \( \mu \)s
Figure 3.6: Three-kernel Sobel implementation. The left-side figures correspond to computing $H_-$, and the right-side to $H_1$. First operation: column-wise convolutions that produce two temporary results from the input. Second: row-wise convolutions on each of the temporaries. Third: combining both with a quadrature add to achieve the soft edge map.

- Row-wise convolution – 2 calls at 93.5 µs each
- Quadrature add – 85 µs

The one-function option has turned out to be the fastest by a factor of two. In retrospect this is obviously because the limiting factor is the time instead of computation: the math behind the Sobel operator is very simple, yet the three-kernel option performs much more back-and-forth to global memory to perform it than does the one-kernel option. Even though the apronning of the one-function option is complex, the fact that it reads from and writes to global memory only once is highly beneficial.

To give a rough idea of the CUDA speed-up, we compare the performance of this Sobel implementation with a CPU version. In particular, while our fastest Sobel kernel ran in 197 µs, OpenCV’s Sobel function (cvSobel)—when run on the same platform and over the same size of image—took an average of 2 ms, a ten-times speed-up.

This Sobel implementation also compares well with a reference implementation developed by NVIDIA for their CUDA SDK, which contains many sample CUDA applications. In particular, their solution takes 274 µsto process a 512-by-512-pixel, 8-bit image (benchmarked on the same system with the same GPU as above). Since the image sizes are different here, we might approach
a normalized comparison by examining the computation time per pixel. There, our kernel spent 0.64 ns per pixel, whereas the NVIDIA kernel spent 1.05 ns. Though it should be noted that the NVIDIA reference implementation used a different type of GPU memory (in particular, texture memory) and provided a few more options than our solution. Ours had the benefit of being able to optimized for only one specific application.

3.4.2 Separable Filter Conversion

The separable filter—despite also involving exogenous operations—was a simple conversion to CUDA. But, because the convolutions were with 9-element vectors instead of 3-element vectors as in the Sobel case, we did not implement these “manually” and instead used constant GPU memory (identified in CUDA by the keyword “\_constant\_”) to store the convolution kernels, and used explicit looping to compute the convolution products. Using device constant memory to store convolution kernels is a common practice [9].

We picked optimal block sizes for both the row-wise and column-wise operations. For the row-wise convolutions, the apron data would lie strictly to the west and east, which—being stored linearly in memory—can be accessed with coalesced reads. To cover the 640-pixel–wide images, we chose to process it in two, 320-pixel halves. We allocated blocks of size 352 (320 + 16 + 16) to have a group of 16 threads to either side to read in the east–west data. After shared data synchronization, the middle 320 threads calculated the result. Thus each thread had to read in only one location from global memory, and thereafter 91% of the threads computed output values.

For the column convolution we used a 16-by-32 thread block, with a width of 16 chosen to achieve coalesced reads. We then chose the height dimension as large as possible (here brushing up against the 512-thread limit of the GPU) to achieve maximum work efficiency from the threads. Since all threads were computing output, this meant some threads had to read in multiple data points to cover the upper four and bottom four rows of the apron. This meant 128 threads (25% of the 512) had to read in two data values instead of just one. But thereafter 100% of the threads computed one output value apiece.

Over the 640-by-480–pixel input images, these convolution functions executed very quickly, with the row-wise pass taking an average of 93 μs, and the column-wise pass taking an average of 136 μs. Despite the fact that both operations were convolving with exactly the same kernel, the discrepancy in their runtimes is due to the greater data coalescing seen in the row-wise version, and the single pass of data loading that was also possible in that domain.
To illustrate the importance of coalescing reads and writes to global memory, we implemented a column-wise convolution with the same layout as the row-wise convolution, where we processed an entire column at once with no concern given to coalescing reads or writes. For this, we allocated a 512-thread block that was arrayed vertically. The top and bottom 16 threads loaded in zero values to shared memory (for padding), and the middle 480 reading in the actual data values. After synchronization, the north and south apron threads exited and the middle 480 threads computed the results. A block of 640-by-1 was required to cover the whole image. The average kernel runtime for this method of column convolution was a surprisingly ruinous 1130 $\mu$s—over eight times slower than the coalesced version, and over twelve times slower than its dual row convolution operation.

3.4.3 Image Summation

With the pre-processing on the synthetic image complete, the remaining steps were to compute the element-wise product with the reference image and sum up the result. The element-wise product is a trivial, one-to-one function whose implementation has already been covered in multiple examples. The sum, however, presented an interesting challenge. Summing an image obviously lies outside the class of same-size operations that we considered in Section 2.4, as it consumes many values and outputs only one. In fact, computing the sum of a set of numbers can be thought of as inherently serial.

Yet a large sum can still be partially parallelized. In particular, with an unlimited number of threads to execute in parallel, the sum of a vector $x$ with $n$ values can be computed in $\log_2 n$ steps. The approach is to add half of the data to the other half in parallel; then add half of that with its other half in parallel; and so on until we converge to the total sum. We present this algorithm below in pseudocode. For simplicity we assume $n$ is a power of 2:
\[ d \leftarrow n/2 \]

Allocate \( d \) threads and \( \text{temp}[d] \) memory

\[ \text{temp}[0 : d - 1] \leftarrow x[0 : d - 1] + x[d : n - 1] \quad / / \text{In parallel} \]

\[ d \leftarrow d/2 \]

while \( d > 0 \) {
    Threads \( d : 2d - 1 \) exit
    \[ \text{temp}[0 : d - 1] \leftarrow \text{temp}[0 : d - 1] + \text{temp}[d : 2d - 1] \quad / / \text{In parallel} \]
    \[ d \leftarrow d/2 \]
}

In detail, we first add the last half of the data to the first half and save the result in temporary memory. For each pass of the loop, the upper half of the threads exits, and the remaining half adds the temporary sums “abandoned” by the exiting threads to their own temporary sums. In the last iteration of the while loop, \( d = 1 \) so the index 1 thread exits and the index 0 thread adds the remaining two temporary results to get the full sum in \( \text{temp}[0] \). This algorithm is shown graphically in Figure 3.7. One can think of this approach as akin to the “divide and conquer” strategy, though the splitting done here is not to make the problem more tractable but more parallelizable.

The problem with \( n \) not being a power of two is that, sometime during the algorithm, \( d \) will take on an odd value. Then the step \( d \leftarrow d/2 \) will really effect \( d \leftarrow \lfloor d/2 \rfloor \). If \( d \) was 5, say, then only two threads will be active on the next pass, and so will add up only 4 of the 5 previous temporary results. This can be fixed by inserting a check into the algorithm to see if \( d \) was non-divisible by 2 in the last step, and then to manually include the “left behind” result. This slightly complicates the inner loop.

We considered how to apply this algorithm to effect the summation of a 640-by-480-pixel image. One option would be to treat the image as a linear array of 307,200 elements; but, as we did not have the minimum number of 153,600 threads to naively apply the above algorithm, clearly we would have to stagger our approach. Another complication with this approach is that 2-D memory in CUDA is not always packed strictly linearly as each row in a 2-D allocation must start at a location that is 0 modulo 16 bytes [1]. Granted the memory here would have been packed linearly
Figure 3.7: Graphical depiction of the parallelized summation algorithm. The original range of input $x$ is shown in green. In the first iteration, $n/2$ threads (with their temporary data shown in cyan) “fold” the data range by adding the first and last halves of it together. At successive iterations, the upper half of the threads exits (turn grey) and the remaining half adds the “abandoned” temporaries to its own sums. In the final iteration, the full sum of the input range is achieved in the first element (orange).

(since $640 \cdot 4 = 2560 = 0 \mod 16$), but in general this might not be the case when summing up an arbitrary-sized image, and we wanted a robust algorithmic solution.

So we decided to apply the above algorithm to separately sum up each row. With only 640 elements in the width dimension, this meant we needed only 320 threads for each row, which was below our GPU-mandated maximum of 512. We could define a block of 320-by-1 threads, and then a grid of 480-by-1 to cover the entire image. However, we had to place the sums of each row into different locations in memory: we could not add them to a final sum as they were computed. This was because the different blocks could be executed in any order (as discussed in Section 2.2), so CUDA provided no synchronization or locking primitives to prevent data conflicts between threads from different blocks attempting to modify a single address simultaneously.

Instead we had each block write its result to a new location in a linear stretch of memory (with the function being provided a pointer for this purpose). The elegance of this solution was that the first pass over all rows would output its temporary results in such a way as to create another “row” of data, and we would only need to call the same kernel function to process this new row to derive
the final result. By design we would also always be making coalesced reads from global memory, and there would be no bank conflicts between threads as they computed the sums. This scheme is diagrammed in Figure 3.8.

![Diagram](image)

**Figure 3.8:** Row-wise strategy for applying the parallelized summation algorithm to sum up a \( w \)-by-\( h \) image (in green). The first call to the kernel (“1”) is gridded 1-by-\( h \) and sums up each row, outputting \( h \) elements in another linear stretch. The second call to the same kernel (“2”) needs only one block to sum up the row of intermediate results to achieve the result (orange).

With the algorithm decided, one remaining option was whether we used the strictly power of two version of the algorithm or not. If we used the algorithm that required a power of two–sized input, we would need to pad the 640-element rows to 1024 elements (or this could be achieved virtually by allocating a 512-thread block and having any threads that read over the edge of the image read a default value of 0 instead). However, this would make the first step of the algorithm wasteful since we would be adding 1024 “virtual locations” when only 640 exist, presenting an efficiency of only 62.5%. When we were later summing up the 480 temporary results, we would also need 256 threads to cover the full 512-element virtual range, though this would be 94% efficient. However, if we used the version of the algorithm robust to non–power of two sizes, then we would need only a 320-thread block to compute the intermediate sums over the 640-element rows, achieving a 100%-efficient loop (and analogously a 240-thread block to sum up the 480 intermediate sums). These functions would be using a slightly more complicated version of the algorithm, but they might benefit from the more efficient use of threads.

As with the Sobel edge map conversion, without an *a priori* way of telling which function would
be the most convenient in this case, we implemented and benchmarked both. To sum up a 640-by-480–pixel image, the first option—with a 512/256-thread pass—took 116 µs, while the second option—with a 320/240-thread pass—took only 93 µs. The second option presents only a 20% speedup, which is small enough to suggest that, while it is optimal with the specific numbers here, it may not be so in all cases. Indeed, when the input length is exactly a power two in both dimensions, we should expect the first option to be the fastest, as both algorithms would be doing the exact same amount of computation, but the first option would be doing so with a more streamlined inner loop.

To verify this conjecture, we benchmarked both versions of the algorithm again but running on a 1024-by-512–pixel image. In this case, the power of two–targeted algorithm was markedly faster, taking only 136 µs compared to the general-\(n\) algorithm’s 194 µs. The power of two algorithm was running with the same block sizes as in the 640-by-480–pixel case (512/256), and it took only 17% longer to process 70% more data. In contrast, the general algorithm had to be run with more threads per block and took 108% longer.

After successfully computing the image sum, we needed only transfer one number back to the GPU for processing. While this represents the least costly amount of memory transfer possible, there is still the irreducible overhead of making the transfer. Thus, instead of transferring each of the \(n\) results for each arm pose back individually, we saved them in a linear stretch of memory on the GPU, and transferred this back only once all \(n\) scores had been computed. The total amount of data transfer was still the same, but by “batching” (as it is called) our transfer we paid as little overhead as possible [29].

3.5 Intermediate Benchmark Results

At this point, we had a working CUDA-optimized version of the entire pipeline. Benchmarking it, however, the end-to-end time of the solution had decreased only to 18 seconds versus the 30 seconds taken by the original. Discarding the irreducible 7 seconds needed for the OpenGL rendering, the image processing computation had dropped from taking 23 seconds to 11 seconds. While this constituted a 2.1-times speedup, it was still far from the order of magnitude that CUDA is generally believed capable of delivering. This was also much less than the individual speed-ups we were seeing with the Sobel edge detection and convolution, which were an order of magnitude faster than optimized CPU versions. To decipher this lack of performance, we used the CUDA Profiler to
benchmark the reference solution.

Figure 3.9 gives a pie chart showing the amount of time spent in the different CUDA stages. As can be seen, nearly three-fourths of the execution time is spent simply in memory transfer. We can understand this by reflecting that the computations involved in this visual servoing pipeline are all so simple. The kernels take an average of only 100 \( \mu s \) to execute, with the slowest one taking at most 250 \( \mu s \). Meanwhile, to evaluate one pose of the arm, we must transfer a 640-by-480–pixel, 4-byte-deep image (i.e., 1.23 MB of data) from OpenGL- to CUDA-accessible memory. This intra-GPU transfer has a maximum bandwidth higher than that of CPU-to-GPU transfer, but, as it was still crossing non-trivial boundaries on the GPU, it so was not as fast as true intra-CUDA memory transfer [1].

Even though we had diagnosed memory transfer as the bottleneck of the CUDA program, there was no way to optimize the transfer. CUDA provides only one function call to do memory transfer (\texttt{cudaMemcpy} [31]), and this was a black box that could not be pried into. So we reflected that the OpenGL was rendering color images, but the CUDA was converting them immediately to grayscale. We investigated whether or not we could reduce this memory transfer load by having OpenGL render one-channel images natively—thus cutting the required data transfer by a factor of four—but this
proved not possible. Thus, not being able to speed up or reduce the amount of memory transfer, we investigated how to increase the efficiency of the existing transfers.

3.6 Color Channel Multiplexing

One source of inefficiency in the above pipeline was the mismatch between the format of the synthetic arm images output by OpenGL and the format used by CUDA. In particular, OpenGL outputs 8-bit, four-channel color images (with red, green, and blue color components, and an alpha channel—all packed into sequential 32-bit values) while the CUDA uses black-and-white (or one-channel grayscale) single-precision floating-point images. Thus the first step of the CUDA pipeline was to recover these 8-bit color values, convert them to floating-point, and average them to get the grayscale value. If we could have OpenGL render only an 8-bit, one-channel image to begin this, this would optimize memory transfer efficiency by a factor of four. However, this proved not feasible. Since we could not discard the extraneous color channels from OpenGL, we decided to use them to send extra information in the form of additional poses.

The synthetic arm model contains a 3-component color for every vertex, given as \((r, g, b)\). Originally we were rendering these three colors in OpenGL and averaging them together on CUDA, but we could average them before rendering and so have the model appear in “simulated” greyscale (i.e., where there are still three color channels, but each holds the same value). Each vertex, instead of its given color value of \((r, g, b)\), would now have the color value \(( (r+g+b)/3, (r+g+b)/3, (r+g+b)/3)\). Now the three channels are obviously redundant since they all carry the same information. Then, instead of placing the same color in all three channels, we could render a pose in only one color channel, in the case of red achieving \(( (r+g+b)/3, 0, 0)\). If we could render another pose in the green channel as \((0, (r'+g'+b')/3, 0)\), and then add this to the red-channel image, we would have two poses superimposed. If we likewise extended this to use the blue channel and the alpha channel, we would be multiplexing four poses together. As all of these would exist in orthogonal color channels, the information would not be corrupted at all.

Implementing this color channel multiplexing required only simple modification of the reference solution. We rendered different versions of the synthetic arm model depending on what channel (color or alpha) we wished the pose to appear in, and we superimposed them all using OpenGL’s accumulation buffer [19]. A sample multi-pose image created using this technique is shown in Figure 3.10.
Figure 3.10: Overlaid poses using color channel multiplexing. (a) shows the image used in computation while (b) gives the same against a black background for better visibility of the multiple poses.

Now, instead of sending only one pose with every 1.23 MB transfer, we would be sending four. And, instead of needing \( n \) transfers from OpenGL to CUDA to process \( n \) poses, we would need only \( \lceil n/4 \rceil \). The only change on the CUDA side was, instead of averaging together the red, green, and blue channels of the input image, we would extract each of the four channels and process them individually. This is a slight work reduction for the CUDA as it no longer has to perform the RGB-to-grayscale conversion: only an integer to floating-point conversion (which it was doing before).

Benchmarking this new version of the function, end-to-end processing time dropped from 18 seconds to 11 seconds. Memory transfer went from taking up 72% of the CUDAs end-to-end processing time to only 41%. The OpenGL processing still took 7 seconds, however, which meant that the time required for image processing in CUDA dropped from 11 seconds to 4 seconds, for a 2.75-times improvement. The cumulative speed-up of the image processing chain (from the original 23 seconds) was now 7.7 times, which was much closer to the order-of-magnitude speed-ups expected from CUDA conversion. Though here memory transfer was still clearly our limiting factor, explaining why we were not seeing the largest speed-ups.
3.7 Optimizing Synthetic Pose Rendering

With the addition of color channel multiplexing, the CUDA image processing pipeline had become highly optimized, such that it now took up only 36% of the end-to-end processing time (4 out of 11 seconds). The rest of this time was dedicated to rendering the arm poses in OpenGL, making this rendering stage the new bottleneck. This then became the focus of our optimization efforts.

The OpenGL took 7 seconds to render the arm at approximately 750 different poses, or around 9.3 ms for each pose (or 107 poses per second). This is comparatively slow for rendering a simple OpenGL model, which can often be rendered at thousands of frames per second. Investigating why the arm model was so expensive to render, we discovered that its level of detail was excessive.

The arm was modeled as 36,000 triangles defined from a pool of 27,000 points, with each point being given a unique color. The mean area of these triangles was only 0.68 mm$^2$—less than a square millimeter—, which was trivially small. Indeed, around 33,000 triangles had areas less than this, such that the median area was only 619 $\mu$m$^2$. Even in places where the arm possessed large, flat sections, the model still needed to use hundreds of triangles to depict them. This represented an excessive number of primitives for OpenGL, and the fact that each vertex was individually colored was also deleterious. In that case, OpenGL had to compute smoothed colors for every pixel that were interpolated from the individual vertex colors [19]. With most of the triangles being microscopic, this was clearly excessive—especially at the poses and ranges we would be rendering the arm at.

The above level of detail became doubly excessive when considering that the visual servoing algorithm's image processing pipeline first blurs the images, then performs edge detection, and then blurs them again. Even if the model were only coarsely detailed, this processing would discard much of what detail there was. This also implies that, if we were to reduce the complexity of the arm model to save on OpenGL rendering time, the final scores calculated per pose should not be greatly affected.

We proposed to achieve this reduction in runtime by simplifying the 36,000-triangle model. While the given model was made of only OpenGL triangle primitives, we would convert this to one that used a far smaller number of general polygons. We planned to do this by designing an algorithm to conflate neighboring “similar” triangles into larger, simpler primitives, with a single mean color each. This would reduce the number of primitives in the model, the number of vertices, and discard all color interpolation.

Figure 3.11 shows a motivating example for this proposed agglomeration algorithm: the square in
Figure 3.11a is composed of 50 finely shaded triangles. This is analogous to the level of detail given in the original arm model. Since OpenGL requires three, 3-element vertices to specify each triangle and a 3-element color for each vertex, rendering this model requires passing 900 numbers to OpenGL. Yet we could replace this finely detailed model with a single quadrilateral where each vertex is given the mean color of the subtended shape, and this would require passing only 15 numbers to OpenGL (12 for the vertices and 3 for the single color). Since our processing blurs out fine details and passes only edge information, this would represent a minimal loss of information, and for significant time savings.

Figure 3.11: Motivation for merging the multiple, finely shaded triangles of the synthetic arm model into simpler equivalent shapes so as to boost rendering speed. The model of (a) requires rendering 50 interpolated-color triangles, but the model of (b) requires drawing only one flat-color quadrilateral—a much faster task for OpenGL. The model of (b) should produce nearly the same score as (a) after being put through the visual servoing pipeline, so this detail reduction should not effect results.

3.7.1 Greedy Agglomerative Algorithm

To achieve this detail reduction, we proposed a greedy agglomerative algorithm to transform the given OpenGL model into a simpler yet mostly equivalent one. This “mesh simplification” is a common strategy in the field of computer graphics [32], and real-time versions of it have even been built with GPGPU [33]. Our algorithm would perform the same function as these, consuming the large number of triangles in the original model and representing them with far fewer general polygons (as in Figure 3.11). We designed our algorithm to merge nearly co-planar neighboring triangles, and to average their colors together into a single color for each polygon. This would reduce the number of vertices, the number of colors, and also the color rendering complexity (since there would no longer
be any need for color interpolation). This approach is similar to (and motivated by) the point cloud segmentation algorithm of [34].

We pose the algorithm in pseudocode below. We start with the 36,000 triangles in a set $Tris$, and an empty set $Polys$ of general polygons. There is also a set $P$ of points in $\mathbb{R}^3$, over which the triangles and polygons were defined. We store each triangle in the same way as the original OpenGL model: as a list of three indices into $P$. Polygons are saved as general-sized lists of indices. The algorithm examines all triangles in $Tris$, promoting one at a time to the form of a polygon. With each polygon it begins an iterative search for all neighboring triangles, searching for those that can be reasonably fused into the growing polygon. Once the current polygon has reached its largest extent, the algorithm moves to the next un-processed triangle. Since the triangles and polygons were all saved as indices, it was easy to determine whether two triangles (or a triangle and a polygon) shared a vertex or an edge, as it required only examining how many indices they had in common, and required no search over coordinates in $\mathbb{R}^3$. 
Polys ← {Tris[0]} // Promote first triangle to a polygon
Tris ← {Tris[1 : n − 1]} // Remove first triangle from list
last = 0
while Tris ≠ ∅ { // Loop over all triangles
    NeighborTris ← {} // Init list of neighbor triangles to Polys[last]
    added ← 0 // Number of triangles added this round
    for all t ∈ Tris{
        if t has 2 or more points in common with Polys[last]
            and the normals of t and Polys[last] have a dot product \( \geq 0.95 \) { 
            added ← added + 1
            incorporate shape of t into Polys[last]
            Tris ← Tris \{t\}
        }
    }
    if added = 0 {
        last ← last + 1 // Advance to next polygon
        Polys[last] ← Tris[0] // Promote next triangle to a polygon
        Tris ← {Tris[1 : n − 1]} // Remove first triangle from list
    }
}

We start by promoting the first triangle to a polygon and removing it from the list of triangles. The while loop proceeds while there are still triangles to process, so by the end of the algorithm all of the triangles should have been absorbed into polygons (though, in the degenerate case, a “polygon” could still be a triangle if there were no neighboring triangles similar enough to be merged with it). Inside the while loop we check all remaining triangles for those which are neighbors of the current polygon—defined as having two or more vertices in common with the growing polygon. If these neighboring triangles have a normal vector highly aligned with the polygon’s normal vector
(calculated using SVD when the number of points in the polygon is larger than 3), the triangle is added to the polygon by incorporating its novel points. Omitted from the above description for brevity is how we also maintain a single color for each polygon, which is calculated as the area-weighted mean of the vertex-averaged colors of all composite triangles.

An illustration of the algorithm’s initialization and first iteration are shown in Figure 3.12.

![Figure 3.12: Greedy agglomerative algorithm illustration. (a) shows the beginning set Tris. (b) shows turning one triangle into the first entry of Polys. (c) shows selecting all neighboring triangles of the current polygon, with the shared points highlighted. (d) shows selecting triangles that are normal-wise aligned with the polygon: the green triangles are passed and the gray triangle is not. (e) shows the updated polygon. While triangles are still being added to the current polygon, the algorithm loops from (e) to (c). Else it loops back to (b) and converts the next free triangle to a polygon, proceeding until all triangles have been consumed.](image)

### 3.7.2 OpenGL Limitations on Polygons

While the algorithm outlined above is simple, the exact process of merging triangles into a growing polygon was complicated by certain limitations of OpenGL. Specifically, while the OpenGL polygon primitive we were using did support arbitrary numbers of vertices, it did not support general-shaped polygons. In particular, OpenGL rendered these polygons as a “triangle fan” [19]. If we supply \( m \) vertices to OpenGL (in the form of a vector \((v[0], \ldots, v[m-1])\)), the first vertex supplied \((v[0])\) is used as an “anchor point” for all of the triangles. Then the first triangle rendered is drawn between the next two vertices provided and this anchor point, and all later triangles are drawn between the last two vertices provided (in order) and the anchor point. I.e., the first triangle rendered has the vertices \((v[0], v[1], v[2])\), the second \((v[0], v[2], v[3])\), and the \( n \)’th \((v[0], v[n], v[n+1])\). While this allows us to render non-convex polygons, we cannot render arbitrary shapes. We must also take care to preserve a meaningful order in our polygon vertices, as the rendered shape depends crucially on...
how the vertices are presented to OpenGL. This is illustrated in Figure 3.13, where we show how presenting the exact same vertices but in a different sequence to OpenGL results in two different shapes.

Figure 3.13: Triangle fan–style rendering. (a) shows a polygon properly drawn when the vertices are provided in the correct order, creating a “fan” of contiguous triangles. (b) shows the same fan with the order of the 0th and 3rd points flipped. The shape becomes distorted.

To preserve this order of the vertices when merging a triangle into a polygon, we had to follow a special procedure. Whenever only one edge of the triangle and polygon overlapped, we had to add the novel point from the triangle between the two shared points already in the polygon’s list. Whenever the triangle shared two edges, we had to instead remove the interior point from the polygon outline. Both of these cases are illustrated in Figure 3.14. The bonus of this is that the number of vertices in our polygon will not grow monotonically as it agglomerates more triangles, and we can subsume \( n \) triangles into one polygon using far fewer than \( n \) vertices. Not shown is the rare case where a polygon absorbs a triangle that would become completely contained within it. In this case, we need only remove all the points from the newly absorbed triangle from the polygon’s vertex list.

Following this procedure at least prevents us from confusing the vertices’ order and thereby distorting the polygon’s shape. It does not, however, ensure that we will always draw the desired shape, as it is still possible to select a poor anchor point that will vitiate proper rendering. In Figure 3.14d, for example, using the topmost vertex for the anchor point would distort the rendering of the red polygon.

Despite this risk, we decided neither to check for nor to correct this in our agglomerative algorithm. First because we were merging triangles in planar regions, and the arm possessed few planar regions of complex shapes whose rendering could be drastically distorted by this mechanism. Also
Figure 3.14: Specialized merging of triangles into polygons. (a) continues the polygon-growing example of Figure 3.12. In (b) we detect two new triangles to merge. In (c) we show that we must add the green point in order to incorporate the upper triangle, but we must remove the orange point to incorporate the lower triangle. (d) shows the new shape and its vertices. Some of these points will be valid to use as the “anchor point” and some will not.

because, even where this distortion happened, we believed it would be only over a small scale and so not visible. This was reinforced by the knowledge that the visual servoing image processing to come only reduces the level of detail, and will thus likely smooth over such errors introduced by the agglomeration procedure.

3.7.3 Effect on Results and Speed-Up

Using the agglomerative procedure outlined above, we produced seven different reduced-detail models from the original arm model. Our finest reduced-detail model contained 2,930 polygons (starting from the original 36,000 triangles) while our coarsest had only 1,060. An example of one such reduced-detail arm model (compared with the original) is given in Figure 3.15.

Now with reduced-detail arm models, it remained to be seen what effects using these models would have on the numerical results and whether they would be acceptable. So we ran our processing pipeline with the original model and each of the reduced-detail models, saving the output scores. These results are plotted for comparison in Figure 3.16. In particular, we observed that, with the reduced-detail models, the scores over the processed images were uniformly slightly lower (around 1–2.5%) than with the original model. This was obvious in how these models were throwing away detail, thus reducing the number of edge pixels, and so causing a proportional depression on the scores calculated from the summed-up element-wise products. But we noted that the shapes of the curves were not altered significantly, nor were the locations of their relative maxima and minima changed. Since our processing pipeline was only concerned with the location of the arg max over pose scores, we should have been able to use one of these reduced-detail models without seeing any
Figure 3.15: Sample results of the greedy agglomerative procedure. (a) shows the original model while (b) shows one of the reduced-detail models. While the smooth, specular regions of (a) contrast with the matte, patchy regions of (b), the essential outlines and color scheme of the reduced-detail model are unperturbed.

change in which pose was voted the strongest.

Figure 3.16: Change in the element-wise product results per pose with reduced-detail models. The x axis shows the pose number while the y axis plots the score (the axis bounds are 720–810). The top blue line in the plot represents the results from the original model, while the lower lines represent those from the reduced-detail versions.

Reassured that our reduced-detail models were not distorting the results, we examined what effect these models had on runtime. Figure 3.17 shows the OpenGL-based processing times for the original model (the bottom bar) versus those with the seven reduced-detail models. The model complexity appears to have a proportional effect on runtime, though the fastest model was the 1,300-polygon model (with the second-smallest number of polygons). While rendering the original
Figure 3.17: End-to-end time spent in OpenGL rendering with the original and the reduced-detail models. The bottommost bar shows baseline of \( \approx 7 \) s, with all other models taking significantly less time.

Model complexity (# polygons):
- 2930
- 2829
- 2411
- 2058
- 1624
- 1300
- 1060
- Baseline

End-to-end OpenGL time (s):

Model at all 750 poses took 7 seconds, rendering the 1,300-polygon model took only 1.7 seconds, a 4.1-times speed-up. The most detailed model (with 2,930 polygons) was the slowest at 3.0 s, but this still offered a 2.3-times speed-up. Obviously the reduced-detail models have been worth significant savings, reducing the time spent in OpenGL to something even less than the time now spent in CUDA, and without significantly altering the results.

3.8 Final Optimizations

Thus far we had optimized the OpenGL–CUDA memory transfer and significantly sped-up the OpenGL rendering by using reduced-detail models of the robot arm. At present our end-to-end processing time stood at 6 seconds (using the 1300-polygon reduced-detail model). Approximately
4 s of that time was spent in the CUDA and 2 s in the OpenGL. This was tantalizingly close to our goal of real-time execution, but still undeniably distant there from.

However, one option that had always been considered for speeding up the reference solution had been to reduce the image size the pipeline was run at, switching from 640-by-480-pixel images down to 320-by-240-pixel images. This was rationalized in how the processing pipeline was only throwing away detail, so reducing the starting resolution by a factor of two was judged acceptable if it could provide a four-times speed-up. This was not implemented for the original, 30-second solution, however, as a factor of four would only have brought it down to approximately 8 s. Yet, with our optimized GPGPU solution now running at 6 s, a factor of four speed-up would bring us close to only a second in execution time.

Thus we implemented this option of halving the resolution. However, instead of changing our pipeline to run over 320-by-240-pixel images, we continued to use 640-by-480-pixel images but rendered four poses at tiled positions within each frame. We had to modify the CUDA to process each block of the image separately, but this had the advantage of reducing the number of transfers between OpenGL and CUDA by a factor of four (while still maintaining the same number of transferred poses). An illustrated example of these tiled images is given in Figure 3.18.
3.8.1 Benchmarking of All Variants

We ran a final set of benchmarks on all variants of the algorithm, both with and without color multiplexing, with and without half-resolution rendering, and with and without the reduced-detail arm model. The results are illustrated in Figure 3.19. Figure 3.19a shows the results with the original arm model while Figure 3.19b shows the results with the 1300-polygon reduced model. The blue component of the bars shows the end-to-end time spent in OpenGL and the red component shows everything else, including the CUDA computation. The OpenGL time is only reduced by changing the arm models (the difference between the (a) and (b) figures) while the CUDA processing time is reduced only by the CMUX and half-resolution steps (represented by the different columns inside each figure). In the most efficient configuration, the end-to-end time processing time was only 2.4 s, representing over a 15-times speed-up from the original 30 s runtime. This was close to running in less than a second, which was our conceptual barrier of a “real-time” solution.

![Figure 3.19: End-to-end processing time for the CUDA visual servoing pipeline with all optimization variants. The left side figure shows times with the original arm model; the right side with the reduced-detail model. The columns correspond, in order, to the full-scale, no-CMUX version; the full-scale CMUX version; the half-scale, no-CMUX version; and the half-scale CMUX version (illustrated by the glyphs beneath).](image)

Finally we experimented with how much of a speed-up was possible by moving the algorithm onto a newer GPU and a better system. No re-design of the algorithm was necessary as all we had to do was re-compile the existing program on a different platform. In particular, we moved from a system
with a NVIDIA 8800 GT to one with a 9800 GTX. With just this change, the end-to-end processing time dropped to just under a second, giving almost a 3-times speed-up versus the older GPU, and breaking the barrier of being able to examine an entire ensemble of synthetic poses in one second. With this we had not only achieved real-time speeds, but also demonstrated that the program could be even more significantly sped-up only by simply migrating to whatever is the newest GPU.

3.9 Conclusion

We took an existing implementation of a visual servoing algorithm that ran with an end-to-end time of 30 s, and we developed an optimized GPGPU conversion built on CUDA that ran in under a second on a modern (though not top-of-the-line) GPU. This met our original goal of a real-time solution, and so would allow this visual servoing algorithm to be used as an integral part of the perceptual feedback systems of a robotics platform.

Our work highlights the key approaches necessary in building an efficient GPGPU and CUDA application. In particular, achieving the fastest kernels involves not only making careful design choices that follow the CUDA best practices (viz., coalescing reads to global memory and employing shared memory), but often requires an assiduity in imagining and benchmarking many alternative implementations of a function to find which performs optimally under the given circumstances. Also, when faced with irreducible bottlenecks (such as we did with memory transfer), it can be necessary to explore solutions that do not “optimize” any code per se but that maximize the efficiency of operations that one is already committed to performing (such as we did with color channel multiplexing to increase the “effective” memory transfer bandwidth). While all these combined to make the GPGPU version much faster than the original, we also demonstrated how simply moving to a newer GPU can itself be worth a non-trivial additional speed-up factor. This itself is a powerful argument for using GPGPU, as, in the near-term, the growth in GPU raw performance is far outstripping the growth in CPU performance [1]. This suggests that, whatever the speed-up between a CPU and a GPGPU version of a program may be now, it is virtually guaranteed to increase with time.
Chapter 4

Sliding-Window Object Detection

One of the principle tasks of computer vision is to recognize instances of an object class in a camera image. Object classes that have received the largest amount of developmental effort include faces [35], people [36], vehicles and animals [37], and office objects (mugs, keyboard, mice, and monitors [38]). The current predominant approach to performing object detection is called the “sliding-window object detector.” This detector excerpts fixed-size rectangular sub-windows from an image and calculates “features” for each (which can be based on image edges, patch correlations, histograms of oriented gradients, wavelet coefficients, etc. [39]). These features are presented to a classifier that will have been trained on labeled data of both positive and negative examples of the object to be recognized. This classifier is most commonly either a linear support vector machine (SVM) or a variant on decision trees. When presented with a feature vector from a novel image sub-window, the classifier labels it as likely containing the object or not. Sliding-window object detectors have been able to achieve significantly high classification rates with very low error rates while being one of the object detection algorithms closest to operating at real-time speeds [40].

We were given an implementation of a sliding-window object detector (SWOD) that made use of high-performance computer vision libraries; yet this implementation was over a factor of fifty away from operating at real-time speeds, which ideally would be the ability to run one object detector over one frame per second. The benefits of being able to run a state-of-the-art object detector in real-time on a robotics platform are obvious, as this is an essential milestone for achieving “human-level robotic vision,” which must not only match the performance of human vision but also its speed.

A preliminary version of this work appeared in [40] with Adam Coates. Though the discussion
of the GPGPU conversion presented here is much more in-depth than what was presented in that work.

4.1 Reference Solution

In this section we provide an in-depth walk-through of the reference SWOD solution we were meant to optimize. This solution was part of the STAIR Vision Library (SVL, where “STAIR” stands for “STanford AI Robot”), and was a C++ implementation that relied heavily on the optimized routines of the OpenCV library to perform its core operations of convolution, integral images, etc. The particular detector we were working with extracted patch features from the image and presented them to a boosted decision tree for classification. The following sections cover the mechanics of the SWOD algorithm, how we computed the feature vectors for each sub-window, and the classifier used to evaluate them.

4.1.1 SWOD Algorithm

The SWOD algorithm is a generalized method of applying a limited classifier (one trained to detect objects only at a single pose and scale) to detect objects at many positions and scales in a larger image [41]. Call the input image to be processed $I$, with general size $w \times h$ (in our case, the image sizes were 640-by-480). At the inner loop of the SWOD algorithm is a classifier that can be presented with images of size $w \times h$ (in the reference solution, 32-by-32 pixels) for which it will return a probability of the object appearing in the frame. To detect objects at all locations in $I$ at the native scale of $w \times h$, the SWOD algorithm extracts all $w \times h$–pixel sub-windows from $I$ and presents them to the classifier, saving the positions and sizes of any detections. In pseudo-code:
\textit{Detections} ← \{\}  // Initialize empty detections list

for \( r = 0 : I_h - h - 1 \)  \\
    \{  \\
    \begin{align*}
    \text{for } c = 0 : I_w - w - 1 \quad &\{ \\
    \text{subimage} ← I(r + : h - 1, c + : w - 1)  // Extract sub-image for consideration  \\
    \text{if } classify(\text{subimage}) > \text{threshold}  \\
    \text{Detections} \cup ← \{r, c, w, h\}  // Save detection
    \}  \\
    \}  \\
\}

We use the notation \( r + : h - 1 \) as shorthand for \( r : r + h - 1 \) to specify the range \( \{k \mid r \leq k \leq r + h - 1\} \).

The call to \textit{classify} invokes the classifier to process the sub-image and return its probability of a “yes” assignment. The \textit{threshold} can be set variably, but is commonly 0.5.

As stated, the algorithm examines all possible sub-windows of \( I \); but, because of spatial coherence, whatever result the classifier returns at a location \((r, c)\), it is very likely to return the same result for the sub-window only a pixel or two off in either the horizontal or vertical directions. Indeed, it is generally assumed that the classifier is designed in such a way so as to be robust to such small pixel increments.

Thus, to speed up the algorithm, often the sliding window’s position is incremented by more than one pixel at a time (common values are from 2 to 10 pixels). We will use \( inc_r \) and \( inc_c \) to indicate these variable-sized increments. Incorporating these, the loops of the algorithm become:

\[
\begin{align*}
\text{for } r = 0 : inc_r : I_h - h - 1 \quad &\{ \\
\text{for } r = 0 : inc_c : I_w - w - 1 \quad &\{ \\
\end{align*}
\]

Here we use the Matlab-style double colon operator \( i : j : k \) to indicate the range \( \{m \mid i \leq m \leq k \wedge m - i = 0 \mod j\} \). A schematic of this SWOD scheme is shown with an example image in Figure 4.1. The sliding window travels over the entire image frame, moving by the given increment, and extracting the passing \( w \)-by-\( h \) sub-windows to present to the classifier.
Figure 4.1: Example of sliding-window object detection. A window is moved over the entire image frame, extracting sub-windows that are presented to the classifier.

The algorithm thus far will detect any objects at the native scale ($w$-by-$h$) in $I$, but, if the object appears in the frame at a larger scale, the classifier will miss it. Thus the SWOD algorithm has an outer loop that performs successive down-sizing of the image, then re-runs the above loop on the smaller image. This allows the classifier to detect objects that appeared at larger scales in the original image.

Call this image down-sizing factor $f$ (in the SVL, $f$ was 1.2, meaning the image became $16.6\%$ smaller with each iteration). The $w$-by-$h$-pixel sub-windows in this new image are processed, and the locations and sizes of any detections are scaled by the factor $f$ to report what the detection would have been on the original image $I$ at its native scale. This down-sizing is repeated until the image reaches some threshold size (often the minimum size of the classifier, which is the case in the SVL). The SWOD algorithm with this outer loop is given in pseudocode below:
\[Detections \leftarrow \{\}\]
\[f = 1.2 \quad // \text{Set successive down-sizing factor}\]
\[f_{\text{cumulative}} = 1 \quad // \text{Initialize scale to 1}\]
\[\hat{I} = I \quad // \text{Initialize image for processing at native scale}\]
\[\text{do}\{\]
\[\quad \text{for } r = 0 : \text{inc}_r : \hat{I}_h - h - 1 \quad \{\]
\[\quad \quad \text{for } c = 0 : \text{inc}_c : \hat{I}_w - w - 1 \quad \{\]
\[\quad \quad \quad \text{subimage} \leftarrow \hat{I}(r+ : h-1, c+ : w-1)\]
\[\quad \quad \quad \text{if } \text{classify}(\text{subimage})\]
\[\quad \quad \quad Detections \cup \leftarrow f_{\text{cumulative}} \cdot \{r, c, w, h\} \quad // \text{Save scaled detection}\]
\[\quad \quad \}\]
\[\quad \}
\[\]
\[f_{\text{cumulative}} \cdot = f \quad // \text{Compound scale factor}\]
\[\hat{I} = \text{resize}(I, f_{\text{cumulative}}) \quad // \text{Scale image to get next-smaller scale}\]
\[\}\text{while}(\hat{I}_h \geq h \land \hat{I}_w \geq w) \quad // \text{Continue while image has classifier-sized sub-windows}\]

This algorithm detects objects in the image from the minimum \(w\)-by-\(h\) resolution of the classifier up to the full size of the image \(I\).

\subsection*{4.1.2 Patch Features}

The SVL's SWOD implementation uses patch features, which are small pieces of other images that are compared with the region to be classified. In detail, these patches are convolved with the sub-window to achieve a “patch response image” describing their pixel-wise similarity/dissimilarity to it. Certain locations from this response image are read off and saved as feature values. The response values from each patch are stacked into a feature vector that represents the sub-window, and this is fed to the classifier.

More formally, we are given a “patch dictionary” \(D\) with \(n_d\) entries. Each entry, \(D_k\), has two
components: a patch image \( (D_{k,\text{patch}}) \), and a location \( (D_{k,\text{location}}) \) [38]. The patch is single-channeled and of a size between a minimum of 4-by-4–pixels to a maximum of 16-by-16–pixels (though the patch need not be square). In the SVL, the patch response image is computed as the normalized cross-correlation coefficient between the patch \( D_{k,\text{patch}} \) and the sub-window. Before feature values are extracted, however, the response image is subjected to a procedure called “max-pooling.” Afterwards the response value at \( D_{k,\text{location}} \) is read out and interred in the feature vector, \( f \nu \), at location \( f \nu[k] \).

We walk through a complete example of this processing below. Consider an object detector trained to recognize coffee mugs with a 32-by-32–pixel classifier. An enlarged test image is shown in Figure 4.2a. We are going to process this window with an example patch shown in Figure 4.2b—which was actually taken from the test image for simplicity. Call the patch size \( w \)-by-\( h \).

![Figure 4.2](image)

**Figure 4.2**: An example sub-window being presented to the classifier (a) and an example image patch (b). For illustration’s sake, the patch was drawn from the test image (location shown with a green rectangle).

First we compute the normalized cross-correlation coefficient (NCCC) between the patch and the image. The formula for normalized cross-correlation coefficient is given below as (4.1). The computation is done relative to the upper-left pixels of both the patch and the image instead of their center pixels. Also, instead of convolving the raw image \( I \) and the raw patch \( D_{k,\text{patch}} \), we convolve corrected versions that have had their means over the \( w \)-by-\( h \) patch-sized window subtracted out (\( \hat{I} \) and \( \hat{D}_{k,\text{patch}} \), respectively):

\[
C(r, c) = \frac{\sum_{i=0}^{h-1} \sum_{j=0}^{w-1} \hat{I}(r + i, c + j) \cdot \hat{D}_{k,\text{patch}}(i, j)}{\sqrt{\left(\sum_{i=0}^{h-1} \sum_{j=0}^{w-1} \hat{I}(r + i, c + j)^2\right) \cdot \left(\sum_{i=0}^{h-1} \sum_{j=0}^{w-1} \hat{D}_{k,\text{patch}}(i, j)^2\right)}}
\]  

(4.1)
\[ i = I - \frac{1}{wh} \sum_{i=0}^{w-1} \sum_{j=0}^{h-1} I(r + i, c + j) \]

\[ \hat{D}_{k,\text{patch}} = D_{k,\text{patch}} - \frac{1}{wh} \sum_{i=0}^{w-1} \sum_{j=0}^{h-1} D_{k,\text{patch}}(i, j) \]

The result of the convolution using the example image and patch of Figure 4.2 is shown in Figure 4.3a. Note, by design, the output lies between 0 and 1, with a correlation coefficient of 1 meaning the two patches are exactly the same; a coefficient of −1 meaning they are exact inverses of each other; and a coefficient of 0 meaning they are unrelated. In Figure 4.3a, the result is near 1 where the patch overlaps with the lower part of the mug handle (where it was, in fact, extracted from). The strong negative result at the lower right corner of the mug itself is where the light background and the dark mug body is the inverse of the dark mug handle and light background of the example patch (making them negatively correlated).

Figure 4.3: Normalized cross-correlation coefficient result and max-pooling result. (a) shows the NCCC response image; (b) the response after max-pooling; and (c) is a red–cyan color composite of (b) with the negative mug image, showing the strong response around the area from where the original patch was extracted.

After convolution, the next step is “max-pooling,” which smears positive results out over a larger area, the purpose of which being to make detections more robust to small changes in the input image. Since the patch features are read out at only one location each ($D_{k,\text{location}}$), max-pooling makes strong positives detectable even if they do not occur on $D_{k,\text{location}}$ exactly but within a small radius of it.

In the SVL, max-pooling is performed with image dilation with a radius $r = 3$, meaning each pixel in the convolution result image $C$ is replaced by the maximum value in a $2r + 1$-by-$2r + 1$-pixel
window centered around it. Formally,

$$C_{\text{maxpooled}}(r, c) = \max_{i=-r,\ldots,r} \max_{j=-r,\ldots,r} C(i, j)$$  \hspace{1cm} \text{(4.2)}$$

The result of max-pooling on the continuing example of Figure 4.3a is given in Figure 4.3b. To better illustrate the final result, in Figure 4.3c we present a color composite of the max-pooled response in cyan with a negative of the test image in red. Note the strong feature response over the lower handle of the mug, as desired.

### 4.1.3 Decision Tree Classifier

The specific classifier used here was a decision tree. This is a simple classifier (one of the first developed in machine learning [42]) that uses a tree structure to make progressively finer-grained split decisions until arriving at a classification result [43].

![Decision Tree](image)

**Figure 4.4:** Schematic of an example decision tree. Each decision node switches on a value from the feature vector, \(x\). Leaf nodes return probabilities of the classification result.

A schematic of an example decision tree is shown in Figure 4.4. A decision tree is composed of two types of nodes. The first type of node contains a binary test: given a feature index \(i\) (into a vector \(x\)) and a threshold value \(t\), the node tests if \(x[i] \geq t\), and, if true, it returns the output of the left child; if false, it returns the output of the right. Nodes of this first type can have only
two child nodes (obviously because of the binary nature of the decision). The second type of node returns a classification result directly: once that node is reached, it returns a probability $p$ that the input vector matches the positive examples the classifier was trained on (conditioned on all of the preceding decisions in the tree). These nodes have no children, and for this reason they are called “decision stumps.”

To classify a feature vector $x$, one passes it to the tree’s root. This node either returns a classification score directly (for a tree of depth 0) or subjects it to a binary test and passes it to either of its children, and so on until a classification result is returned. If the tree is $m$ nodes deep, then $m$ maximum decisions can be made in the classification on any one vector $x$. The tree is not required to be complete, thus certain decision branches can be shorter than others.

The decision tree classifier is a simple model; so much so that one might hardly expect it to perform well on complex machine learning problems, and especially the computer vision task. Indeed, the decision tree is capable only of dividing up the general $\mathbb{R}^n$ vector space into rectilinear subdivisions, and then only with divisions that are axis-aligned. For a simple, linearly separable dataset, unless the separating line is axis-aligned, the decision tree would require an infinite tree depth to capture the boundary to arbitrary precision. But decision trees are typically limited to depths of 3 or 4, and so are only able to approximate non-axis-aligned boundaries over finite ranges. Allowing a tree of infinite depth would doubtless over-fit the data, leading to poor generalization error (the expected error of the classifier on a test set of data that it does not see during training).

![Figure 4.5: Axis-aligned–linearly separable data being classified by a decision tree. The decision tree can achieve near-perfect classification with only one decision node.](image-url)
Motivating examples are shown in Figure 4.5 and Figure 4.6. Figure 4.5 shows the case with an axis-aligned–linearly separable dataset, which the decision tree is able to classify almost perfectly using only one decision boundary. In Figure 4.6, however, this same data is rotated by 45°, and decision trees with depths from 1 to 6 can only make successive approximations to this boundary.

Applying the decision tree to the object detection problem, obviously the chances are remote that, for some n-dimensional feature space collected from images of mugs, say, that the decision boundary between the object and non-object classes would be capture-able by a decision tree of moderate depth. Even if there existed such an ideal feature set, finding it would be perhaps no less difficult than solving the general object detection problem to begin with. Nevertheless, decision trees can be applied to this space through the important modification of the “boosting” algorithm. Decision trees are also useful because of their simplicity, which makes them one of the fastest models to train, and also one of the most amenable to using exceptionally large training datasets [40].

We do not cover the common training algorithms for decision trees (such as [44]) as we were only
concerned with optimizing the testing phase of the SWOD pipeline, where it was run on a novel image to obtain all classification results. Since we were not interested in optimizing the training procedure, these algorithms are not discussed.

4.1.4 Boosted Decision Trees

Boosting is a generalized machine learning procedure to train a series of “weak classifiers” (like the decision tree, that cannot make very complex decisions) in an ensemble manner to create a single “strong classifier” that performs well on complex problems [43]. It relies on a loop wherein a weak classifier is trained on a weighted dataset, i.e., each labeled training point, \((x_i, y_i)\), is now also associated with a weight \(w_i\). Together these weights define a probability distribution over the samples (i.e., \(\sum_{i=1}^{n} w_i = 1\)). After training each weak classifier, call it \(c\), the samples are re-weighted such that the samples that \(c\) classified correctly receive less weight, and those that \(c\) got wrong receive more weight. In the next iteration of the loop, a new classifier \(c'\) is trained on these newly weighted samples, and thus focuses on those samples that all previous classifiers got wrong. By training multiple weak classifiers to focus on each “hard” segment of the training set, we can combine them to create a much more powerful (and more accurate) final classifier.

We present boosting in pseudo-code below. After training each classifier \(c\) on the weighted samples, its weighted error rate \((\epsilon)\) is calculated, along with a weighting \((\alpha)\) for the classifier. Samples are then re-weighted, re-normalized, and the next loop begins. Not given is the exit condition, which can either be a set maximum number of iterations, or a convergence criterion on the incremental error rate, \(\epsilon\) (such as when it becomes so low that there is no need to continue training). Once training exits, the algorithm returns an ensemble classifier made of the weighted sum of all of the classifiers, \(c\), trained in the inner loop, and weighted relative to their individual accuracy (determined by \(\alpha\)).
$w_i \leftarrow \frac{1}{n}$ $\forall i$ // Initialize weights to be uniform

$C \leftarrow \{}$ // Set list of weighted classifiers to empty

do {

Train new classifier $c$ on $\{(x_i, y_i, w_i)\}_{i=1}^n$

$\epsilon \leftarrow \sum_{i=1}^n w_i 1\{c(x_i) \neq y_i\}$ // Weighted error rate of $c$

$\alpha \leftarrow \frac{1}{2} \log \frac{1 - \epsilon}{\epsilon}$ // Calculate alpha parameter

$C \cup \leftarrow \{(c, \alpha)\}$ // Save weighted classifier

// Re-weight samples

$w_i \leftarrow \begin{cases} w_i e^{-\alpha}, & c(x_i) \neq y_i \\ w_i e^{\alpha}, & c(x_i) = y_i \end{cases}$

where $z$ is a normalization constant to ensure $\sum_{i=1}^n w_i = 1$

} // Define output classifier as weighted ensemble

$\hat{C}(x) \leftarrow \sum_{i=1}^n \frac{C[i].\alpha \cdot C[i].c(x)}{\sum_{j=1}^n C[j].\alpha}$

In our reference SWOD implementation, the classifier used an ensemble of 200 boosted decision trees. While a single decision tree was inept at classifying an object, such an ensemble—when trained on a large dataset—proved able to achieve exceptionally high accuracy with near-perfect precision and over 95% recall [40].

4.1.5 Generalized Multi-Channel Images

Traditionally, a “multi-channel” image has been one where the multiple channels were used to convey color information, such as with multiple “primary” colors used for additive or subtractive mixing (as in the RGB and CMYK color spaces, respectively), or with hue and saturation channels (as in the HSV/HSL color space), or with luminance and chromaticity channels (as in the YCbCr color
space) [18]. But the concept of multi-channel images can be expanded to cover any type of multimodal visual information.

For the purposes of object detection, a multi-channel image is understood simply as a vector of multiple images, where each is of the same size and corresponds at a per-pixel level to the others, but which can contain information from any sensor modality. Commonly these extra channels are used for sensor information completely orthogonal to or complementary to traditional visual information. Examples include edge maps computed from an intensity channel [45], infrared camera images [46], 3-D sensors that create depth maps [47], and polarimetric sensors [48]. Integrating novel sensor modalities with existing vision systems has proven effective at increasing object detector performance [47].

The images we dealt with in the SWOD pipeline contained three such multi-modal channels. The first was a grayscale channel that contained the intensity camera image. The second was a soft edge map derived from the first channel. The third was a depth (or range) image obtained from a 3-D sensor (specifically, a high-resolution line-scanning laser [23]) which showed the distance of all objects from the camera. An example of all three channels for a typical image is given in Figure 4.7.

![Figure 4.7](image.png)

Figure 4.7: An image with generalized “channels.” (a) is an intensity (grayscale) image; (b) is a soft edge map; and (c) is a depth image.

Because of this generalization to multiple image channels, we must modify our definition of the patch dictionary slightly. In particular, each entry of our patch dictionary must now contain not only a patch and a location ($D_{k,patch}$ and $D_{k,location}$) but an index, $D_{k,channel}$, identifying which channel of the multi-channel image vector it is meant to be run on. The patch images we use for features can now be drawn from intensity images, from soft edge maps, and from depth images.
<table>
<thead>
<tr>
<th>Number of patches</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>3.3</td>
</tr>
<tr>
<td>87</td>
<td>22.2</td>
</tr>
</tbody>
</table>

Table 4.1: Preliminary SVL benchmarks for small patch dictionaries.

### 4.1.6 Reference Solution Benchmarks

We benchmarked the SVL reference solution to show how much optimization was necessary to bring us within range of real-time execution. The following numbers were obtained on a quad-core Intel Core 2 Duo at 2.4 GHz with 4 GB of RAM (a modern though not high-performance system). We ran the SVL SWOD with a native image resolution of 640-by-480 over a three-channel image as above. We ran classifiers with two different sizes of patch dictionaries, and using the OpenCV implementation of boosted decision trees.

The patch dictionaries used here were very small, yet the speeds were far from being real-time. This highlighted just what a significant speed-up would be needed with CUDA conversion.

### 4.2 Analysis of Potential CUDA Pipeline

Whereas our reference solution for the visual servoing algorithm was a straightforward, standalone application, the reference solution here came in the form of a large computer vision library, and we were concerned with optimizing only one facet of the library’s capabilities. Considering the spectrum of possible approaches, we might begin by picking the costliest operation of the library’s SWOD pipeline and moving that onto the GPU. For the SVL, benchmarking analysis of the above tests showed that over 80% of the time of the SWOD pipeline was spent performing convolution (with the OpenCV function `cvMatchTemplate`). Replacing this with a CUDA implementation of convolution would be an easy start; however, even if the convolution were optimized to take “zero” time compared to the rest of the pipeline, we could at best achieve a 5-times speedup. This was significantly below what we needed to achieve real-time speeds, which meant replacing more of the pipeline would obviously be necessary. So we undertook an analysis of the entire SWOD pipeline and examined the trade-offs (complexities versus speed-ups) of moving each segment onto the GPU.

We covered the steps of the SWOD pipeline above, which we now recapitulate:

1. Convolution – Process multi-channel input image to get response images for each patch feature
2. Max-pooling – Post-processing step on response images
3. Collect feature vectors – Extract values from response images to build feature vector
4. Evaluate decision trees – Consume feature vectors into vectors of evaluated decision trees
5. Evaluate classifier – Consume decision tree vectors into a single vector of classification results

This omits the outermost loop wherein the image is scaled down and the whole procedure repeated. Now we annotate each step by considering its complexity. We shall say we have \( m \) numbers of sliding windows to consider, \( n \) features in our classifier (and so \( n \) patch feature responses to compute per image), and \( t \) trees in our decision tree. The input images are taken to be \( w \)-by-\( h \) in size, and the maximum patch feature dimension size is \( p \). The max-pooling radius is \( r \). Finally there are \( c \) images in our multi-channel image vector.

1. Convolution
   - Complexity – \( O(nc(wh)p^2) \) – To compute \( nc \) convolution results
   - Memory – \( O(nc(wh)) \) – To store them

2. Max-pooling
   - Complexity – \( O(nc(wh)r^2) \) – To compute the \( r \)-radius max-pooling
   - Memory – \( O(nc(wh)) \) – To store them (same as above)

3. Collect feature vectors
   - Complexity – \( O(mn) \) – To read the total number of locations out
   - Memory – \( O(mn) \) – Storage space required

4. Evaluate decision trees
   - Complexity – \( O(mt) \) – Evaluating each tree on each sub-window
   - Memory – \( O(mt) \) – Space for \( t \) tree results for each of \( m \) windows

5. Evaluate classifier
   - Complexity – \( O(mt) \) – Summing up \( mt \) numbers
   - Memory – \( O(m) \) – Single probability result for each of \( m \) windows
Now we look at what will be the trade-offs of replacing each of these steps. The first step, convolution, is highly amenable to acceleration with CUDA, as per-pixel parallelization militates against the $wh$ term in the complexity. However, the memory complexity of replacing only the convolution step is staggering. While the costs of transferring the test images (only three channels thereof) and the few patches to the GPU are negligible, returning the $nc$ $w$-by-$h$ results from the GPU would be very expensive. With three-channel 640-by-480-pixel images and, say, 100 features, this constitutes a transfer bulk of 369 MB, which could take over a second. While CUDA can greatly accelerate CPU-based operations, this CPU-to-GPU memory transfer is one of the added costs of doing GPGPU (i.e., that has no analogue on the purely CPU-based side) and so only detracts from the final speed-up. Moving the max-pooling step onto the GPU, as well, would also benefit from per-pixel parallelization, but this would still leave the large $ncwh$ memory transfer burden.

We could greatly reduce the memory transfer burden, however, by moving the third step onto the GPU—the collection of feature vectors. Then we would need to transfer only an amount $mn$ back instead of $ncwh$. We know $m$ is less than $wh$ since the sliding windows are stepped through the image at four-pixel increments, meaning $m$ is 16 times less than $wh$, and so we have saved a factor of 16 in memory transfer. While moving convolution and max-pooling onto the GPU would accelerate the computation, we must not forget to reduce the amount of CPU-to-GPU memory transfer, as that remained the speed-limiting factor with the visual servoing algorithm’s GPGPU version even after heavy optimization.

There is also a benefit to moving step 4—the evaluation of decision trees—onto the GPU, as its computational complexity of $mt$ can be parallelized either in terms of trees ($t$) or in terms of sub-windows ($m$). This transforms the memory burden from $mn$ to $mt$, which could be better or worse depending on whether there are more trees than feature values or not.

The pipeline thus far will have created an $m$-by-$t$ matrix of decision tree results. The final step—classifier evaluation—is equivalent to summing this matrix along the tree (or $t$) dimension. This can be performed in parallel, which argues for moving this step onto the GPU. This would also reduce the memory transfer burden down to an absolute minimum of $m$: one number (the classification result) for each sliding window under consideration. If every possible sliding window is considered (under the stepping-by-four-pixels regimen), this constitutes a total memory cost of $wh/16$, which is far less than the $nwh$ we started with when we considered only moving the convolution onto the GPU.
We have determined that writing an entirely new, CUDA-based SWOD pipeline is the best approach. This effects the smallest amount of CPU-to-GPU memory transfer, and all of the composite operations are parallelizable, thus presenting no bottlenecks to CUDA conversion.

4.3 Conversion of Convolution

The 2-D convolution represented the most computationally intensive step of the SWOD pipeline, but also one most easily able to be sped up by CUDA conversion. Yet, because we were converting the entire SWOD pipeline to CUDA, then, since convolution was the most expensive step on the CPU, it was likely to remain the most expensive step on the GPU. Despite the facility of converting convolution to CUDA, we still had to optimize our approach so as not to weigh down the rest of the computational pipeline.

After studying other convolution examples in the NVIDIA SDK, we decided on a method for implementing the normalized cross-correlation coefficient (NCCC) calculation required by the SVL’s patch features. We chose a square, 16-by-16-thread block (common for image processing), and, for simplicity, anchored the template matching operation relative to the upper-left corners of the reference image and the patches. This way, the input pixel apron would lie only east (the \( w - 1 \) additional columns), south (the \( h - 1 \) additional rows), and southeast (the combination of both), not in all directions. We thus followed the approach of Section 2.5.3 and had each thread compute one output value, while some threads read in multiple input values.

While the image data was to be stored in global memory, we employed an optimization for storing the patch data. Since the images being processed could change but the patch images themselves would remain constant, we followed the form of other NVIDIA convolution implementations and stored the patch data in \texttt{constant} memory on the device [9]. This type of memory is globally accessible on the GPU but can only be written by memory transfers from the CPU; thus it is faster to read and does not need to be cached into shared memory as does typical read/write GPU memory.

In the next sections we will discuss how we devised an exact algorithmic strategy for efficiently computing the complicated form of the normalized cross-correlation coefficient given in (4.1). That section concludes by presenting a streamlined kernel for the NCCC operation, which will contain some novel CUDA optimizations.
4.3.1 Efficient Calculation of NCCC

As discussed previously, calculating NCCC involves not merely computing the element-wise product of the sub-window and the patch, but the convolution with mean-subtracted versions of the sub-window and patch, and then normalization by the squared sums of the same. To see how we could compute this most efficiently with CUDA, we revisit the form of (4.1):

\[
C(r,c) = \sum_{i=0}^{h-1} \sum_{j=0}^{w-1} (I(r+i,c+j) - \bar{I}_{r,c}) \cdot (D_{k,patch}(i,j) - \bar{D}_{k,patch}) \\
\cdot \left( \sum_{i=0}^{h-1} \sum_{j=0}^{w-1} (I(r+i,c+j) - \bar{I}_{r,c})^2 \right)^{-1/2} \\
\cdot \left( \sum_{i=0}^{h-1} \sum_{j=0}^{w-1} (\hat{D}_{k,patch}(i,j) - \bar{D}_{k,patch})^2 \right)^{-1/2} 
\]

(4.3)

\[
\bar{I}_{r,c} = \frac{1}{w \cdot h} \sum_{i=0}^{h-1} \sum_{j=0}^{w-1} I(r+i,c+j)
\]

Dropping various subscripts to simplify (namely .patch), we have:

\[
C(r,c) = \frac{\sum_{i,j} (I(r+i,c+j) - \bar{I}_{r,c}) \cdot (D_{k}(i,j) - \bar{D}_{k})}{\sqrt{\left( \sum_{i,j} (I(r+i,c+j) - \bar{I}_{r,c})^2 \right) \cdot \left( \sum_{i,j} (\hat{D}_{k}(i,j) - \bar{D}_{k})^2 \right)}} 
\]

(4.4)

Since there will be a finite number of patches used, we could pre-compute the patch mean values \(\bar{D}_{k}\) and the sum of squared patch normalized values \(\bar{D}_{k, squared} = \sqrt{\sum_{i,j} (\hat{D}_{k}(i,j) - \bar{D}_{k})^2}\). But, while this works for the patch, it cannot be done for the image, so the formula as written still requires us to compute \(\bar{I}_{r,c}\) before we compute the main convolutional product. This requires two passes over the convolution sub-window.

We can fix this if we expand the summations in the numerator as follows:

\[
C(r,c) = \frac{\sum_{i,j} I(r+i,c+j)D_{k}(i,j) - I(r+i,c+j)\bar{D}_{k} - D_{k}(i,j)\bar{I}_{r,c} + \bar{I}_{r,c}\bar{D}_{k}}{\bar{D}_{k, squared} \cdot \sqrt{\left( \sum_{i,j} (I(r+i,c+j) - \bar{I}_{r,c})^2 \right)}}
\]

(4.5)
Dropping indices and examining only the numerator:

\[
\sum_{i,j} (ID_k - ID_k - D_k \bar{I}_{r,c} + \bar{I}_{r,c} \bar{D}_k)
\]  
(4.6)

\[
= \sum_{i,j} (ID_k) - \sum_{i,j} (ID_k) - \sum_{i,j} (D_k \bar{I}_{r,c}) + \sum_{i,j} (\bar{I}_{r,c} \bar{D}_k)
\]  
(4.7)

\[
= \sum_{i,j} (ID_k) - \bar{D}_k \sum_{i,j} (I) - \bar{I}_{r,c} \sum_{i,j} (D_k) + wh \bar{I}_{r,c} \bar{D}_k
\]  
(4.8)

\[
= \sum_{i,j} (ID_k) - wh \bar{I}_{r,c} \bar{D}_k - wh \bar{I}_{r,c} \bar{D}_k + wh \bar{I}_{r,c} \bar{D}_k
\]  
(4.9)

\[
= \sum_{i,j} (ID_k) - wh \bar{I}_{r,c} \bar{D}_k
\]  
(4.10)

We can similarly expand the image sub-window sum in the denominator:

\[
\sum_{i,j} (I(r + i, c + j) - \bar{I}_{r,c})^2
\]  
(4.11)

\[
= \sum_{i,j} (I^2) - \sum_{i,j} (2I \bar{I}_{r,c}) + \sum_{i,j} (\bar{I}_{r,c}^2)
\]  
(4.12)

\[
= \sum_{i,j} (I^2) - 2I_{r,c} \sum_{i,j} (I) + wh \bar{I}_{r,c}^2
\]  
(4.13)

\[
= \sum_{i,j} (I^2) - 2wh \bar{I}_{r,c}^2 + wh \bar{I}_{r,c}^2
\]  
(4.14)

\[
= \sum_{i,j} (I^2) - wh \bar{I}_{r,c}^2
\]  
(4.15)

Re-combining the equations, we have:

\[
C(r, c) = \frac{\sum_{i,j} (ID_k) - wh \bar{I}_{r,c} \bar{D}_k}{\bar{D}_{k, squared} \cdot \sqrt{\sum_{i,j} (I^2) - wh \bar{I}_{r,c}^2}}
\]  
(4.16)

The benefit of the form of (4.16) is that all of the values can be computed with only one pass through the image sub-window. \(\sum_{i,j} (ID_k)\) is the plain element-wise product between the image and the patch, as if we were doing plain convolution. \(\bar{I}_{r,c}\) can be computed merely by summing up
the values of $I$ we see as we perform the first loop. Similarly, $\sum_{i,j} I^2$ can be computed by summing up the values of $I^2$ simultaneously.

As one final optimization, we note that the value of $\bar{I}_{r,c}$ requires normalizing by $wh$ to compute, and floating-point division is a costly operation. A better approach is to have the value $1/wh$ pre-computed (perhaps passed into the kernel) and to multiply by this value instead wherever normalization is called for. Applying this strategy, (4.16) becomes:

$$C(r, c) = \frac{\sum_{i,j}(ID_k) - \frac{wh}{wh}\left(\sum_{i,j} I\right)D_k}{\bar{D}_{k,\text{scaled}} \cdot \sqrt{\sum_{i,j}(I^2) - \frac{wh}{wh}\left(\sum_{i,j} I\right)^2}}$$

(4.17)

$$C(r, c) = \frac{\sum_{i,j}(ID_k) - \left(\sum_{i,j} I\right)\bar{D}_k}{\bar{D}_{k,\text{scaled}} \cdot \sqrt{\sum_{i,j}(I^2) - \frac{1}{wh}\left(\sum_{i,j} I\right)^2}}$$

(4.18)

With this form, we need to compute these three sums on one pass through the image:

$$\text{sumConvolution} = \sum_{i=0}^{h-1} \sum_{j=0}^{w-1} I(r+i, c+j)D_{k,\text{patch}}(i, j)$$

(4.19)

$$\text{sumImage} = \sum_{i=0}^{h-1} \sum_{j=0}^{w-1} I(r+i, c+j)$$

(4.20)

$$\text{sumImageSquared} = \sum_{i=0}^{h-1} \sum_{j=0}^{w-1} (I(r+i, c+j))^2$$

(4.21)

Combining them with the form of (4.16) will then give us the normalized cross-correlation coefficient result. This is rendered as a kernel below:

```c
__shared__ float data[DATA_SIZE];

// Data is read in. Patch is in patchData.
float sumConvolution = 0.0f;
float sumImage = 0.0f;
float sumImageSquared = 0.0f;
for (int i=0; i<h; ++i) {
  for (int j=0; j<w; ++j) {
    const float dataValue = data[ (r+i)*dataWidth + (c+j) ];
    const float patchValue = patchData[ i*patchWidth + j ];
    sumConvolution += dataValue * patchValue;
    sumImage += dataValue;
  }
}
```
\begin{verbatim}
80 sumImageSquared += dataValue * dataValue;
81
82 // Ninv is 1.0/float( patchWidth * patchHeight );
83 float result = __fdividef( sumConvolution - patchMean * sumImage,
84 sqrtf( sumImageSquared - Ninv*sumImage*sumImage ) * patchSquaredMean );
\end{verbatim}

\subsection{4.3.2 Loop Unrolling and Templating}

The CUDA optimization strategies we presented in Section 2 were part of the general best practices in writing CUDA. While following these rules is important, once they have been employed, there are even more ways to optimize complex functions. One of these is loop unrolling. This is a practice that originated in the CPU domain, of replacing an explicit loop with the sequential instructions that are executed by each iteration of the loop. For example, we present a simple loop below:

\begin{verbatim}
for ( int i=0; i<5; ++i )
    c[i] += a[i]/float(i+1);
\end{verbatim}

If we were to unroll this loop, it would become:

\begin{verbatim}
c[0] += a[0]/float(1);
c[1] += a[1]/float(2);
c[2] += a[2]/float(3);
c[3] += a[3]/float(4);
c[4] += a[4]/float(5);
\end{verbatim}

Oftentimes, the latter style of explicitly writing out all of the instructions to execute is faster than constructing the loop statements “dynamically” by use of the loop variable and checking the termination conditions of the loop on every pass.

A limitation on loop unrolling is that it can only be applied when the size of the loop is statically known (i.e., at compile time). This is the case in the example given above, but, were the loop from 0 up to some value \( n \), whose value were not constant, then loop unrolling would not be impossible. The only variant of loop unrolling would could employ was if we knew a lower bound on \( n \), such as \( n \geq k \). Then we could unroll the first \( k \) iterations of the loop and write a physical loop that handled the rest of the iterations.

Loop unrolling is most efficient when the loops are small (less than 50 iterations, say) and the expense of storing and bringing in the extra, explicit instructions is outweighed by the amount of time saved not by having to execute the loop control mechanisms. Also, if one wishes to unroll
multiple nested loops, then the size of each loop must be statically known (though unrolling multiple levels of loops can be less efficient as the number of instructions is exponential in the sizes of all loops).

The convolution function we have posited above is ripe for loop unrolling. It contains only two small nested loops: one over the width and one over the height of the patch. Yet, to template over these parameters, we had to make them statically known to the program—i.e., the patch width and height could not simply be passed as arguments to the function. A compromise was C++-style templating, which allows one to explicitly create different versions of a function based on statically known integral values. A simple example is given below:

```c++
// template<int N>
float foo(float *x) {
    float ret = 0.0f;
    for (int i=0; i<N; ++i)
        ret += x[i];
    return ret;
}

int main(int argc, char *argv[]) {
    if (condition) {
        float x5 = foo<5>(x);
    } else {
        float x10 = foo<10>(x);
    }
}
```

The two calls on lines 10 and 11 go to different versions of the function `foo`, written based on the value of the templating variable `N` (this is an example of “heterogeneous polymorphism,” in contrast to the “homogenous polymorphism” brand of templating that other languages such as Java and C♯ use [49]). While there is no explicit loop unrolling visible here, a good compiler can unroll small loops itself when the dimensions are statically known [50]. The checking of `condition` in the main loop then allows us to dynamically dispatch to one of two functions, each of which uses a different statically known `N` to unroll its inner loop.

We demonstrate loop unrolling with a kernel that executes the simple program of (4.23). Assume we have many threads computing this loop (now of size `n`) over vector inputs `c` and `a`:

```c
__global__ void foo(float *c, float *a, int n) {
```
As it is, the loop over $i$ cannot be unrolled because its length (as defined by $n$) is not known at compile time. We can fix this by templating with $n$ as an int parameter $N$:

```
const int ix = n * threadIdx.x;
for (int i=0; i<n; ++i )
c[ix + i] += a[ix + i]/float(i+1);
```

The loop in this code is now unrollable. We have also inserted the CUDA directive "\#pragma unroll" before the loop to explicitly request loop unrolling. Also note the (perhaps confusing) "<...><<<...>>>") syntax used to call templated kernels. We mentioned previously (Section 2.1) that the triple bracket syntax was derived from templated function syntax, and now one can see them both functioning side by side.

While the above construct lets us use loop unrolling, it leaves the problem that the templating parameter must still be statically known. If this is not the case (as with our patch width and heights), we can simply follow the form of (4.25) and set some dynamic dispatch control code before the kernel call, switching on the value of non-statically known variable to call different versions of the templated function:

```
int main() {
  switch (n) {
    case 5: foo<5><<< 1, 256 >>>(c, a); break;
    case 10: foo<10><<< 1, 256 >>>(c, a); break;
    case 15: foo<15><<< 1, 256 >>>(c, a); break;
  }
}
```
<table>
<thead>
<tr>
<th>Templating</th>
<th>Average execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>1.74</td>
</tr>
<tr>
<td>Patch width</td>
<td>1.32</td>
</tr>
<tr>
<td>Patch height</td>
<td>2.01</td>
</tr>
<tr>
<td>Patch width and height</td>
<td>1.52</td>
</tr>
</tbody>
</table>

Table 4.2: Average runtimes over all patches for convolutions with different styles of templating.

While this lets use loop unrolling, it comes at the price of this extra control code that must sit before the function call. Though, in terms of end-to-end runtime, typically the overhead incurred is far under-cut by the savings from loop unrolling. The main drawback is then really the unsightliness of the code duplication necessary for this dynamic dispatch. Where before we had only one function call, now we need one version of the call for every value of our templated parameters.

Applying this to the convolution example, we see we can directly apply this technique to unroll the loops over patch width and height that lie inside of the kernel (4.22). But, to handle all patches from size 4-by-4 up to 16-16, we would need 13 call versions for both the individually width- and the height-templated versions, and 169 call versions for the double templating version. To militate against such code bloat, we decided to template at a coarser grain of 4-pixel increments. Thus, when templating over patch widths, we would handle sizes of only 4, 8, 12, and 16 pixels (four options). In the double templating case, we would need only 16 call versions. The trade-off is that now all patch data must be padded to one of these sizes: a convolution with a 9-by-6–pixel patch, say, must be run as a 12-by-8–pixel convolution.

We implemented the NCCC calculation in a kernel following the ultimate form derived in Section 4.3.1, and we performed loop unrolling over the coarse grain described above. We ran benchmarks where we convolved all patches from size 1-by-1 up to 16-by-16 (even though patches smaller than 4-by-4 were not used in the SVL, we ran the numbers for completeness) on a 640-by-480–pixel image. The runtimes per convolution are given (in milliseconds) in Figure 4.8, and we report the average runtimes for each version in Table 4.2.

Obviously the runtimes are roughly proportional to the size of the patch, \( wh \), (and so the size of the inner loops) in all versions. For the version with no templating (the upper left figure), the runtimes vary for each patch size run. In the functions with templating, the runtimes are nearly uniform across the “templating bins” (where smaller patches’ runtimes are “rounded up” to the
Figure 4.8: Benchmarking results of variously templated versions of the normalized cross-correlation coefficient calculation kernel of Section 4.3.1. All runtimes are given in milliseconds, and all color scales are identical.

runtimes of the next padding size). Since the scales of the heat maps are all identical, though, we can see immediately that templating over width is the most effective, reducing maximum runtime to around 3.55 ms from the version with no templating whose max lay was 5.36 ms (a 51% speed-up). Templating over the height, however, actually seemed detrimental. But this is explainable if the height-wise templating did not increase efficiency at all, and so the function only suffered from the “rounded up” runtimes to process the extra rows. Meanwhile, templating with both width and height also underperformed the version with only width-wise unrolling.

We may reason that only width-wise unrolling is beneficial because, in the kernel of (4.22), the loop over width was the inner loop, and the loop over height was the outer (an order chosen to favor sequential reads to memory). Loop unrolling only the outer loop is equivalent to writing multiple versions of the inner loop, which saves only $O(h)$ loop overhead. Unrolling the inner loop, however, saves $O(w)$ overhead $h$ times. Thus we experimented with unrolling over height but with the order
of the `for` loops reversed, with width on the outside and height on the inside, going from:

```c
#pragma unroll
for ( int i = 0; i < H; ++i ) {
    for ( int j = 0; j < w; ++j ) {
        (4.29)
    }
}
to:

```c
for ( int j = 0; j < w; ++j ) {
    #pragma unroll
    for ( int i = 0; i < H; ++i ) {
        (4.30)
    }
}
```

Benchmarking this version of height-wise unrolling, the average runtime was lower, but had dropped from 2.01 ms only to 1.84 ms, making it still the most expensive option. From these results, we decided to use only the width-wise-templated version of convolution.

### 4.4 Integral Images

The above implementation of the normalized cross-correlation coefficient calculation worked, but it proved expensive. A typical CUDA convolution over a 640-by-480-pixel image took an average of around 1 ms, but this function above took over 3 ms. This is obviously because the inner loop of the algorithm contains roughly three times the work as a plain convolution, with three multiply-accumulates instead of just one. The additional work is necessary to compute the image sum and the squared image sum. However, it is a well-known technique in image processing that the computation of image sums can be sped up with integral images, as shown by Viola and Jones in [51] (indeed, this is the technique that the SVL uses for computing its response images).

An integral image is defined as the pixel-wise integration of another image. Starting from an input image $I$ of size $w$-by-$h$, each value of the integral image is the sum (or integral) of all other pixel values in $I$ that lie to the left and above the integral image pixel location. Specifically, the value of the integral image’s pixels are given as:

$$I_{\text{integral}}(r, c) = \sum_{i=1}^{r} \sum_{j=1}^{c} I(i, j)$$  \hspace{1cm} (4.31)

With this, computing the sum over any rectangular region in $I$ can be accomplishable as the addition of only four value from the integral image. In particular, for the sum starting at location
(r, c) and with a size of (w, h), the values to be added from the integral image are:

\[
\sum_{i=r}^{r+h-1} \sum_{j=c}^{c+w-1} I(i, j) = I_{\text{integral}}(r + h - 1, c + w - 1) - I_{\text{integral}}(r + h - 1, c - 1)
- I_{\text{integral}}(r - 1, c + w - 1) + I_{\text{integral}}(r - 1, c - 1) \tag{4.32}
\]

This can also be shown mathematically by successively conflating the integrals of (4.32):

Figure 4.9: Illustration of integral images. The heat map shows the integral image (non-decreasing in value from left to right and top to bottom). To calculate the sum of the original image over region IV, the four pixels shown are added as (4) – (1) – (3) + (2).

This is illustrated in Figure 4.9. To achieve the sum of the region labeled “IV” in the image, we start by accumulating the value of the integral image at pixel location 4. This gives us the sum of the regions I, II, III, and IV. Next we subtract the values of the integral image at locations 1 and 3, which subtracts the sums from the regions I and II, and III and II, respectively. Thus our running sum is “I + II + III + IV – I – II – III – II,” or “IV – II.” We have doubly subtracted the sum from region II, and so must add back the value of the integral image at location 2 to achieve the desired result.

This can also be shown mathematically by successively conflating the integrals of (4.32):
\[ I_{wh} - I_{0h} = \sum_{i=1}^{r+h-1} \sum_{j=1}^{c+w-1} I(i,j) - \sum_{i=1}^{r+h-1} \sum_{j=1}^{c-1} I(i,j) \]
\[ = \sum_{i=1}^{r+h-1} \left( \sum_{j=1}^{c+w-1} I(i,j) - \sum_{j=1}^{c-1} I(i,j) \right) \]
\[ = \sum_{i=1}^{r+h-1} c+w-1 \sum_{j=1}^{c} I(i,j) \]
(4.33)

\[ I_{w0} - I_{00} = \sum_{i=1}^{r-1} \sum_{j=1}^{c+w-1} I(i,j) - \sum_{i=1}^{r-1} c-1 \sum_{j=1}^{c} I(i,j) \]
\[ = \sum_{i=1}^{r-1} c+w-1 \sum_{j=1}^{c} I(i,j) - \sum_{j=1}^{c-1} I(i,j) \]
\[ = \sum_{i=1}^{r-1} c+w-1 \sum_{j=1}^{c} I(i,j) \]
(4.34)

\[ I_{wh} - I_{0h} - I_{w0} + I_{00} = I_{wh} - I_{0h} - (I_{w0} - I_{00}) \]
\[ = \sum_{i=1}^{r+h-1} \sum_{j=1}^{c+w-1} I(i,j) - \sum_{i=1}^{r-1} \sum_{j=1}^{c+w-1} I(i,j) \]
\[ = \sum_{j=c}^{c+w-1} \left( \sum_{i=1}^{r+h-1} I(i,j) - \sum_{i=0}^{r-1} I(i,j) \right) \]
\[ = \sum_{j=c}^{c+w-1} \sum_{i=r}^{r+h-1} I(i,j) \]
(4.35)

For computing the normalized cross-correlation coefficient, we currently must compute the image sum and the squared image sum over every sub-window, which amounts to tripling the computational work. But, with an integral image and a squared integral image (i.e., the integral image over the squared pixel values) of \( I \), the extra sums could each be computed in constant time. The main loop would return to being no more expensive than mere convolution.

### 4.4.1 Integral Image Precision Errors

Our first attempt to include integral images in our convolution procedure, however, encountered problems with numerical errors. Typically, integral images are computed with double-precision floating-point accuracy. This is essential with a 640-by-480–pixel image, where the “summit pixel”
in the integral image must represent the value of 1.2 million image pixels without loss. If there is
error in the upper-range values of the integral image, then window sums computed based on those
values will be in error. Any error in the integral images is intolerable here as the NCCC calculation
is sensitive to the precise values of the sums (which are used in the normalization factors). The
output values of the NCCC calculation are supposed to lie between $-1$ and $1$, but this condition
can be violated if there are any errors in the chain of calculation.

Yet the version of CUDA we were using provided support only for single-precision floating-point
numbers. Double-precision was available on some GPUs, but only then with a high computational
cost. We did not wish to vitiate the optimization provided by integral images by invoking the higher
cost of using double-precision numbers; so first we implemented integral images with single-precision
floating-point.

![Figure 4.10: Errors when computing integral images with single-precision floating-point numbers.](image)
The top row pertains to integral images and the bottom row to squared integral images. The left
column shows ground truth results with double-precision. The middle column shows the absolute
error versus the single-precision–based integral images. The right column is a thresholded version
of the middle column, showing correct pixel values in blue with incorrect values in red.

The results of these tests with single-precision integral images are shown in Figure 4.10. We
generated a random image with integer pixel values between 0 and 255 (as they are in the SVL
pipeline) and computed the integral and squared integral images in both double and single precision.
The top row shows the results for the integral image and the bottom row shows the results for the squared integral image. The left column shows ground truth result with double-precision, and the middle column shows the absolute error with the single-precision–based image. The right column is a thresholded version of the middle column, showing pixels in error in red and correct pixels in blue. While error accumulates slowly with the integral image, only manifesting itself in the upper reaches of the sum, error appears in the squared integral almost immediately.

Yet the magnitudes of the errors seen in Figure 4.10 are small, with the mean pixel-wise error being $5.29 \cdot 10^{-6}\%$ and $1.99 \cdot 10^{-5}\%$ for the integral and squared integral images, respectively. One could hope that that whatever errors “propagate” into the NCCC result will themselves be insignificant. This hope is repudiated by Figure 4.11, however, where we actually calculated the effect of using single-precision integral images upon a patch feature response image (the NCCC calculation). The ground truth response image is given in (a); that obtained with single-precision integral images is given in (b); and the absolute error between (a) and (b) are shown in (c). While the features match over most of the image, large errors (up to 1.2 in magnitude) manifest in the lower right corner of the image, where the errors in the integral images become most egregious. Since the feature values should range only from $-1$ to 1, this means we have some pixels that have flipped in sign relative to the ground truth image. This level of error is obviously unacceptable.

![Figure 4.11](image)

Figure 4.11: Errors in the patch feature response image (NCCC result of (4.1)) resulting from the use of single-precision numbers for the integral images. (a) shows the ground truth feature values; (b) the values from using single-precision integral images; (c) the absolute error between the two images.

### 4.4.2 Block-Wise Integral Images

Searching for a solution to the single-precision problem, we examined other ways to get the benefits of integral images without having to compute the full sums (which showed errors only near the ends
of the computation). One way to fix this is with block-wise integral images. Instead of computing the integral uninterrupted over the whole image, we would compute contiguous integrals only over square-shaped blocks of the image. For a block size of $d$, the pixel values of these block-wise integral images would be:

$$I_{\text{block,int}}(r,c) = \left\lceil \frac{(r-1)/d}{d} \right\rceil + \sum_{i=\lfloor (r-1)/d \rfloor + 1}^{\lfloor (r-1)/d \rfloor + d} I(i,j)$$

An example block-wise integral image is shown in the heat map of Figure 4.12. The benefit of using block-wise images is the sums inside the blocks are smaller than the whole image sums, so we can possibly tune the block size so as to have no errors inside of each block’s integral. The drawback is the increased complexity of computing image sums. When the region we want the sum over lies completely within one block, then the sum can be computed with only four summations as with a regular integral image. When this region lies over the boundaries of several blocks, however, the task becomes more complicated, possibly taking up to nine summations.

![Figure 4.12: Sample block-wise integral image showing for integrated blocks. To compute the sum of the dotted-outlined image region requires computing a convoluted sum between the nine labeled pixels.](image)

With block-wise integral images, we can possibly pick a block size that will avoid passing errors
to the final computation. This analysis is shown in Figure 4.13. These are the thresholded error images from Figure 4.10, with possible block sizes drawn over them. For the integral image error image on the left, choosing a block size of 512 would still leave us with sums too large to represent without loss, but a block size of 256 would make us reasonably certain of not having any errors in our results. For the squared integral image on the right, however, even using a block size of 32 would probably let us see errors in the top few pixels. Only using the incredibly small block size of 16 would we be assured of having no errors.

Figure 4.13: How selecting the block size in block-wise integral images can affect whether error is carried over into the result. This uses the integral error images of Figure 4.10.

Though, as we saw in Figure 4.11, the upper-left 128-by-128-pixel block of the response image did not seem to contain significant error. If we implemented block-wise integral images with that a block size of 128, then, the integral image would not be in error, and perhaps the errors in the squared integral image would not significantly skew our results. Yet, in converting the SWOD pipeline to CUDA, we did not want to make any compromises on matching the output between the two versions where it was possible to avoid. Accepting error in one sector of the conversion would make further development of the library difficult as these various “hacks” would always have to be re-examined for suitability every time a change was made. Using a block size of 16 for the integral images would have been too tedious and computationally limiting (as even the CUDA thread block size would be larger), so we decided to seek another solution. Yet, had block-wise integral images been able to eliminate error with reasonable block sizes, we might have employed them in our conversion, and they would likely be a feasible solution for other applications.
4.4.3 Row-wise Integral Images

The next solution we explored was in the same spirit of block-wise integral images but reduced even further. Instead of computing the integral over a limited 2-D range, we imagined computing it over a limited 1-D range, or doing block-wise integral images only over rows. Formally,

\[ I_{\text{row, int}}(r, c) = \left\lfloor \frac{(c-1)}{d} + d \right\rfloor + \sum_{j=\left\lfloor \frac{(c-1)}{d} + 1 \right\rfloor}^{\left\lfloor \frac{(c-1)}{d} + d \right\rfloor} I(r, j) \] (4.37)

This way, for a block size of \( d \), we are only summing up \( d \) pixels over a block instead of \( d^2 \), so these row-wise integral errors should be robust to error for much larger values of \( d \) than were the block-wise integral images. Indeed, we saw above that even a block size of 32 for squared integral images was too much, as it involved summing 1024 values. But, with row-wise integral images, we could use a block size of 512 and likely be under the limit. A sample row-wise integral image for a non-random input image with a block size of 256 is shown in Figure 4.14. We verified that, with this block size, neither the integral nor the squared integral image had any errors.

![Figure 4.14: Example row-wise integral image over a non-random 640-by-480-pixel input with a block size of 256.](image)

A benefit of row-wise integral images is that we need to add only two values to get the sum in a region over a row. When the region lies over a seam, we need to make only one more addition. The trade-off is, to get the full sum over a 2-D image region, we must of course add up all of the sums across the rows, meaning a loop over the height of the region is required. In the context of
computing the NCCC result, however, this is not too deleterious. We are already running a double loop over the patch width and height to compute the convolutional product, and we would only need to include this summation over row-wise integrals in the outermost loop over height. We show the altered kernel to compute the NCCC result in this manner below:

```c
float sumConvolution = 0.0f;
float sumImage = 0.0f;
float sumImageSquared = 0.0f;
for (int i = 0; i < h; ++i) {
    sumImage += rowIntegral[(r+i)*dataWidth + w + c] - rowIntegral[(r+i)*dataWidth + c];
    sumImageSquared += rowSquaredIntegral[(r+i)*dataWidth + w + c] - rowSquaredIntegral[(r+i)*dataWidth + c];
    for (int j = 0; j < w; ++j) {
        const float dataValue = data[(r+i)*dataWidth + (c+j)];
        const float patchValue = patchData[i*patchWidth + j];
        sumConvolution += dataValue * patchValue;
    }
}
```

The original form of our kernel to compute the NCCC result without integral images (4.22) computed the integral and squared integral sums in the innermost loop, being $O(wh)$ in work. With integral images, this would have been reduced to $O(1)$, but unfortunately we could not use integral images—nor even block-wise integral images for the same $O(1)$—because of numerical errors. This solution with row-wise integral images, however, introduces no error and is only $O(h)$ in work.

Another benefit of using the row-wise integral image is that it can be computed endogenously, whereas block-wise integral images could not be. While block-wise integration obviously is endogenous, to compute the result with a block size of $d$, we would have needed $d^2$ threads to be able to compute each block’s result in a self-contained manner. As this is outside of our current thread limit even for small values of $d$, the resulting kernels would likely have been convoluted, and we might have had to use several stages to achieve the final result (as in Section 3.4.3). With the row-wise integral images, however, the numbers of threads required to compute each block is only $d$, which means these functions can be easily implemented.
### 4.4.4 Integral Image Benchmarking

With an integral image solution, we set about benchmarking its effect on the convolution pipeline. In the benchmarking trials, we ran one instance of both computations (convolution with and without integral images) with each patch size from 4-by-4 pixels up to 16-by-16 pixels. The raw results for convolution function runtimes are given in Figure 4.15. Obviously using integral images has sped up the results measurably, reducing the average runtime from 1.70 µs over all patch sizes to only 1.17 µs (a gain of 45%).

![Figure 4.15: Benchmarking of NCCC functions, (a) without integral images, and (b) with. (c) shows the relative speed-up from (a) to (b). While the smallest patch sizes can be processed faster without integral images, they help significantly on the larger patch sizes.](image)

What remained to be seen was whether this speed-up compensated for the extra time required to compute the integral images. In the trial, computing the two integral images (of size 640-by-480) cost a total of 1.12 µs. This is less than the average cost of one convolution, so obviously this is not a significant draw. In the SVL pipeline, we would only need to call this function once for every channel in our multi-channel image, or only three times in the current pipeline. Compared to the hundreds of convolutions we would be doing over each image, this is trivial.

Though there was one complication to mention. It is important to note that we did not use exact form of (4.38) when implementing this convolution. Instead of computing the sum over the column region as in the lines 9–12, we pre-computed the image sum over the patch width with a separate kernel. Thus the integral images fed into the convolution were not actually row-wise integral images but images where each pixel represented the sum of the next \( p_w \) columns (where \( p_w \) was obviously the width of the patch). The sum accumulation step in the kernel thus became:

```cpp
1 \text{sumImage} += \text{rowIntegral}[ (r+i) \times \text{dataWidth} + c ];
2 \text{sumImageSquared} += \text{rowSquaredIntegral}[ (r+i) \times \text{dataWidth} + w + c ];
```
We opted for these pre-computed patch-width sums because they significantly simplified the convolution kernel. The computation presented in (4.38) was deceptively simple as it had only two terms; we did not show the checks for whether the window lay over a seam in the integral image, and so required a third term for correction. Placing this in a separate function reduced the amount of math and checking done in the already burgeoning convolution function. Also, with this modification the use of the integral images became endogenous, as we no longer had to read in integral image data to the east of the block’s position.

The drawback was that now the integral images had to be processed to find the patch-width sums before every convolution, thus saddling the cost of convolution with that of this new kernel. However, observe that we were computing these sums over the patch widths, which were constrained to only ever take on 13 values (specifically, from 4 pixels to 12 pixels). Even if we had to compute 400 patch feature responses, we should only need to perform this patch-width pre-computation 13 times. Though, if we were to process the patches in any order, we would have to have pre-computed these 13 versions of the row-wise integral image before we began. This represents a large amount of data to allocate on the GPU. We could optimize this away and keep only one version of the row-wise integral image at a time by reordering the patches to group those of the same width to be processed sequentially. This was the approach we employed in the above trial.

The cost of this patch width normalization step turned out to be a trivial 125 µs per kernel. Also, we only needed to run this 26 times (13 times for all patch widths, and then once each for both the integral and the integral squared images), which allotted a total runtime of 3.25 ms. This was nearly a hundred times less than the total time spent on convolution, so the burden was trivial.

With this we were confident that we had achieved an optimized convolution function that made an effective use of integral images.

4.5 Conversion of Max-Pooling

We now cover how we implemented an optimized version of the max-pooling step. Given a response image, \( R \), we wished to perform image dilation with radius \( r \) to achieve the max-pooled result:

\[
R_{\text{max,pooled}}(i, j) = \max_{m=-r,\ldots,r} \max_{n=-r,\ldots,r} R(m, n)
\]  

(4.40)
To implement this exact equation in CUDA, however, would be onerous. This is obviously an
exogenous function, and, as written, its input data apron lies to all directions.

But, as part of the SWOD conversion, we were told that the locations attached to features value
(i.e., \(D_{k, \text{location}}\)) would never ask for invalid values. Specifically, the features would never ask for a
response in the top \(r\) rows of the image, the left \(r\) columns of the image, or likewise so far to the
right or down that part of the convolution template would lie outside of the sliding window. Because
of this constraint, we could instead compute the function:

\[
R'_{\text{max,pooled}}(i, j) = \max_{m=0,\ldots,2r+1} \max_{n=0,\ldots,2r+1} R(m, n)
\]

(4.41)

Obviously this computes most of the same results of (4.40), but with an offset. Because of this offset,
(4.41) does not compute the values of (4.40) that lie along its top \(r\) rows and left \(r\) columns. But we
would never need these values anyway because of the guarantee given above about which locations
would be polled. Therefore computing either (4.40) or (4.41) would give us all needed values, with
only an offset required to reinterpret the results from (4.41). Given that, (4.41) is much easier to
compute as it involves an input apron that lies only to the east, south, and southeast. This was the
version we decided to implement.

A final optimization we made was based on the specific value of \(r\) used in the SVL. As this was
3, we were taking maxima over 7-by-7–pixel windows of the image. We could easily have written a
kernel that read all of the data into shared memory and had each thread compute its 49-point max,
but this would have been a large inner loop. An alternative would be to instead take the maximum
over 4-by-4–pixel windows (so only 16 values), then, from that temporary result, take a four-point
maximum of the form:

\[
M_{\text{fourpoint}}(i, j) = \max\{M_{\text{fourpoint}}(i, j), M_{\text{fourpoint}}(i, j + 3),
M_{\text{fourpoint}}(i + 3, j), M_{\text{fourpoint}}(i + 3, j + 3)\}
\]

(4.42)

Thus the desired max over the full 7-by-7–pixel window would be achieved by combining four maxima
from overlapping 4-by-4–pixel windows. A sum could not be broken down this way as it would end
up double-counting the overlapped values, but the idempotence of the max operation allows this.
The benefit of doing the max this way would be that the threads would have inner loops of 16
<table>
<thead>
<tr>
<th>Method</th>
<th>Execution time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-kernel 7-by-7 max</td>
<td>381</td>
</tr>
<tr>
<td>4-by-4 max</td>
<td>222</td>
</tr>
<tr>
<td>4-point max</td>
<td>153</td>
</tr>
<tr>
<td>Two-kernel 7-by-7 max</td>
<td>373</td>
</tr>
</tbody>
</table>

Table 4.3: Benchmarks of different 7-by-7–window max computations.

iterations instead of 49. The drawback would be that two kernels would be needed to compute the full product (one for the 4-by-4 max, and one for the 4-point max).

To determine which of these options was the fastest, we implemented and benchmarked both. The results are given in Table 4.3. The full, 7-by-7–pixel max took 381 µs, compared to 373 µs for the two-kernel solution. While the two-kernel option is slightly faster, the extra overhead of making the two calls should easily absorb the < 10 µs advantage.

A final twist was added to this result when we discovered that CUDA provided functions for doing elementary mathematical operations outside of the standard C operators (+, −, /, *), including a function `fmaxf` for returning the maximum of two floating-point numbers. Our existing kernel had an inner loop of:

1. `const float newValue = data[ ix + data_width * i + j ];`
2. `if ( newValue > runningMax ) {`
3. `   runningMax = newValue;`
4. `}`

(4.43)

We experimented with replacing this code with:

1. `const float newValue = data[ ix + data_width * i + j ];`
2. `runningMax = fmaxf( runningMax, newValue );`

(4.44)

Surprisingly, this led to a dramatic speed-up for the max-finding functions. Table 4.4 gives the results. The 7-by-7 max function saw an end-to-end speed-up of 30%. The boost was less dramatic for the 4-by-4–window max at only 18%, obviously because this function computes far fewer maxima. Now the one-kernel solution is the clear overall winner, running 17% faster than the two-kernel solution.
### Table 4.4: Benchmarks of Table 4.3 redone using the fmaxf function.

<table>
<thead>
<tr>
<th>Method</th>
<th>Execution time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-by-7 max</td>
<td>294</td>
</tr>
<tr>
<td>4-by-4 max</td>
<td>191</td>
</tr>
<tr>
<td>4-point max</td>
<td>153</td>
</tr>
<tr>
<td>Two-kernel method</td>
<td>344</td>
</tr>
</tbody>
</table>

#### 4.6 Conversion of Classifier Evaluation

At this point in the processing chain, we had calculated a feature vector for every sliding window under consideration. The remaining step was to evaluate all of the decision trees on each feature vector and sum up their responses to achieve the classification result.

#### 4.6.1 Individual Decision Tree Evaluation

On the CPU, decision trees are typically (and fittingly) implemented as tree data structures. A pointer is saved to the root node, which saves pointers to its children, which in turn save pointers to their children, etc. When the root node is presented with a feature vector $fv$ and polled for its result, it evaluates its binary threshold and passes the vector to one of its children so as to return its result. This passing along continues until a numerical result is found, which is passed up through all of the recursive calls (or, if this style of “tail recursion” has been optimized by the compiler, it takes only one recursive call [50]). Example code is given below.

```c
class DecisionTree {
    int index;
    float threshold, value;
    DecisionTree *leftChild, *rightChild;
    ...

    float classify(float *x) {
        if ( index == -1 ) return value;
        if ( x[index] > threshold )
            return leftChild->classify(x);
        else
            return rightChild->classify(x);
    }
}
```
Despite its simplicity, this structure could not be ported directly to CUDA as that domain prohibits recursive calls. One workaround would be to store the tree in a simple, linear data structure as it is easy to collapse the data of a decision tree into a vector. Then the tree could be evaluated with only a loop instead of with a recursive function:

```c
struct DecisionTree {
    int index, rightChild, leftChild;
    float threshold, value;
}
float EvalDecisionTree(DecisionTree *tree, float *x)
{
    int i = 0;
    for (;;) {
        if (tree[i].index == -1) return tree[i].value;
        if (x[tree[i].index] > tree[i].threshold)
            i = tree[i].leftChild;
        else
            i = tree[i].rightChild;
    }
}
```

The benefit of this structure is that it can be ported to CUDA. Yet we also know that our trees will have a fixed depth, so another option is to “unroll” this loop into a nested if–else statement of fixed complexity. For example, for a tree with a maximum depth of 2 decision nodes, we would have the function:

```c
float EvalDecisionTree(DecisionTree *tree, float *x)
{
    if (tree[ROOT].index == -1) return tree[ROOT].value;
    if (x[tree[ROOT].index] > tree[ROOT].threshold) {
        if (tree[LCHILD].index == -1) return tree[LCHILD].value;
        if (x[tree[LCHILD].index] > tree[LCHILD].threshold)
            return tree[LLCHILD].value;
        else
            return tree[LRCHILD].value;
    } else {
        if (tree[RCHILD].index == -1) return tree[RCHILD].value;
        if (x[tree[RCHILD].index] > tree[RCHILD].threshold)
            return tree[RLCHILD].value;
        else
            return tree[RRCHILD].value;
    }
}
```
else
    return tree[RR_CHILD].value;
}

The values ROOT, L_CHILD, R_CHILD, etc., are statically known indices into the decision vector. Given the proven benefits of loop unrolling (Section 4.3.2), we decided to implement this former option.

We wrote a decision tree evaluation function for trees with a maximum decision node depth of three, giving them a maximum number of 15 nodes ($2^4 - 1$, or seven splitting nodes and eight decision stumps). However, if we had implemented the tree in the struct form above, this would naively have taken 180 bytes of storage per tree (with three 4-byte numbers required to describe the feature vector index, feature threshold, and return value of each of the fifteen nodes). But we knew there was always going to be at least one decision, so the root node would never return a value (saving 4 bytes). And, as the deepest layer of nodes would only ever return values, they needed neither feature vector indices nor thresholds (saving $8 \cdot 2 \cdot 4 = 64$ bytes). Thus we needed only 112 bytes to describe a tree: seven four-byte ints, and twenty-one four-byte floats.

For storing the trees on the GPU, we opted not for any struct-based form, as we were already going to be treating the trees as linear memory. Instead, we saved a decision tree in the form of an int and a float array (the first with seven elements and the second with twenty-one). For our ensemble of decision trees, we simply allocated larger arrays and concatenated all of the individual tree descriptions inside of them. Thus the description of tree $i$ inside of the ensemble started at offset $7i$ inside of the int array and $21i$ inside of the float array. We reasoned that the coding complexity of maintaining these two arrays with different internal strides was trivial compared to the simplicity of having the trees stored thus completely transparently.

### 4.6.2 Decision Tree Ensemble Evaluation

The above section described a kernel that will read in a decision tree, a feature vector, evaluate the tree on the vector, and write out its result. In the SWOD pipeline, obviously this function must be run $t$ times for each tree, and $m$ times for each sliding window’s feature vector. A simple solution would be to execute $mt$ threads in an arbitrarily blocked and gridded space, where each thread looks up its $x$ index to find out which tree it should load, and its $y$ index to find which vector it should load, and writes out its result to an $m$-by-$t$ matrix. But it should seem wasteful to load both a tree, a feature vector, perform one trivial computation (which consists along of navigating some nested
logic), and immediately exit the thread.

An alternative scheme was to have a thread load a tree description, then loop over several feature vectors. This will amortize the cost of reading in the tree, which, though small, does involve non-trivial transactions with global memory. The dual to this scheme would be to load in a feature vector and keep it in memory while looping over tree descriptions, thus amortizing the cost of loading in the feature vector. Both of these should be faster than the naive approach of above.

We decided to implement the first scheme because it posed several advantages over the second. First, there were typically many more sliding windows (and so feature vectors) than trees, i.e., \( m \gg t \). Thus looping over \( t \)—as in the second scheme—would mean we would have to allocate a minimum of \( m \) threads (possibly thousands) to process all feature vectors. But the dual, where we loop over \( m \), would mean a minimum of \( t \) threads (only hundreds). Also, in the second scheme, each thread could loop for a maximum of \( m \) iterations, giving us much more power to optimize by exploring how many iterations a thread can make before we face diminishing returns. If we could only ever loop over \( t \) iterations, our power to experiment would be diminished.

Another benefit of this approach relates to shared memory. In the first scheme, where each thread in a block “owns” a separate tree, the threads can cooperate to load a feature vector into shared memory. This will be useful as feature vectors are potentially nearly a thousand elements long. The threads in a block can then all make coalesced reads to global memory. When they are evaluating their owned trees on the feature vector, their reads will be going to shared memory, and, being so sparse, the probability of bank conflicts would be minimal. In contrast, in the second scheme, where each thread owns a feature vector and they all loop over trees, there is comparatively little benefit to using shared memory as each tree description is only 112-bytes-long.

We have decided to lay out our thread block in the tree dimension. Threads will cooperate to read in one feature vector at a time: if \( n > t \), then each thread will read in up to \( \lceil n/t \rceil \) values from the vector. Also, each block will evaluate \( k \) feature vectors in its inner loop. This layout is illustrated in Figure 4.16.

Of course a crucial degree of freedom is what value should \( k \) take. If \( k = 1 \), then we have the naive solution, only with threads cooperating to read the feature vectors into shared memory instead of all making separate transactions to global memory. As we increase \( k \), each thread is doing more work sequentially, thus reducing the parallelism of the operation, but increasingly amortizing the costs of reading in all of the tree descriptions (and of allocating thread blocks).

To determine the optimal range of \( k \), we ran a benchmarking test with \( t = 200 \) trees (the same as
with our reference classifier), \( m = 1000 \) feature vectors, \( n \) taking on a typical value in the hundreds, and with \( k \) varying between 1 and 512. At \( k = 1 \), 1000 blocks of threads would be allocated, and, at \( k = 512 \), only two blocks would be. The end-to-end runtime results are plotted versus \( k \) in Figure 4.17.

From the figure we clearly see that the naive solution of having each block process only one feature vector is the most inefficient solution, taking over 3.5 ms to process all 1000 feature vectors. The optimal range appears with \( k \) around 70–80. From \( k \in [1, 80] \), the runtime has been exponentially
improving with increasing $k$. Thereafter the plot assumes a distinctly step-like appearance. The first step is observed around $k = 167$, with additional ones at 200, 250, 333, 500. These are obviously the quantities $1000/6, 1000/5, 1000/4, 1000/3, 1000/2$, or when the kernel transitions from being executed with only 6 blocks to 5 blocks, to 4 blocks, etc. From these results we chose $k = 100$ for our default setting.

4.6.3 Ensemble Evaluation

The above sections have discussed how to the $m$-by-$n$ matrix of feature values into an $m$-by-$t$ matrix of individual decision tree results. The remaining step was to sum this matrix along the $t$ dimension to achieve the $m$-element vector of final classifications. This we accomplished via a simple application of the same style of summation algorithm described in Section 3.4.3.

We did apply one optimization to reduce memory allocation on the GPU, however. Naively we would need separate storage for both the features and the decision tree results. But, as we were "consuming" the feature values to produce classifications (i.e., they would not be needed after we had the classification result), we could overwrite the feature vectors in place with their "classification vectors"—the intermediate results of all the decision trees before being summed up. In the case where $t > n$ (i.e., there were more trees than features), we could allocate $mt$ memory, and use a stride of $t$ when writing in the $n$-element feature vectors. If $t < n$, then we would only need to allocate $mn$ memory, use a stride of $n$, but sum up only the first $t$ results in the summation step.

4.7 Benchmark Results

We have presented a methodical discussion of how we converted the entire SVL SWOD pipeline to CUDA, but the actual developmental process was more piecemeal. Thus we obtained a series of incremental speed-up benchmarks as work proceeded. We outline these results below.

The reference SVL solution had an end-to-end runtime of 22.2 s when processing a 2-channel, 640-by-480–pixel image with an 87-patch dictionary. The first milestone in our conversion saw us place all of the feature extraction on the GPU. The CUDA code thus included a brute force NCCC calculation (without integral images), max-pooling, compiling the feature vectors, and transferring the $mn$ values back to the CPU (for CPU-based decision tree evaluation). In this configuration, the end-to-end runtime was reduced to 2.80 s, nearly an 8-times speed-up. For this first milestone,
though, we used the existing SVL pipeline, and only replaced the feature extraction with CUDA-based functions.

For the second milestone, we created a single function that would eventually handle the entire CUDA-based SWOD pipeline. Whatever was not converted yet to CUDA (including the decision trees) was implemented with outside calls, but we centralized all of the GPGPU operations. We also optimized the previous functions (including using templating in the convolution function), and the end-to-end runtime dropped to 2.10 s, for a total speed-up of 10.6-times.

In the third milestone we finally moved the decision tree evaluation onto the GPU, reducing the amount of data transfer back to the CPU from \( mn \) to the ideal \( m \). This reduced runtime to 1.60 s for nearly a 14-times speed-up. Finally we applied integral images to the NCCC calculation, and achieved a final runtime of 1.24 s, or almost an 18-times speed-up. While this is does not achieve 1 frame per second, it is very close to real-time, and achieves much more serviceable feedback for a robot versus the 22.2 seconds per frame we began with.

One could point out that this benchmark was achieved with only a relatively small 87-patch dictionary, so this CUDA object detector would run even slower on larger, more realistic patch dictionaries. While this is true, we also observed that, as the dictionaries got larger, the CPU version saturated on data throughput faster than the GPU. Thus, while the GPU got slower as the dictionaries increased in size, its speed-up over the CPU became greater.

In our final benchmarking experiments, we ran with much larger dictionaries (from sizes of 500 patches up to nearly 900 patches), and we ran on two GPUs: the 8800 GT that we have been benchmarking on until now, and the 9800 GTX on the system we mentioned in Section 3.8.1. The results are presented graphically in Figure 4.18. While the reference solution had runtimes on the order of 5–6 minutes, the fastest GPU (the 9800 GTX) ran in only 3–4 seconds. In particular, the 8800 GT showed an average speed-up of 35-times, and the 9800 GTX showed an average speed-up of over 90-times. One caveat to this was that reference benchmarks were no longer made with the OpenCV decision trees as they had before, and instead used an SVL decision tree class. But decision tree evaluation was always only a trivial component of the total CPU-side runtime as compared to the convolution.

While a runtime of 3–4 seconds falls short of our goal of one frame per second object detection, it approaches a nearly two-order-of-magnitude speed-up over the CPU implementation, and this brings state-of-the-art object detection algorithm tantalizing within the range of real-time operation. Also, even though the largest-dictionary detectors did not run as quickly as desired, the smaller
87-patch dictionary now ran in well under a second with the new GPU. This suggests that one could use a two-tiered approach to object detection, first scanning a scene with a less-accurate, small-dictionary detector, and intermittently running a slower but more accurate large-dictionary detector, including performing temporal integration of the various detections. Such coarse/fine techniques have commonly been explored to aid the speed and accuracy of object detectors [52].

4.8 Conclusion

We showed in detail how we analyzed the reference solution sliding-window object detection pipeline and derived what would be the optimal pipeline for a CUDA version. We discussed each step in this translation process, showing all of the choices and incremental optimizations that were made to achieve peak performance from each component. We reiterated many of the optimization strategies seen first with the CUDA visual servoing algorithm, and we explored many new approaches, including focused line-level optimizations (such as loop unrolling and CUDA-specific functions), and how to
effectively port existing image processing optimizations to the GPU domain (viz., integral images). The end-to-end speed-ups we achieved were nearly a factor of 20 on older machines with small patch dictionaries, and over 90 with newer GPUs and larger patch dictionaries. Again we have demonstrated how to bring a state-of-the-art computer vision technique into the domain of real-time operation, thus enabling a powerful algorithm to be included in the robot’s perceptive loop.
Chapter 5

Conclusion

In this thesis, we presented the CUDA environment for programming GPGPU applications, and we applied it to develop real-time versions of two modern computer vision techniques: visual servoing and sliding-window object detection. We first provided a thorough primer to CUDA, introducing both its abilities and its limitations. Next we developed a novel taxonomy of function classes which defines the complexity for CUDA conversion over a large corpus of algorithms. Based on a function’s input–output data interdependence, this taxonomy describes the form of the most basic CUDA kernel necessary to implement it, its rough complexity, and the major degrees of freedom in its design.

In converting the visual servoing algorithm to use GPGPU, we surpassed our goal of real-time execution. In particular, we reduced the end-to-end runtime of the algorithm from 30 s on the CPU to under a second on a modern GPU, achieving over a thirtyfold speed-up. In the process of conversion, we explored both the general strategies for writing efficient CUDA code, and the specific facets of CUDA’s behavior that lead to optimal performance. We discovered that achieving the fastest kernels requires not only respecting the CUDA best practices, but often conceptualizing many options for implementing a specific function and benchmarking all variants to find the best choice under the circumstances. We also found that, when posed with certain bottlenecks, we had to make use of creative solutions that did not speed-up any existing code but rather optimized what were irreducible burdens (such as where we used color channel multiplexing to circumvent a bottleneck in OpenGL-to-CUDA memory transfer).

Our conversion of the sliding-window object detector was much more involved. In the process we applied many of the lessons learned above with the visual servoing conversion, and also pushed
into new realms, showing that achieving an efficient CUDA conversion is not only a matter of one-to-one re-implementation. For example, to optimize computing a complicated form of convolution (specifically the normalized cross-correlation coefficient), we first had to streamline the form of the equation before we wrote any code. Next we had to design an efficient CUDA kernel that incorporated the line-level optimizations of loop unrolling and templating; and afterwards we still had to use integral images (an optimization familiar from the CPU domain) to achieve the best performance. Next, when we converted the evaluation of the boosted decision tree classifier, we proved again the necessity of first analyzing the problem thoroughly to discern the optimal conversion scheme, and then benchmarking over the free parameters of the implementation to achieve peak performance. When optimizing the runtime of a program, it does well to treat this as a true optimization problem, and so approach it by rigorously identifying the important variables and finding the minimum runtime along those axes. From this work, our CUDA sliding-window object detector achieved real-time execution rates of greater than one frame per second on detectors that used small patch dictionaries. Detectors with much larger patch dictionaries ran at speeds of a few seconds per frame, however, and so were not strictly real-time, but they still saw speed-ups of nearly two orders of magnitude over their CPU implementations.

Our work has demonstrated the efficacy of GPGPU—and of CUDA, in particular—in two ways. The first is in terms of the raw speed-ups that were achieved, which were near two orders of magnitude in both domains explored. This made it possible for these state-of-the-art algorithms to be included in the perceptive loop of a robotics platform, and so brought these research algorithms into the realm of real-world application. This is important as computer vision is ostensibly the quest to endow computers with human-level vision, but “human-level vision” tacitly implies that computers must not only match the versatility and power of human vision but also its speed; otherwise it would simply be a curiosity. Indeed, even if an artificial intelligence were built tomorrow, if it thought at speeds a hundred times slower than humans, it would be of little practical significance. Thus speeding up the execution of complex algorithms is an important dual goal to that of the research of those algorithms itself. While pure research continues exploring ever more intricate models to solve “AI-hard” problems, parallel to this is the work—such as in this thesis—that seeks to pull these increasingly complex constructions back into the realm of feasible execution times. This itself facilitates further research progress since it speeds the developmental cycle of receiving feedback from the implementation of novel ideas.

The second way in which we have shown the efficacy of GPGPU is by exemplifying the speed-ups
that were achieved simply by moving to newer GPUs. Switching to the next hardware generation offered a clean twofold or threefold speed-up over the old GPU, with the only cost being that of recompiling on a new system, without changing any of the existing code. While such small speed-ups themselves might be facilely dismissed, the fact of them provides a powerful motivation for moving to GPGPU, as the GPU’s performance edge over the CPU is already dramatic and will only increase for the foreseeable future. The 1990’s was an era of leap-frogging processor clock speeds, that applications could be reliably expected to double in execution speed every year as the new hardware was released. In the 2000’s, however, clock speeds saturated, and manufacturers have even begun reducing them as they focus on increasing power efficiency. But GPGPU offers a portal back into such relentlessly increasing performance trends as the GPU’s advantage comes not from its clock speed but rather from its massive parallelism and its focus on raw computational throughput. While clock speeds look to remain static in the near term, the continual shrinking in feature sizes means that the GPU’s parallelism—and thus its ability to out-perform the CPU—will only increase.

Apart from its speed advantages, GPGPU is also proving desirable in the dimension of cost. GPU-based supercomputers are now being realized that have performance abilities on par with traditional supercomputers, but, since they require only a handful of GPUs versus hundreds of old CPU cores, their costs can be on the order of a thousand times less [53]. Combined, GPGPU’s benefits are staggering: applications that are already much faster than their CPU counterparts; associated hardware costs that are much lower to achieve equivalent performance; and the fact that both of these trends are presently skewing farther in GPGPU’s favor.
Bibliography


