An Investigation on Utilising a Modern Hybrid Multicore System for Object Tracking

Author: Max Marlon Baird
Supervisor: Dr. Sven-Bodo Scholz

A thesis submitted in fulfilment of the requirements for the degree of MSc.

in the

School of Mathematical and Computer Sciences

August 2015
Declaration of Authorship

I, Max Marlon Baird, declare that this thesis titled, ‘An Investigation on Utilising a Modern Hybrid Multicore System for Object Tracking’ and the work presented in it is my own. I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: 

Date:
“Why did the multithreaded chicken cross the road? - to To other side. get the”

Unknown
Abstract

Object Tracking is compute intensive by nature where the filtering aspect can benefit from data parallel techniques. However, authoring software for parallel computing requires much tailoring in the interest of efficiency which may have a negative effect on portability. Our work investigates the degree of effort required to port existing object tracking algorithms. We then seek to compare performance between hand-coded implementations against auto generated code. Our results show that software migration can be a tedious and time consuming process and that simulating and adapting existing functionality necessitates tedious translation.
Acknowledgements

I would like to sincerely thank my colleagues and especially my supervisor, without whom I would not have gotten thus far.
## Contents

Declaration of Authorship .................................................. i

Abstract .................................................................................. iii

Acknowledgements ...................................................................... iv

Contents ................................................................................... v

List of Figures ............................................................................. vii

1 Introduction ............................................................................ 1

2 Objectives ................................................................................ 3

3 Literature Review ..................................................................... 4

3.1 Background .......................................................................... 4

3.1.1 Concurrency and Parallelism ........................................ 4

3.1.2 The plateau of Sequential Computing ............................ 5

3.1.3 Parallel Computing ........................................................ 6

3.1.4 Object Tracking ............................................................. 8

3.1.5 Particle Filters ............................................................... 10

3.1.5.1 Bayesian Estimation .................................................. 11

3.1.5.2 Basic Particle Filter Algorithm ............................... 12

3.2 Parallel Technologies applied to Object Tracking ............... 13

3.2.1 OpenCL ....................................................................... 14

3.2.2 CUDA .......................................................................... 15

3.3 GPU-Based Object Tracking .............................................. 15

3.4 Single Assignment C ......................................................... 17

3.5 Software Portability ........................................................... 18

4 Requirements Analysis ....................................................... 19

5 Professional, ethical, legal and social issues .......................... 20

6 Overview of Prior Work ......................................................... 21
List of Figures

3.1 Taxonomy of tracking methods [Yilmaz et al., 2006].  

3.2 Flow diagram for the particle filter algorithm [Cabido et al., 2012a]. After the prediction stage at time $t$ the particle filter evaluation starts at a new time step $t+1$.  

6.1 Sequential algorithm of detection tracking system [Limprasert et al., 2013].  

6.2 Parallel algorithm of detection tracking system [Limprasert et al., 2013].  

8.1 SaC and C++ likelihood runtimes  

vii
For Stephanie...
Chapter 1

Introduction

Video based visual tracking is a complex task which consists of estimating the position of objects [Cabido et al., 2012a]. Aside from being useful in areas such as robotics and augmented reality, it also has practical applications in video surveillance, human-computer interaction and recreational activities (e.g. sports) all of which require robust and efficient tracking algorithms. Often object tracking is one aspect of a larger program which necessitates a greater demand for real time processing with a small amount of false positives. Achieving this facilitates more compute time for post-processing steps.

Object tracking essentially consists of two steps: target representation and location and filtering and data association where our work has a focus on the latter. Filtering helps with the dynamic tracking of an object after it has been identified. Two commonly employed filters are the Kalman filter and the particle filter, our work targets the particle filter. Particle filtering is a probabilistic algorithm widely applied to tracking, which takes advantage of knowledge about previous states of the system thus reducing computational cost of an exhaustive search over an entire image [Montemayor et al., 2004].

Even though the particle filter tries to be efficient in reducing computational cost, it is still (by current standards) considered to be computationally intensive. Despite being superior to the Kalman filter, the particle filter is only now being adopted for practical use due to the increased availability of compute power which is necessary for this approach. More specifically, the availability of compute power comes from the evolution of parallel hardware which is highly beneficial for data-parallel programs. The particle filter algorithm can be implemented as a data parallel program since two of its steps involve independent data processing.
A common target platform for particle filter implementation is the graphics processing unit (GPU) which has hardware optimized for parallel processing. Particle filtering algorithms can therefore be naturally expressed on such platforms. Nevertheless, it is still quite hard to efficiently implement the algorithm particularly if the intention is to target diverse platforms e.g multi-core processors and general purpose GPUs.

Our work bears a dependency on existing object tracking software, therefore our first interest is that of portability. In our case this means the degree of effort required to transport and adapt it to a new environment. The existing code has a reliance on third party tools and libraries thus collectively forming external (not part of core code) elements which need to be transported to the new environment. The adaptation we perform involves modifications we needed to make to the original software.

Our work then looks at how well high-level languages fare when they are being used to implement data-parallel algorithms which need to execute on different platforms. In particular, one high level language that will be looked at is Single Assignment C (SAC), developed in house to determine how well the language is suited to:

1. Express these algorithms and
2. What kind of performance is achievable for different platforms

The approach taken is to implement the compute-intense aspect of the particle filtering algorithm in SAC and then compare the performance of the SAC generated code against the hand-written code (as many as available) on different platforms (multi-core CPUs and GPUs). Discrepancy in measurements will be analyzed in order to determine where they stem from and potentially re-write the SAC implementation to improve performance.

Most hand written implementations are highly tuned to take advantage of specific hardware, so our work also entails a measure of scalability and portability of the generated SAC code. Our work makes use of a previously implemented object tracking system that implements a particle filter. We aim to discover what is required to port existing object tracking code to a system on which it was not intended.
Chapter 2

Objectives

Writing an object tracking algorithm is difficult particularly if it needs to have fast response times close to real time. The only opportunity to achieve fast response times is to have parallel processing programs. Writing parallel programs to achieve these time limits can prove quite difficult. We are interested in determining whether we can use a high level programming language and have compiler level tool chains synthesize parallel code which can be run on GPUs and multi-core systems. As an experimental test we will be looking at the programming language Single Assignment C (SAC). We first need to determine the difficulty of adapting previously written code (approximately four years old) to a more recent environment.

The objectives of this work are as follows:

1. Determine the portability of an object tracking system to an alternative platform.

2. Determine how the performance of SAC generated code compares against existing hand-coded tracking algorithms.
Chapter 3

Literature Review

This review of the literature commences by over viewing the requisite background knowledge for the research then proceeds with discussing parallel computing technologies and their application to object tracking algorithms. The review then discusses the philosophy of SAC and its potential application to parallelize object tracking. It closes by examining software portability and the possible challenges one may face when porting software to an unintended platform.

3.1 Background

This section serves to provide a background on sequential and parallel computing as well as object tracking. Firstly, a brief history on sequential architectures and its ultimate limitation(s) is provided as a means of justifying the migration from sequential to parallel computing. Secondly, we then proceed to briefly outline the rise of parallel computing as well as its current limitations. Finally, a background to object tracking is provided.

3.1.1 Concurrency and Parallelism

In the era of programming and hardware architecture, processes were sequential by default which meant their instructions were executed consecutively. Parallelism was not possible but concurrency was; concurrent execution achieves a sort of pseudo-parallelism where multiple processes can commence and complete through being executed in overlapping time periods. Concurrency, in this sense, is enforced because of the hardware architecture: a single core processor is only capable of executing a single instruction at any instant.
At this point it is important to make a distinction between the meaning of parallel and concurrent within the context of programming. The words parallel and concurrent are generally considered synonyms; but not so in programming where they are used to describe fundamentally different concepts [Marlow, 2013]. A parallel program is one which performs its computations on different processors simultaneously (this definition implies a multi-core or multi-processor architecture). Conversely, a concurrent program is a programming structuring technique in which there are multiple threads of control [Marlow, 2013]. Each thread seemingly executes “at the same time”, where their executions are sequential, but interleaved thus realizing an illusion of parallelism.

Relevant to the prior distinction made between concurrency and parallelism, there are two related programming paradigms: deterministic and nondeterministic. A deterministic programming paradigm is one in which each program can give only one result, whereas a nondeterministic paradigm can have varying results depending on an aspect of execution [Marlow, 2013]. Concurrent programming is inherently non-deterministic since execution is interleaved and events can be triggered at any time (e.g. from a user interface, network, database etc.). Whereas parallelism is primarily deterministic because the goal of parallelism is to simply arrive at the answer as fast as possible i.e. the output is always the same for a given input.

Deterministic parallelism is ideal because it provides performance enhancement while allowing for sequential reasoning, testing and debugging. Concurrent parallelism is possible but often comes with overhead which can be attributed to the loss of determinism.

Nevertheless, in the wider scope of programming and software engineering, concurrency and parallelism are not mutually exclusive. Often concurrency is needed to persist a responsive interface while compute-intensive tasks are performed through parallelism in the background.

### 3.1.2 The plateau of Sequential Computing

The speed of a process executing on single core architecture is limited to how many clock cycles the underlying processor is able to achieve. Prior to the rise of multi-core architectures, processes were able to automatically gain a speed up through the advancement of electronic manufacturing technology delivering faster single core processors. There was therefore an association of performance with high processor clock frequencies [Pase and Eckl, 2005].
In 1965, Moore’s law quantified the astounding growth of all the new technology of semiconductors by stating that the density of components per integrated circuit would continue to double at regular intervals [Schaller, 1997]. Accordingly, computing speed and accessibility has been able to grow phenomenally for more than four decades [Pase and Eckl, 2005]. However, of recent Moore’s law has begun to show signs of diminishing returns. This characteristic may be attributed to a quantum limit imposed by physics which places an absolute restriction to resolution that science and engineering can achieve [Powell, 2008].

Essentially, an increase in clock speeds of a processor also increases the waste heat produced by the processor. Inexpensively cooling faster processors poses a challenge which is offset in some ways by smaller transistors requiring less electricity. However, transistors made too small begin to adopt quantum mechanics characteristics which makes their functionality unreliable. These and other factors are what contributed to the end of the the “clock speed race” and gave rise to multi-core processors which then naturally led to parallel computing.

3.1.3 Parallel Computing

The multi-core era commenced with innovations in hardware architecture such as hyperthreading and dual-core processors. These innovations provided relatively inexpensive parallel computing resources for desktop computers [Trobec et al., 2009]. Multi-core processors therefore superseded their sequential counterpart with the understanding that while higher clock rates may not be achievable, being able to execute instructions in parallel was the next acceptable alternative. However, taking advantage of these new resources requires parallel programming techniques. A next generation processor no longer guarantees a speedup since clock speeds are unlikely to vary by much from the previous generation.

Since automatic speed ups are no longer a positive side effect of newer generation processors, emphasis needs to be placed on the concepts of parallel programming in order to make efficient use of underlying parallel resources. Regardless of programming language, paradigm, technology etc., the flow of execution of any process needs ultimately goes through a processor. If multi-core processors are dominant, the need for parallel programming will therefore extend to all areas of software development.

Early literature in parallel computing recognized that parallelism can bring about significant improvements in computing speed [Nikhil et al., 1987] which would be orders of magnitude faster than the then current supercomputers. Though this recognition it
was realized that most of the computing models at the time were based on von Neumann, sequential architectures, which did not seem extensible in any straightforward way to parallel machines [Nikhil et al., 1987].

This led to explorations in automatic parallelizations on sequential code in an attempt to relieve programmers from performing manual parallelism. Parallelizing compilers is one technique which exploits loops as good sources of parallelism, where there are no loop-carried dependencies [Eigenmann and Hoelfinger, 2001], of course this entails data-dependence analysis and program analysis for correctness verification. Progress has been made in the field of dynamic parallelization techniques such as speculative multithreading which attempts to optimistically transform a sequential program into a parallel one at runtime [Yiapanis et al., 2013]. In any case, automatic approaches that attempt to parallelize serial code simply cannot deal with the fundamental shifts in algorithm structure required for effective parallelization [McCool et al., 2012].

The initial designers of computer architecture used serial machine languages to simply the programming interface, this propagated to higher level languages, tools and ultimately to programmers who developed a serial mental model of the computer. Currently, programming practice, theory, data structures and algorithms have a strong bias toward serial execution [McCool et al., 2012]. This makes for a poor match to current parallel computer architecture. At this point it can be argued that the tables have turned with respect to sequential and parallel computing; previously, sequential programs automatically benefited from performance improvement with each generation of single core processors. As it currently stands, this will no longer be the case; instead, efficiently parallelized applications will make good use of current multi-core processors and should be able to automatically scale to even better performance on future processors.

Above all, parallel computing is not the ultimate answer to performance enhancement for programs. The parallel execution time for programs cannot be arbitrarily reduced by employing parallel resources [Rauber and Rünger, 2013]. When considering overall improvement, Amdahal’s law should be applied. Essentially, the overall speedup of a program which employs parallel techniques is limited by any sequential operations of execution which cannot be parallelized. Examples of such operations include disk I/O, network transfers or simply shipping data across the bus. Other restrictions may arise because of the algorithm being implemented and its inherent data dependencies. Taking these into account means there will be a ceiling to how much improvement can be attained no matter how many processors are employed. While speedup is attainable through parallelizing computation, the bottleneck, when considering the overall execution, lies with sections of the program which currently cannot be parallelized.
3.1.4 Object Tracking

Object tracking is an important problem in computer vision [Rymut and Kwolek, 2010], and is a prerequisite for analyzing and understanding visual data. Object tracking, as it relates to video technology, involves the automatic location of a moving object (or objects) as it transitions between frames. Generally, an object tracker works by detecting the object of interest then establishing correspondence between instances of the object across frames [Yilmaz et al., 2006]. Performing these steps discretely requires the object tracker to obtain possible object regions in every frame, via an object detection algorithm, and then correlate these objects across frames. If the steps are performed in union, the object region and correlation is simultaneously estimated through iterative updates of the object’s location and region information from previous frames.

The first and second steps are technically referred to as target representation and localization and filtering and data association [Comaniciu et al., 2003] respectively. The former is a bottom-up process and copes with changes of appearance of the target while the latter is top-down and deals with the dynamics of the tracked object and learning of scene priors [Comaniciu et al., 2003]. The manner in which these two components of the algorithm is combined influences the robustness and efficiency of the tracker. The applied combination method is therefore dependent on the characteristic(s) of the intended problem domain. For example, face tracking in a normally crowded scene would require greater emphasis to be placed on target representation rather than target dynamics.

With respect to target representation, a tracking object is defined as anything of interest for further analysis where its initially observed shape and appearance are candidates for future identification. The shape attributes commonly considered for object representation include points, primitive geometric shapes and object silhouette and contour. While the attributes to represent the appearance at features are probability densities of object appearance, templates and active appearance models. The shape and appearance attributes presented here are not exhaustive; [Yilmaz et al., 2006] provides a more complete list with accompanying descriptions. The localization step tied to target representation is necessary so that objects of interest can be detected and their location determined.

Methods of object tracking typically fall within a specific hierarchical category, Figure 3.1 depicts a taxonomy of tracking methods. A description of each method is external to the scope of our research; the figure is included to serve as a global view of object
tracking categories and to introduce the category under which our research is oriented: probabilistic (statistical) methods.

With point tracking, objects are represented by points where point detection is based on the object’s state (position and motion) from the prior frame. Because of this approach, point tracking algorithms need to discreetly perform the general tracking steps. This category of object tracking encompasses two broad classes, deterministic and statistical methods; each method has its trade-offs when it comes to the problems specific to point tracking. In the interest of brevity and maintaining focus, further mention of deterministic methods is intentionally omitted; following, we continue with statistical methods.

In the process of object tracking it is quite possible for the object of interest to undergo random deviations from it’s “normal” shape [Yilmaz et al., 2006], for instance, maneuvering vehicles. Statistical methods have merits in solving these specific tracking problems by taking into account such uncertainties. The object position in an image is derived from measures obtained from a detection mechanism, the object state can then be determined by Single Object State Estimation or Multiobject Data Association and State Estimation which handles single and multiple object states respectively. Of particular interest to our work is the tracking of multiple objects using filters.

Determining the state for multiple objects requires the use of either Kallman or particle filters. Kallman filters typically work well for an object whose state distribution is assumed to be Gaussian but if this assumption is untrue this filter will give poor estimations [Yilmaz et al., 2006]. This limitation can be overcome by using particle filtering [Kitagawa, 1987]. However, the application of either filter requires a prior solution
to the correlation problem. A nearest neighbor approach can be used but introduces complications if objects are close to each other. Nevertheless, there are statistical data association techniques to handle these problems, details of which can be found in [Bar-Shalom, 1987].

While video object tracking finds many practical applications ranging from robotics, surveillance, augmented reality and human computer vision; the current state-of-the-art is still far from achieving results comparable to human performance [Santner et al., 2010]. Difficulties in tracking objects can arise due to abrupt object motion, changing appearance patterns of both the object and the scene and object-to-object and object-to-scene occlusions [Yilmaz et al., 2006]. The core problem encountered is the occurrence of what can be classified as “noise” while tracking objects. Specific examples of noise in this context refers to background clutter, scale, rotation, perspective projection, changes in illumination and occlusions [Rymut and Kwolek, 2010]. The challenge is to track the object irrespective of noise; attempts of reliably overcoming this challenge introduces complexity thus resulting in a time-consuming process [Rymut and Kwolek, 2010]. It is important that object detection occurs quickly and accurately where real-time performance is expected with minimal false positives [Smedt et al., 2014]. Efficient (in terms of speed) real-time detection proves beneficial since more time is available for post-processing steps.

### 3.1.5 Particle Filters

As indicated in Section 3.1.4, particle filtering is applicable to many areas inclusive of computer vision. They are superior to Kalman filters but come with higher computational requirements which limits the practicality of their use for real-time applications [Chitchian and van Amesfoort, 2012]. However, the advent of parallel architectures and data-parallel programming models has served to offset this computational limitation.

Particle filtering is an implementation of recursive Bayesian filtering using (sequential) Monte Carlo simulations where its goal is to estimate a stochastic process (a collection of random variables, representing the evolution of random values over time) given some noisy observations. Particle filters are non-parametric in nature (i.e. they make no assumptions about the probability distributions of the variables being assessed), and so are not bound to a particular distribution form and are compatible with non-linear state transition functions [Chitchian and van Amesfoort, 2012]. This attribute of particle filters means that they can be extended to tackle a range of problems, which is why they can raise computation cost.
Before examining the basic particle filter algorithm, a brief digress into Bayesian Estimation (which particle filtering mechanizes) is necessary. The following is an excerpt from [Chitchian and van Amesfoort, 2012].

### 3.1.5.1 Bayesian Estimation

Bayesian estimation computes a *Probability Density Function* (a function that describes the relative likelihood for a random variable to take on a given value) for the state of a dynamic system over the range of possible values. Suppose $x$ is a quantity which we wish to infer from the measurement $z$, $p(x)$ represents all knowledge regarding this quality prior to the actual measurement. This distribution is called the *prior probability distribution*. The conditional probability $p(x \mid z)$ is called the *posterior probability distribution* and represents our knowledge of $x$ having incorporated the measurement data. However, in this case, the inverse probability $p(z \mid x)$ is needed as it directly relates to the measurement characteristics where the distribution is unknown in advance. *Bayes rule* allows for the calculation of the conditional probability based on its inverse.

In order to discuss how the *Bayes filter* calculates the state estimate, the dynamics of the system needs to be modeled. Let $x_k$ denote the state time $k$, and $z_k$ denote the set of all measurements acquired at time $k$. Assuming the system exhibits Markov properties [Markov, 1960], the state $x_k$ depends only on the previous state $x_{k-1}$. The evolution of the state is governed by the probability distribution: $p(x_k \mid x_{k-1})$, which is referred to as the *state transition probability*. The measurements of the state follow the probability distribution $p(z_k \mid x_k)$ which is called the *measurement probability*.

The Bayes filter calculates the state estimate, from an initial state $p(x_0)$, recursively in two steps:

1. **Predict**: The state estimate from the previous step is used to predict the current state. This estimate is known as the *a priori* estimate, as it does not incorporate any measurements from the current time step.

   
   $$
   p(x_k) = \int p(x_k \mid x_{k-1}) p(x_{k-1} \mid z_{k-1}) \, dx_{k-1}
   $$

2. **Update**: The state estimate derived from the previous step is updated according to the actual measurements done on the system. This estimate is referred to as the *a posteriori* estimate.

   $$
   p(x_k \mid z_k) = np(z_k \mid x_k) \, p(x_k)
   $$
3.1.5.2 Basic Particle Filter Algorithm

As previously mentioned, particle filtering is based on the Bayesian filter introduced in Section 3.1.5.1, which uses sequential Monte Carlo Simulations. Detailed information on Monte Carlo methods can be found in the book by [Kroese et al., 2013]. The posterior is represented by a finite set of random samples drawn from the posterior with associated weights. The posterior is represented by a set of particles where each particle \( x_k[m] \) is an instantiation of the state at time \( t \); \( m \) represents the index of the particle in question.

The prediction step generates each particle \( x_k[m] \) from the previous state \( x_{k-1}[m] \) by sampling the state transition probability \( p(x_k | x_{k-1}) \).

In the update step, when measurement \( z_k \) is available, each particle is assigned a weight \( w_i[m] \) according to:

\[
    w_i[m] = p(z_k | x_i[k])
\]

With a large enough particle population, the weighted set of particles \( \{x_i[k], w_i[k], i = 0, \ldots, N\} \) becomes a discrete weighted approximation of the true posterior \( p(x_k | z_k) \).

Each step performs discrete operations on each particle, this property of the algorithm well suits a data parallel programming model. Which leads to the next section discussing the parallel technologies applied to object tracking. Other filtering techniques are available, most notably the Kalman filter [Kalman, 1960], but a detailed discussion of these would prove out of context. Figure 3.2 is a graphical representation of the algorithm.
3.2 Parallel Technologies applied to Object Tracking

Visual tracking is a well-known compute intensive procedure in computer vision [Cabido et al., 2012b], the nature of these procedures can exceed the capabilities of the CPU. This does not mean that CPUs are simply not able to execute these computations, only that they’re unable to do so fast enough to be practical in terms of real-time visual tracking. Hardware has been evolving toward multi-core architectures which are inherently parallel and Graphics Processing Units (GPUs) is one such architecture which can be exploited for general purpose computing.
What makes this evolution attractive to object tracking techniques has to do with a typical characteristic of tracking algorithms; *independent data processing*. Tracking algorithms with this characteristic can benefit from a data parallel programming model which maps and scales quite well to parallel architectures such as multi-core CPUs and massively parallel GPUs. Underlying parallel hardware architecture facilitates a natural mapping to data parallel programs and has initiated object tracking research [Cabido et al., 2012b, Chitchian and van Amesfoort, 2012, Jáuregui and Horain, 2010] into general purpose computations on GPUs (GP-GPU).

The GPU is specialized for compute-intensive, highly parallel computation and is well suited to address problems that can be expressed as data-parallel computations [NVIDIA, 2015]. The following sub-sections serve to describe specific parallel technologies that are currently utilized when parallelizing tracking algorithms using a GP-GPU.

### 3.2.1 OpenCL

Today’s computers often include highly parallel compute units such as GPUs, CPUs among other processors and it is important for software developers to find ways to fully utilize these heterogeneous processing platforms [Opencl and Aaftab Munshi, 2014]. Generally, creating applications for parallel architectures is challenging, however, doing same for heterogeneous systems poses an even greater challenge since multicore compute units can architecturally be very different.

While general purpose GPU programming can leverage compute power in addition to that of the CPU, it is usually vendor and/or hardware specific. This makes developing to simultaneously use multiple compute sources in a generic way, from a single code base, laborious.

Recognition of these issues has driven the development of OpenCL in order to ease the programming burden when developing for heterogeneous systems. OpenCL is a framework which supports execution on multi-core central processing units, digital signal processors, field programmable gate arrays, graphics processing units, and heterogeneous accelerated processing units [Gaster et al., 2012]. Implementing a solution designed with the OpenCL specification is scalable and allows a seamless “stitching together” of diverse heterogeneous compute devices, such as the aforementioned, from one or many manufacturers.

OpenCL is more than a language, it is a framework consisting of an API for coordinating parallel computation across heterogeneous processors and a runtime system to
support software development. The standard supports both data and task based parallel models and interoperates with OpenGL and other graphics APIs.

OpenCL targets programmers interested in authoring efficient, parallel code that is portable since it is impractical for the programmer to be constantly aware of the current state of all compute devices.

3.2.2 CUDA

CUDA is defined as “A General-Purpose Parallel Computing Platform and Programming Model” [NVIDIA, 2015], this technology is specific to NVIDIA GPUs and was introduced by said company in 2006. The aim of CUDA is to be a preferable option to CPUs for computationally complex problems.

The CUDA parallel programming model is designed to allow for transparently scaling parallelism which leverages increasing numbers of cores. CUDA’s abstractions of thread groups, shared memories and barrier synchronization provide fine-grained data and thread parallelism within course-grained data and task parallelism. With these abstractions, problems which target CUDA can be divided into course blocks (sub-problems), where each block is solved by co-operative parallel threads. Each block of threads can be scheduled on any available GPU processor and executed in any order, concurrently or sequentially therefore requiring the runtime system to be aware of the physical multiprocessor count [NVIDIA, 2015].

NVIDIA'S GPUs consist of arrays of Streaming Multiprocessors (SMs) facilitating automatic scalability because a CUDA multi-threaded program is partitioned into blocks of independently executing threads where each block is mapped to multiprocessor. A GPU with more multiprocessors automatically executes the program in less time than a GPU with fewer [NVIDIA, 2015].

3.3 GPU-Based Object Tracking

This section discusses the limitations of current parallel approaches to object tracking. A discussion which is a necessary prerequisite in order to establish a foundation which serves as the premise of our work.

Recent advancement in hardware has made the GPU a viable option for compute intensive tasks required for reliable, real-time object tracking. While there has been much success [Cabido et al., 2012a, Choi and Christensen, 2013, Limprasert et al.,
2013, Rymut and Kwolek, 2010, Sinha et al., 2006] this usually comes with highly
tuning applications for specific GPU architectures in order gain improvement. GPU
architectures are are usually vendor and model specific, therefore, manually tuning an
application for a particular architecture does not necessarily mean the application will
be able to take full advantage of future or alternate architectures without additional
manual tuning. With GPU computing there is the overarching slow down that comes
from the communication delay between the host and GPU, which has been acknowl-
edged and taken into consideration by some studies [Rymut and Kwolek, 2010, Sinha
et al., 2006].

To clarify, implementations may be easily ported to more recent hardware/architec-
tures and may even achieve a greater speed up, the issue is whether the implementa-
tions, by default, will be making the most efficient use of the newer hardware.

In recognition of effective filtering being key to reliable object tracking [Rymut and
Kwolek, 2010] has used CUDA to take advantage of the architectural properties of
NVIDIA’s GeForce 8800 graphics processor used on the NVIDIA GeForce 9800 GT
graphics card. While their work resulted in over a 40-fold speedup, their implemen-
tation was manually tuned to a GPU with 112 CUDA cores, where they specified 32
thread blocks of 128 threads each. The problem with their implementation is its lack
of scalability with the next graphics card within the same series (Geforce 9800 GTX),
which has 128 CUDA cores. For more recent cards the difference in architecture is ex-
pected to increase.

Work done by [Sinha et al., 2006] has achieved a speedup that was a 20 times im-
provement over the CPU. Again, this implementation is quite specific as it runs ex-
clusively with NVIDIA hardware and was only tested on two specific graphics cards.
Efforts were also made to partition their algorithm to minimize data read-back from
the GPU. Seamless transference of their implementation to more powerful hardware
would not prove trivial because the implementation details are hardware dependent
and vendor specific.

[Smedt et al., 2014] has approached the problem of object detection using OpenCL
which is able to target multiple compute devices. Since object detection is a data par-
allel operation they decided to use the GPU as the target device since its hardware is
optimized for data parallelism. By using the GPU as the computing device requires
decisions to be made regarding workgroup and work-item sizes. For efficiency, these
decisions need to be made based on the intended hardware which ultimately means
specific, manual tuning.
Other work [Cabido et al., 2009, Chitchian and van Amesfoort, 2012] specific to parallelizing particle filtering and thus improving object tracking has been done but they entail some form of tuning or the other which has a negative impact on portability and scalability. Our work is an experimentation with Single Assignment C; hardware independent approach to data-parallel computation.

3.4 Single Assignment C

Single Assignment C (SAC) is defined as a functional array language for efficient multi-threaded execution [Grelck and Scholz, 2006]. SAC aims to combine high level array programming with fully automatic resource management so that programmers can work with a simple abstraction of underlying complex data structures and without concern of low-level hardware details. This enables high productivity (since less time needs to be spent on optimizing for concurrency) and maintenance.

A positive side effect of SAC’s aims is that it tends to lessen the divide between following good software engineering principles and developing efficient parallel code. What serves as key in achieving SAC’s aim and this side effect is that the language incorporates optimization and parallelization into the language design. This allows for a more “traditional” (sequential-like) programming experience since parallel features are not “tacked-on” as an afterthought.

SAC is adopted from the C programming language which is fairly well known and standard thus facilitating a natural transition for newcomers. The high-level, implicitly parallel language features of SAC achieves encouraging runtime behavior because the compiler does code restructuring into a seemingly obfuscated representation (to humans) that is quite machine friendly [Grelck and Scholz, 2006]. Its single array data structure means that its design favors array-intensive tasks which can be expressed and executed in a data-parallel manner.

Our work builds on research by [Limprasert et al., 2013] which employs a particle filter developed for a parallel implementation on a GPU. Our aim is to see how the particle filter can be formulated in SAC and to then compare with the manual GPU-based filter and analyze potential differences in performance.
3.5 Software Portability

Software portability is becoming increasingly recognized as a desirable attribute for software products and porting is a technique which extends the value and life of a software unit [Mooney, 1993]. [Brown, 2003] defines portable software as that which can, with reasonable effort, be run on computers other than the one for which it was originally written. Because our work builds on previously completed software and we ultimately seek to examine performance of this software on an unintended platform; determining portability is a necessary first step of our study. Portability, for our work, is contextualized to the effort required for transportation and adaptation of existing software.

Software can be represented in multiple forms from a high to low level where these levels are between its creation and execution [Mooney, 2004]. Each representation is a candidate for adaptation giving the following levels of portability:

- Source Portability (software adapted at source level then recompiled for new target)
- Binary Portability (porting in executable form)
- Intermediate Portability (a representation which falls between source and binary)

The level of portability we determine to be applicable to our case is source portability. We port the existing software at this level because no intermediate representation exists and a binary port would prove difficult because of runtime library dependencies.

In the realm of parallel computing, there are various kinds of memory architectures, shared memorymultiprocessors and clusters of work stations with new technologies extending this range [McColl, 1996]. Authoring parallel programs to achieve acceptable performance often requires platform specific tailoring. Designing parallel software with a consideration of portability in the face of such architectural diversity can prove to be a challenge.

The hardware target for the existing parallel software we port is a GPU. GP-GPU (General Purpose GPU) programming typically entails architecture specific tailoring in order to achieve performance gains. Our work first investigates the portability of the existing sequential and parallel object tracking implementations.
Chapter 4

Requirements Analysis

The primary requirement of our work is to explore the challenges of migrating an existing object tracking system to a platform. Successfully achieving this would then progress into implementing a step of the particle filtering phase of object tracking using Single Assignment C.

Secondary requirements include investigating any discrepancies in performance between our implementation and existing implementations. This will be done to determine where and why they occur, and, if our implementation can be further improved, we will do so as long as time permits.
Chapter 5

Professional, ethical, legal and social issues

Given the core of component of our research to be an analysis of a parallelization technique to real time object tracking, the experimental aspect of the research can entail the use of publicly placed cameras. These cameras are intended track the movement of objects (in this case, people) in order to determine the technique’s efficiency.

That being said, we are aware of the potential to inadvertently record the movement of individuals who may not be aware of an active object tracking session. To this end, any recorded video will be treated in accordance with the strictest data protection laws. After experimentation, recorded information and all copies will be destroyed.

Finally, if becomes necessary to include samples of video and/or images as part of the submission of our research report, any identifying sections (faces, vehicle number plates, etc.) will be appropriately obscured unless authorized.
Chapter 6

Overview of Prior Work

Our research is primarily based on previous work done by former PhD student Wasit Limpresart at Heriot-Watt University. Because of this, a brief description of his work is a necessary prerequisite before delving into the methodology and other aspects of this report. The sections which follow detail the aspect of his research which is apt to our work.

6.1 Visual Tracking

Limprasert’s work, titled “Real-Time People Tracking in a Camera Network”, draws parallels between the importance of visual perception for humans and visual tracking as a fundamental element in certain scientific domains. His research utilizes object tracking, where objects of interest, in this case people, are tracked in real-time. The tracking framework employs a particle filter with both a sequential and parallel implementation, the latter being intended for GPU execution. His work concludes with a performance comparison between the implementations and results in the GPU implementation having a more significant speed-up ratio.

6.1.1 Sequential Implementation

The sequential implementation targets a standard CPU and consists of a detection module thus making the system fully automatic, i.e. initiates tracking without human intervention. The detection module is used in combination with a tracking module based on an ellipsoid model that allows many cameras to work in unison [Limprasert et al., 2013]. The detection module processes images from each camera to produce a foreground image and find a new subject, while the tracking module simultaneously
estimates the new subject’s states and activates a tracker for the detected subject. Each module was developed individually and later combined into a single system.

The sequential algorithm is designed in accordance with the traditional convention of single instruction execution. The probability function used by the particle filter needs to be applied to many particles, this results in the processor sequentially performing the identical calculation for each particle repeatedly. The algorithm was implemented in C++ and used OpenCV to read and display output. A visual representation of the algorithm is depicted below.

![Sequential algorithm of detection tracking system](image)

**Figure 6.1:** Sequential algorithm of detection tracking system [Limprasert et al., 2013].

Of interest to our research is the particle filtering step of the algorithm, with a specific focus on likelihood function within the particle filter.

### 6.1.2 Parallel Implementation

With respect to the parallel implementation, the particle filter has hundreds of particles that express the probability density. Determining the likelihood of each particle
prior work requires a complex likelihood calculation similar to ray tracing in computer graphics. Since likelihood calculations per particle are independent of each other, the operation lends itself well to data parallel computation. Limprasert’s research thus made use of the aforementioned technique in combination with reduction, skip ahead and cache memory.

More specifically, the GPU hardware used was the Nvidia GeForce GTS250, this facilitated the use of CUDA for parallel processing. His implementation parallelized each function in the tracking framework and their execution was performed on the GPU. Reference Figure 6.2 below for a diagram of the GPU implementation.

![Parallel algorithm of detection tracking system](image)

**Figure 6.2:** Parallel algorithm of detection tracking system [Limprasert et al., 2013].

The slowest profiled functions were the detection and likelihood functions, both of which were parallelized using the data-parallel technique. The detection function is divided into grids and each grid is processed by a thread. The likelihood function takes the same approach by having each thread process the computation of a particle.
Chapter 7

Methodology

7.1 Central Research Questions

The practical aspect of our work makes use of sequential and parallel implementations (by [Limprasert et al., 2013]) of an object tracking algorithm which utilizes a particle filter. The questions central to our research are as follows:

1. What are the challenges of migrating an existing object tracking system to an unintended platform?

2. What is the practicality of formulating a particle filter using an auto-parallelizing language such as SaC?

3. Will the automatically generated code using the SAC compiler tool chain have better performance and portability?

7.1.1 Problems To Address

It is necessary to take into consideration the challenges one can face when performing code migration. This is important, especially in our case, where the characteristic of the existing implementation, for performance reasons, may have been specifically tailored and thus tightly coupled to its intended hardware platform. Migration may not only pose difficulties performance wise, but also build wise. Being written approximately four years prior, third party libraries utilized then may no longer offer legacy support or simply may not exist for our intended environment. Further, assuming a successful build, writing an adaptation of a particular aspect of the particle filter requires us to construct an accurate mental model of the system and understand its
data flows. The level of difficulty of this has a direct dependency on the technical documentation available.

A justification for determining the practicality, as it relates to the second question, is necessary since we are essentially seeking to replace functions initially written using a standard language (in this case, C++ and CUDA) with SAC functions. The SAC functions therefore need to be written, using a black box Software Engineering concept, so that they can be seamlessly integrated with the existing code. The SAC code must accept the same input and produce the same output irrespective of how the intermediate processing is executed. Though based on the C Programming language, the current evolution of SAC only officially supports primitive data types which mandates a necessary translation step for the input data into a more “SAC friendly” format.

With respect to the third research question in section 7.1, we aim to examine how efficiently the SAC generated code serves to alleviate the programmer’s burden since performance tuning is manual and can be highly architecturally specific. When introducing their parallel GPU design, [Limprasert et al., 2013] admits, “For a worker a task can be performed in a sequential mode. When a number of worker increase the task can be split and performed in a parallel mode, where many subtasks are performed at the same time. The idea is very simple but in practice data transfer between the workers or processing cores has a limit. This limit makes the task splitting difficult and programmers have to concern about the different levels of memory architecture in the target hardware.”.

Programmers of parallel systems are required to amass and comprehend technical architectural knowledge of the target hardware so that it may be used as economically as possible. A negative side-effect of this endeavor is the authoring of code that cannot be easily migrated to alternative architectures without (possibly non-trivial) adaptation in the interest of performance. Tedious code migration may be attributed, in part, to the continuous rapid evolution of hardware, most notably GPUs.

7.1.2 Approach

This section provides an overview of our approach to the primary research in order to contextualise our methodology.

The existing sequential and parallel implementations are respectively written in C++ and C++ with CUDA; each application is built using the Microsoft Visual Studio
2008 IDE for a Windows platform. The implementation details of the prior applications direct the initial step of our approach: its adaptation for a Linux platform.

Adaptation of the existing code is done primarily due to SAC currently being unavailable for the Windows platform and to reduce unintended discrepancies in result data. To this end, we need an identical hardware and software platform for accurate performance measurement. Each application has an “offline” variant meaning that a live stream of video is not necessary during execution, instead, a series of static images on disc simulated live input. In keeping with maintaining accurate and consistent results we prefer the offline variants since mirroring the same live input across executions would prove challenging, and result in measurements relating to the number of objects being tracked. It is important to note that evaluating the efficiency of the object tracking algorithm in general is not central to our research, only to investigate the difficulty and evaluate a particular aspect: particle filtering.

At this point of our work we make the decision to first proceed with the sequential version of the existing implementation with the intention of applying the identical steps (from this point) to the parallel implementation.

The next step entails profiling the adapted code to confirm our expectations of the anticipated bottleneck function(s). In this case, the likelihood step of the particle filter is highlighted as the primary consumer of processor time, a SAC equivalent of this function is then attempted. An analysis of the inputs and outputs of the likelihood function is performed to allow for a verification of correctness of the SAC implementation. This analysis entails breaking down and expanding any complex data types to a primitive form and persisting their values to file per function call.

We persist the inputs and outputs of the likelihood function to serve two purposes:

- As a convenient way to provide input to our implementation and allow for external testing prior to integration. This simplifies development and eliminates the burden of performing simultaneous integration.
- Enables a verification of functionality.

Finally, we compare the output of our SAC implementation against the output of the original implementation in order to determine correct functionality. We then repeat this procedure for the parallel aspect of our work.
Chapter 8

Findings

In this section we discuss findings relevant to our research questions mentioned in section 7.1. We will first look at the portability challenges faced followed by the practicality of formulating an aspect of the particle filter using SaC. Finally, the performance results of our implementation will be compared to the existing implementation.

8.1 Portability Challenges

The migration challenges discussed in the following subsections aim to outline the challenges faced for each implementation and what was required to achieve a successful build. Porting effort and runtime evaluation for each will also be discussed. Our goal here is to migrate the existing sequential and parallel object tracking system from a Windows to Linux environment. We now proceed to first outline the generic issues associated with migrating both versions.

8.1.1 General Challenges

The overarching problem with the existing code is the lack of technical documentation which stagnated the attainment of successful builds. This phase of our work consumed a significant percentage of our available time since the specification of the correct version of dependent libraries required dynamic discovery. This discovery process was iterative and typically entailed attempting a build, then determining what libraries were required and attaining those libraries. External to the aforementioned documentation issue, legacy libraries for the Linux platform often had sub dependencies no longer officially supported for our distribution of Linux (see Appendix 11). This served to further compound the issue and required various workarounds.
Additionally, an issue shared by both versions of the existing code was their use of the OpenCV (Open Source Computer Vision) library. OpenCV is an library of image processing functions aimed at real time computer vision. Each existing implementation therefore made use of functions from within this library. A stalling point in our work upon encountering this library is attributed to a lack of prior knowledge and experience. The OpenCV binaries required by the existing object tracking system needed to be built specific to the architecture of our environment. This led to a time consuming (again attributed to lack of experience) process of configuring and executing the necessary build tools.

A final commonality was that of determining the correct compiler version to build each system. Through using a recent distribution of Linux (see Appendix 11) the default compiler was initially attempted for compilation. Through constant compilation failure we were forced to meticulously search library documentation in order to determine the correct compiler version (g++4.4).

8.1.2 Sequential C++

Apart from the generic issues discussed in section 8.1, the only other challenges faced involved re-writing pieces of Windows specific code to allow for compilation in a Linux environment. One example of this is the replacement of the Windows “DWORD” data type with a suitable Linux compilable “unsigned long” equivalent. Other code changes involved replacing Windows specific function calls with their Linux equivalents. For example; an interface function was created to mask calls to a Windows timing function where these calls were passed to a Linux equivalent.

8.1.2.1 Porting Effort

The sequential implementation was attempted after achieving prior success with its parallel counterpart. Admittedly, the prior experience gained through achieving a CUDA build (see 8.1.3) resulted in this being a less tedious process. Also, since this version of the object tracking system was sequential and targeted for the CPU, there were no additional platform specific build configurations to resolve. The porting effort in this instance did consume some amount of our time, but in the end did not prove to be intensely difficult.
8.1.3 CUDA

Prior to the general challenges discussed in 8.1.1, approaches were taken that did not work out for various reasons. Because we were working with code written in CUDA, we initially attempted a configuration using the latest version of the API. Unfortunately, we were unable to achieve this for our Linux platform for the following two reasons:

1. The latest version of the CUDA toolkit had technical incompatibilities with our environment.
2. The parallel implementation was written for a version of CUDA with functions no longer supported by the most current.

With respect to point one, we commenced by installing the latest version of the CUDA toolkit and iteratively worked our way back through previous versions until a compatible toolkit was identified. The more recent versions of CUDA did not work because of incompatibility issues with our environment (which is not in agreement with the official documentation). Other Linux distributions were attempted before falling back to earlier versions of the toolkit. This phase of our work was especially arduous due to the graphical Linux system being broken after every failed attempt with no simple way of recovering. In the interest of time, we took the path of resuming with a fresh installation of the Linux distribution of choice. We were eventually also forced to regress to an earlier CUDA capable device before attaining a successful toolkit installation. Essentially, the balance required at this stage was compatibility among the hardware, operating system and the existing parallel implementation.

The existing object tracking system utilized CUDA version 3.2; according to NVIDIA’s documentation, the toolkit underwent radical changes after version 4.0. This meant that software written using version 4.0 and before needed some form of refactoring in order make use of more recent toolkits. In the interest of time, and staying within project scope, we opted not to perform any code refactoring. The reason for desiring a more recent toolkit version was to automatically take better advantage of CUDA compatible hardware.

Similar to its sequential counterpart, some modification minor code changes were necessary in order to fully migrate from a Windows to Linux environment. It is worthy to make mention of several additional libraries required which meant an even more time consuming build configuration process.
8.1.3.1 Porting Effort

The porting effort for this version of the object tracker proved to be significantly more challenging and time consuming than its sequential counterpart. The challenges faced during the initial phase of configuring in combination with the general challenges (8.1.1) resulted in the porting of this version quite a formidable task.

8.2 SaC Adaptation

The SaC software of the object tracking algorithm did not currently exist; we therefore identified the most performance intensive function of the existing code for Sac adaptation. After profiling the existing sequential implementation, the most intensive step of the object tracking system revealed itself to be the likelihood function of the particle filter.

Our intention here with the SaC implementation of the likelihood function was to precisely simulate its execution as if it was being used in a complete object tracking system. To this end, the approach we took was one of extracting all the data passed into the existing likelihood function, for each call, and using this extracted data as input for our likelihood function.

Because SaC does not currently support complex data types, any data of a non-primitive type needed to be simplified to its primitive form. This process involved examining the composition of records (structures) being used by the object tracking algorithm so that the types of their primitive members could be determined. Out of the eight arguments of the likelihood function, six of these were complex data types and all being references to a respective array. For each function call we persisted the data of all parameters to file; this required code (see Appendix B) which expanded every complex type to its primitive form before persisting to file.

The body of the existing likelihood function was copied verbatim as the body of SaC implementation, necessitating minor code adjustment to use array indexes to access “structure members” instead of conventional member access. Because of the verbatim implementation, a separate SaC function was written to read the previously persisted files and re-construct them in a way that mirrored their initial complex types (see Appendix B).
The characteristic of one particular structure member of a complex data type was that it referenced a potentially large character array. SaC arrays currently cannot be constructed using strings or individual characters so an alternate representation, which allowed for identical functionality, had to be formulated.

The likelihood function performs several iterative operations on multiple arrays which makes it a prime candidate for re-writing using with-loops; a native SaC construct which enables code optimizations. Attempts were made in this direction but were eventually abandoned due to time constraints and a lack of complete understanding of the likelihood algorithm.

### 8.2.1 Adaptation Effort

The SaC adaptation effort proved laborious due to the “type conversion” required. Having to adapt code initially written in a language supportive of complex types mandated a conversion step for the SaC adaptation process. The difficulty was further compounded by the necessary reconstructing of the data types to fit the verbatim code. Admittedly, a more in depth understanding of the likelihood algorithm may have rendered the latter step unnecessary.

### 8.3 Performance Results

The sequential likelihood SaC function is compared against the sequential C++ implementation where the results are depicted in table 8.1 and accompanying graph 8.1.

<table>
<thead>
<tr>
<th>Execution</th>
<th>SaC runtime (seconds)</th>
<th>C++ runtime (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>235.3088</td>
<td>62.62</td>
</tr>
<tr>
<td>2</td>
<td>236.8217</td>
<td>62.79</td>
</tr>
<tr>
<td>3</td>
<td>236.0877</td>
<td>62.9</td>
</tr>
</tbody>
</table>

Table 8.1: Comparison of SaC and C++ likelihood runtimes

Our results show a significantly better performance of the initial C++ implementation of the likelihood function. However, the poor performance of SaC is attributed to the very poor code adaptation which does not take advantage of native language features to achieve better performance. As stated in section 8.2, the SaC implementation is a verbatim copy from the original source.
Figure 8.1: SaC and C++ likelihood runtimes
Chapter 9

Discussion

The objectives of our work were to determine portability of an object tracking algorithm and to determine how well SaC generated code compares performance wise to the said algorithm. The findings of our work indicate reasonable, time consuming challenges when attempting to migrate the parallel implementation of an object tracking algorithm. Our discussion first covers the portability challenges followed by remarks on the performance findings.

The primary problem of using pre-existing code, in this case, was a lack of technical documentation. Simply attaining a successful build proved to be quite a challenge. This aspect of our work consumed a significant percentage of our time because the specification of the dependent libraries was not documented. Our migration attempt from Windows to Linux was therefore especially tedious. Migration was basically an iterative process of determining missing libraries from compiler errors. Resolving required libraries rarely proved to be enough; the correct version needed to be figured out as well. Doing so for software written approximately four years ago was difficult since legacy libraries had to be sought.

A secondary challenge of working with pre-existing code, especially of this complexity, is the time required to form a working mental model of the system. Establishing an understanding of how the data flows through the code required reverse engineering and profiling. Working out how the system fit together required building up a clear picture of dependencies and formulating an architectural view of the components and how they interface with each other. Once again, this took time to accomplish.

The domain of heterogeneous computing inherently poses portability challenges if similar performance is required for varying computing platforms. Usually, to gain the best
performance an application is tailored to a specific combination of hardware and software. Therefore, merely attaining a successful compilation on another platform does not necessarily guarantee the same performance without architecture specific tuning. In this case, the portability challenge resided primarily in the building phase of migration due to library dependencies; Linux equivalents of Windows libraries were needed.

With respect to the performance findings, our SaC implementation did not achieve better performance even though all operations performed by the likelihood function are array based. At this point, we believe that the outcome is a result of code very poorly adapted for SaC and does not take advantage of the SaC language features. More time is needed to better adapt the multiple array operations so that automatic code optimizations can take place within the SaC compiler.
Chapter 10

Conclusion

In examining the portability of object tracking code, to an unintended platform, with the intent of gaining better performance by using an auto-parallelizing language provided insight to portability and performance. The particular challenges for our setting was the migration of the existing sequential and parallel implementation from Windows to Linux. Porting the existing software was a two phase process of transportation followed by adaptation. It is during the transportation phase of our work we encountered the most time consuming difficulties of resolving build errors. Adaptation merely required a few minor code changes which were specific to the Windows platform.

The SaC software did not exist prior and was crafted by adapting the most compute intense function of the existing implementations, the likelihood function. While SaC is designed for array based programming, we found it’s performance was less efficient than that of the existing C++ version. However, this may be directly attributed to that fact that almost no adaptation steps were taken to take advantage of SaC’s optimization features.

10.1 Future Work

Future work would entail the continued adaptation of our implementation of the likelihood function to make full use of the SaC language features. We would also like to continue on with developing a parallel SaC implementation for performance evaluation against the existing parallel implementation. Embarking on these pieces of research, particularly the latter, would provide much better results about the efficiency and portability of auto-generated, parallel code.
10.2 Reflection

Through attempting this project I have been able to gain experience on what is involved with building software for a Linux platform. Most of my acquired skills and insights are especially technical and, though quite necessary, out of the scope this discussion. In hindsight, I was more excited about the testing and evaluation phases of this work and therefore over-looked the time that might have been required for the initial phases, i.e. to port the existing software.

Another learning experience for came from my realization of the time that is needed to build a mental model of software written by someone other than myself. I wrote the SaC implementation as simply as possible and basically took an “as long as it works” approach. This step was intended to serve only as a verification of correctness where I would have iteratively constructed a more SaC compliant version. However, I quickly realized how much at a loss I was when it came to doing so simply because I was not entirely clear on the technical functionality of the existing code. Having a high-level, general understanding was not enough.

Overall, through this project I’ve learned the importance of performing a careful evaluation of each step necessary to achieve an experiment and to not overlook any phase as trivial. I sincerely did not anticipate porting to have been such a time consuming and technical process. Learning about the importance of establishing a working mental model of software I need to adapt came as a welcome surprise. Though the project did not go as far, and achieve the level of success, as I desired, it has been a worthy learning experience.
Chapter 11

Appendix A

11.1 Technology & Platform Details

This appendix provides specific details of the hardware and software used for our research. Necessary technology details are given to further support the reproducibility of the methodology and the reliability and validity of our findings. Exact hardware details and version numbers for software libraries are given.

11.2 Hardware

The memory, CPU and GPU details are presented here.

11.2.1 Memory

The memory information was obtained using the following Linux command. Reference section 11.3 for specific software details.

$ dmidecode --type memory

The output is as follows:

Handle 0x003F, DMI type 17, 28 bytes
Memory Device
Array Handle: 0x003D
Error Information Handle: Not Provided
Total Width: 64 bits
Data Width: 64 bits
Size: 2048 MB
Form Factor: DIMM
Set: 1
Locator: XMM1
Bank Locator: Not Specified
Type: DDR3
Type Detail: Synchronous
Speed: 1333 MHz
Manufacturer: JEDEC ID:02 FE
Serial Number: DD0F6EE0
Asset Tag: Not Specified
Part Number: EBJ21UE8BDF0-DJ-F
Rank: 2

Handle 0x0041, DMI type 17, 28 bytes
Memory Device
Array Handle: 0x003D
Error Information Handle: Not Provided
Total Width: 64 bits
Data Width: 64 bits
Size: 2048 MB
Form Factor: DIMM
Set: 2
Locator: XMM3
Bank Locator: Not Specified
Type: DDR3
Type Detail: Synchronous
Speed: 1333 MHz
Manufacturer: JEDEC ID:02 FE
Serial Number: DA0F6814
Asset Tag: Not Specified
Part Number: EBJ21UE8BDF0-DJ-F
Rank: 2
11.2.2 CPU

The CPU information was obtained using the following Linux command. Reference section 11.3 for further software details.

```
$ lscpu
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>x86_64</td>
</tr>
<tr>
<td>CPU op-mode(s)</td>
<td>32-bit, 64-bit</td>
</tr>
<tr>
<td>Byte Order</td>
<td>Little Endian</td>
</tr>
<tr>
<td>CPU(s)</td>
<td>8</td>
</tr>
<tr>
<td>On-line CPU(s) list</td>
<td>0-7</td>
</tr>
<tr>
<td>Thread(s) per core</td>
<td>2</td>
</tr>
<tr>
<td>Core(s) per socket</td>
<td>4</td>
</tr>
<tr>
<td>Socket(s)</td>
<td>1</td>
</tr>
<tr>
<td>NUMA node(s)</td>
<td>1</td>
</tr>
<tr>
<td>Vendor ID</td>
<td>GenuineIntel</td>
</tr>
<tr>
<td>CPU family</td>
<td>6</td>
</tr>
<tr>
<td>Model</td>
<td>30</td>
</tr>
<tr>
<td>Stepping</td>
<td>5</td>
</tr>
<tr>
<td>CPU MHz</td>
<td>1199.000</td>
</tr>
<tr>
<td>BogoMIPS</td>
<td>5585.99</td>
</tr>
<tr>
<td>Virtualization</td>
<td>VT-x</td>
</tr>
<tr>
<td>L1d cache</td>
<td>32K</td>
</tr>
<tr>
<td>L1i cache</td>
<td>32K</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256K</td>
</tr>
<tr>
<td>L3 cache</td>
<td>8192K</td>
</tr>
<tr>
<td>NUMA node0 CPU(s)</td>
<td>0-7</td>
</tr>
</tbody>
</table>

Table 11.1: CPU Details
11.2.3 GPU

The GPU details were obtained using the following command:

```
$ sudo lspci -v -s 01:00.0
```

```
01:00.0 VGA compatible controller: NVIDIA Corporation GF119 \[GeForce GT 520\] (rev a1) (prog-if 00 [VGA controller])
Physical Slot: 1
Flags: bus master, fast devsel, latency 0, IRQ 49
Memory at f2000000 (32-bit, non-prefetchable) [size=16M]
Memory at e8000000 (64-bit, prefetchable) [size=128M]
Memory at f0000000 (64-bit, prefetchable) [size=32M]
I/O ports at 1100 [size=128]
[virtual] Expansion ROM at f3080000 [disabled] [size=512K]
Capabilities: [60] Power Management version 3
Capabilities: [68] MSI: Enable+ Count=1/1 Maskable- 64bit+
Capabilities: [78] Express Endpoint, MSI 00
Capabilities: [b4] Vendor Specific Information: Len=14 <?>
Capabilities: [100] Virtual Channel
Capabilities: [128] Power Budgeting <?>
Capabilities: [600] Vendor Specific Information: ID=0001 Rev=1 Len=024 <?>
Kernel driver in use: nvidia
```

11.3 Software

This section lists the details of the executing environment along with the versions of SDKs and other tools used. The list also includes tools and libraries needed to build the existing implementation.

The listing of the details of the execution environment are as follows:

- Linux eklow 3.16.0-43-generic #58~14.04.1-Ubuntu SMP
  Mon Jun 22 10:21:20 UTC 2015 x86_64 x86_64 x86_64 GNU/Linux
- OpenCV 2.2.0
- g++-4.4 (Ubuntu/Linaro 4.4.7-8ubuntu1) 4.4.7
- CUDA 4.0
- flycapture2-2.7.3.19-i386-pkg (1).tgz
- sac2c v1.00-beta (Haggis And Apple)product rev 18629 linux-gnu_x86_64
Chapter 12

Appendix B

12.1 Code Listings

This appendix provides the code listings for the likelihood function as implemented in SAC, as well as other pieces of code written which assisted in the process of extracting and transforming data into a “SAC Friendly” format.

12.1.1 Sequential Likelihood Implementation

Below is the complete listing the likelihood function, written in SAC, that first reads the data for each iteration from file.

```c
#define IMG_HEIGHT 0
#define IMG_WIDTH 1
#define IMG_IMG_DATA 2
#define IMG_WIDTH_STEP 3
#define IMG_N_CHANNELS 4

#define X_EX 0
#define X_VS 1
#define X_H 2
#define X_SX 3
#define X_SY 4
#define X_VX 5
#define X_VY 6

#define EPARAM_EA 0
#define EPARAM_EB 1
#define EPARAM_EC 2
#define EPARAM_XC 3
#define EPARAM_YC 4
#define EPARAM_AREA 5
#define EPARAM_X1 6
#define EPARAM_Y1 7
#define EPARAM_X2 8
```
# define EPARAM_Y2 9
use StdIO : all;
use Array : all;
#define BUFFER_SIZE 64
#define ITERATIONS 794
#define kmax 16
#define nmax 512
#define spf 4
#define sqspf 16.0f
#define invisibleweight 0.06f

float[.,.,.] loadData(String::string file_name)
{
    int member_amount, iterations, items_between_iterations;
    path = "var_dump2/raw/";
    meta_path = String::+("meta_", file_name);
    err, stream = fopen(String::+(path,meta_path), "r");
    amt, member_amount, iterations, items_between_iterations =
        fscanf(stream, "%d %d %d");
    fclose(stream);
    printf("%s: %d %d %d\n", String::+(path,meta_path),
        member_amount, iterations, items_between_iterations);
    err, stream = fopen(String::+(path,file_name), "r");
    data = genarray([iterations * items_between_iterations], 0.0f);
    index = 0;
    while(!feof(stream))
    {
        l = fscanf(stream, BUFFER_SIZE);
        if(String::strcmp("\n", l) != 0)
        {
            data[index] = String::tof(l);
            index++;
        }
    }
    fclose(stream);
    collection = reshape([iterations, (items_between_iterations /
        member_amount), member_amount], data);
    return (collection);
}

int likelihood(float[.,.] numofgrids, float[.,.] grids, float[.,.] Eparam_grid,
    float[.,.] sum_grid, float[.,.] X, float[.,.] Eparam_X,
    float[.,.] weight, float[.,.] img, float[.,.] imgData)
{
    float xt, yt;
float h, w;
int numoflayer;
int ismember, actpx;
int img_height;
int img_width;
int x,y,n,k,m,i,temp;
float normsum;
normsum=0.0f;
actpx=0;
w = genarray([kmax*nmax], 0.0f);
sumps = genarray([kmax*nmax], 0.0f);
kernellvl = genarray([kmax], 0.0f);
LUT = [0.0f, sqspf*1.0000f, sqspf*0.6667f, sqspf*0.5714f, sqspf *0.5333f, sqspf*0.5161f];
inomega = genarray([kmax], 0);
sum_grid = genarray(shape(sum_grid), 0.0f);

// initialization
C = with{
    (0, 0) <= [i,j] < [kmax,kmax]{if(i==j){res=1;}
        else{res=0;}}: res;
}:
    genarray([kmax,kmax], 0);
C = reshape([1, kmax*kmax], C);

img_height = toi(img[0][IMG_HEIGHT]);
img_width = toi(img[0][IMG_WIDTH]);
img_widthStep = toi(img[0][IMG_WIDTH_STEP]);
img_nChannels = toi(img[0][IMG_N_CHANNELS]);

for (y=0; y<img_height; y+=spf){
    for (x=0; x<img_width; x+=spf){
        if(toi(imgData[(y*img_widthStep+x*img_nChannels)][0]) != 0){
            actpx++;
            ismember=0;
            for(n=0; n<nmax; n++){
                numoflayer=0;
                for(k=0; k<kmax; k++){
                    inomega[k]=0;
                    kernellvl[k]=0.0f;
                    if(toi(X[k*nmax+n][X_VS])==1){
                        if(toi(Eparam_X[k*nmax+n][EPARAM_X1])<x && toi(Eparam_X[k*nmax+n][EPARAM_Y1])<y && x<toi(Eparam_X[k*nmax+n][EPARAM_X2]) && y<toi(Eparam_X[k*nmax+n][EPARAM_Y2])){
                            xt=tof(x)-Eparam_X[k*nmax+n][EPARAM_XC];
                            yt=tof(y)-Eparam_X[k*nmax+n][EPARAM_YC];
                            h = 1.0f - Eparam_X[k*nmax+n][EPARAM_EA]*xt*xt - Eparam_X[k*nmax+n][EPARAM_EB]*xt*yt - Eparam_X[k*nmax+n][EPARAM_EC]*yt*yt;
                            if(-0.2f<h){
                                kernellvl[k]=h;
                                numoflayer++;
                                ismember=1;
                            }else if(0.5f<h)inomega[k]=1;
                        }
                    }
                }
            }
        }
    }
}
if (0 < numoflayer) {
    if (numoflayer < 5) {
        for (k = 0; k < kmax; k++) {
            if (toi (X[k*nmax+n][X_VS]) == 1) {
                sumps[k*nmax+n] = sumps[k*nmax+n] + LUT[numoflayer] * kernellvl[k];
            }
        }
        else printf("numoflayer out of bound: %d \n", numoflayer);
    }
    for (k = 0; k < kmax - 1; k++) {
        if (inomega[k] == 1) {
            for (m = k + 1; m < kmax; m++) {
                if (inomega[m] == 1) {
                    C[k*kmax+m] = 1;
                    C[m*kmax+k] = 1;
                }
            }
        }
    }
    // compute number of active pixels in detection ellipses
    if (ismember == 0) {
        temp = toi (numofgrids[0]);
        for (i = 0; i < temp; i++) {
            xt = x - Eparam_grid[i][EPARAM_XC];
            yt = y - Eparam_grid[i][EPARAM_YC];
            h = 1 - Eparam_grid[i][EPARAM_EA] * xt * xt - Eparam_grid[i][EPARAM_EB] * xt * yt -
            Eparam_grid[i][EPARAM_EC] * yt * yt;
            if (0.0f < h) {
                // see note
                sum_grid[i] = sum_grid[i] + sqspf;
            }
        }
    }
    // compute weight
    for (n = 0; n < nmax; n++) {
        for (k = 0; k < kmax; k++) {
            if (X[k*nmax+n][X_VS] > 0.0f) {
                normsum = sumps[k*nmax+n] / (0.5f * Eparam_X[k*nmax+n][EPARAM_AREA]);
                w[k*nmax+n] = Math::exp(-10 * (1 - normsum) * (1 - normsum));
            } else {
                w[k*nmax+n] = invisibleweight;
            }
            weight[k*nmax+n] = 1.0f;
        }
    }
    // joint weight
    for (k = 0; k < kmax; k++) {
        for (m = 0; m < kmax; m++) {
            }
if(C[k*kmax+m] == 1) {
    for(n = 0; n < nmax; n++) {
        weight[k*nmax+n] = weight[k*nmax+n] * w[m*nmax+n];
    }
}

return actpx;
}

int main() {
    numofgrids = loadData("raw_numofgrids.txt");
grids = loadData("raw_grids.txt");
Eparam_grids = loadData("raw_Eparam_grids.txt");
sum_grid = loadData("raw_sum_grids.txt");
X = loadData("raw_x.txt");
weight = loadData("raw_weight.txt");
img = loadData("raw_img.txt");
Eparam_X = loadData("raw_Eparam_X.txt");
imgData = loadData("raw_img_imgData.txt");

actpx = with{
    (. <= [i] <= .){
        res = likelihood(numofgrids[i], grids[i], Eparam_grids[i],
                        sum_grid[i], X[i], Eparam_X[i], weight[i], img[i], imgData[i]);
    }: res;
}: genarray([[ITERATIONS], 0]);

print(actpx);
return (0);
}
12.1.2 Meta Input Generator

This SAC program was used to generate meta data for each data file to be used by the sequential likelihood implementation. The resulting meta file consisted of three numbers in one line representing the amount of members of a data type (just one if the type is primitive), iterations and items between iterations. The meta file of the corresponding input file is first read where these values are used for correct parsing.

```c
use StdIO: all;
use Array: all;

#define BUFFER_SIZE 64

#define RAW_SUM_GRID_MEMBER_AMT 1
#define RAW_GRIDS_MEMBER_AMT 2
#define RAW_X_MEMBER_AMT 7
#define RAW_EPARAM_MEMBER_AMT 10
#define RAW_IMG_MEMBER_AMT 5
#define RAW_WEIGHT_MEMBER_AMT 1

void generate_meta(String::string fileName, int memberAmount)
{
    path = "var_dump2/raw/";
    err, stream = fopen(String::+(path, fileName), "r");
    iterations = 0;
    items_between_iterations = 0;
    index = 0;

    while (!feof(stream))
    {
        l = fscanf(stream, BUFFER_SIZE);
        if(String::strcmp("\n", l) == 0)
        {
            iterations++;
        }
    }

    itemsCount = genarray([iterations], 0);
    rewind(stream);

    while (!feof(stream))
    {
        l = fscanf(stream, BUFFER_SIZE);
        if(String::strcmp("\n", l) == 0)
        {
            itemsCount[index] = items_between_iterations;
            items_between_iterations = 0;
            index++;
        }
        else
        {
            items_between_iterations++;
        }
    }
```
fclose ( stream );

flag = 0;
itemsCount [0] = itemsCount [1];

// Because this file has a very long line which may be causing errors
if ( String :: strcmp ("raw_img_imgData.txt", fileName) == 0 )
{
    itemsCount [0] = itemsCount [2];
    itemsCount [1] = itemsCount [2];
}

for (i = 0; i < iterations; i++)
{
    for (j = 0; j < iterations; j++)
    {
        if ( itemsCount [i] != itemsCount [j] )
        {
            flag = 1;
        }
    }
}

if ( flag == 1 )
{
    printf ( "Warning: Error with: %s\n", fileName );
    print ( itemsCount );
}

outFile = String :: +( "meta_", fileName );
err, stream = fopen ( String :: +( path, outFile ), "w" );

fprintf ( stream, "%d %d %d", memberAmount, iterations, itemsCount [0] );

fclose ( stream );

printf ( "All is equal\n" );
printf ( "Items between: %d\n", itemsCount [0] );
printf ( "Iterations: %d\n", iterations );
}

int main()
{
    int member_amount;
    err, stream = fopen ( "raw_list.txt", "r" );
    err, stream2 = fopen ( "raw_list_group.txt", "r" );

    while ( !feof ( stream ) )
    {
        file_name = fscans ( stream, 32 );
        amt, member_amount = fscanf ( stream2, "%d" );

        if ( member_amount != 0 )
        {
            // Further code...
        }
    }
103    printf("\n");
104    generate_meta(file_name, member_amount);
105  }
106 }
107
108 fclose(stream);
109 fclose(stream2);
110 return (0);
111}

12.1.3 Data Dump Header

In order to analyze the existing implementation some augmentation of its code was necessary. The code in this listing defines a generic data type (a “Dump_Item”) along with size and member name information. The function dump_vars is executed each time the likelihood function is called, where it iteratively writes to file the values of each argument.

```c
#include <stdio.h>
#include <stdlib.h>

#define PATH_LENGTH 64
#define NAME_LENGTH 16
#define VALUE_LENGTH 128
#define MYPOINT_MEMBER_AMT 2
#define EPARAM_MEMBER_AMT 10
#define SUM_GRID_MEMBER_AMT 1
#define STATE_MEMBER_AMT 7
#define WEIGHT_MEMBER_AMT 1
#define IPL_IMAGE_MEMBER_AMT 5
#define IPL_IMG_DATA_AMT 1
#define NAME_LEN 16
#define NUM_OF_GRIDS_MEMBER_AMT 1

typedef struct Dump_Item{
    char name[NAME_LENGTH];
    char *value;
} DUMP_ITEM;

cchar MYPOINT_NAMES[MYPOINT_MEMBER_AMT][NAME_LEN] = {"X", "Y"};
cchar EPARAM_NAMES[EPARAM_MEMBER_AMT][NAME_LEN] = {"Ea", "Eb", "Ec",
    "xc", "yc", "area", "x1", "y1", "x2", "y2"};
cchar SUM_GRID_NAMES[SUM_GRID_MEMBER_AMT][NAME_LEN] = {"sum_grid"};
cchar STATE_NAMES[STATE_MEMBER_AMT][NAME_LEN] = {"ex", "vs", "h", "sx",
    "sy", "vx", "vy"};
cchar WEIGHT_NAMES[WEIGHT_MEMBER_AMT][NAME_LEN] = {"weight"};
cchar IPL_IMAGE_NAMES[IPL_IMAGE_MEMBER_AMT][NAME_LEN] = {"height",
    "width", "imageData", "widthstep", "nChannels"};
cchar NUM_OF_GRID_NAMES[WEIGHT_MEMBER_AMT][NAME_LEN] = {"numofgrids "};
cchar IPL_IMG_DATA_NAMES[IPL_IMAGE_MEMBER_AMT][NAME_LEN] = {"imageData"};
```
void dump_vars(DUMP_ITEM *dItem, char const *file_name, long int item_count)
{
    long int i;
    FILE *fp, *fp2;
    char path[PATH_LENGTH] = "var_dump/";
    char raw_path[PATH_LENGTH] = "var_dump/raw_";
    const char delimiter = '_';
    strcat(path, file_name);
    strcat(raw_path, file_name);
    fp = fopen(path, "a");
    fp2 = fopen(raw_path, "a");
    if(fp == NULL )
    {
        printf("Error opening %s for writing!\n", path);
        exit(-1);
    }
    if(fp2 == NULL )
    {
        printf("Error opening %s for writing!\n", raw_path);
        exit(-1);
    }
    fprintf(fp, "%c\n", delimiter);
    fprintf(fp2, "%c\n", delimiter);
    for(i = 0; i < item_count; i++)
    {
        fprintf(fp, "%s: %s\n", dItem[i].name, dItem[i].value);
        fprintf(fp2, "%s\n", dItem[i].value);
        free(dItem[i].value);
    }
    fclose(fp);
    fclose(fp2);
}

12.1.4 Data Dump Functions

void get_memory(char **ptr, int len, const char *function_name)
{
    *ptr = (char *)malloc(sizeof(char) * len);
    if(ptr == NULL )
    {
        printf("Could not allocate memory in function!!: %s\n", function_name);
        exit(-1);
    }
}
void getMypoint(Mypoint *mPoint, DUMP_ITEM *dItem, int index) {
    strcpy(dItem->name, MYPOINT_NAMES[index]);
    get_memory(&dItem->value, VALUE_LENGTH, "getMypoint");

    switch(index) {
    case 0: sprintf(dItem->value, "%f", mPoint->x); break;
    case 1: sprintf(dItem->value, "%f", mPoint->y); break;
    default: printf("Invalid index in function getMypoint: %d\n", index);
             exit(-1);
    }
}

void getEparam(Eparam *eParam, DUMP_ITEM *dItem, int index) {
    strcpy(dItem->name, EPARAM_NAMES[index]);
    get_memory(&dItem->value, VALUE_LENGTH, "getEparam");

    switch(index) {
    case 0: sprintf(dItem->value, "%f", eParam->Ea); break;
    case 1: sprintf(dItem->value, "%f", eParam->Eb); break;
    case 2: sprintf(dItem->value, "%f", eParam->Ec); break;
    case 3: sprintf(dItem->value, "%f", eParam->Ea); break;
    case 4: sprintf(dItem->value, "%f", eParam->Ea); break;
    case 5: sprintf(dItem->value, "%f", eParam->Ea); break;
    case 6: sprintf(dItem->value, "%d", eParam->x1); break;
    case 7: sprintf(dItem->value, "%d", eParam->y1); break;
    case 8: sprintf(dItem->value, "%d", eParam->x2); break;
    case 9: sprintf(dItem->value, "%d", eParam->y2); break;
    default: printf("Invalid index in function getEparam: %d\n", index);
             exit(-1);
    }
}

void getFloat(float *f, DUMP_ITEM *dItem, char *name) {
    strcpy(dItem->name, name);
    get_memory(&dItem->value, VALUE_LENGTH, "getFloat");
    sprintf(dItem->value, "%f", *f);
}

void getInt(int *i, DUMP_ITEM *dItem, char *name) {
    strcpy(dItem->name, name);
    get_memory(&dItem->value, VALUE_LENGTH, "getInt");
    sprintf(dItem->value, "%d", *i);
}

void getState(state *state, DUMP_ITEM *dItem, int index) {
    strcpy(dItem->name, STATE_NAMES[index]);
    get_memory(&dItem->value, VALUE_LENGTH, "getState");
switch(index)
{
    case 0: sprintf(dItem->value, "%f", _state->ex); break;
    case 1: sprintf(dItem->value, "%f", _state->vs); break;
    case 2: sprintf(dItem->value, "%f", _state->h); break;
    case 3: sprintf(dItem->value, "%f", _state->sx); break;
    case 4: sprintf(dItem->value, "%f", _state->sy); break;
    case 5: sprintf(dItem->value, "%f", _state->vx); break;
    case 6: sprintf(dItem->value, "%f", _state->vy); break;
    default:
        printf("Invalid index in function getState: %d\n", index);
        exit(-1);
}

void convertChars(char *str)
{
    sprintf(str, "%d", strlen(str));
}

void getIplImage(IplImage *img, DUMP_ITEM *dItem, int index)
{
    strcpy(dItem->name, IPL_IMAGE_NAMES[index]);
    if(index == 2)
    {
        get_memory(&dItem->value, img->height*img->width, "getIplImage");
        convertChars(img->imageData);
    }
    else
    {
        get_memory(&dItem->value, VALUE_LENGTH, "getIplImage");
    }
    switch(index)
    {
    case 0: sprintf(dItem->value, "%d", img->height); break;
    case 1: sprintf(dItem->value, "%d", img->width); break;
    case 2: sprintf(dItem->value, "%s", img->imageData); break;
    case 3: sprintf(dItem->value, "%d", img->widthStep); break;
    case 4: sprintf(dItem->value, "%d", img->nChannels); break;
    default:
        printf("Invalid index in function getIplImage: %d\n", index);
        exit(-1);
    }
}

void getIpImageData(IplImage *img, DUMP_ITEM *dItem, char *name)
{
    strcpy(dItem->name, name);
    get_memory(&dItem->value, img->height*img->width, "getIpImageData");
    sprintf(dItem->value, "%s", img->imageData);
void dump_test (int numofgrids, Mypoint *grids, Eparam *Eparam_grid,
   float *sum_grid,
   state *X, Eparam *Eparam_X, float *weight, IplImage *img) {

    long int numofgridsAmt = NUM_OF_GRIDS_MEMBER_AMT;
    long int pointItemAmt = numofgrids*MYPOINT_MEMBER_AMT;
    long int eparamItemAmt = numofgrids*EPARAM_MEMBER_AMT;
    long int sumGridItemAmt = numofgrids;
    long int xItemAmt = kmax*nmax*STATE_MEMBER_AMT;
    long int weightItemAmt = kmax*nmax;
    int imgItemAmt = IPL_IMAGE_MEMBER_AMT;
    int imgDataAmt = IPL_IMG_DATA_AMT;

    // May need to do dynamic allocation if mysterious segfaults occur
    DUMP_ITEM numofgridsItems [numofgridsAmt];
    DUMP_ITEM myPointItems [pointItemAmt];
    DUMP_ITEM eparamItems [eparamItemAmt];
    DUMP_ITEM sumGridItems [sumGridItemAmt];
    DUMP_ITEM *xItems = (DUMP_ITEM *) malloc (sizeof (struct Dump_Item) * xItemAmt); // For large array sizes
    DUMP_ITEM weightItems [weightItemAmt];
    DUMP_ITEM imgItems [imgItemAmt];
    DUMP_ITEM imgDataItems [imgDataAmt];
    long int i, j, index;

    // Collect numofgrids values
    for (i = 0; i < numofgridsAmt; i++) {
        getInt (&numofgrids, &numofgridsItems[i], NUM_OF_GRID_NAMES[0]);
    }
    dump_vars(numofgridsItems, "numofgrids.txt", numofgridsAmt);

    // Collect all grids values
    for (i = 0, index = 0; i < numofgrids; i++) {
        for (j = 0; j < MYPOINT_MEMBER_AMT; j++, index++)
            getMypoint (&grids[i], &myPointItems[index], j);
    }
    dump_vars(myPointItems, "grids.txt", pointItemAmt);

    // Collect all Eparam_grid values
    for (i = 0, index = 0; i < numofgrids; i++) {
        for (j = 0; j < EPARAM_MEMBER_AMT; j++, index++)
            getEparam (&Eparam_grid[i], &eparamItems[index], j);
    }
}
Appendix B. Code Listings

```c
dump_vars(eparamItems, "Eparam_grids.txt", eparamItemAmt);

// Collect all sum_grid values
for(i=0; i < numofgrids; i++)
{
    getFloat(&sum_grid[i], &sumGridItems[i], SUM_GRID_NAMES[0]);
}
dump_vars(sumGridItems, "sum_grids.txt", sumGridItemAmt);

// Collect all x values
for(i=0, index=0; i < (kmax*nmax); i++)
{
    for(j =0; j < STATE_MEMBER_AMT; j++, index++)
    {
        getState(&x[i], &xItems[index], j);
    }
}
dump_vars(xItems, "x.txt", xItemAmt);
free(xItems);

// Collect all Eparam_X values
for(i=0, index=0; i < numofgrids; i++)
{
    for(j=0; j < EPARAM_MEMBER_AMT; j++, index++)
    {
        getEparam(&Eparam_X[i], &eparamItems[index], j);
    }
}
dump_vars(eparamItems, "Eparam_X.txt", eparamItemAmt);

// Collect all weight values
for(i=0; i < (kmax*nmax); i++)
{
    getFloat(&weight[i], &weightItems[i], WEIGHT_NAMES[0]);
}
dump_vars(weightItems, "weight.txt", weightItemAmt);

// Collect all img values
for(i=0; i<imgItemAmt; i++)
{
    getIplImage(img, &imgItems[i], i);
}
dump_vars(imgItems, "img.txt", imgItemAmt);

// getIpImageData(img, &imgDataItems[0], IPL_IMG_DATA_NAMES[0]);
// dump_vars(imgDataItems, "img_imgData.txt", imgDataAmt);
```
Chapter 13

Appendix C

13.1 Performance Measurement

Table 13.2 presents the performance profile of the existing object tracking code where the “biggest eater” is the likelihood function. The profiling tool used was gprof, the meaning of the columns of the aforementioned table are summarized in table 13.1 below.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>% Time</td>
<td>The percentage of the total running time of the program used by this function.</td>
</tr>
<tr>
<td>Cumulative Seconds</td>
<td>A running sum of the number of seconds accounted, for by this function and those listed above it.</td>
</tr>
<tr>
<td>Self Seconds</td>
<td>The number of seconds accounted for by this function alone. This is the major sort for this listing.</td>
</tr>
<tr>
<td>Calls</td>
<td>The number of times this function was invoked, if this function is profiled, else blank.</td>
</tr>
<tr>
<td>Self ms/call</td>
<td>The average number of milliseconds spent in this function per call, if this function is profiled, else blank.</td>
</tr>
<tr>
<td>Total ms/call</td>
<td>The average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.</td>
</tr>
<tr>
<td>Name</td>
<td>The name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.</td>
</tr>
</tbody>
</table>

Table 13.1: Performance Profile Column Meanings
Table 13.2: Performance Profile of Existing Object Tracker
Bibliography


